Programming Computer Vision with Python

Jan Erik Solem

Programming Computer Vision with Python

Contents

Preface

Today, images and video are everywhere. Online photo sharing sites and social networks have them in the billions. Search engines will produce images of just about any conceivable query. Practically all phones and computers come with built in cameras. It is not uncommon for people to have many gigabytes of photos and videos on their devices.

Programming a computer and designing algorithms for understanding what is in these images is the field of computer vision. Computer vision powers applications like image search, robot navigation, medical image analysis, photo management and many more.

The idea behind this book is to give an easily accessible entry point to hands-on computer vision with enough understanding of the underlying theory and algorithms to be a foundation for students, researchers and enthusiasts. The Python programming language, the language choice of this book, comes with many freely available powerful modules for handling images, mathematical computing and data mining.

When writing this book I have had the following principles as a guideline. The book should:

- be written in an exploratory style. Encourage readers to follow the examples on their computers as they are reading the text.
- promote and use free and open software with a low learning threshold. Python was the obvious choice.
- be complete and self-contained. Not complete as in covering all of computer vision (this book is far from that!) but rather complete in that all code is presented and explained. The reader should be able to reproduce the examples and build upon them directly.
- be broad rather than detailed, inspiring and motivational rather than theoretical.

In short: act as a source of inspiration for those interested in programming computer vision applications.

Prerequisites and Overview

What you need to know

- Basic programming experience. You need to know how to use an editor and run scripts, how to structure code as well as basic data types. Familiarity with Python or other scripting style languages like Ruby or Matlab will help.
- Basic mathematics. To make full use of the examples it helps if you know about matrices, vectors, matrix multiplication, the standard mathematical functions and concepts like derivatives and gradients. Some of the more advanced mathematical examples can be easily skipped.

What you will learn

- Hands-on programming with images using Python.
- Computer vision techniques behind a wide variety of real-world applications.
- Many of the fundamental algorithms and how to implement and apply them yourself.

The code examples in this book will show you object recognition, content-based image retrieval, image search, optical character recognition, optical flow, tracking, 3D reconstruction, stereo imaging, augmented reality, pose estimation, panorama creation, image segmentation, de-noising, image grouping and more.

Chapter Overview

Chapter 1 Introduces the basic tools for working with images and the central Python modules used in the book. This chapter also covers many fundamental examples needed for the remaining chapters.

Chapter 2 Explains methods for detecting interest points in images and how to use them to find corresponding points and regions between images.

Chapter 3 Describes basic transformations between images and methods for computing them. Examples range from image warping to creating panoramas.

Chapter 4 Introduces how to model cameras, generate image projections from 3D space to image features and estimate the camera viewpoint.

Chapter 5 Explains how to work with several images of the same scene, the fundamentals of multiple-view geometry and how to compute 3D reconstructions from images.

Chapter 6 Introduces a number of clustering methods and shows how to use them for grouping and organizing images based on similarity or content.

Chapter 7 Shows how to build efficient image retrieval techniques that can store image representations and search for images based on their visual content.

Chapter 8 Describes algorithms for classifying image content and how to use them recognizing objects in images.

Chapter 9 Introduces different techniques for dividing an image into meaningful regions using clustering, user interactions or image models.

Chapter 10 Shows how to use the Python interface for the commonly used OpenCV computer vision library and how to work with video and camera input.

Introduction to Computer Vision

Computer vision is the automated extraction of information from images. Information can mean anything from 3D models, camera position, object detection and recognition to grouping and searching image content. In this book we take a wide definition of computer vision and include things like image warping, de-noising and augmented reality[1].

Sometimes computer vision tries to mimic human vision, sometimes uses a data and statistical approach, sometimes geometry is the key to solving problems. We will try to cover all of these angles in this book.

Practical computer vision contains a mix of programming, modeling, and mathematics and is sometimes difficult to grasp. I have deliberately tried to present the material with a minimum of theory in the spirit of *"as simple as possible but no simpler"*. The mathematical parts of the presentation are there to help readers understand the algorithms. Some chapters are by nature very math heavy (chapters 4 and 5 mainly). Readers can skip the math if they like and still use the example code.

[1] These examples produce new images and are more image processing than actually extracting information from images.

Python and NumPy

Python is the programming language used in the code examples throughout this book. Python is a clear and concise language with good support for input/output, numerics, images and plotting. The language has some peculiarities such as indentation and compact syntax that takes getting used to. The code examples assume you have Python 2.6 or later as most packages are only available for these versions. The upcoming Python 3.x version has many language differences and is not backward compatible with Python 2.x or compatible with the ecosystem of packages we need (yet).

Some familiarity with basic Python will make the material more accessible for readers. For beginners to Python, Mark Lutz' book [20] and the online documentation at `http://www.python.org/` are good starting points.

When programming computer vision we need representations of vectors and matrices and operations on them. This is handled by Python's `NumPy` module where both vectors and matrices are represented by the `array` type. This is also the representation we will use for images. A good `NumPy` reference is Travis Oliphant's free book [24]. The documentation at `http://numpy.scipy.org/` is also a good starting point if you are new to `NumPy`. For visualizing results we will use the `Matplotlib` module and for more advanced mathematics, we will use `SciPy`. These are the central packages you will need and will be explained and introduced in Chapter 1.

Besides these central packages there will be many other free Python packages used for specific purposes like reading JSON or XML, loading and saving data, generating graphs, graphics programming, web demos, classifiers and many more. These are usually only needed for specific applications or demos and can be skipped if you are not interested in that particular application.

It is worth mentioning IPython, an interactive Python shell that makes debugging and experimentation easier. Documentation and download available at `http://ipython.org/`.

Notation and Conventions

Code is given in a special boxed environment with color highlighting (in the electronic version) and looks like this:

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

# plot the points
plot(x,y)
```

Text is typeset according to these conventions:

Italic is used for definitions, filenames and variable names.

`Typewriter` is used for functions and Python modules.

`Small constant width` is used for console printout and results from calls and APIs.

`Hyperlink` is used for URLs (clickable in the electronic version).

Plain text is used for everything else.

Mathematical formulas are given inline like this $f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$ or centered independently

$$f(\mathbf{x}) = \sum_i w_i x_i + b \ ,$$

and are only numbered when a reference is needed.

In the mathematical sections we will use lowercase (s, r λ, θ, ...) for scalars, uppercase (A, V, H, ...) for matrices (including I for the image as an array) and lowercase bold ($\mathbf{t}$, $\mathbf{c}$, ...) for vectors. We will use $\mathbf{x} = [x, y]$ and $\mathbf{X} = [X, Y, Z]$ to mean points in 2D (images) and 3D respectively.

Acknowledgments

I'd like to express my gratitude to everyone involved in the development and production of this book. The whole O'Reilly team has been helpful. Special thanks to Andy Oram (O'Reilly) for editing, and Paul Anagnostopoulos (Windfall) for efficient production work.

Many people commented on the various drafts of this book as I shared them online. Klas Josephson and Håkan Ardö deserves lots of praise for thorough comments and feedback. Fredrik Kahl and Pau Gargallo helped with fact checks. Thank you all readers for encouraging words and for making the text and code examples better. Receiving emails from strangers sharing their thoughts on the drafts was a great motivator.

Finally, I'd like to thank my friends and family for support and understanding when I spend nights and weekends on writing. Most thanks of all to my wife Sara, my long time supporter.

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 is free and available from `http://www.pythonware.com/products/pil/`.

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

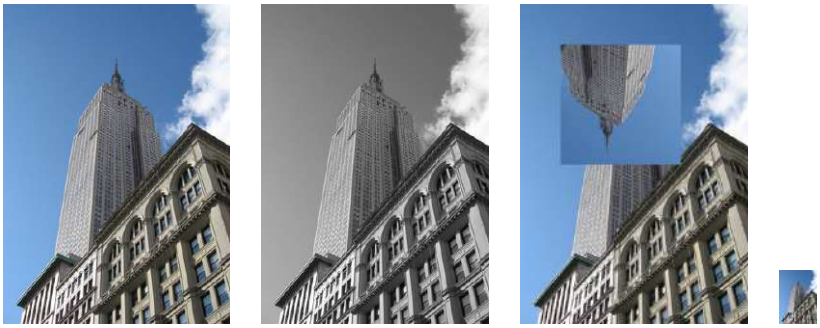


Figure 1.1: Examples of processing images with PIL.

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.

```
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 *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 f.endswith('.jpg')]
```

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 counter clockwise 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. `Matplotlib` produces high quality figures like many of the illustrations used in this book. `Matplotlib`'s `PyLab` interface is the set of functions that allow 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

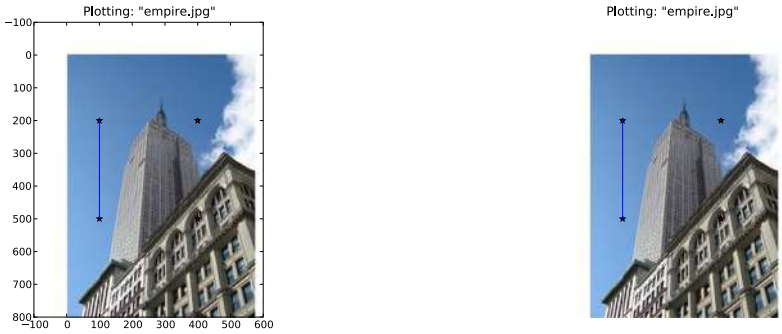


Figure 1.2: Examples of plotting with `Matplotlib`. An image with points and a line with and without showing the axes.

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 Tables 1.1, 1.2 and 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.

color	
’b’	blue
’g’	green
’r’	red
’c’	cyan
’m’	magenta
’y’	yellow
’k’	black
’w’	white

Table 1.1: Basic color formatting commands for plotting with PyLab.

line style	
’-’	solid
’- -’	dashed
’:’	dotted

Table 1.2: Basic line style formatting commands for plotting with PyLab.

marker	
’.’	point
’o’	circle
’s’	square
’*’	star
’+’	plus
’x’	x

Table 1.3: Basic plot marker formatting commands for plotting with PyLab.

```
from PIL import Image
from pylab import *

# read image to array
im = array(Image.open('images/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.

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 let's 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()
```

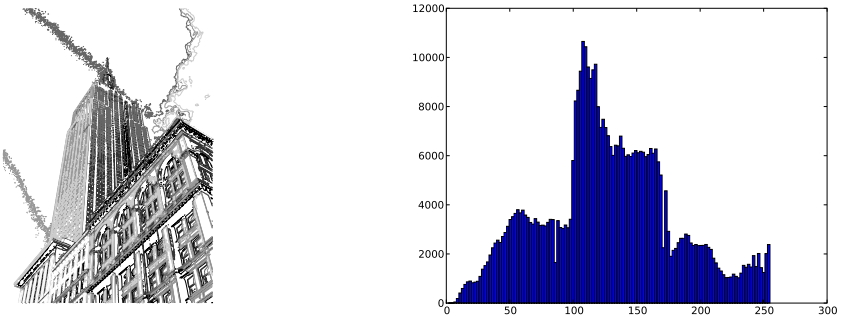


Figure 1.3: Examples of visualizing image contours and plotting image histograms with `Matplotlib`.

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[1]. The array object let's 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

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

like a list (or list of lists) but 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

```
(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 from row j
im[:,i] = 100         # set all values in column i to 100
im[:100,:50].sum()    # the sum of the values of the first 100 rows and 50 columns
im[50:100,50:100]     # rows 50-100, columns 50-100 (100th not included)
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 backwards. 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 $0 \ldots 255$ (or if you like $0 \ldots 1$) to itself (meaning that the output has the same range as the input). Here are some examples.

```
from PIL import Image
from numpy import *

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 $100 \ldots 200$ 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())
```

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

```
2 255
0 253
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, for example 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.

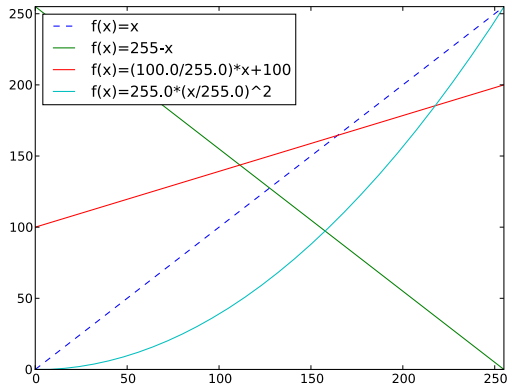


Figure 1.4: Example graylevel transforms. Three example functions together with the identity transform showed as a dashed line.

Figure 1.5: Graylevel transforms. Applying the functions in Figure 1.4. (left) Inverting the image with $f(x) = 255 - x$, (center) clamping the image with $f(x) = (100/255)x + 100$, (right) quadratic transformation with $f(x) = 255(x/255)^2$.

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*.

```
def imresize(im,sz):
  """ Resize an image array using PIL. """
  pil_im = Image.fromarray(uint8(im))

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

This function will come in handy later.

Histogram equalization

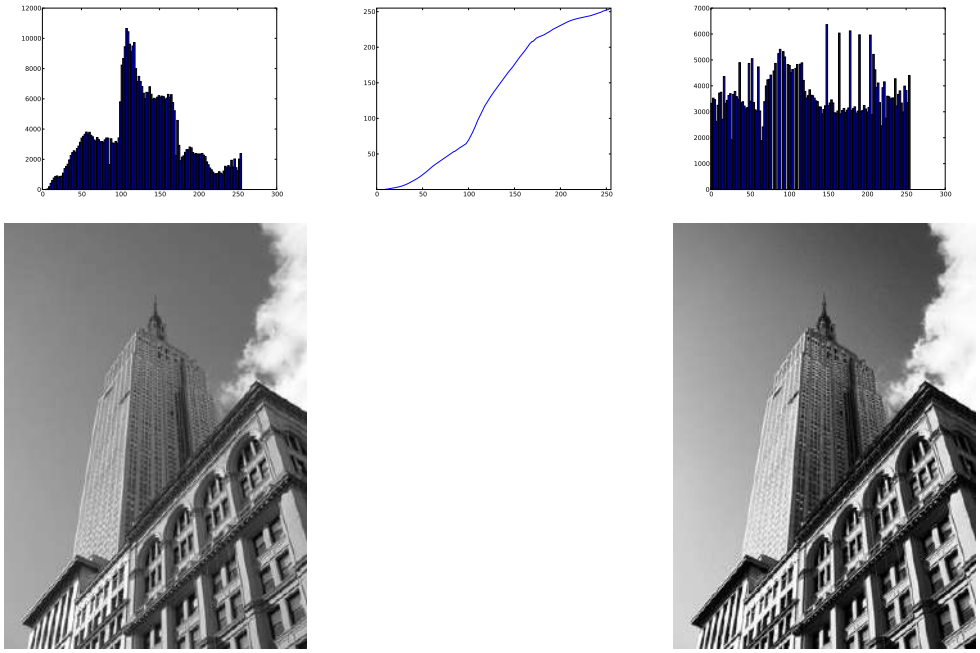
A very useful example of a graylevel transform is *histogram equalization*. This transform flattens the graylevel histogram of an image so that all intensities are as equally common as possible. This is often a good way to normalize image intensity before further processing and also a way to increase 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 $0 \ldots 1$. Try this on an image like this:



before transform after

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.

```
from PIL import Image
from numpy import *

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

Figure 1.6 and 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.

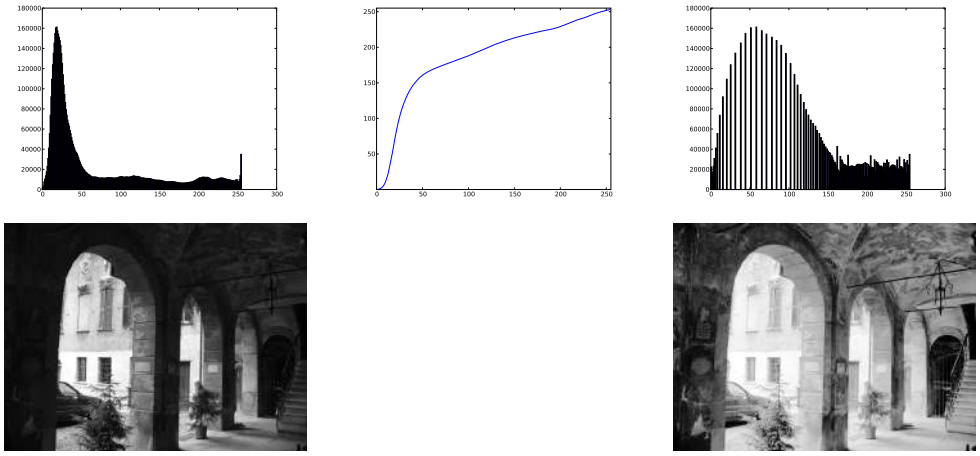


Figure 1.7: Example of histogram equalization. On the left is the original image and histogram. The middle plot is the graylevel transform function. On the right is the image and histogram after histogram equalization.

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*.

```
def compute_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.

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 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, for example using `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.

```
from PIL import Image
from numpy import *

def pca(X):
  """ Principal Component Analysis
    input: X, matrix with training data stored as flattened arrays in rows
    return: projection matrix (with important dimensions first), variance 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 ones we want
    S = sqrt(e)[::-1] # reverse since eigenvalues are in increasing order
```

```
    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 num_data

  # 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 $0 \ldots (n-1)$. 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 2359 fonts are from a collection of freely available fonts[2]. 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 size
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')
```

[2]Images courtesy of Martin Solli, `http://webstaff.itn.liu.se/~marso/`, collected and rendered from publicly available free fonts.

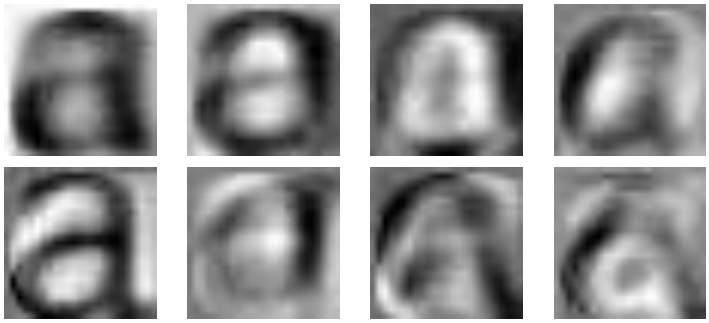


Figure 1.8: The mean image (top left) and the first seven modes, i.e. the directions with most variation.

```
# 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()
```

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:

```
# save mean and principal components
f = open('font_pca_modes.pkl', 'wb')
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#module-pickle`.

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`.

Lastly, `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:

```
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.

(a) (b) (c) (d)

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 $\sigma = 10$.

Figure 1.9 shows examples of an image blurred with increasing σ. Larger values gives less details. 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.

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, 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*

$$|\nabla I| = \sqrt{I_x^2 + I_y^2} \; ,$$

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

$$I_x = I * D_x \;\; \text{and} \;\; I_y = I * D_y \; .$$

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

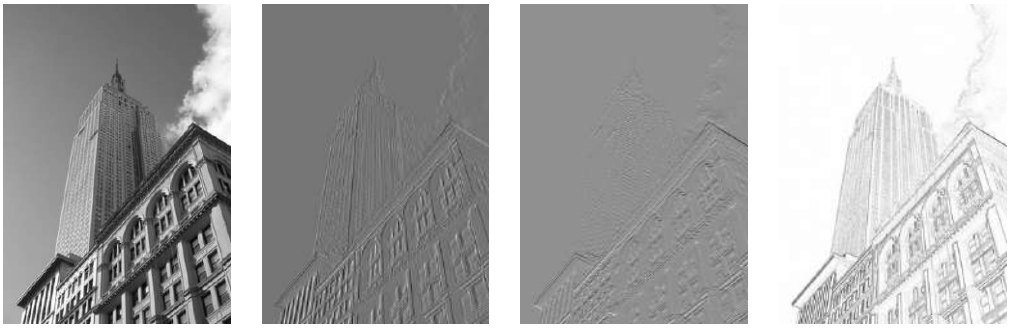
$$D_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \;\; \text{and} \;\; D_y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \; .$$

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)
```



(a) (b) (c) (d)

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.

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,

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

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.

Morphology - counting objects

Morphology (or *mathematical morphology*) is a framework and a collection of image processing methods for measuring and analyzing basic shapes. Morphology is usually applied to binary images but can be used with grayscale also. A *binary image* is an image in which each pixel takes 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.

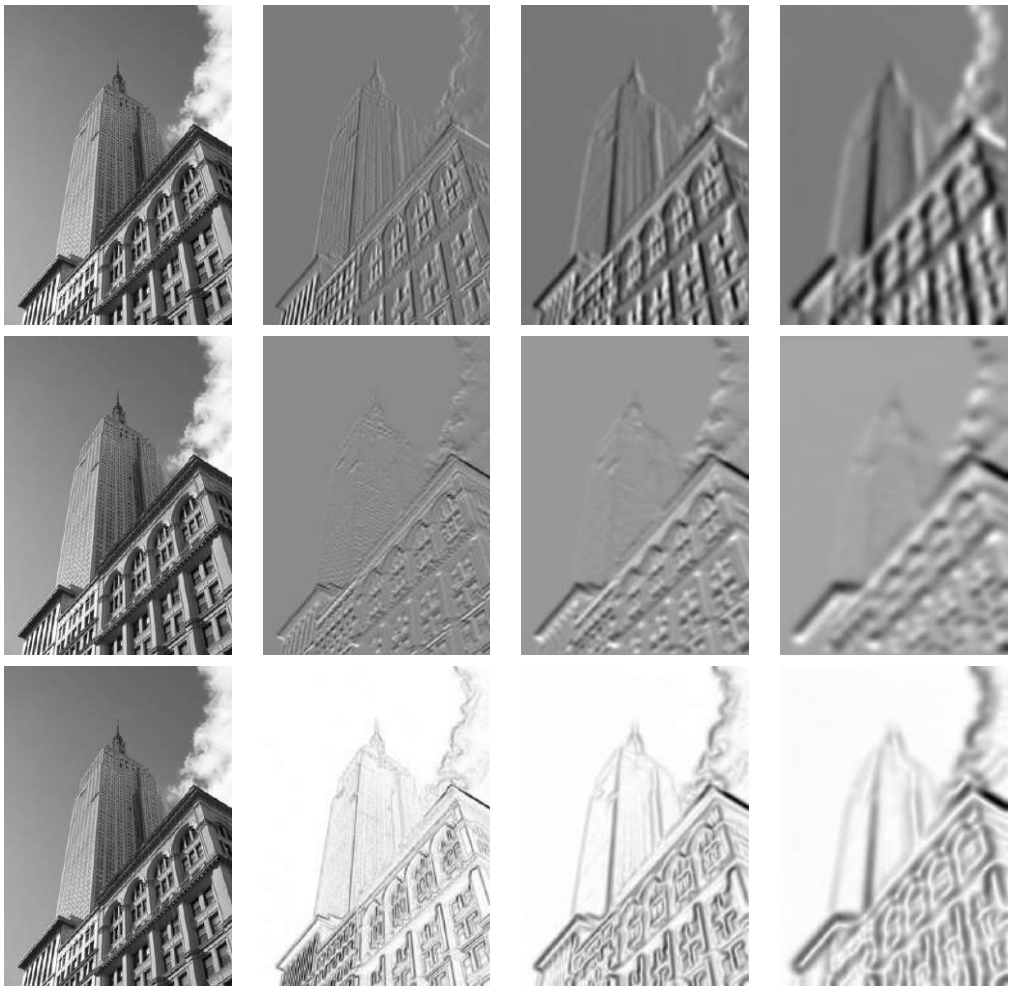
Consider the binary image in Figure 1.12a[3]. Counting the objects in that image can be done using

```
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 with 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 1.12b 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 operation called binary opening, we can remove them.

```
# morphology - opening to separate objects better
im_open = morphology.binary_opening(im,ones((9,5)),iterations=2)

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

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



(a) (b) (c) (d)

Figure 1.11: An example of computing image derivatives using Gaussian derivatives. (top) x-derivative, (middle) y-derivative and (bottom) gradient magnitude. (a) original image in grayscale, (b) Gaussian derivative filter with $\sigma = 2$, (c) with $\sigma = 5$, (d) with $\sigma = 10$.

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 changed. The image after opening and the corresponding label image are shown in Figure 1.12c-d. 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`.

Useful SciPy modules

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

Reading and writing .mat files 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.[4] Many images in this book are created just like this.

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

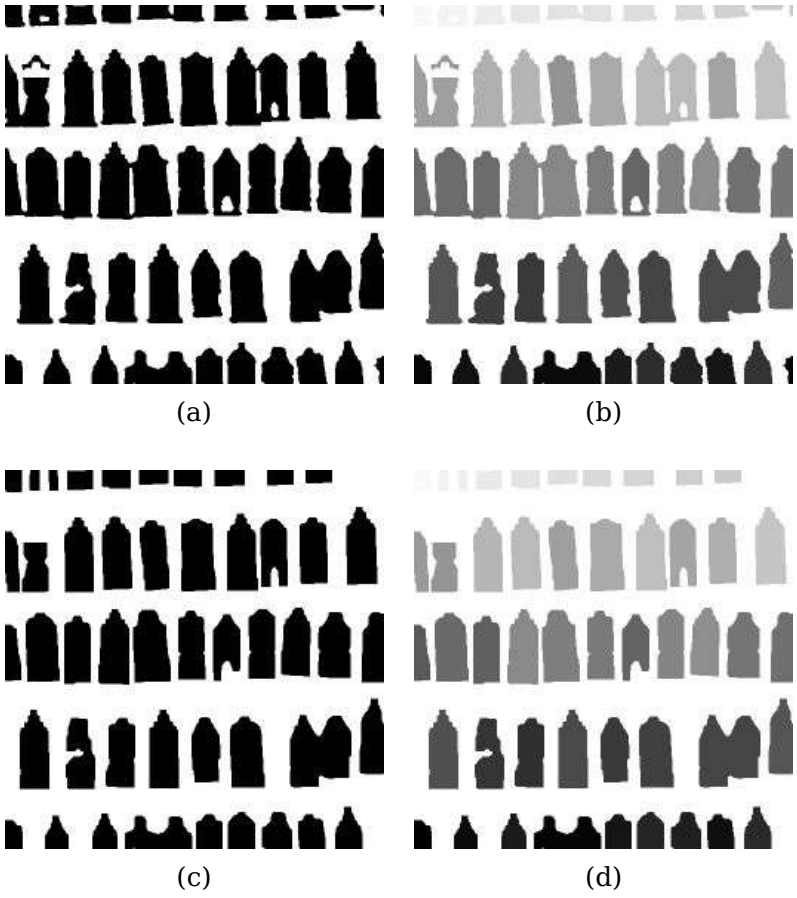


Figure 1.12: An example of morphology. Binary opening to separate objects followed by counting them. (a) original binary image, (b) label image corresponding to the original, grayvalues indicate object index, (c) binary image after opening, (d) label image corresponding to the opened image.

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

```
import scipy.misc
scipy.misc.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

$$J(I) = \int |\nabla I| d\mathbf{x} \ . \tag{1.1}$$

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

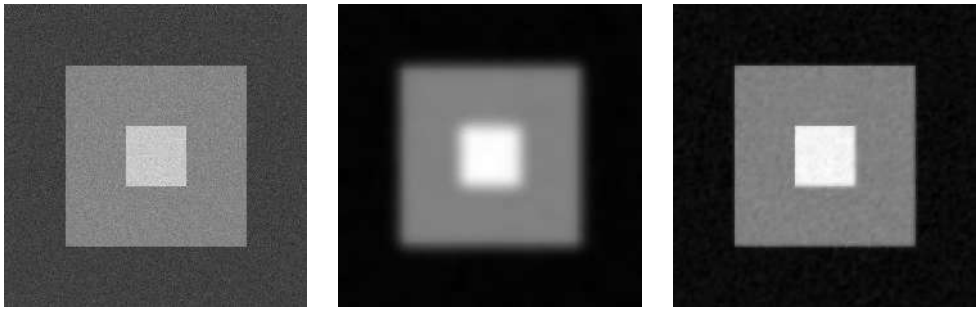
$$\min_U \ ||I - U||^2 + 2\lambda J(U),$$

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 allow "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=100):
  """ An implementation of the Rudin-Osher-Fatemi (ROF) denoising model
    using the numerical procedure presented in eq (11) A. Chambolle (2005).

    Input: noisy input image (grayscale), initial guess for U, weight of
    the TV-regularizing term, steplength, tolerance for stop criterion.

    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 gradient
    GradUy = roll(U,-1,axis=0)-U # y-component of U's gradient

    # 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-component
    RyPy = roll(Py,1,axis=0) # right y-translation of y-component

    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



(a) (b) (c)

Figure 1.13: An example of ROF de-noising of a synthetic example. (a) original noisy image, (b) image after Gaussian blurring ($\sigma = 10$). (c) image after ROF de-noising.

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
import scipy.misc
scipy.misc.imsave('synth_rof.pdf',U)
scipy.misc.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.

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

```
from PIL import Image
```

(a) (b) (c)

Figure 1.14: An example of ROF de-noising of a grayscale image. (a) original image, (b) image after Gaussian blurring ($\sigma = 5$). (c) image after ROF de-noising.

```
from pylab import *
import rof

im = array(Image.open('empire.jpg').convert('L'))
U,T = rof.denoise(im,im)

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

The result should look something like Figure 1.14c, 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.

From Chapter 2 and onwards we assume PIL, `NumPy` and `Matplotlib` to be 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.

Chapter 2

Local Image Descriptors

This chapter is about finding corresponding points and regions between images. Two different types of local descriptors are introduced with methods for matching these between images. These local features will be used in many different contexts throughout this book and are an important building block in many applications such as creating panoramas, augmented reality, and computing 3D reconstructions.

2.1 Harris corner detector

The *Harris corner detection* algorithm (or sometimes the Harris & Stephens corner detector) is one of the simplest corner indicators available. The general idea is to locate interest points where the surrounding neighborhood shows edges in more than one direction, these are then image corners.

We define a matrix $\mathbf{M}_I = \mathbf{M}_I(\mathbf{x})$, on the points $\mathbf{x}$ in the image domain, as the positive semi-definite, symmetric matrix

$$\mathbf{M}_I = \nabla I \, \nabla I^T = \begin{bmatrix} I_x \\ I_y \end{bmatrix} \begin{bmatrix} I_x & I_y \end{bmatrix} = \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \ , \tag{2.1}$$

where as before ∇I is the image gradient containing the derivatives I_x and I_y (we defined the derivatives and the gradient on page 32). Because of this construction, $\mathbf{M}_I$ has rank one with eigenvalues $\lambda_1 = |\nabla I|^2$ and $\lambda_2 = 0$. We now have one matrix for each pixel in the image.

Let W be a weight matrix (typically a Gaussian filter G_σ), the component-wise convolution

$$\overline{\mathbf{M}}_I = W * \mathbf{M}_I \ , \tag{2.2}$$

gives a local averaging of $\mathbf{M}_I$ over the neighboring pixels. The resulting matrix $\overline{\mathbf{M}}_I$ is sometimes called a *Harris matrix*. The width of W determines a region of interest

around $\mathbf{x}$. The idea of averaging the matrix $\mathbf{M}_I$ over a region like this is that the eigenvalues will change depending on the local image properties. If the gradients vary in the region, the second eigenvalue of $\overline{\mathbf{M}}_I$ will no longer be zero. If the gradients are the same, the eigenvalues will be the same as for $\mathbf{M}_I$.

Depending on the values of ∇I in the region, there are three cases for the eigenvalues of the Harris matrix, $\overline{\mathbf{M}}_I$:

- If λ_1 and λ_2 are both large positive values, then there is a corner at $\mathbf{x}$.
- If λ_1 is large and $\lambda_2 \approx 0$, then there is an edge and the averaging of $\mathbf{M}_I$ over the region doesn't change the eigenvalues that much.
- If $\lambda_1 \approx \lambda_2 \approx 0$ then there is nothing.

To distinguish the important case from the others without actually having to compute the eigenvalues, Harris and Stephens [12] introduced an indicator function

$$\det(\overline{\mathbf{M}}_I) - \kappa \operatorname{trace}(\overline{\mathbf{M}}_I)^2 \ .$$

To get rid of the weighting constant κ, it is often easier to use the quotient

$$\frac{\det(\overline{\mathbf{M}}_I)}{\operatorname{trace}(\overline{\mathbf{M}}_I)^2}$$

as an indicator.

Let's see what this looks like in code. For this we need the `scipy.ndimage.filters` module for computing derivatives using Gaussian derivative filters as described on page 33. The reason is again that we would like to suppress noise sensitivity in the corner detection process.

First add the corner response function to a file *harris.py* which will make use of the Gaussian derivatives. Again the parameter σ defines the scale of the Gaussian filters used. You can also modify this function to take different scales in the x and y directions as well as a different scale for the averaging to compute the Harris matrix.

```
from scipy.ndimage import filters

def compute_harris_response(im,sigma=3):
  """ Compute the Harris corner detector response function
    for each pixel in a graylevel image. """

  # derivatives
  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)

  # compute components of the Harris matrix
  Wxx = filters.gaussian_filter(imx*imx,sigma)
  Wxy = filters.gaussian_filter(imx*imy,sigma)
  Wyy = filters.gaussian_filter(imy*imy,sigma)

  # determinant and trace
  Wdet = Wxx*Wyy - Wxy**2
  Wtr = Wxx + Wyy

  return Wdet / Wtr
```

This gives an image with each pixel containing the value of the Harris response function. Now it is just a matter of picking out the information needed from this image. Taking all points with values above a threshold with the additional constraint that corners must be separated with a minimum distance is an approach that often gives good results. To do this, take all candidate pixels, sort them in descending order of corner response values and mark off regions too close to positions already marked as corners. Add the following function to *harris.py*.

```
def get_harris_points(harrisim,min_dist=10,threshold=0.1):
  """ Return corners from a Harris response image
    min_dist is the minimum number of pixels separating
    corners and image boundary. """

  # find top corner candidates above a threshold
  corner_threshold = harrisim.max() * threshold
  harrisim_t = (harrisim > corner_threshold) * 1

  # get coordinates of candidates
  coords = array(harrisim_t.nonzero()).T

  # ...and their values
  candidate_values = [harrisim[c[0],c[1]] for c in coords]

  # sort candidates
  index = argsort(candidate_values)

  # store allowed point locations in array
  allowed_locations = zeros(harrisim.shape)
  allowed_locations[min_dist:-min_dist,min_dist:-min_dist] = 1

  # select the best points taking min_distance into account
  filtered_coords = []
  for i in index:
    if allowed_locations[coords[i,0],coords[i,1]] == 1:
```

```
      filtered_coords.append(coords[i])
      allowed_locations[(coords[i,0]-min_dist):(coords[i,0]+min_dist),
            (coords[i,1]-min_dist):(coords[i,1]+min_dist)] = 0

  return filtered_coords
```

Now you have all you need to detect corner points in images. To show the corner points in the image you can add a plotting function to *harris.py* using `Matplotlib` as follows.

```
def plot_harris_points(image,filtered_coords):
  """ Plots corners found in image. """

  figure()
  gray()
  imshow(image)
  plot([p[1] for p in filtered_coords],[p[0] for p in filtered_coords],'*')
  axis('off')
  show()
```

Try running the following commands:

```
im = array(Image.open('empire.jpg').convert('L'))
harrisim = harris.compute_harris_response(im)
filtered_coords = harris.get_harris_points(harrisim,6)
harris.plot_harris_points(im, filtered_coords)
```

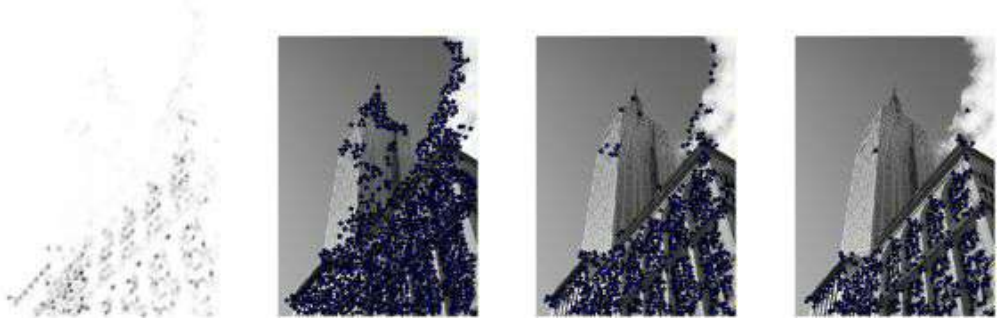
The image is opened and converted to grayscale. Then the response function is computed and points selected based on the response values. Finally, the points are plotted overlaid on the original image. This should give you a plot like the images in Figure 2.1.

For an overview of different approaches to corner detection, including improvements on the Harris detector and further developments, see for example `http://en.wikipedia.org/wiki/Corner_detection`.

Finding corresponding points between images

The Harris corner detector gives interest points in images but does not contain an inherent way of comparing these interest points across images to find matching corners. What we need is to add a descriptor to each point and a way to compare such descriptors.

An *interest point descriptor* is a vector assigned to an interest point that describes the image appearance around the point. The better the descriptor, the better your correspondences will be. With *point correspondence* or *corresponding points* we mean points in different images that refer to the same object or scene point.



(a) (b) (c) (d)

Figure 2.1: An example of corner detection with the Harris corner detector. (a) the Harris response function, (b), (c) and (d) corners detected with threshold 0.01, 0.05, and 0.1 respectively.

Harris corner points are usually combined with a descriptor consisting of the graylevel values in a neighboring image patch together with normalized cross correlation for comparison. An *image patch* is almost always a rectangular portion of the image centered around the point in question.

In general, *correlation* between two (equally sized) image patches $I_1(\mathbf{x})$ and $I_2(\mathbf{x})$ is defined as

$$c(I_1, I_2) = \sum_{\mathbf{x}} f(I_1(\mathbf{x}), I_2(\mathbf{x})) \ ,$$

where the function f varies depending on the correlation method. The sum is taken over all positions $\mathbf{x}$ in the image patches. For *cross correlation* $f(I_1, I_2) = I_1 \ I_2$, and then $c(I_1, I_2) = I_1 \cdot I_2$ with $\cdot$ denoting the scalar product (of the row- or column-stacked patches). The larger the value of $c(I_1, I_2)$, the more similar the patches I_1 and I_2 are[1].

Normalized cross correlation is a variant of cross correlation defined as

$$ncc(I_1, I_2) = \frac{1}{n-1} \sum_{\mathbf{x}} \frac{(I_1(\mathbf{x}) - \mu_1)}{\sigma_1} \cdot \frac{(I_2(\mathbf{x}) - \mu_2)}{\sigma_2} \ , \tag{2.3}$$

where n is the number of pixels in a patch, μ_1 and μ_2 are the mean intensities, and σ_1 and σ_2 are the standard deviations in each patch respectively. By subtracting the mean and scaling with the standard deviation, the method becomes robust to changes in image brightness.

To extract image patches and compare them using normalized cross correlation, you need two more functions in *harris.py*. Add these:

[1]Another popular function is $f(I_1, I_2) = (I_1 - I_2)^2$ which gives *sum of squared differences* (*SSD*).

```
def get_descriptors(image,filtered_coords,wid=5):
  """ For each point return pixel values around the point
    using a neighbourhood of width 2*wid+1. (Assume points are
    extracted with min_distance > wid). """

  desc = []
  for coords in filtered_coords:
    patch = image[coords[0]-wid:coords[0]+wid+1,
              coords[1]-wid:coords[1]+wid+1].flatten()
    desc.append(patch)

  return desc

def match(desc1,desc2,threshold=0.5):
  """ For each corner point descriptor in the first image,
    select its match to second image using
    normalized cross correlation. """

  n = len(desc1[0])

  # pair-wise distances
  d = -ones((len(desc1),len(desc2)))
  for i in range(len(desc1)):
    for j in range(len(desc2)):
      d1 = (desc1[i] - mean(desc1[i])) / std(desc1[i])
      d2 = (desc2[j] - mean(desc2[j])) / std(desc2[j])
      ncc_value = sum(d1 * d2) / (n-1)
      if ncc_value > threshold:
        d[i,j] = ncc_value

  ndx = argsort(-d)
  matchscores = ndx[:,0]

  return matchscores
```

The first function takes a square grayscale patch of odd side length centered around the point, flattens it and adds to a list of descriptors. The second function matches each descriptor to its best candidate in the other image using normalized cross correlation. Note that the distances are negated before sorting since a high value means better match. To further stabilize the matches, we can match from the second image to the first and filter out the matches that are not the best both ways. The following function does just that.

```
def match_twosided(desc1,desc2,threshold=0.5):
  """ Two-sided symmetric version of match(). """
```

```
  matches_12 = match(desc1,desc2,threshold)
  matches_21 = match(desc2,desc1,threshold)

  ndx_12 = where(matches_12 >= 0)[0]

  # remove matches that are not symmetric
  for n in ndx_12:
    if matches_21[matches_12[n]] != n:
      matches_12[n] = -1

  return matches_12
```

The matches can be visualized by showing the images side-by-side and connecting matched points with lines using the following code. Add these two functions to *harris.py*:

```
def appendimages(im1,im2):
  """ Return a new image that appends the two images side-by-side. """

  # select the image with the fewest rows and fill in enough empty rows
  rows1 = im1.shape[0]
  rows2 = im2.shape[0]

  if rows1 < rows2:
    im1 = concatenate((im1,zeros((rows2-rows1,im1.shape[1]))),axis=0)
  elif rows1 > rows2:
    im2 = concatenate((im2,zeros((rows1-rows2,im2.shape[1]))),axis=0)
  # if none of these cases they are equal, no filling needed.

  return concatenate((im1,im2), axis=1)


def plot_matches(im1,im2,locs1,locs2,matchscores,show_below=True):
  """ Show a figure with lines joining the accepted matches
    input: im1,im2 (images as arrays), locs1,locs2 (feature locations),
    matchscores (as output from 'match()'),
    show_below (if images should be shown below matches). """

  im3 = appendimages(im1,im2)
  if show_below:
    im3 = vstack((im3,im3))

  imshow(im3)

  cols1 = im1.shape[1]
  for i,m in enumerate(matchscores):
    if m>0:
      plot([locs1[i][1],locs2[m][1]+cols1],[locs1[i][0],locs2[m][0]],'c')
```

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

Figure 2.2 shows an example of finding such corresponding points using normalized cross correlation (in this case with 11×11 pixels in a patch) using the commands:

```
wid = 5
harrisim = harris.compute_harris_response(im1,5)
filtered_coords1 = harris.get_harris_points(harrisim,wid+1)
d1 = harris.get_descriptors(im1,filtered_coords1,wid)

harrisim = harris.compute_harris_response(im2,5)
filtered_coords2 = harris.get_harris_points(harrisim,wid+1)
d2 = harris.get_descriptors(im2,filtered_coords2,wid)

print 'starting matching'
matches = harris.match_twosided(d1,d2)

figure()
gray()
harris.plot_matches(im1,im2,filtered_coords1,filtered_coords2,matches)
show()
```

If you only want to plot a subset of the matches to make the visualization clearer, substitute *matches* with for example *matches[:100]* or a random set of indices.

As you can see in Figure 2.2, there are quite a lot of incorrect matches. This is because cross correlation on image patches is not as descriptive as more modern approaches. As a consequence, it is important to use robust methods for handling these correspondences in an application. Another problem is that these descriptors are not invariant to scale or rotation and the choice of patch sizes affect the results.

In recent years there has been a lot of development in improving feature point detection and description. Let's take a look at one of the best algorithms in the next section.

2.2 SIFT - Scale-Invariant Feature Transform

One of the most successful local image descriptors in the last decade is the *Scale-Invariant Feature Transform* (*SIFT*), introduced by David Lowe in [17]. SIFT was later refined and described in detail in the paper [18] and has stood the test of time. SIFT includes both an interest point detector and a descriptor. The descriptor is very robust and is largely the reason behind the success and popularity of SIFT. Since its introduction many alternatives have been proposed with essentially the same type of descriptor. The descriptor is nowadays often combined with many different interest point detectors (and region detectors for that matter) and sometimes even applied densely

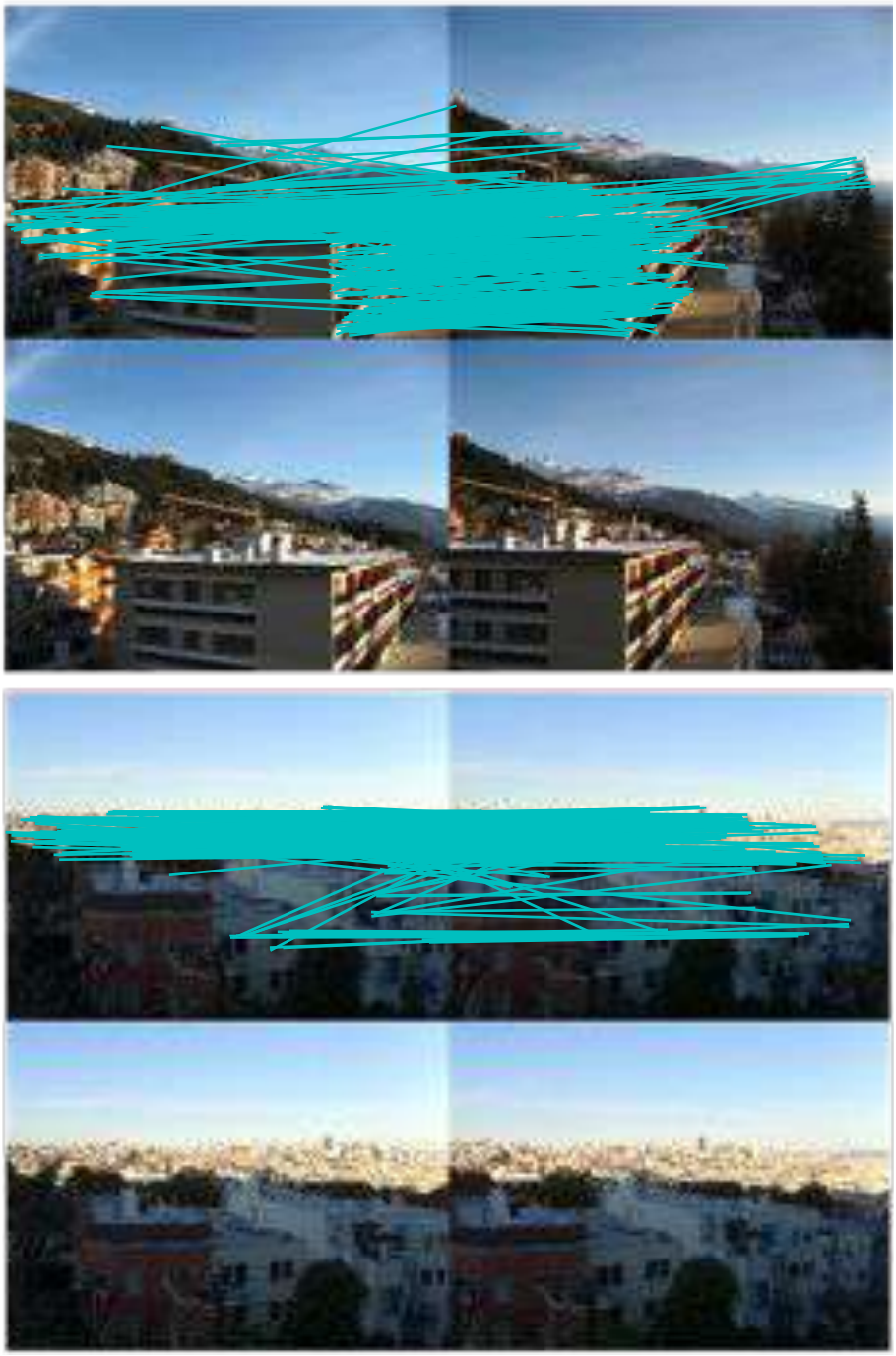


Figure 2.2: Example of matches resulting from applying normalized cross correlation to patches around Harris corner points.

across the whole image. SIFT features are invariant to scale, rotation and intensity and can be matched reliably across 3D viewpoint and noise. A brief overview is available online at `http://en.wikipedia.org/wiki/Scale-invariant_feature_transform`.

Interest points

SIFT interest point locations are found using *difference-of-Gaussian* functions

$$D(\mathbf{x},\sigma) = [G_{k\sigma}(\mathbf{x}) - G_{\sigma}(\mathbf{x})] * I(\mathbf{x}) = [G_{k\sigma} - G_{\sigma}] * I = I_{k\sigma} - I_{\sigma} \ ,$$

where G_σ is the Gaussian 2D kernel described on page 31, I_σ the G_σ-blurred grayscale image and k a constant factor determining the separation in scale. Interest points are the maxima and minima of $D(\mathbf{x},\sigma)$ across both image location and scale. These candidate locations are filtered to remove unstable points. Points are dismissed based on a number of criteria like low contrast and points on edges. The details are in the paper.

Descriptor

The interest point (keypoint) locator above gives position and scale. To achieve invariance to rotation, a reference direction is chosen based on the direction and magnitude of the image gradient around each point. The dominant direction is used as reference and determined using an orientation histogram (weighted with the magnitude).

The next step is to compute a descriptor based on the position, scale and rotation. To obtain robustness against image intensity, the SIFT descriptor uses image gradients (compare that to normalized cross correlation above that uses the image intensities). The descriptor takes a grid of subregions around the point and for each subregion computes an image gradient orientation histogram. The histograms are concatenated to form a descriptor vector. The standard setting uses 4×4 subregions with 8 bin orientation histograms resulting in a 128 bin histogram ($4 * 4 * 8 = 128$). Figure 2.3 illustrates the construction of the descriptor. The interested reader should look at [18] for the details or `http://en.wikipedia.org/wiki/Scale-invariant_feature_transform` for an overview.

Detecting interest points

To compute SIFT features for images we will use the binaries available with the open source package VLFeat [36]. A full Python implementation of all the steps in the algorithm would not be very efficient and really is outside the scope of this book. VLFeat is available at `http://www.vlfeat.org/`, with binaries for all major platforms.

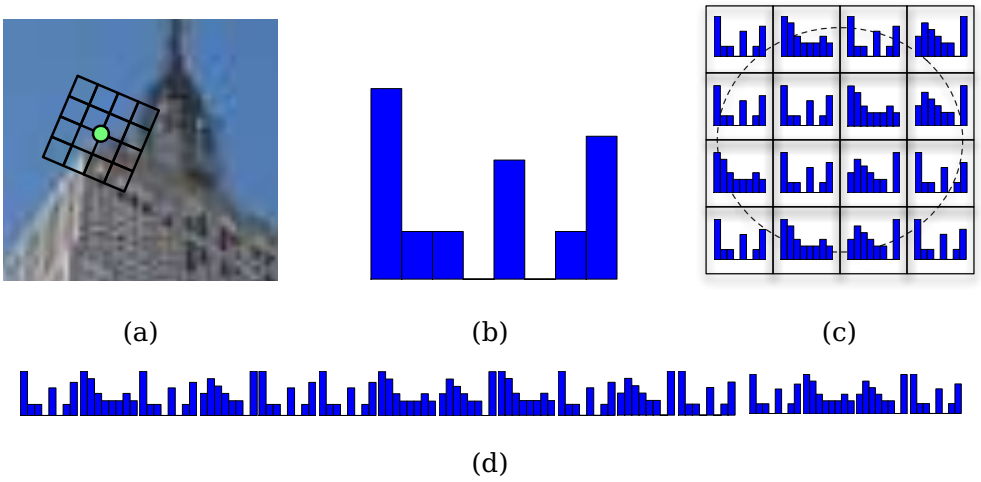


Figure 2.3: An illustration of the construction of the feature vector for the SIFT descriptor. (a) a frame around an interest point, oriented according to the dominant gradient direction. (b) an 8 bin histogram over the direction of the gradient in a part of the grid. (c) histograms are extracted in each grid location. (d) the histograms are concatenated to form one long feature vector.

The library is written in C but has a command line interface that we can use. There is also a Matlab interface and a Python wrapper `http://github.com/mmmikael/vlfeat/` if you prefer that to the binaries used here. The Python wrapper can be a little tricky to install on some platforms due to its dependencies so we will focus on the binaries instead. There is also an alternative SIFT implementation available at Lowe's website `http://www.cs.ubc.ca/~lowe/keypoints/` (Windows and Linux only).

Create a file *sift.py* and add the following function that calls the executable.

```
def process_image(imagename,resultname,params="--edge-thresh 10 --peak-thresh 5"):
  """ Process an image and save the results in a file. """

  if imagename[-3:] != 'pgm':
    # create a pgm file
    im = Image.open(imagename).convert('L')
    im.save('tmp.pgm')
    imagename = 'tmp.pgm'

  cmmd = str("sift "+imagename+" --output="+resultname+
        " "+params)
  os.system(cmmd)
  print 'processed', imagename, 'to', resultname
```

The binaries need the image in grayscale .pgm format, so if another image format is used we first convert to a temporary .pgm file. The result is stored in a text file in an easy to read format. The files look something like this

```
318.861 7.48227 1.12001 1.68523 0 0 0 1 0 0 0 0 0 11 16 0 ...
318.861 7.48227 1.12001 2.99965 11 2 0 0 1 0 0 0 173 67 0 0 ...
54.2821 14.8586 0.895827 4.29821 60 46 0 0 0 0 0 0 99 42 0 0 ...
155.714 23.0575 1.10741 1.54095 6 0 0 0 150 11 0 0 150 18 2 1 ...
42.9729 24.2012 0.969313 4.68892 90 29 0 0 0 1 2 10 79 45 5 11 ...
229.037 23.7603 0.921754 1.48754 3 0 0 0 141 31 0 0 141 45 0 0 ...
232.362 24.0091 1.0578 1.65089 11 1 0 16 134 0 0 0 106 21 16 33 ...
201.256 25.5857 1.04879 2.01664 10 4 1 8 14 2 1 9 88 13 0 0 ...
...
...
```

where each row contains the coordinates, scale and rotation angle for each interest point as the first four values, followed by the 128 values of the corresponding descriptor. The descriptor is represented with the raw integer values and is not normalized. This is something you will want to do when comparing descriptors. More on that later.

The example above shows the first part of the first eight features found in an image. Note that the two first rows have the same coordinates but different rotation. This can happen if several strong directions are found at the same interest point.

Here's how to read the features to `NumPy` arrays from an output file like the one above. Add this function to *sift.py*,

```
def read_features_from_file(filename):
  """ Read feature properties and return in matrix form. """

  f = loadtxt(filename)
  return f[:,:4],f[:,4:] # feature locations, descriptors
```

Here we used the `NumPy` function `loadtxt()` to do all the work for us.

If you modify the descriptors in your Python session writing the result back to feature files can be useful. The function below does this for you using `NumPy`'s `savetxt()`.

```
def write_features_to_file(filename,locs,desc):
  """ Save feature location and descriptor to file. """
  savetxt(filename,hstack((locs,desc)))
```

This uses the function `hstack()` that horizontally stacks the two arrays by concatenating the rows so that the descriptor part comes after the locations on each row.

Having read the features, visualizing them by plotting their locations in the image is a simple task. Just add `plot_features()` as below to your file *sift.py*.

```
def plot_features(im,locs,circle=False):
  """ Show image with features. input: im (image as array),
    locs (row, col, scale, orientation of each feature). """
```

```
  def draw_circle(c,r):
    t = arange(0,1.01,.01)*2*pi
    x = r*cos(t) + c[0]
    y = r*sin(t) + c[1]
    plot(x,y,'b',linewidth=2)

  imshow(im)
  if circle:
    for p in locs:
      draw_circle(p[:2],p[2])
  else:
    plot(locs[:,0],locs[:,1],'ob')
  axis('off')
```

This will plot the location of the SIFT points as blue dots overlaid on the image. If the optional parameter *circle* is set to "True", circles with radius equal to the scale of the feature will be drawn instead using the helper function `draw_circle()`.

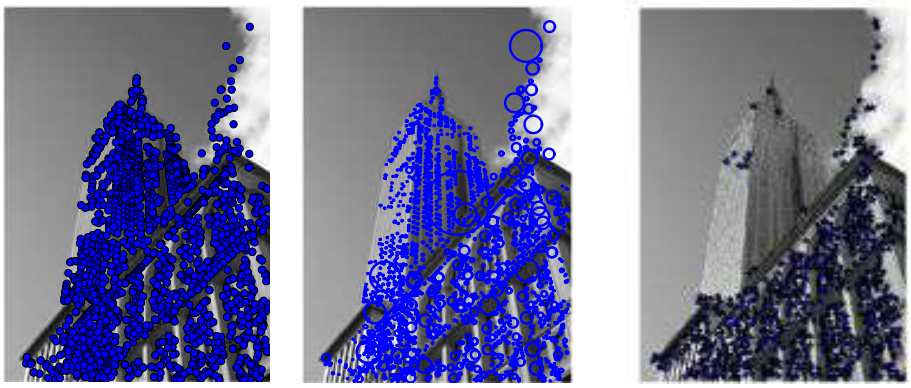
The following commands

```
import sift

imname = 'empire.jpg'
im1 = array(Image.open(imname).convert('L'))
sift.process_image(imname,'empire.sift')
l1,d1 = sift.read_features_from_file('empire.sift')

figure()
gray()
sift.plot_features(im1,l1,circle=True)
show()
```

will create a plot like the one in Figure 2.4b with the SIFT feature locations shown. To see the difference compared to Harris corners, the Harris corners for the same image is shown to the right (Figure 2.4c). As you can see the two algorithms select different locations.

Matching descriptors

A robust criteria (also introduced by Lowe) for matching a feature in one image to a feature in another image is to use the ratio of the distance to the two closest matching features. This ensures that only features that are distinct enough compared to the other features in the image are used. As a consequence, the number of false matches is lowered. Here's what this matching function looks like in code. Add `match()` to *sift.py*.



(a) (b) (c)

Figure 2.4: An example of extracting SIFT features for an image. (a) SIFT features (b) SIFT features shown with circle indicating the scale of the feature (c) Harris points for the same image for comparison.

```
def match(desc1,desc2):
  """ For each descriptor in the first image,
    select its match in the second image.
    input: desc1 (descriptors for the first image),
    desc2 (same for second image). """

  desc1 = array([d/linalg.norm(d) for d in desc1])
  desc2 = array([d/linalg.norm(d) for d in desc2])

  dist_ratio = 0.6
  desc1_size = desc1.shape

  matchscores = zeros((desc1_size[0],1),'int')
  desc2t = desc2.T # precompute matrix transpose
  for i in range(desc1_size[0]):
    dotprods = dot(desc1[i,:],desc2t) # vector of dot products
    dotprods = 0.9999*dotprods
    # inverse cosine and sort, return index for features in second image
    indx = argsort(arccos(dotprods))

    # check if nearest neighbor has angle less than dist_ratio times 2nd
    if arccos(dotprods)[indx[0]] < dist_ratio * arccos(dotprods)[indx[1]]:
      matchscores[i] = int(indx[0])

  return matchscores
```

This function uses the angle between descriptor vectors as distance measure. This makes sense only after we have normalized the vectors to unit length[2]. Since the matching is one-sided, meaning that we are matching each feature to all features in the other image, we can pre-compute the transpose of the matrix containing the descriptor vectors containing the points in the second image so that we don't have to repeat this exact same operation for each feature.

To further increase the robustness of the matches, we can reverse the procedure and match the other way (from the features in the second image to features in the first) and only keep the correspondences that satisfy the matching criteria both ways (same as what we did for the Harris points). The function `match_twosided()` does just this:

```
def match_twosided(desc1,desc2):
  """ Two-sided symmetric version of match(). """

  matches_12 = match(desc1,desc2)
  matches_21 = match(desc2,desc1)

  ndx_12 = matches_12.nonzero()[0]

  # remove matches that are not symmetric
  for n in ndx_12:
    if matches_21[int(matches_12[n])] != n:
      matches_12[n] = 0

  return matches_12
```

To plot the matches we can use the same functions used in *harris.py*. Just copy the functions `appendimages()` and `plot_matches()` and add them to *sift.py* for convenience (you could also import *harris.py* and use them from there if you like).

Figures 2.5 and 2.6 shows some examples of SIFT feature points detected in image pairs together with pair-wise matches returned from the function `match_twosided()`.

Figure 2.7 shows another example of matching features found in two images using `match()` and `match_twosided()`. As you can see, using the symmetric (two-sided) matching condition removes the incorrect matches and keeps the good ones (some correct matches are also removed).

With detection and matching of feature points we have everything needed to apply these local descriptors to a number of applications. The coming two chapters will add geometric constraints on correspondences in order to robustly filter out the incorrect

[2] In the case of unit length vectors the scalar product (without the `arccos()`) is equivalent to the standard Euclidean distance.

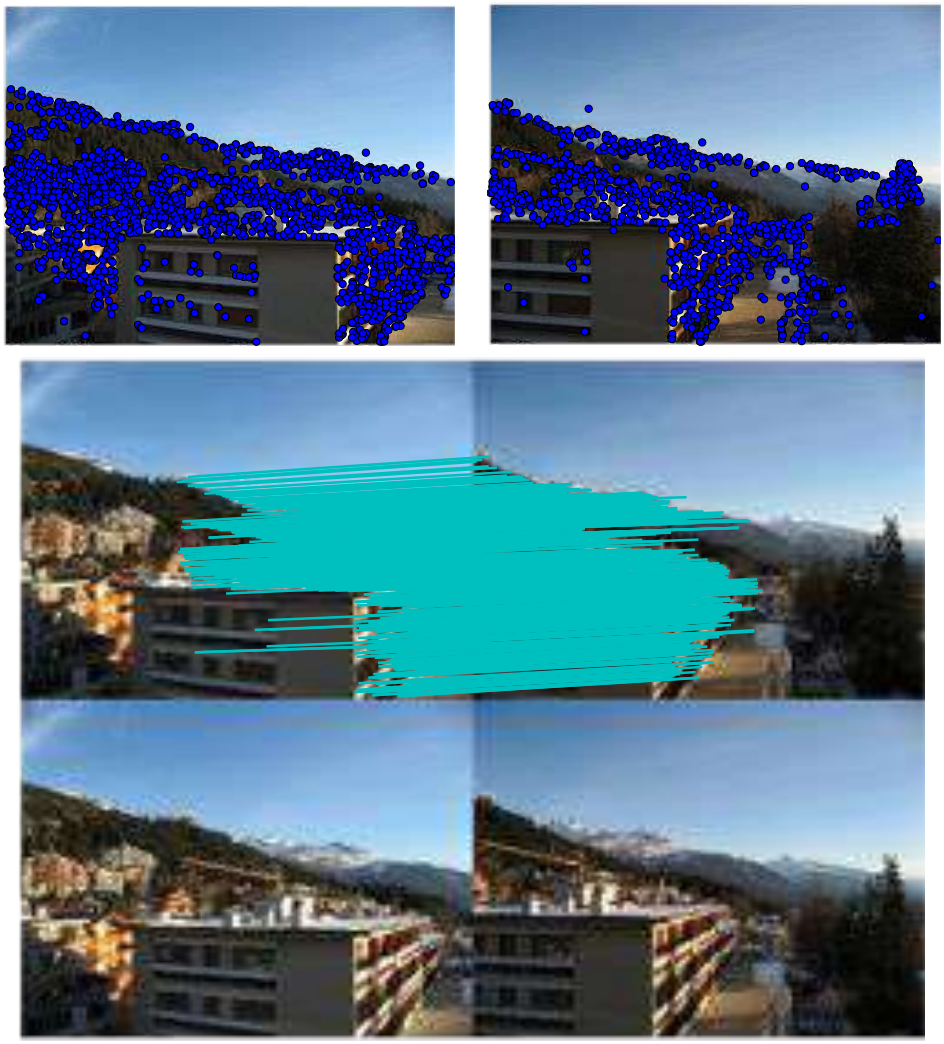


Figure 2.5: An example of detecting and matching SIFT features between two images.

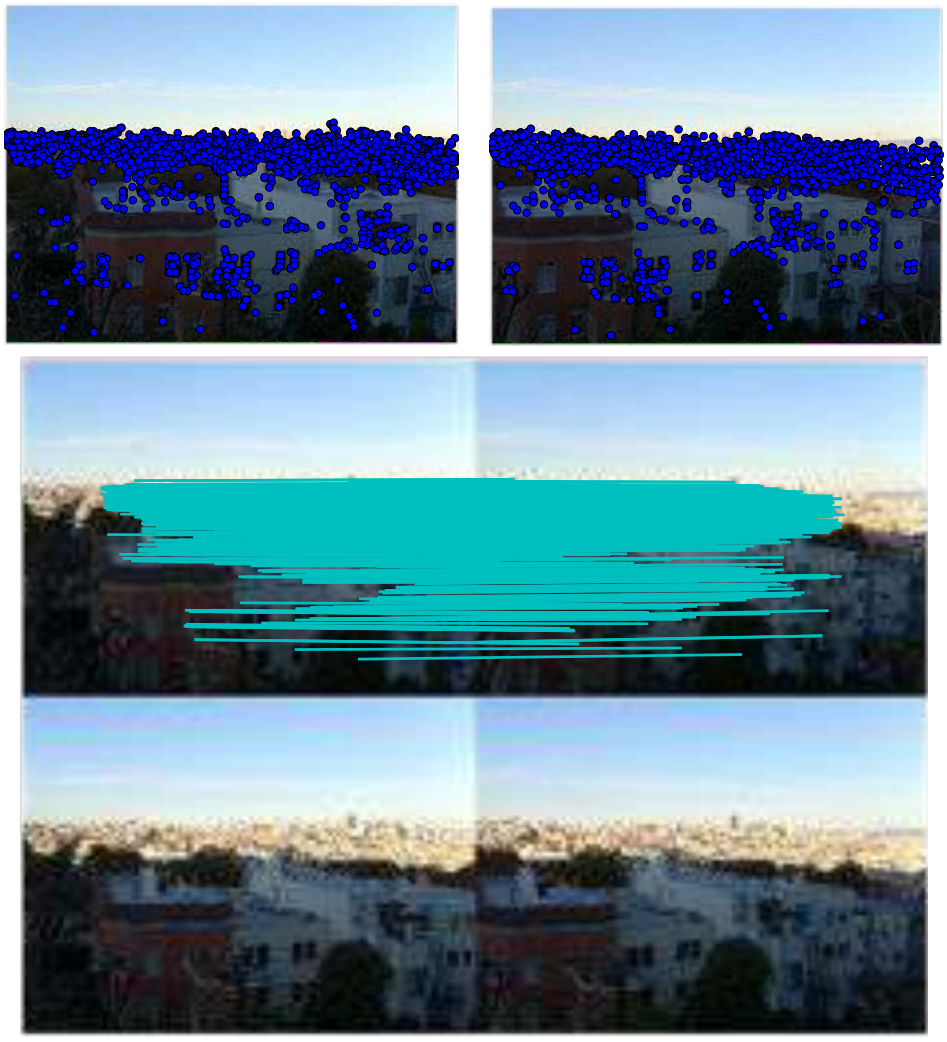
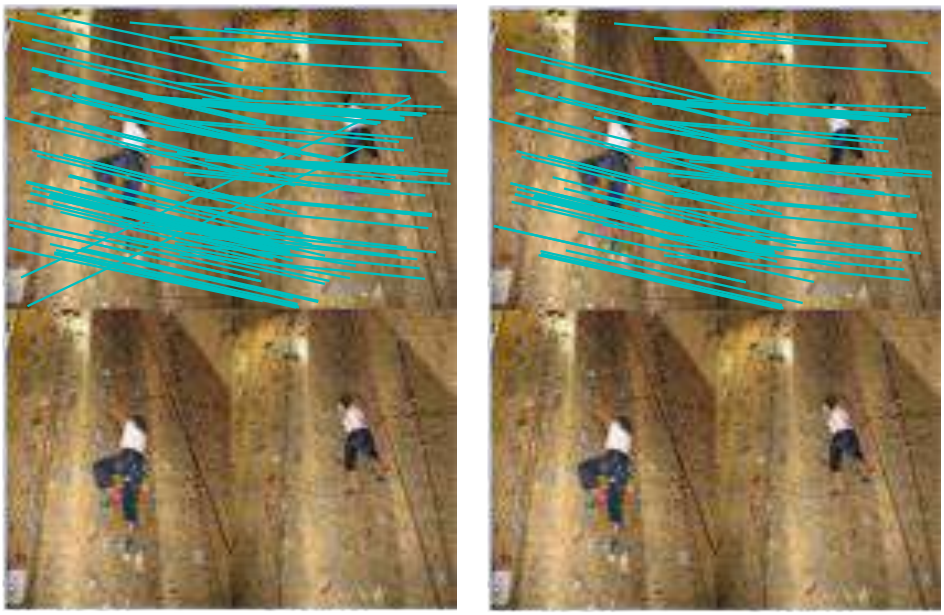


Figure 2.6: An example of detecting and matching SIFT features between two images.



(a) (b)

Figure 2.7: An example of matching SIFT features between two images. (a) matches from features in the left image without using the two-sided match function (b) the remaining matches after using the two-sided version.

ones and apply local descriptors to examples such as automatic panorama creation, camera pose estimation, and 3D structure computation.

2.3 Matching Geotagged Images

Let's end this chapter by looking at an example application of using local descriptors for matching images with geotags. Geotagged images are images with GPS coordinates either added manually by the photographer or automatically by the camera.

Downloading geotagged images from Panoramio

One source of geotagged images is the photo-sharing service Panoramio (`http://www.panoramio.com/`), owned by Google. Like many web services, Panoramio has an API to access content programmatically. Their API is simple and straight-forward and is described at `http://www.panoramio.com/api/`. By making a HTTP GET call to a url like this

```
http://www.panoramio.com/map/get_panoramas.php?order=popularity&set=public&
from=0&to=20&minx=-180&miny=-90&maxx=180&maxy=90&size=medium
```

where *minx*, *miny*, *maxx*, *maxy* define the geographic area to select photos from (minimum longitude, latitude, maximum longitude and latitude, respectively). You will get the response in easy-to-parse JSON format. JSON is a common format for data transfer between web services and is more lightweight than XML and other alternatives. You can read more about JSON at `http://en.wikipedia.org/wiki/JSON`.

An interesting location with two distinct views is the White house in Washington D.C. which is usually photographed from Pennsylvania Avenue from the south side or from the north. The coordinates (latitude, longitude) are:

```
lt=38.897661
ln=-77.036564
```

To convert to the format needed for the API call, subtract and add a number from these coordinates to get all images within a square centered around the White house. The call

```
http://www.panoramio.com/map/get_panoramas.php?order=popularity&set=public&
from=0&to=20&minx=-77.037564&miny=38.896662&maxx=-77.035564&maxy=38.898662&size=medium
```

returns the first 20 images within the coordinate bounds (± 0.001), ordered according to popularity. The response looks something like this

```
{ "count": 349,
"photos": [{"photo_id": 7715073, "photo_title": "White House", "photo_url":
"http://www.panoramio.com/photo/7715073", "photo_file_url":
"http://mw2.google.com/mw-panoramio/photos/medium/7715073.jpg", "longitude":
-77.036583, "latitude": 38.897488, "width": 500, "height": 375, "upload_date":
"10 February 2008", "owner_id": 1213603, "owner_name": "***", "owner_url":
"http://www.panoramio.com/user/1213603"}
,
{"photo_id": 1303971, "photo_title": "ÃĄÃt'Îl'ÃĆÃĘÂt'ÃĹÃšâL'ěÃĆÃĺâĹđ", "photo_url":
"http://www.panoramio.com/photo/1303971", "photo_file_url":
"http://mw2.google.com/mw-panoramio/photos/medium/1303971.jpg", "longitude":
-77.036353, "latitude": 38.897471, "width": 500, "height": 336, "upload_date":
"13 March 2007", "owner_id": 195000, "owner_name": "***", "owner_url":
"http://www.panoramio.com/user/195000"}
...
...
]}
```

To parse this JSON response we can use the `simplejson` package. Simplejson is available at `http://github.com/simplejson/simplejson` and there is online documentation available on the project page.

If you are running Python 2.6 or later there is no need to use `simplejson` as there is a JSON library included with these later versions of Python. To use the built in one, just import like this

```
import json
```

If you want to use `simplejson` where available (it is faster and could contain newer features than the built in one) a good idea is to import with a fallback, like this

```
try: import simplejson as json
except ImportError: import json
```

The following code will use the `urllib` package that comes with Python to handle the requests and then parse the result using `simplejson`.

```
import os
import urllib, urlparse
import simplejson as json

# query for images
url = 'http://www.panoramio.com/map/get_panoramas.php?order=popularity&\
    set=public&from=0&to=20&minx=-77.037564&miny=38.896662&\
    maxx=-77.035564&maxy=38.898662&size=medium'
c = urllib.urlopen(url)

# get the urls of individual images from JSON
j = json.loads(c.read())
imurls = []
for im in j['photos']:
```

Figure 2.8: Images taken at the same geographic location (square region centered around the White house) downloaded from panoramio.com.

```
  imurls.append(im['photo_file_url'])

# download images
for url in imurls:
  image = urllib.URLopener()
  image.retrieve(url, os.path.basename(urlparse.urlparse(url).path))
  print 'downloading:', url
```

As you can easily see by looking at the JSON output, it is the "photo_file_url" field we are after. Running the code above, you should see something like this in your console.

```
downloading: http://mw2.google.com/mw-panoramio/photos/medium/7715073.jpg
downloading: http://mw2.google.com/mw-panoramio/photos/medium/1303971.jpg
downloading: http://mw2.google.com/mw-panoramio/photos/medium/270077.jpg
downloading: http://mw2.google.com/mw-panoramio/photos/medium/15502.jpg
...
...
```

Figure 2.8 shows the 20 images returned for this example. Now we just need to find and match features between pairs of images.

Matching using local descriptors

Having downloaded the images, we now need to extract local descriptors. In this case we will use SIFT descriptors as described in the previous section. Let's assume that the images have been processed with the SIFT extraction code and the features are stored in files with the same name as the images (but with file ending ".sift" instead of ".jpg"). The lists *imlist* and *featlist* are assumed to contain the filenames. We can do a pairwise matching between all combinations as follows.

```
import sift

nbr_images = len(imlist)

matchscores = zeros((nbr_images,nbr_images))
for i in range(nbr_images):
  for j in range(i,nbr_images): # only compute upper triangle
    print 'comparing ', imlist[i], imlist[j]

    l1,d1 = sift.read_features_from_file(featlist[i])
    l2,d2 = sift.read_features_from_file(featlist[j])

    matches = sift.match_twosided(d1,d2)

    nbr_matches = sum(matches > 0)
    print 'number of matches = ', nbr_matches
    matchscores[i,j] = nbr_matches

# copy values
for i in range(nbr_images):
  for j in range(i+1,nbr_images): # no need to copy diagonal
    matchscores[j,i] = matchscores[i,j]
```

We store the number of matching features between each pair in *matchscores*. The last part of copying the values to fill the matrix completely is not necessary since this "distance measure" is symmetric, it just looks better that way. The *matchscores* matrix for these particular images looks like this:

```
662 0 0 2 0 0 0 0 1 0 0 1 2 0 3 0 19 1 0 2
0 901 0 1 0 0 0 1 1 0 0 1 0 0 0 0 0 0 1 2
0 0 266 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
2 1 0 1481 0 0 2 2 0 0 0 2 2 0 0 0 2 3 2 0
0 0 0 0 1748 0 0 1 0 0 0 0 0 2 0 0 0 0 0 1
0 0 0 0 0 1747 0 0 1 0 0 0 0 0 0 0 0 1 1 0
0 0 0 2 0 0 555 0 0 0 1 4 4 0 2 0 0 5 1 0
0 1 0 2 1 0 0 2206 0 0 0 1 0 0 1 0 2 0 1 1
1 1 0 0 0 1 0 0 629 0 0 0 0 0 0 0 1 0 0 20
0 0 0 0 0 0 0 0 0 829 0 0 1 0 0 0 0 0 0 2
0 0 0 0 0 0 1 0 0 0 1025 0 0 0 0 0 1 1 1 0
1 1 0 2 0 0 4 1 0 0 0 528 5 2 15 0 3 6 0 0
```

```
2 0 0 2 0 0 4 0 0 1 0 5 736 1 4 0 3 37 1 0
0 0 1 0 2 0 0 0 0 0 0 2 1 620 1 0 0 1 0 0
3 0 0 0 0 0 2 1 0 0 0 15 4 1 553 0 6 9 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2273 0 1 0 0
19 0 0 2 0 0 0 2 1 0 1 3 3 0 6 0 542 0 0 0
1 0 0 3 0 1 5 0 0 0 1 6 37 1 9 1 0 527 3 0
0 1 0 2 0 1 1 1 0 0 1 0 1 0 1 0 0 3 1139 0
2 2 0 0 1 0 0 1 20 2 0 0 0 0 0 0 0 0 0 499
```

Using this as a simple distance measure between images (images with similar content have higher number of matching features), we can now connect images with similar visual content.

Visualizing connected images

Let's visualize the connections between images defined by them having matching local descriptors. To do this we can show the images in a graph with edges indicating connections. We will use the `pydot` package (`http://code.google.com/p/pydot/`) which is a Python interface to the powerful GraphViz graphing library. Pydot uses Pyparsing (`http://pyparsing.wikispaces.com/`) and GraphViz (`http://www.graphviz.org/`) but don't worry, all of them are easy to install in just a few minutes.

Pydot is very easy to use. The following code snippet illustrates this nicely by creating a graph illustrating a tree with depth two and branching factor five adding numbering to the nodes. The graph is shown in Figure 2.9. There are many ways to customize the graph layout and appearance. For more details, see the Pydot documentation or the description of the DOT language used by GraphViz at `http://www.graphviz.org/Documentation.php`.

```
import pydot

g = pydot.Dot(graph_type='graph')

g.add_node(pydot.Node(str(0),fontcolor='transparent'))
for i in range(5):
  g.add_node(pydot.Node(str(i+1)))
  g.add_edge(pydot.Edge(str(0),str(i+1)))
  for j in range(5):
    g.add_node(pydot.Node(str(j+1)+'-'+str(i+1)))
    g.add_edge(pydot.Edge(str(j+1)+'-'+str(i+1),str(j+1)))
g.write_png('graph.jpg',prog='neato')
```

Let's get back to our example with the geotagged images. To create a graph showing potential groups of images, we create an edge between nodes if the number of matches is above a threshold. To get the images in the graph you need to use the full path of each image (represented by the variable *path* in the example below). To

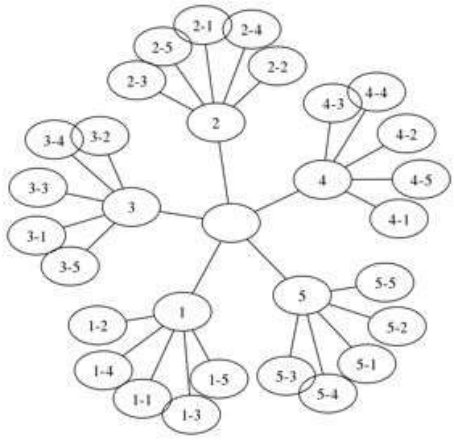


Figure 2.9: An example of using pydot to create graphs.

make it look nice we also scale each image to a thumbnail with largest side 100 pixels. Here's how to do it:

```
import pydot

threshold = 2 # min number of matches needed to create link

g = pydot.Dot(graph_type='graph') # don't want the default directed graph
for i in range(nbr_images):
  for j in range(i+1,nbr_images):
    if matchscores[i,j] > threshold:
      #first image in pair
      im = Image.open(imlist[i])
      im.thumbnail((100,100))
      filename = str(i)+'.png'
      im.save(filename) # need temporary files of the right size
      g.add_node(pydot.Node(str(i),fontcolor='transparent',
            shape='rectangle',image=path+filename))

      # second image in pair
      im = Image.open(imlist[j])
      im.thumbnail((100,100))
      filename = str(j)+'.png'
      im.save(filename) # need temporary files of the right size
      g.add_node(pydot.Node(str(j),fontcolor='transparent',
            shape='rectangle',image=path+filename))

      g.add_edge(pydot.Edge(str(i),str(j)))
```

```
g.write_png('whitehouse.png')
```

The result should look something like Figure 2.10 depending on which images you download. For this particular set, we see two groups of images, one from each side of the White house.

This application was a very simple example of using local descriptors for matching regions between images. For example, we did not use any verification on the matches. This can be done (in a very robust way) using concepts that we will define in the coming two chapters.

Exercises

1. Modify the function for matching Harris corner points to also take a maximum pixel distance between points for them to be considered as correspondences in order to make matching more robust.

2. Incrementally apply stronger blur (or ROF de-noising) to an image and extract Harris corners. What happens?

3. An alternative corner detector to Harris is the FAST corner detector. There are a number of implementations including a pure Python version available at `http://www.edwardrosten.com/work/fast.html`. Try this detector, play with the sensitivity threshold, and compare the corners with the ones from our Harris implementation.

4. Create copies of an image with different resolutions (for example by halving the size a few times). Extract SIFT features for each image. Plot and match features to get a feel for how and when the scale independence breaks down.

5. The VLFeat command line tools also contain an implementation of Maximally Stable Extremal Regions (MSER), `http://en.wikipedia.org/wiki/Maximally_stable_extremal_regions`, a region detector that finds blob like regions. Create a function for extracting MSER regions and pass them to the descriptor part of SIFT using the "--read-frames" option and one function for plotting the ellipse regions.

6. Write a function that matches features between a pair of images and estimates the scale difference and in-plane rotation of the scene based on the correspondences.

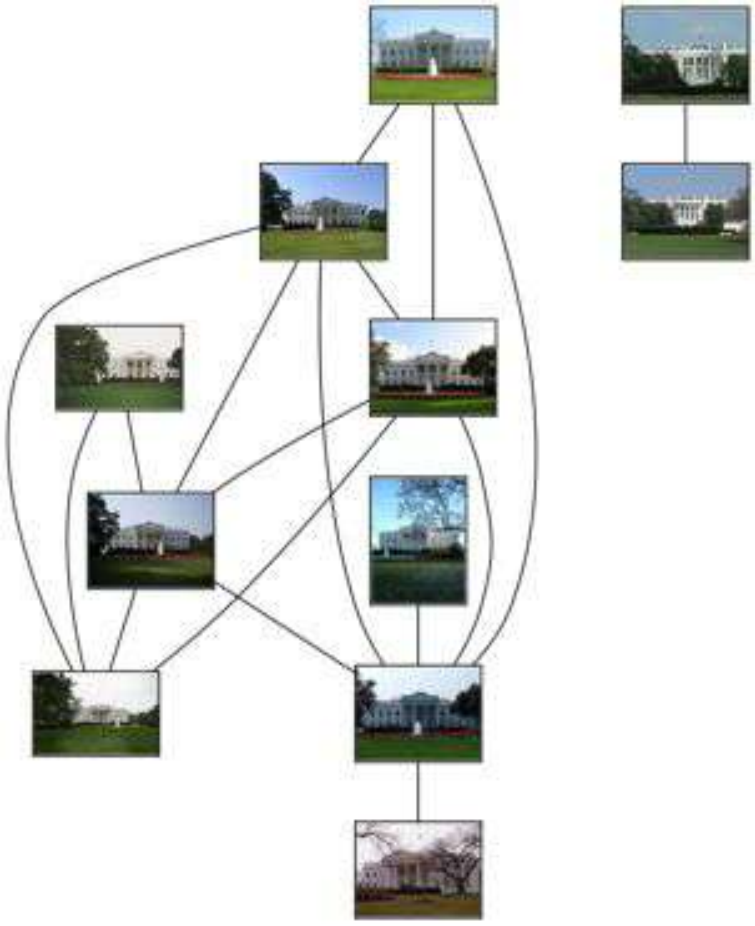


Figure 2.10: An example of grouping images taken at the same geographic location using local descriptors.

7. Download images for a location of your choice and match them as in the White house example. Can you find a better criteria for linking images? How could you use the graph to choose representative images for geographic locations?

Chapter 3

Image to Image Mappings

This chapter describes transformations between images and some practical methods for computing them. These transformations are used for warping, image registration and finally we look at an example of automatically creating panoramas.

3.1 Homographies

A *homography* is a 2D projective transformation that maps points in one plane to another. In our case the planes are images or planar surfaces in 3D. Homographies have many practical uses such as registering images, rectifying images, texture warping and creating panoramas. We will make frequent use of them. In essence a homography H maps 2D points (in homogeneous coordinates) according to

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & h_9 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix} \quad \text{or} \quad \mathbf{x}' = H\mathbf{x} \ .$$

Homogeneous coordinates is a useful representation for points in image planes (and in 3D as we will see later). Points in homogeneous coordinates are only defined up to scale so that $\mathbf{x} = [x, y, w] = [\alpha x, \alpha y, \alpha w] = [x/w, y/w, 1]$ all refer to the same 2D point. As a consequence, the homography H is also only defined up to scale and has eight independent degrees of freedom. Often points are normalized with $w = 1$ to have a unique identification of the image coordinates x,y. The extra coordinate makes is easy to represent transformations with a single matrix.

Create a file *homography.py* and add the following functions to normalize and convert to homogeneous coordinates.

```
def normalize(points):
```

```
  """ Normalize a collection of points in
    homogeneous coordinates so that last row = 1. """

  for row in points:
    row /= points[-1]
  return points


def make_homog(points):
  """ Convert a set of points (dim*n array) to
    homogeneous coordinates. """

  return vstack((points,ones((1,points.shape[1]))))
```

When working with points and transformations we will store the points column-wise so that a set of n points in 2 dimensions will be a $3 \times n$ array in homogeneous coordinates. This format makes matrix multiplications and point transforms easier. For all other cases we will typically use rows to store data, for example features for clustering and classification.

There are some important special cases of these projective transformations. An *affine transformation*

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a_1 & a_2 & t_x \\ a_3 & a_4 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad \text{or} \quad \mathbf{x}' = \begin{bmatrix} A & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix} \mathbf{x} \ ,$$

preserves $w = 1$ and can not represent as strong deformations as a full projective transformation. The affine transformation contains an invertible matrix A and a translation vector $\mathbf{t} = [t_x, t_y]$. Affine transformations are used for example in warping.

A *similarity transformation*

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s\cos(\theta) & -s\sin(\theta) & t_x \\ s\sin(\theta) & s\cos(\theta) & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad \text{or} \quad \mathbf{x}' = \begin{bmatrix} sR & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix} \mathbf{x} \ ,$$

is a rigid 2D transformation that also includes scale changes. The scalar s specifies scaling, R is a rotation of an angle θ and $\mathbf{t} = [t_x, t_y]$ is again a translation. With $s = 1$ distances are preserved and it is then a *rigid transformation*. Similarity transformations are used for example in image registration.

Let's look at algorithms for estimating homographies and then go into examples of using affine transformations for warping, similarity transformations for registration and finally full projective transformations for creating panoramas.

The direct linear transformation algorithm

Homographies can be computed directly from corresponding points in two images (or planes). As mentioned earlier, a full projective transformation has eight degrees of freedom. Each point correspondence gives two equations, one each for the x and y coordinates, and therefore four point correspondences are needed to compute H.

The *direct linear transformation* (DLT) is an algorithm for computing H given four or more correspondences. By rewriting the equation for mapping points using H for several correspondences we get an equation like

$$\begin{bmatrix} -x_1 & -y_1 & -1 & 0 & 0 & 0 & x_1x_1' & y_1x_1' & x_1' \\ 0 & 0 & 0 & -x_1 & -y_1 & -1 & x_1y_1' & y_1y_1' & y_1' \\ -x_2 & -y_2 & -1 & 0 & 0 & 0 & x_2x_2' & y_2x_2' & x_2' \\ 0 & 0 & 0 & -x_2 & -y_2 & -1 & x_2y_2' & y_2y_2' & y_2' \\ & \vdots & & \vdots & & \vdots & & \vdots & \end{bmatrix} \begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \\ h_5 \\ h_6 \\ h_7 \\ h_8 \\ h_9 \end{bmatrix} = \mathbf{0} \ ,$$

or $A\mathbf{h} = \mathbf{0}$ where A is a matrix with twice as many rows as correspondences. By stacking all corresponding points a least squares solution for H can be found using singular value decomposition (SVD). Here's what it looks like in code. Add the function below to *homography.py*.

```
def H_from_points(fp,tp):
  """ Find homography H, such that fp is mapped to tp
    using the linear DLT method. Points are conditioned
    automatically. """

  if fp.shape != tp.shape:
    raise RuntimeError('number of points do not match')

  # condition points (important for numerical reasons)
  # --from points--
  m = mean(fp[:2], axis=1)
  maxstd = max(std(fp[:2], axis=1)) + 1e-9
  C1 = diag([1/maxstd, 1/maxstd, 1])
  C1[0][2] = -m[0]/maxstd
  C1[1][2] = -m[1]/maxstd
  fp = dot(C1,fp)

  # --to points--
  m = mean(tp[:2], axis=1)
  maxstd = max(std(tp[:2], axis=1)) + 1e-9
```

```
  C2 = diag([1/maxstd, 1/maxstd, 1])
  C2[0][2] = -m[0]/maxstd
  C2[1][2] = -m[1]/maxstd
  tp = dot(C2,tp)

  # create matrix for linear method, 2 rows for each correspondence pair
  nbr_correspondences = fp.shape[1]
  A = zeros((2*nbr_correspondences,9))
  for i in range(nbr_correspondences):
    A[2*i] = [-fp[0][i],-fp[1][i],-1,0,0,0,
          tp[0][i]*fp[0][i],tp[0][i]*fp[1][i],tp[0][i]]
    A[2*i+1] = [0,0,0,-fp[0][i],-fp[1][i],-1,
          tp[1][i]*fp[0][i],tp[1][i]*fp[1][i],tp[1][i]]

  U,S,V = linalg.svd(A)
  H = V[8].reshape((3,3))

  # decondition
  H = dot(linalg.inv(C2),dot(H,C1))

  # normalize and return
  return H / H[2,2]
```

The first thing that happens in this function is a check that the number of points are equal. If not an exception is thrown. This is useful for writing robust code but we will only use exceptions in very few cases in this book to make the code samples simpler and easier to follow. You can read more about exception types at `http://docs.python.org/library/exceptions.html` and how to use them at `http://docs.python.org/tutorial/errors.html`.

The points are conditioned by normalizing so that they have zero mean and unit standard deviation. This is very important for numerical reasons since the stability of the algorithm is dependent of the coordinate representation. Then the matrix A is created using the point correspondences. The least squares solution is found as the last row of the matrix V of the SVD. The row is reshaped to create H. This matrix is then de-conditioned and normalized before returned.

Affine transformations

An affine transformation has six degrees of freedom and therefore three point correspondences are needed to estimate H. Affine transforms can be estimated using the DLT algorithm above by setting the last two elements equal to zero, $h_7 = h_8 = 0$.

Here we will use a different approach, described in detail in [13] (page 130). Add the following function to *homography.py*, which computes the affine transformation matrix from point correspondences.

```
def Haffine_from_points(fp,tp):
  """ Find H, affine transformation, such that
    tp is affine transf of fp. """

  if fp.shape != tp.shape:
    raise RuntimeError('number of points do not match')

  # condition points
  # --from points--
  m = mean(fp[:2], axis=1)
  maxstd = max(std(fp[:2], axis=1)) + 1e-9
  C1 = diag([1/maxstd, 1/maxstd, 1])
  C1[0][2] = -m[0]/maxstd
  C1[1][2] = -m[1]/maxstd
  fp_cond = dot(C1,fp)

  # --to points--
  m = mean(tp[:2], axis=1)
  C2 = C1.copy() #must use same scaling for both point sets
  C2[0][2] = -m[0]/maxstd
  C2[1][2] = -m[1]/maxstd
  tp_cond = dot(C2,tp)

  # conditioned points have mean zero, so translation is zero
  A = concatenate((fp_cond[:2],tp_cond[:2]), axis=0)
  U,S,V = linalg.svd(A.T)

  # create B and C matrices as Hartley-Zisserman (2:nd ed) p 130.
  tmp = V[:2].T
  B = tmp[:2]
  C = tmp[2:4]

  tmp2 = concatenate((dot(C,linalg.pinv(B)),zeros((2,1))), axis=1)
  H = vstack((tmp2,[0,0,1]))

  # decondition
  H = dot(linalg.inv(C2),dot(H,C1))

  return H / H[2,2]
```

Again the points are conditioned and de-conditioned as in the DLT algorithm. Let's see what these affine transformations can do with images in the next section.

3.2 Warping images

Applying an affine transformation matrix H on image patches is called *warping* (or *affine warping*) and is frequently used in computer graphics but also in several computer vision algorithms. A warp can easily be performed with SciPy using the ndimage package. The command

```
transformed_im = ndimage.affine_transform(im,A,b,size)
```

transforms the image patch *im* with A a linear transformation and $\mathbf{b}$ a translation vector as above. The optional argument *size* can be used to specify the size of the output image. The default is an image with the same size as the original. To see how this works, try running the following commands:

```
from scipy import ndimage

im = array(Image.open('empire.jpg').convert('L'))
H = array([[1.4,0.05,-100],[0.05,1.5,-100],[0,0,1]])
im2 = ndimage.affine_transform(im,H[:2,:2],(H[0,2],H[1,2]))

figure()
gray()
imshow(im2)
show()
```

This gives a result like the image to the right in Figure 3.1. As you can see, missing pixel values in the result image are filled with zeros.

Image in image

A simple example of affine warping is to place images, or parts of images, inside another image so that they line up with specific areas or landmarks.

Add the function image_in_image() to *warp.py*. This function takes two images and the corner coordinates of where to put the first image in the second.

```
def image_in_image(im1,im2,tp):
  """ Put im1 in im2 with an affine transformation
    such that corners are as close to tp as possible.
    tp are homogeneous and counter-clockwise from top left. """

  # points to warp from
  m,n = im1.shape[:2]
  fp = array([[0,m,m,0],[0,0,n,n],[1,1,1,1]])

  # compute affine transform and apply
  H = homography.Haffine_from_points(tp,fp)
```

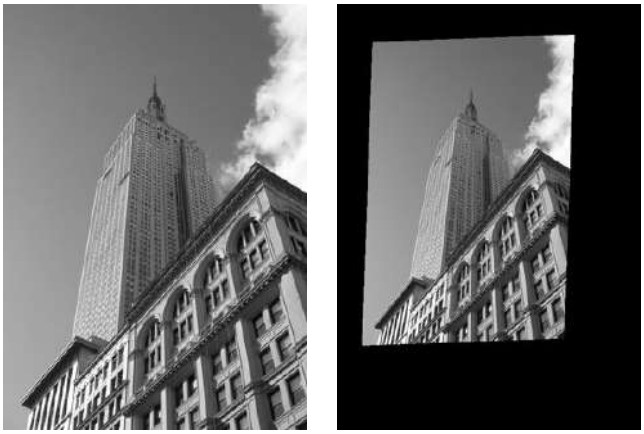


Figure 3.1: An example of warping an image using an affine transform, (left) original, (right) image after warping with *ndimage.affine_transform()*.

```
  im1_t = ndimage.affine_transform(im1,H[:2,:2],
          (H[0,2],H[1,2]),im2.shape[:2])
  alpha = (im1_t > 0)

  return (1-alpha)*im2 + alpha*im1_t
```

As you can see, there is not much needed to do this. When blending together the warped image and the second image we create an *alpha map* which defines how much of each pixel to take from each image. Here we use the fact that the warped image is filled with zeros outside the borders of the warped area to create a binary alpha map. To be really strict we could have added a small number to the potential zero pixels of the first image, or done it properly, see exercises at the end of the chapter. Note that the image coordinates are in homogeneous form.

To try this function, let's insert an image on a billboard in another image. The following lines of code will put the leftmost image of Figure 3.2 into the second image. The coordinates were determined manually by looking at a plot of the image (in PyLab figures the mouse coordinates are shown near the bottom). PyLab's `ginput()` could of course also have been used.

```
import warp

# example of affine warp of im1 onto im2
im1 = array(Image.open('beatles.jpg').convert('L'))
im2 = array(Image.open('billboard_for_rent.jpg').convert('L'))

# set to points
```

Figure 3.2: An example of placing an image inside another image using an affine transformation.

```
tp = array([[264,538,540,264],[40,36,605,605],[1,1,1,1]])

im3 = warp.image_in_image(im1,im2,tp)

figure()
gray()
imshow(im3)
axis('equal')
axis('off')
show()
```

This puts the image on the upper part of the billboard. Note again that the landmark coordinates *tp* are in homogeneous coordinates. Changing the coordinates to

```
tp = array([[675,826,826,677],[55,52,281,277],[1,1,1,1]])
```

will put the image on the lower left "for rent" part.

The function `Haffine_from_points()` gives the best affine transform for the given point correspondences. In the example above those were the image corners and the corners of the billboard. If the perspective effects are small, this will give good results. The top row of Figure 3.3 shows what happens if we try to use an affine transformation to a billboard image with more perspective. It is not possible to transform all four corner points to their target locations with the same affine transform (a full projective transform would have been able to do this though). If you want to use an affine warp so that all corner points match, there is a useful trick.

For three points an affine transform can warp an image so that the three correspondences match perfectly. This is because an affine transform has six degrees of

freedom and three correspondences give exactly six constraints (x and y coordinates must match for all three). So if you really want the image to fit the billboard using affine transforms, you can divide the image into two triangles and warp them separately. Here's how to do it.

```
# set from points to corners of im1
m,n = im1.shape[:2]
fp = array([[0,m,m,0],[0,0,n,n],[1,1,1,1]])

# first triangle
tp2 = tp[:,:3]
fp2 = fp[:,:3]

# compute H
H = homography.Haffine_from_points(tp2,fp2)
im1_t = ndimage.affine_transform(im1,H[:2,:2],
          (H[0,2],H[1,2]),im2.shape[:2])

# alpha for triangle
alpha = warp.alpha_for_triangle(tp2,im2.shape[0],im2.shape[1])
im3 = (1-alpha)*im2 + alpha*im1_t

# second triangle
tp2 = tp[:,[0,2,3]]
fp2 = fp[:,[0,2,3]]

# compute H
H = homography.Haffine_from_points(tp2,fp2)
im1_t = ndimage.affine_transform(im1,H[:2,:2],
          (H[0,2],H[1,2]),im2.shape[:2])

# alpha for triangle
alpha = warp.alpha_for_triangle(tp2,im2.shape[0],im2.shape[1])
im4 = (1-alpha)*im3 + alpha*im1_t

figure()
gray()
imshow(im4)
axis('equal')
axis('off')
show()
```

Here we simply create the alpha map for each triangle and then merge all images together. The alpha map for a triangle can be computed by simply checking for each pixel if that pixel's coordinates has a convex combination of the triangle's corner points

that have all coefficients positive[1]. That means the pixel is inside the triangle. Add the following function `alpha_for_triangle()`, which was used in the example above, to *warp.py*.

```
def alpha_for_triangle(points,m,n):
  """ Creates alpha map of size (m,n)
    for a triangle with corners defined by points
    (given in normalized homogeneous coordinates). """

  alpha = zeros((m,n))
  for i in range(min(points[0]),max(points[0])):
    for j in range(min(points[1]),max(points[1])):
      x = linalg.solve(points,[i,j,1])
      if min(x) > 0: #all coefficients positive
        alpha[i,j] = 1
  return alpha
```

This is an operation your graphics card can do extremely fast. Python is a lot slower than your graphics card (or a C/C++ implementation for that matter) but it works just fine for our purposes. As you can see at the bottom of Figure 3.3, the corners now match.

Piecewise affine warping

As we saw in the example above, affine warping of triangle patches can be done to exactly match the corner points. Let's look at the most common form of warping between a set of corresponding points, *piecewise affine warping*. Given any image with landmark points we can warp that image to corresponding landmarks in another image by triangulating the points into a triangle mesh and then warping each triangle with an affine transform. These are standard operations for any graphics and image processing library. Here we show how to do this using `PyLab` and `SciPy`.

To triangulate points, *Delaunay triangulation* is often used. An implementation of Delaunay triangulation comes included in `Matplotlib` (but outside the `PyLab` part) and can be used like this:

```
import matplotlib.delaunay as md

x,y =  array(random.standard_normal((2,100)))
centers,edges,tri,neighbors = md.delaunay(x,y)

figure()
for t in tri:
```

[1] A *convex combination* is a linear combination $\sum_j \alpha_j \mathbf{x}_i$ (in this case of the triangle points) such that all coefficients α_j are non-negative and sum to 1.

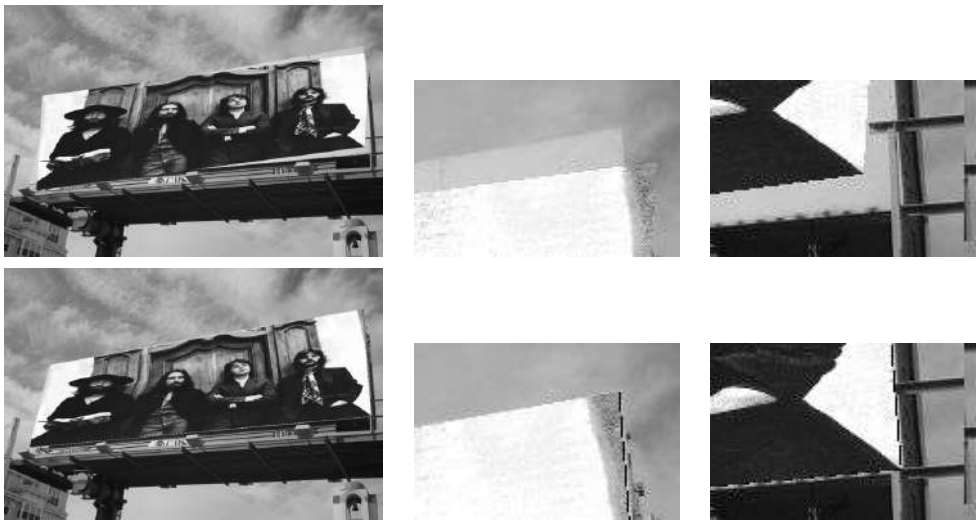


Figure 3.3: Comparing an affine warp of the full image with an affine warp using two triangles. The image is placed on a billboard with some perspective effects. (top) using an affine transform for the whole image results in a bad fit. The two right hand corners are enlarged for clarity. (bottom) using an affine warp consisting of two triangles gives an exact fit.

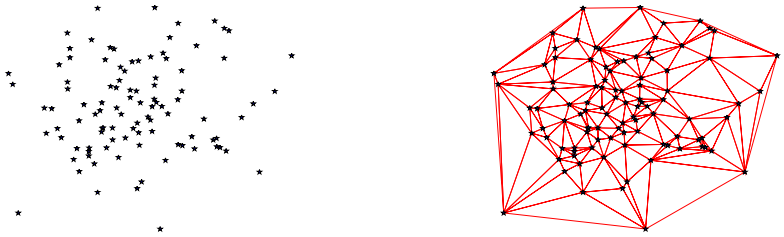


Figure 3.4: An example of Delaunay triangulation of a set of random 2D points.

```
    t_ext = [t[0], t[1], t[2], t[0]] # add first point to end
    plot(x[t_ext],y[t_ext],'r')

plot(x,y,'*')
axis('off')
show()
```

Figure 3.4 shows some example points and the resulting triangulation. Delaunay triangulation chooses the triangles so that the minimum angle of all the angles of the triangles in the triangulation is maximized[2]. There are four outputs of `delaunay()` of which we only need the list of triangles (the third of the outputs). Create a function in *warp.py* for the triangulation.

```
import matplotlib.delaunay as md

def triangulate_points(x,y):
  """ Delaunay triangulation of 2D points. """

  centers,edges,tri,neighbors = md.delaunay(x,y)
  return tri
```

The output is an array with each row containing the indices in the arrays x and y for the three points of each triangle.

Let's now apply this to an example of warping an image to a non-flat object in another image using 30 control points in a 5 by 6 grid. Figure 3.5 shows an image to be warped to the facade of the "turning torso". The target points were manually selected using `ginput()` and stored in the file *turningtorso_points.txt*.

[2]The edges are actually the dual graph of a Voronoi diagram, see `http://en.wikipedia.org/wiki/Delaunay_triangulation`.

First we need a general warp function for piecewise affine image warping. The code below does the trick, where we also take the opportunity to show how to warp color images (you simply warp each color channel).

```
def pw_affine(fromim,toim,fp,tp,tri):
  """ Warp triangular patches from an image.
    fromim = image to warp
    toim = destination image
    fp = from points in hom. coordinates
    tp = to points in hom.  coordinates
    tri = triangulation. """

  im = toim.copy()

  # check if image is grayscale or color
  is_color = len(fromim.shape) == 3

  # create image to warp to (needed if iterate colors)
  im_t = zeros(im.shape, 'uint8')

  for t in tri:
    # compute affine transformation
    H = homography.Haffine_from_points(tp[:,t],fp[:,t])

    if is_color:
      for col in range(fromim.shape[2]):
        im_t[:,:,col] = ndimage.affine_transform(
          fromim[:,:,col],H[:2,:2],(H[0,2],H[1,2]),im.shape[:2])
    else:
      im_t = ndimage.affine_transform(
          fromim,H[:2,:2],(H[0,2],H[1,2]),im.shape[:2])

    # alpha for triangle
    alpha = alpha_for_triangle(tp[:,t],im.shape[0],im.shape[1])

    # add triangle to image
    im[alpha>0] = im_t[alpha>0]

  return im
```

Here we first check if the image is grayscale or color and in the case of colors, we warp each color channel. The affine transform for each triangle is uniquely determined so we use `Haffine_from_points()`. Add this function to the file *warp.py*.

To use this function on the current example, the following short script puts it all together:

```
import homography
```

```
import warp

# open image to warp
fromim = array(Image.open('sunset_tree.jpg'))
x,y = meshgrid(range(5),range(6))
x = (fromim.shape[1]/4) * x.flatten()
y = (fromim.shape[0]/5) * y.flatten()

# triangulate
tri = warp.triangulate_points(x,y)

# open image and destination points
im = array(Image.open('turningtorso1.jpg'))
tp = loadtxt('turningtorso1_points.txt') # destination points

# convert points to hom. coordinates
fp = vstack((y,x,ones((1,len(x)))))
tp = vstack((tp[:,1],tp[:,0],ones((1,len(tp)))))

# warp triangles
im = warp.pw_affine(fromim,im,fp,tp,tri)

# plot
figure()
imshow(im)
warp.plot_mesh(tp[1],tp[0],tri)
axis('off')
show()
```

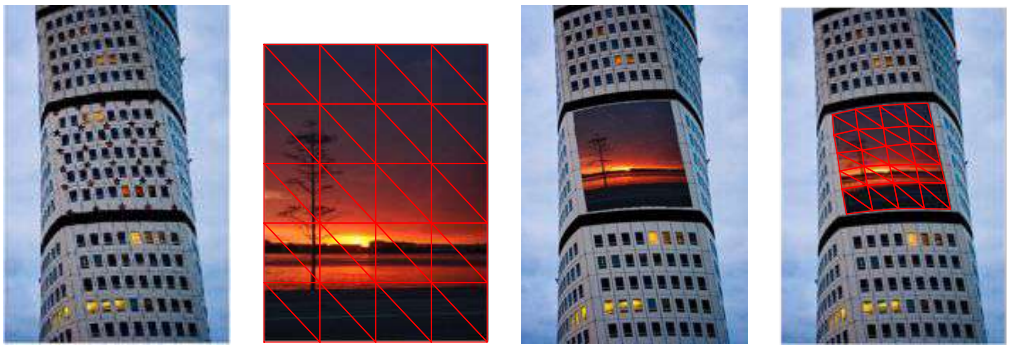
The resulting image is shown in Figure 3.5. The triangles are plotted with the following helper function (add this to *warp.py*):

```
def plot_mesh(x,y,tri):
  """ Plot triangles. """

  for t in tri:
    t_ext = [t[0], t[1], t[2], t[0]] # add first point to end
    plot(x[t_ext],y[t_ext],'r')
```

This example should give you all you need to apply piece-wise affine warping of images to your own applications. There are many improvements that can be made to the functions used, let's leave some to the exercises and the rest to you.

Registering images

Image registration is the process of transferring images so that they are aligned in a common coordinate frame. Registration can be rigid or non-rigid and and is an



(a) (b) (c) (d)

Figure 3.5: An example of piecewise affine warping using Delaunay triangulated landmark points. (a) the target image with landmarks. (b) image with triangulation. (c) with warped image. (d) with warped image and triangulation.

important step in order to be able to do image comparisons and more sophisticated analysis.

Let's look at an example of rigidly registering a set of face images so that we can compute the mean face and face appearance variations in a meaningful way. In this type of registration we are actually looking for a similarity transform (rigid with scale) to map correspondences. This is because the faces are not all at the same size, position and rotation in the images.

In the file *jkfaces.zip* are 366 images of a single person (one for each day in 2008)[3]. The images are annotated with eye and mouth coordinates in the file *jkfaces.xml*. Using the points, a similarity transformation can be computed and the images warped to a normalized coordinate frame using this transformation (which as mentioned, includes scaling). To read XML files we will use `minidom` that comes with Python's built in `xml.dom` module.

The XML file looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<faces>
        <face file="jk-002.jpg" xf="46" xm="56" xs="67" yf="38" ym="65" ys="39"/>
        <face file="jk-006.jpg" xf="38" xm="48" xs="59" yf="38" ym="65" ys="38"/>
        <face file="jk-004.jpg" xf="40" xm="50" xs="61" yf="38" ym="66" ys="39"/>
        <face file="jk-010.jpg" xf="33" xm="44" xs="55" yf="38" ym="65" ys="38"/>
...
...
</faces>
```

[3]Images are courtesy of JK Keller (with permission), see `http://jk-keller.com/daily-photo/` for more details.

To read the coordinates from the file, add the following function that uses `minidom` to a new file *imregistration.py*.

```
from xml.dom import minidom

def read_points_from_xml(xmlFileName):
  """ Reads control points for face alignment. """

  xmldoc = minidom.parse(xmlFileName)
  facelist = xmldoc.getElementsByTagName('face')
  faces = {}
  for xmlFace in facelist:
    fileName = xmlFace.attributes['file'].value
    xf = int(xmlFace.attributes['xf'].value)
    yf = int(xmlFace.attributes['yf'].value)
    xs = int(xmlFace.attributes['xs'].value)
    ys = int(xmlFace.attributes['ys'].value)
    xm = int(xmlFace.attributes['xm'].value)
    ym = int(xmlFace.attributes['ym'].value)
    faces[fileName] = array([xf, yf, xs, ys, xm, ym])
  return faces
```

The landmark points are returned in a Python dictionary with the filename of the image as key. The format is; *xf,yf* coordinates of the leftmost eye in the image (the person's right), *xs,ys* coordinates of the rightmost eye and *xm,ym* mouth coordinates. To compute the parameters of the similarity transformation we can use a least squares solution. For each point $\mathbf{x}_i = [x_i, y_i]$ (in this case there are three of them), the point should be mapped to the target location $[\hat{x}_i, \hat{y}_i]$ as

$$\begin{bmatrix} \hat{x}_i \\ \hat{y}_i \end{bmatrix} = \begin{bmatrix} a & -b \\ b & a \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix} \ .$$

Taking all three points, we can rewrite this as a system of equations with the unknowns a, b, t_x, t_y like this

$$\begin{bmatrix} \hat{x}_1 \\ \hat{y}_1 \\ \hat{x}_2 \\ \hat{y}_2 \\ \hat{x}_3 \\ \hat{y}_3 \end{bmatrix} = \begin{bmatrix} x_1 & -y_1 & 1 & 0 \\ y_1 & x_1 & 0 & 1 \\ x_2 & -y_2 & 1 & 0 \\ y_2 & x_2 & 0 & 1 \\ x_3 & -y_3 & 1 & 0 \\ y_3 & x_3 & 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ t_x \\ t_y \end{bmatrix} \ .$$

Here we used the parameterization of similarity matrices

$$\begin{bmatrix} a & -b \\ b & a \end{bmatrix} = s \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} = sR \ ,$$

with scale $s = \sqrt{a^2 + b^2}$ and rotation matrix R.

More point correspondences would work the same way and only add extra rows to the matrix. The least squares solution is found using `linalg.lstsq()`. This idea of using least squares solutions is a standard trick that will be used many times in this book. Actually this is the same as used in the DLT algorithm earlier.

The code looks like this (add to *imregistration.py*):

```
from scipy import linalg

def compute_rigid_transform(refpoints,points):
  """ Computes rotation, scale and translation for
    aligning points to refpoints. """

  A = array([ [points[0], -points[1], 1, 0],
        [points[1],  points[0], 0, 1],
        [points[2], -points[3], 1, 0],
        [points[3],  points[2], 0, 1],
        [points[4], -points[5], 1, 0],
        [points[5],  points[4], 0, 1]])

  y = array([ refpoints[0],
        refpoints[1],
        refpoints[2],
        refpoints[3],
        refpoints[4],
        refpoints[5]])

  # least sq solution to mimimize ||Ax - y||
  a,b,tx,ty = linalg.lstsq(A,y)[0]
  R = array([[a, -b], [b, a]]) # rotation matrix incl scale

  return R,tx,ty
```

The function returns a rotation matrix with scale as well as translation in the x and y directions. To warp the images and store new aligned images we can apply `ndimage.affine_transform()` to each color channel (these are color images). As reference frame, any three point coordinates could be used. Here we will use the landmark locations in the first image for simplicity.

```
from scipy import ndimage
from scipy.misc import imsave
import os

def rigid_alignment(faces,path,plotflag=False):
  """ Align images rigidly and save as new images.
    path determines where the aligned images are saved
    set plotflag=True to plot the images. """
```

```
  # take the points in the first image as reference points
  refpoints = faces.values()[0]

  # warp each image using affine transform
  for face in faces:
    points = faces[face]

    R,tx,ty = compute_rigid_transform(refpoints, points)
    T = array([[R[1][1], R[1][0]], [R[0][1], R[0][0]]])

    im = array(Image.open(os.path.join(path,face)))
    im2 = zeros(im.shape, 'uint8')

    # warp each color channel
    for i in range(len(im.shape)):
      im2[:,:,i] = ndimage.affine_transform(im[:,:,i],linalg.inv(T),offset=[-ty,-tx])

    if plotflag:
      imshow(im2)
      show()

    # crop away border and save aligned images
    h,w = im2.shape[:2]
    border = (w+h)/20

    # crop away border
    imsave(os.path.join(path, 'aligned/'+face),im2[border:h-border,border:w-border,:])
```

Here we use the `imsave()` function to save the aligned images to a sub-directory "aligned".

The following short script will read the XML file containing filenames as keys and points as values and then register all the images to align them with the first one.

```
import imregistration

# load the location of control points
xmlFileName = 'jkfaces2008_small/jkfaces.xml'
points = imregistration.read_points_from_xml(xmlFileName)

# register
imregistration.rigid_alignment(points,'jkfaces2008_small/')
```

If you run this you should get aligned face images in a sub-directory. Figure 3.6 shows six sample images before and after registration. The registered images are cropped slightly to remove the undesired black fill pixels that may appear at the borders of the images.

Figure 3.6: Sample images before (top) and after rigid registration (bottom).

Now let's see how this affects the mean image. Figure 3.7 shows the mean image for the unaligned face images next to the mean image of the aligned images (note the size difference due to cropping the borders of the aligned images). Although the original images show very little variation in size of the face, rotation and position, the effects on the mean computation is drastic.

Not surprisingly, using badly registered images also has a drastic impact on the computation of principal components. Figure 3.8 shows the result of PCA on the first 150 images from this set without and with registration. Just as with the mean image, the PCA-modes are blurry. When computing the principal components we used a mask consisting of an ellipse centered around the mean face position. By multiplying the images with this mask before stacking them we can avoid bringing background variations into the PCA-modes. Just replace the line that creates the matrix in the PCA example in Section 1.3 (page 29) with:

```
immatrix = array([mask*array(Image.open(imlist[i]).convert('L')).flatten()
        for i in range(150)],'f')
```

where *mask* is a binary image of the same size, already flattened.

3.3 Creating Panoramas

Two (or more) images that are taken at the same location (that is, the camera position is the same for the images) are homographically related. This is frequently used for creating panoramic images where several images are stitched together into one big mosaic. In this section we will explore how this is done.

Figure 3.7: Comparing mean images. (left) without alignment. (right) with three-point rigid alignment.

RANSAC

RANSAC, short for "RANdom SAmple Consensus", is an iterative method to fit models to data that can contain outliers. Given a model, for example a homography between sets of points, the basic idea is that the data contains *inliers*, the data points that can be described by the model, and *outliers*, those that do not fit the model.

The standard example is the case of fitting a line to a set of points that contains outliers. Simple least squares fitting will fail but RANSAC can hopefully single out the inliers and obtain the correct fit. Let's look at using *ransac.py* from `http://www.scipy.org/Cookbook/RANSAC` which contains this particular example as test case. Figure 3.10 shows an example of running `ransac.test()`. As you can see, the algorithm selects only points consistent with a line model and correctly finds the right solution.

RANSAC is a very useful algorithm which we will use in the next section for homography estimation and again for other examples. For more information, see the original paper by Fischler and Bolles [11], Wikipedia `http://en.wikipedia.org/wiki/RANSAC` or the report [40].

Robust homography estimation

We can use this RANSAC module for any model. All that is needed is a Python class with `fit()` and `get_error()` methods, the rest is taken care of by *ransac.py*. Here we are interested in automatically finding a homography for the panorama images using a set of possible correspondences. Figure 3.11 shows the matching correspondences found automatically using SIFT features by running the following commands:

Figure 3.8: Comparing PCA-modes of unregistered and registered images. (top) the mean image and the first nine principal components without registering the images beforehand. (bottom) the same with the registered images.

Figure 3.9: Five images of the main university building in Lund, Sweden. The images are all taken from the same viewpoint.

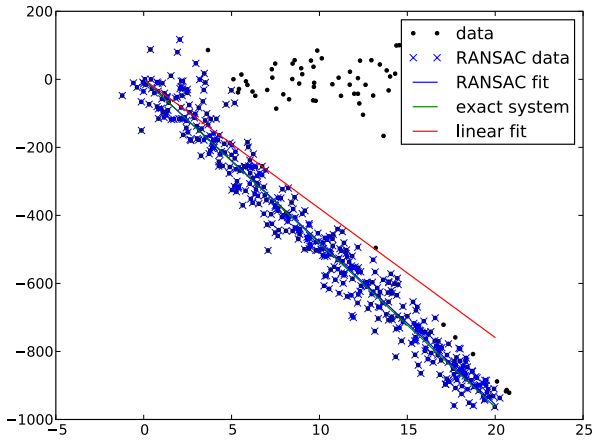


Figure 3.10: An example of using RANSAC to fit a line to points with outliers.

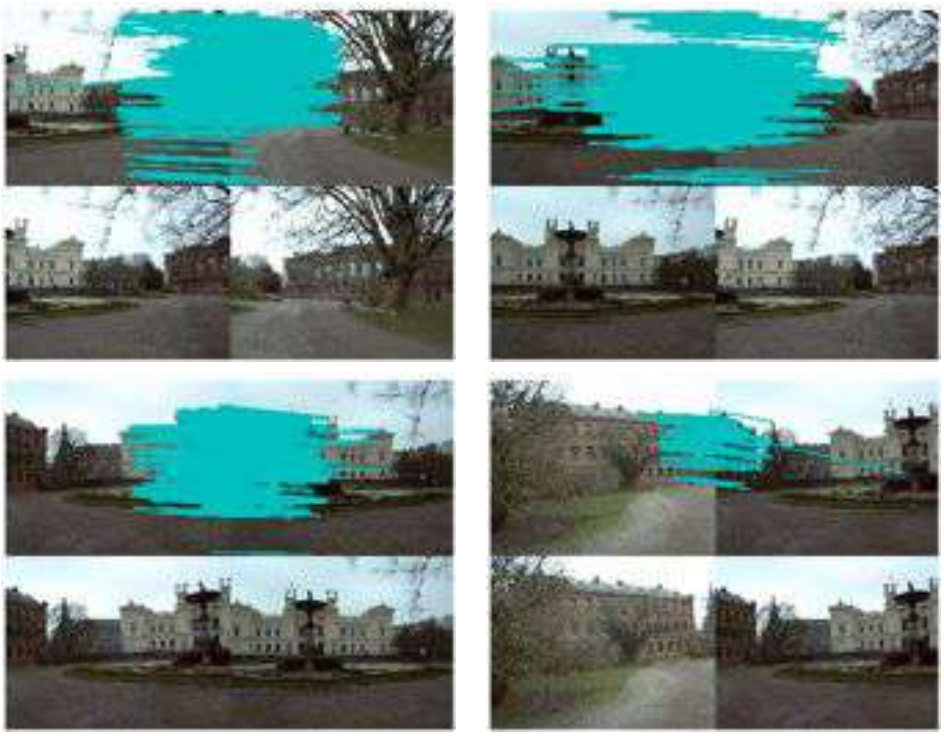


Figure 3.11: Matching correspondences found between consecutive image pairs using SIFT features.

```
import sift

featname = ['Univ'+str(i+1)+'.sift' for i in range(5)]
imname = ['Univ'+str(i+1)+'.jpg' for i in range(5)]
l = {}
d = {}
for i in range(5):
  sift.process_image(imname[i],featname[i])
  l[i],d[i] = sift.read_features_from_file(featname[i])

matches = {}
for i in range(4):
  matches[i] = sift.match(d[i+1],d[i])
```

It is clear from the images that not all correspondences are correct. SIFT is actually a very robust descriptor and gives fewer false matches than for example Harris points with patch correlation, but still it is far from perfect.

To fit a homography using RANSAC we first need to add the following model class to *homography.py*.

```
class RansacModel(object):
  """ Class for testing homography fit with ransac.py from
    http://www.scipy.org/Cookbook/RANSAC"""

  def __init__(self,debug=False):
    self.debug = debug

  def fit(self, data):
    """ Fit homography to four selected correspondences. """

    # transpose to fit H_from_points()
    data = data.T

    # from points
    fp = data[:3,:4]
    # target points
    tp = data[3:,:4]

    # fit homography and return
    return H_from_points(fp,tp)


  def get_error( self, data, H):
    """ Apply homography to all correspondences,
      return error for each transformed point. """

    data = data.T

    # from points
    fp = data[:3]
    # target points
    tp = data[3:]

    # transform fp
    fp_transformed = dot(H,fp)

    # normalize hom. coordinates
    for i in range(3):
      fp_transformed[i] /= fp_transformed[2]

    # return error per point
    return sqrt( sum((tp-fp_transformed)**2,axis=0) )
```

As you can see, this class contains a `fit()` method which just takes the four correspondences selected by *ransac.py* (they are the first four in *data*) and fits a homogra-

phy. Remember, four points are the minimal number to compute a homography. The method `get_error()` applies the homography and returns the sum of squared distance for each correspondence pair so that RANSAC can chose which points to keep as inliers and outliers. This is done with a threshold on this distance. For ease of use, add the following function to *homography.py*.

```
def H_from_ransac(fp,tp,model,maxiter=1000,match_theshold=10):
  """ Robust estimation of homography H from point
    correspondences using RANSAC (ransac.py from
    http://www.scipy.org/Cookbook/RANSAC).

    input: fp,tp (3*n arrays) points in hom. coordinates. """

  import ransac

  # group corresponding points
  data = vstack((fp,tp))

  # compute H and return
  H,ransac_data = ransac.ransac(data.T,model,4,maxiter,match_theshold,10,return_all=True)
  return H,ransac_data['inliers']
```

The function also lets you supply the threshold and the minimum number of points desired. The most important parameter is the maximum number of iterations, exiting too early might give a worse solution, too many iterations will take more time. The resulting homography is returned together with the inlier points.

Apply RANSAC to the correspondences like this.

```
# function to convert the matches to hom. points
def convert_points(j):
  ndx = matches[j].nonzero()[0]
  fp = homography.make_homog(l[j+1][ndx,:2].T)
  ndx2 = [int(matches[j][i]) for i in ndx]
  tp = homography.make_homog(l[j][ndx2,:2].T)
  return fp,tp

# estimate the homographies
model = homography.RansacModel()

fp,tp = convert_points(1)
H_12 = homography.H_from_ransac(fp,tp,model)[0] #im 1 to 2

fp,tp = convert_points(0)
H_01 = homography.H_from_ransac(fp,tp,model)[0] #im 0 to 1

tp,fp = convert_points(2) #NB: reverse order
H_32 = homography.H_from_ransac(fp,tp,model)[0] #im 3 to 2
```

```
tp,fp = convert_points(3) #NB: reverse order
H_43 = homography.H_from_ransac(fp,tp,model)[0] #im 4 to 3
```

In this example image number 2 is the central image and the one we want to warp the others to. Image 0 and 1 should be warped from the right and image 3 and 4 from the left. The matches were computed from the rightmost image in each pair, therefore we reverse the order of the correspondences for the images warped from the left. We also take only the first output (the homography) as we are not interested in the inlier points for this warping case.

Stitching the images together

With the homographies between the images estimated (using RANSAC) we now need to warp all images to a common image plane. It makes most sense to use the plane of the center image (otherwise the distortions will be huge). One way to do this is to create a very large image, for example filled with zeros, parallel to the central image and warp all the images to it. Since all our images are taken with a horizontal rotation of the camera we can use a simpler procedure, we just pad the central image with zeros to the left or right to make room for the warped images. Add the following function which handles this to *warp.py*.

```
def panorama(H,fromim,toim,padding=2400,delta=2400):
  """ Create horizontal panorama by blending two images
    using a homography H (preferably estimated using RANSAC).
    The result is an image with the same height as toim. 'padding'
    specifies number of fill pixels and 'delta' additional translation. """

  # check if images are grayscale or color
  is_color = len(fromim.shape) == 3

  # homography transformation for geometric_transform()
  def transf(p):
    p2 = dot(H,[p[0],p[1],1])
    return (p2[0]/p2[2],p2[1]/p2[2])

  if H[1,2]<0: # fromim is to the right
    print 'warp - right'
    # transform fromim
    if is_color:
      # pad the destination image with zeros to the right
      toim_t = hstack((toim,zeros((toim.shape[0],padding,3))))
      fromim_t = zeros((toim.shape[0],toim.shape[1]+padding,toim.shape[2]))
      for col in range(3):
        fromim_t[:,:,col] = ndimage.geometric_transform(fromim[:,:,col],
```

```
                    transf,(toim.shape[0],toim.shape[1]+padding))
  else:
    # pad the destination image with zeros to the right
    toim_t = hstack((toim,zeros((toim.shape[0],padding))))
    fromim_t = ndimage.geometric_transform(fromim,transf,
                (toim.shape[0],toim.shape[1]+padding))
else:
  print 'warp - left'
  # add translation to compensate for padding to the left
  H_delta = array([[1,0,0],[0,1,-delta],[0,0,1]])
  H = dot(H,H_delta)
  # transform fromim
  if is_color:
    # pad the destination image with zeros to the left
    toim_t = hstack((zeros((toim.shape[0],padding,3)),toim))
    fromim_t = zeros((toim.shape[0],toim.shape[1]+padding,toim.shape[2]))
    for col in range(3):
      fromim_t[:,:,col] = ndimage.geometric_transform(fromim[:,:,col],
                    transf,(toim.shape[0],toim.shape[1]+padding))
  else:
    # pad the destination image with zeros to the left
    toim_t = hstack((zeros((toim.shape[0],padding)),toim))
    fromim_t = ndimage.geometric_transform(fromim,
                transf,(toim.shape[0],toim.shape[1]+padding))

# blend and return (put fromim above toim)
if is_color:
  # all non black pixels
  alpha = ((fromim_t[:,:,0] * fromim_t[:,:,1] * fromim_t[:,:,2] ) > 0)
  for col in range(3):
    toim_t[:,:,col] = fromim_t[:,:,col]*alpha + toim_t[:,:,col]*(1-alpha)
else:
  alpha = (fromim_t > 0)
  toim_t = fromim_t*alpha + toim_t*(1-alpha)

return toim_t
```

For a general `geometric_transform()` a function, describing the pixel to pixel map, needs to be specified. In this case `transf()` does this by multiplying with H and normalizing the homogeneous coordinates. By checking the translation value in H we can decide if the image should be padded to the left or the right. When the image is padded to the left, the coordinates of the points in the target image changes so in the "left" case a translation is added to the homography. For simplicity we also still use the trick of zero pixels for finding the alpha map.

Now use this function on the images as follows

```
#warp the images
```

Figure 3.12: Horizontal panorama automatically created from SIFT correspondences. (top) the full panorama. (bottom) a crop of the central part.

```
delta = 2000 #for padding and translation

im1 = array(Image.open(imname[1]))
im2 = array(Image.open(imname[2]))
im_12 = warp.panorama(H_12,im1,im2,delta,delta)

im1 = array(Image.open(imname[0]))
im_02 = warp.panorama(dot(H_12,H_01),im1,im_12,delta,delta)

im1 = array(Image.open(imname[3]))
im_32 = warp.panorama(H_32,im1,im_02,delta,delta)

im1 = array(Image.open(imname[j+1]))
im_42 = warp.panorama(dot(H_32,H_43),im1,im_32,delta,2*delta)
```

Note that, in the last line, *im_32* is already translated once. The resulting panorama image is shown in Figure 3.12. As you can see there are effects of different exposure and edge effects at the boundaries between individual images. Commercial panorama software has extra processing to normalize intensity and smooth transitions to make the result look even better.

Exercises

1. Create a function that takes the image coordinates of a square (or rectangular) object, for example a book, a poster or a 2D bar code, and estimates the transform that takes the rectangle to a full on frontal view in a normalized coordinate system. Use `ginput()` or the strongest Harris corners to find the points.

2. Write a function that correctly determines the alpha map for a warp like the one in Figure 3.1.

3. Find a data set of your own that contains three common landmark points (like in the face example or using a famous object like the Eiffel tower). Create aligned images where the landmarks are in the same position. Compute mean and median images and visualize them.

4. Implement intensity normalization and a better way to blend the images in the panorama example to remove the edge effects in Figure 3.12.

5. Instead of warping to a central image, panoramas can be created by warping on to a cylinder. Try this for the example in Figure 3.12.

6. Use RANSAC to find several dominant homography inlier sets. An easy way to do this is to first make one run of RANSAC, find the homograohy with the largest consistent subset, then remove the inliers from the set of matches, then run RANSAC again to get the next biggest set, and so on.

7. Modify the homography RANSAC estimation to instead estimate affine transformations using three point correspondences. Use this to determine if a pair of images contains a planar scene, for example using the inlier count. A planar scene will have a high inlier count for an affine transformation.

8. Build a *panograph* (`http://en.wikipedia.org/wiki/Panography`) from a collection (for example from Flickr) by matching local features and using least-squares rigid registration.

Chapter 4

Camera Models and Augmented Reality

In this chapter we will look at modeling cameras and how to effectively use such models. In the previous chapter we covered image to image mappings and transforms. To handle mappings between 3D and images the projection properties of the camera generating the image needs to be part of the mapping. Here we show how to determine camera properties and how to use image projections for applications like augmented reality. In the next chapter, we will use the camera model to look at applications with multiple views and mappings between them.

4.1 The Pin-hole Camera Model

The *pin-hole camera* model (or sometimes *projective camera* model) is a widely used camera model in computer vision. It is simple and accurate enough for most applications. The name comes from the type of camera, like a camera obscura, that collects light through a small hole to the inside of a dark box or room. In the pin-hole camera model, light passes through a single point, the *camera center*, $\mathbf{C}$, before it is projected onto an *image plane*. Figure 4.1 shows an illustration where the image plane is drawn in front of the camera center. The image plane in an actual camera would be upside down behind the camera center but the model is the same.

The projection properties of a pin-hole camera can be derived from this illustration and the assumption that the image axis are aligned with the x and y axis of a 3D coordinate system. The *optical axis* of the camera then coincides with the z axis and the projection follows from similar triangles. By adding rotation and translation to put a 3D point in this coordinate system before projecting, the complete projection

transform follows. The interested reader can find the details in [13] and [25, 26].

With a pin-hole camera, a 3D point $\mathbf{X}$ is projected to an image point $\mathbf{x}$ (both expressed in homogeneous coordinates) as

$$\lambda \mathbf{x} = P\mathbf{X} \ . \tag{4.1}$$

Here the 3×4 matrix P is called the *camera matrix* (or *projection matrix*). Note that the 3D point $\mathbf{X}$ has four elements in homogeneous coordinates, $\mathbf{X} = [X, Y, Z, W]$. The scalar λ is the *inverse depth* of the 3D point and is needed if we want all coordinates to be homogeneous with the last value normalized to one.

The camera matrix

The camera matrix can be decomposed as

$$P = K \left[R \mid \mathbf{t} \right] \ , \tag{4.2}$$

where R is a rotation matrix describing the orientation of the camera, $\mathbf{t}$ a 3D translation vector describing the position of the camera center, and the intrinsic *calibration matrix* K describing the projection properties of the camera.

The calibration matrix depends only on the camera properties and is in a general form written as

$$K = \begin{bmatrix} \alpha f & s & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix} \ .$$

The *focal length*, f, is the distance between the image plane and the camera center. The skew, s, is only used if the pixel array in the sensor is skewed and can in most cases safely be set to zero. This gives

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \ , \tag{4.3}$$

where we used the alternative notation f_x and f_y, with $f_x = \alpha f_y$.

The *aspect ratio*, α is used for non-square pixel elements. It is often safe to assume $\alpha = 1$. With this assumption the matrix becomes

$$K = \begin{bmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix} \ .$$

Besides the focal length, the only remaining parameters are the coordinates of the *optical center* (sometimes called the *principal point*), the image point $\mathbf{c} = [c_x, c_y]$ where

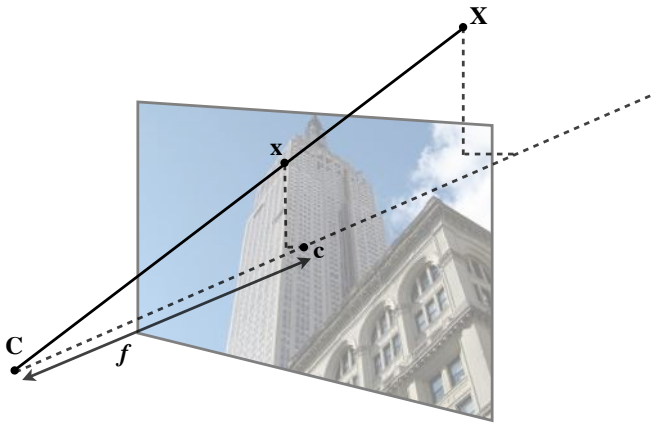


Figure 4.1: The pin-hole camera model. The image point $\mathbf{x}$ is at the intersection of the image plane and the line joining the 3D point $\mathbf{X}$ and the camera center $\mathbf{C}$. The dashed line is the optical axis of the camera.

the optical axis intersects the image plane. Since this is usually in the center of the image and image coordinates are measured from the top left corner, these values are often well approximated with half the width and height of the image. It is worth noting that in this last case the only unknown variable is the focal length f.

Projecting 3D points

Let's create a camera class to handle all operations we need for modeling cameras and projections.

```
from scipy import linalg

class Camera(object):
  """ Class for representing pin-hole cameras. """

  def __init__(self,P):
    """ Initialize P = K[R|t] camera model. """
    self.P = P
    self.K = None # calibration matrix
    self.R = None # rotation
    self.t = None # translation
    self.c = None # camera center
```

```
  def project(self,X):
    """ Project points in X (4*n array) and normalize coordinates. """

    x = dot(self.P,X)
    for i in range(3):
      x[i] /= x[2]
    return x
```

The example below shows how to project 3D points into an image view. In this example we will use one of the Oxford multi-view datasets, the "Model House" data set, available at `http://www.robots.ox.ac.uk/~vgg/data/data-mview.html`. Download the 3D geometry file and copy the "house.p3d" file to your working directory.

```
import camera

# load points
points = loadtxt('house.p3d').T
points = vstack((points,ones(points.shape[1])))

# setup camera
P = hstack((eye(3),array([[0],[0],[-10]])))
cam = camera.Camera(P)
x = cam.project(points)

# plot projection
figure()
plot(x[0],x[1],'k.')
show()
```

First we make the points into homogeneous coordinates and create a `Camera` object with a projection matrix before projection the 3D points and plotting them. The result looks like the middle plot in Figure 4.2.

To see how moving the camera changes the projection, try the following piece of code that incrementally rotates the camera around a random 3D axis.

```
# create transformation
r = 0.05*random.rand(3)
rot = camera.rotation_matrix(r)

# rotate camera and project
figure()
for t in range(20):
  cam.P = dot(cam.P,rot)
  x = cam.project(points)
  plot(x[0],x[1],'k.')
show()
```

Figure 4.2: An example of projecting 3D points. (left) sample image. (middle) projected points into a view. (right) trajectory of projected points under camera rotation. Data from the Oxford "Model House" dataset.

Here we used the helper function `rotation_matrix()` which creates a rotation matrix for 3D rotations around a vector (add this to *camera.py*).

```
def rotation_matrix(a):
  """ Creates a 3D rotation matrix for rotation
    around the axis of the vector a. """
  R = eye(4)
  R[:3,:3] = linalg.expm([[0,-a[2],a[1]],[a[2],0,-a[0]],[-a[1],a[0],0]])
  return R
```

Figure 4.2 shows one of the images from the sequence, a projection of the 3D points and the projected 3D point tracks after the points have been rotated around a random vector. Try this example a few times with different random rotations and you will get a feel for how the points rotate from the projections.

Factoring the camera matrix

If we are given a camera matrix P of the form in equation (4.2), we need to be able to recover the internal parameters K and the camera position and pose $\mathbf{t}$ and R. Partitioning the matrix is called *factorization*. In this case we will use a type of matrix factorization called *RQ-factorization*.

Add the following method to the *Camera* class.

```
def factor(self):
  """ Factorize the camera matrix into K,R,t as P = K[R|t]. """

  # factor first 3*3 part
  K,R = linalg.rq(self.P[:,:3])

  # make diagonal of K positive
```

```
  T = diag(sign(diag(K)))
  if linalg.det(T) < 0:
    T[1,1] *= -1

  self.K = dot(K,T)
  self.R = dot(T,R) # T is its own inverse
  self.t = dot(linalg.inv(self.K),self.P[:,3])

  return self.K, self.R, self.t
```

RQ-factorization is not unique, there is a sign ambiguity in the factorization. Since we need the rotation matrix R to have positive determinant (otherwise coordinate axis can get flipped) we can add a transform T to change the sign when needed.

Try this on a sample camera to see that it works:

```
import camera

K = array([[1000,0,500],[0,1000,300],[0,0,1]])
tmp = camera.rotation_matrix([0,0,1])[:3,:3]
Rt = hstack((tmp,array([[50],[40],[30]])))
cam = camera.Camera(dot(K,Rt))

print K,Rt
print cam.factor()
```

You should get the same printout in the console.

Computing the camera center

Given a camera projection matrix, P, it is useful to be able to compute the camera's position in space. The camera center, $\mathbf{C}$ is a 3D point with the property $P\mathbf{C} = 0$. For a camera with $P = K\,[R \mid \mathbf{t}]$ this gives

$$K[R \mid \mathbf{t}]\mathbf{C} = KR\mathbf{C} + K\mathbf{t} = 0 \ ,$$

and the camera center can be computed as

$$\mathbf{C} = -R^T\mathbf{t} \ .$$

Note that the camera center is independent of the intrinsic calibration K, as expected.

Add the following method for computing the camera center according to the formula above and/or returning the camera center to the `Camera` class.

```
def center(self):
  """ Compute and return the camera center. """
```

```
  if self.c is not None:
    return self.c
  else:
    # compute c by factoring
    self.factor()
    self.c = -dot(self.R.T,self.t)
    return self.c
```

This concludes the basic functions of our `Camera` class. Now, let's see how to work with this pin-hole camera model.

4.2 Camera Calibration

Calibrating a camera means determining the internal camera parameters, in our case the matrix K. It is possible to extend this camera model to include radial distortion and other artifacts if your application needs precise measurements. For most applications however, the simple model in equation (4.3) is good enough. The standard way to calibrate cameras is to take lots of pictures of a flat checkerboard pattern. For example, the calibration tools in OpenCV use this approach, see [3] for details.

A simple calibration method

Here we will look at a simple calibration method. Since most of the parameters can be set using basic assumptions (square straight pixels, optical center at the center of the image) the tricky part is getting the focal length right. For this calibration method you need a flat rectangular calibration object (a book will do), measuring tape or a ruler and preferable a flat surface. Here's what to do:

- Measure the sides of your rectangular calibration object. Let's call these dX and dY.
- Place the camera and the calibration object on a flat surface so that the camera back and calibration object are parallel and the object is roughly in the center of the camera's view. You might have to raise the camera or object to get a nice alignment.
- Measure the distance from the camera to the calibration object. Let's call this dZ.
- Take a picture and check that the setup is straight, meaning that the sides of the calibration object align with the rows and columns of the image.

- Measure the width and height of the object in pixels. Let's call these dx and dy.

See Figure 4.3 for an example of a setup. Now, using similar triangles (look at Figure 4.1 to convince yourself that) the following relation gives the focal lengths:

$$f_x = \frac{dx}{dX}dZ \quad , \quad f_y = \frac{dy}{dY}dZ \quad .$$

For the particular setup in Figure 4.3, the object was measured to be 130 by 185 mm, so $dX = 130$ and $dY = 185$. The distance from camera to object was 460 mm, so $dZ = 460$. You can use any unit of measurement, it doesn't matter, only the ratios of the measurements matter. Using `ginput()` to select four points in the image, the width and height in pixels was 722 and 1040. This means that $dx = 722$ and $dy = 1040$. Putting these values in the relationship above gives

$$f_x = 2555 \quad , \quad f_y = 2586 \quad .$$

Now, it is important to note that *this is for a particular image resolution*. In this case the image was 2592×1936 pixels. Remember that the focal length and the optical center are measured in pixels and scale with the image resolution. If you take other image resolutions (for example a thumbnail image) the values will change. It is convenient to add the constants of your camera to a helper function like this

```
def my_calibration(sz):
  row,col = sz
  fx = 2555*col/2592
  fy = 2586*row/1936
  K = diag([fx,fy,1])
  K[0,2] = 0.5*col
  K[1,2] = 0.5*row
  return K
```

This function then takes a size tuple and returns the calibration matrix. Here we assume the optical center to be the center of the image. Go ahead and replace the focal lengths with their mean if you like, for most consumer type cameras this is fine. Note that the calibration is for images in landscape orientation. For portrait orientation, you need to interchange the constants. Let's keep this function and make use of it in the next section.

4.3 Pose Estimation from Planes and Markers

In Chapter 3 we saw how to estimate homographies between planes. Combining this with a calibrated camera makes it possible to compute the camera's pose (rotation and

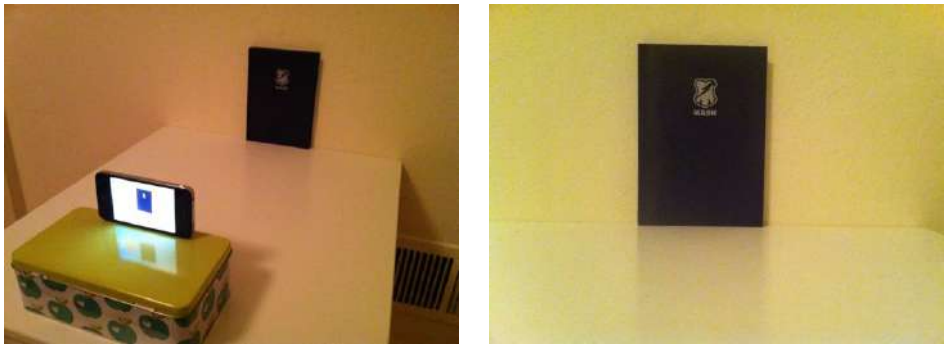


Figure 4.3: A simple camera calibration setup. (left) an image of the setup used. (right) the image used for the calibration. Measuring the width and height of the calibration object in the image and the physical dimensions of the setup is enough to determine the focal length.

translation) if the image contains a planar marker object. This is marker object can be almost any flat object.

Let's illustrate with an example. Consider the two top images in Figure 4.4, the following code will extract SIFT features in both images and robustly estimate a homography using RANSAC.

```
import homography
import camera
import sift

# compute features
sift.process_image('book_frontal.JPG','im0.sift')
l0,d0 = sift.read_features_from_file('im0.sift')

sift.process_image('book_perspective.JPG','im1.sift')
l1,d1 = sift.read_features_from_file('im1.sift')

# match features and estimate homography
matches = sift.match_twosided(d0,d1)
ndx = matches.nonzero()[0]
fp = homography.make_homog(l0[ndx,:2].T)
ndx2 = [int(matches[i]) for i in ndx]
tp = homography.make_homog(l1[ndx2,:2].T)

model = homography.RansacModel()
H = homography.H_from_ransac(fp,tp,model)
```

Now we have a homography that maps points on the marker (in this case the book)

in one image to their corresponding locations in the other image. Let's define our 3D coordinate system so that the marker lies in the X-Y plane ($Z = 0$) with the origin somewhere on the marker.

To check our results we will need some simple 3D object placed on the marker. Here we will use a cube and generate the cube points using the function:

```
def cube_points(c,wid):
  """ Creates a list of points for plotting
    a cube with plot. (the first 5 points are
    the bottom square, some sides repeated). """
  p = []
  #bottom
  p.append([c[0]-wid,c[1]-wid,c[2]-wid])
  p.append([c[0]-wid,c[1]+wid,c[2]-wid])
  p.append([c[0]+wid,c[1]+wid,c[2]-wid])
  p.append([c[0]+wid,c[1]-wid,c[2]-wid])
  p.append([c[0]-wid,c[1]-wid,c[2]-wid]) #same as first to close plot

  #top
  p.append([c[0]-wid,c[1]-wid,c[2]+wid])
  p.append([c[0]-wid,c[1]+wid,c[2]+wid])
  p.append([c[0]+wid,c[1]+wid,c[2]+wid])
  p.append([c[0]+wid,c[1]-wid,c[2]+wid])
  p.append([c[0]-wid,c[1]-wid,c[2]+wid]) #same as first to close plot

  #vertical sides
  p.append([c[0]-wid,c[1]-wid,c[2]+wid])
  p.append([c[0]-wid,c[1]+wid,c[2]+wid])
  p.append([c[0]-wid,c[1]+wid,c[2]-wid])
  p.append([c[0]+wid,c[1]+wid,c[2]-wid])
  p.append([c[0]+wid,c[1]+wid,c[2]+wid])
  p.append([c[0]+wid,c[1]-wid,c[2]+wid])
  p.append([c[0]+wid,c[1]-wid,c[2]-wid])

  return array(p).T
```

Some points are reoccurring so that `plot()` will generate a nice looking cube.

With a homography and a camera calibration matrix, we can now determine the relative transformation between the two views.

```
# camera calibration
K = my_calibration((747,1000))

# 3D points at plane z=0 with sides of length 0.2
box = cube_points([0,0,0.1],0.1)

# project bottom square in first image
```

```
cam1 = camera.Camera( hstack((K,dot(K,array([[0],[0],[-1]])) )) )
# first points are the bottom square
box_cam1 = cam1.project(homography.make_homog(box[:,:5]))

# use H to transfer points to the second image
box_trans = homography.normalize(dot(H,box_cam1))

# compute second camera matrix from cam1 and H
cam2 = camera.Camera(dot(H,cam1.P))
A = dot(linalg.inv(K),cam2.P[:,:3])
A = array([A[:,0],A[:,1],cross(A[:,0],A[:,1])]).T
cam2.P[:,:3] = dot(K,A)

# project with the second camera
box_cam2 = cam2.project(homography.make_homog(box))


# test: projecting point on z=0 should give the same
point = array([1,1,0,1]).T
print homography.normalize(dot(dot(H,cam1.P),point))
print cam2.project(point)
```

Here we use a version of the image with resolution 747×1000 and first generate the calibration matrix for that image size. Next points for a cube at the origin is created. The first five points generated by `cube_points()` correspond to the bottom, which in this case will lie on the plane defined by $Z = 0$, the plane of the marker. The first image (top left in Figure 4.4) is roughly a straight frontal view of the book and will be used as our template image. Since the scale of the scene coordinates is arbitrary, we create a first camera with matrix

$$P_1 = K \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 \end{bmatrix} ,$$

which has coordinate axis aligned with the camera and placed above the marker. The first five 3D points are projected onto the image. With the estimated homography we can transform these to the second image. Plotting them should show the corners at the same marker locations (see top right in Figure 4.4).

Now, composing P_1 with H as a camera matrix for the second image,

$$P_2 = HP_1 ,$$

will transform points on the marker plane $Z = 0$ correctly. This means that the first two columns and the fourth column of P_2 are correct. Since we know that the first 3×3 block should be KR and R is a rotation matrix, we can recover the third column

by multiplying P_2 with the inverse of the calibration matrix and replacing the third column with the cross product of the first two.

As a sanity check we can project a point on the marker plane with the new matrix and check that it gives the same as the same point transformed with the first camera and the homography. You should get the same printout in your console.

Visualizing the projected points can be done like this.

```
im0 = array(Image.open('book_frontal.JPG'))
im1 = array(Image.open('book_perspective.JPG'))

# 2D projection of bottom square
figure()
imshow(im0)
plot(box_cam1[0,:],box_cam1[1,:],linewidth=3)

# 2D projection transferred with H
figure()
imshow(im1)
plot(box_trans[0,:],box_trans[1,:],linewidth=3)

# 3D cube
figure()
imshow(im1)
plot(box_cam2[0,:],box_cam2[1,:],linewidth=3)

show()
```

This should give three figures like the images in Figure 4.4. To be able to reuse these computations for future examples, we can save the camera matrices using Pickle.

```
import pickle

with open('ar_camera.pkl','w') as f:
  pickle.dump(K,f)
  pickle.dump(dot(linalg.inv(K),cam2.P),f)
```

Now we have seen how to compute the camera matrix given a planar scene object. We combined feature matching with homographies and camera calibration to produce a simple example of placing a cube in an image. With camera pose estimation, we now have the building blocks in place for creating simple augmented reality applications.

4.4 Augmented Reality

Augmented reality (*AR*) is a collective term for placing objects and information on top of image data. The classic example is placing a 3D computer graphics model so that

Figure 4.4: Example of computing the projection matrix for a new view using a planar object as marker. Matching image features to an aligned marker gives a homography that can be used to compute the pose of the camera. (top left) template image with a blue square. (top right) an image taken from an unknown viewpoint with the same square transformed with the estimated homography. (bottom) a cube transformed using the estimated camera matrix.

it looks like it belongs in the scene, and moves naturally with the camera motion in the case of video. Given an image with a marker plane as in the section above, we can compute the camera's position and pose and use that to place computer graphics models so that they are rendered correctly. In this last section of our camera chapter we will show how to build a simple AR example. We will use two tools for this, PyGame and PyOpenGL.

PyGame and PyOpenGL

PyGame is a popular package for game development that easily handles display windows, input devices, events and much more. PyGame is open source and available from `http://www.pygame.org/`. It is actually a Python binding for the SDL game engine. For installation instructions, see the Appendix. For more details on programming with PyGame, see for example [21].

PyOpenGL is the Python binding to the OpenGL graphics programming interface. OpenGL comes pre-installed on almost all systems and is a crucial part for graphics performance. OpenGL is cross platform and works the same across operating systems. Take a look at `http://www.opengl.org/` for more information on OpenGL. The getting started page (`http://www.opengl.org/wiki/Getting_started`) has resources for beginners. PyOpenGL is open source and easy to install, see the Appendix for details. More information can be found on the project website `http://pyopengl.sourceforge.net/`.

There is no way we can cover any significant portion of OpenGL programming, we will instead just show the important parts, for example how to use camera matrices in OpenGL and setting up a basic 3D model. Some good examples and demos are available in the PyOpenGL-Demo package (`http://pypi.python.org/pypi/PyOpenGL-Demo`). This is a good place to start if you are new to PyOpenGL.

We want to place a 3D model in a scene using OpenGL. To use PyGame and PyOpenGL for this application we need to import the following at the top of our scripts:

```
from OpenGL.GL import *
from OpenGL.GLU import *
import pygame, pygame.image
from pygame.locals import *
```

As you can see we need two main parts from OpenGL. The GL part contains all functions stating with "gl", which you will see are most of the ones we need. The GLU part is the OpenGL Utility library and contains some higher-level functionality. We will mainly use it to set up the camera projection. The `pygame` part sets up the window and event controls, and `pygame.image` is used for loading image and creating OpenGL textures. The `pygame.locals` is needed for setting up the display area for OpenGL.

The two main components of setting up an OpenGL scene are the projection and model view matrices. Let's get started and see how to create these matrices from our pin-hole cameras.

From camera matrix to OpenGL format

OpenGL uses 4×4 matrices to represent transforms (both 3D transforms and projections). This is only slightly different from our use of 3×4 camera matrices. However, the camera-scene transformations are separated in two matrices, the *GL_PROJECTION* matrix and the *GL_MODELVIEW* matrix. GL_PROJECTION handles the image formation properties and is the equivalent of our internal calibration matrix K. GL_MODELVIEW handles the 3D transformation of the relation between the objects and the camera. This corresponds roughly to the R and t part of our camera matrix. One difference is that the coordinate system is assumed to be centered at the camera so the GL_MODELVIEW matrix actually contains the transformation that places the objects in front of the camera. There are many peculiarities with working in OpenGL, we will comment on them as they are encountered in the examples below.

Given that we have a camera calibrated so that the calibration matrix K is known, the following function translates the camera properties to an OpenGL projection matrix.

```
def set_projection_from_camera(K):
  """ Set view from a camera calibration matrix. """

  glMatrixMode(GL_PROJECTION)
  glLoadIdentity()

  fx = K[0,0]
  fy = K[1,1]
  fovy = 2*arctan(0.5*height/fy)*180/pi
  aspect = (width*fy)/(height*fx)

  # define the near and far clipping planes
  near = 0.1
  far = 100.0

  # set perspective
  gluPerspective(fovy,aspect,near,far)
  glViewport(0,0,width,height)
```

We assume the calibration to be of the simpler form in (4.3) with the optical center at the image center. The first function `glMatrixMode()` sets the working ma-

trix to GL_PROJECTION and subsequent commands will modify this matrix[1]. Then `glLoadIdentity()` sets the matrix to the identity matrix, basically reseting any prior changes. We then calculate the vertical field of view in degrees with the help of the image height and the camera's focal length as well as the aspect ratio. An OpenGL projection also has a near and far clipping plane to limit the depth range of what is rendered. We just set the near depth to be small enough to contain the nearest object and the far depth to some large number. We use the GLU utility function `gluPerspective()` to set the projection matrix and define the whole image to be the view port (essentially what is to be shown). There is also an option to load a full projection matrix with `glLoadMatrixf()` similar to the model view function below. This is useful when the simple version of the calibration matrix is not good enough.

The model view matrix should encode the relative rotation and translation that brings the object in front of the camera (as if the camera was at the origin). It is a 4×4 matrix that typically looks like this

$$\begin{bmatrix} R & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix} ,$$

where R is a rotation matrix with columns equal to the direction of the three coordinate axis and $\mathbf{t}$ is a translation vector. When creating a model view matrix the rotation part will need to hold all rotations (object and coordinate system) by multiplying together the individual components.

The following function shows how to take a 3×4 pin-hole camera matrix with the calibration removed (multiply P with K^{-1}) and create a model view.

```
def set_modelview_from_camera(Rt):
  """ Set the model view matrix from camera pose. """

  glMatrixMode(GL_MODELVIEW)
  glLoadIdentity()

  # rotate teapot 90 deg around x-axis so that z-axis is up
  Rx = array([[1,0,0],[0,0,-1],[0,1,0]])

  # set rotation to best approximation
  R = Rt[:,:3]
  U,S,V = linalg.svd(R)
  R = dot(U,V)
  R[0,:] = -R[0,:] # change sign of x-axis

  # set translation
```

[1]This is an odd way to handle things, but there are only two matrices to switch between, GL_PROJECTION and GL_MODELVIEW, so it is manageable.

```
  t = Rt[:,3]

  # setup 4*4 model view matrix
  M = eye(4)
  M[:3,:3] = dot(R,Rx)
  M[:3,3] = t

  # transpose and flatten to get column order
  M = M.T
  m = M.flatten()

  # replace model view with the new matrix
  glLoadMatrixf(m)
```

First we switch to work on the GL_MODELVIEW matrix and reset it. Then we create a 90 degree rotation matrix since the object we want to place needs to be rotated (you will see below). Then we make sure that the rotation part of the camera matrix is indeed a rotation matrix in case there are errors or noise when we estimated the camera matrix. This is done with SVD and the best rotation matrix approximation is given by $R = UV^T$. The OpenGL coordinate system is a little different so we flip the x-axis around. Then we set the model view matrix M by multiplying the rotations. The function `glLoadMatrixf()` sets the model view matrix and takes an array of the 16 values of the matrix taken *column-wise*. Transposing and then flattening accomplishes this.

Placing virtual objects in the image

The first thing we need to do is to add the image (the one we want to place virtual objects in) as a background. In OpenGL this is done by creating a quadrilateral, a *quad*, that fills the whole view. The easiest way to do this is to draw the quad with the projection and model view matrices reset so that the coordinates go from -1 to 1 in each dimension.

This function loads an image, converts it to an OpenGL texture and places that texture on the quad.

```
def draw_background(imname):
  """ Draw background image using a quad. """

  # load background image (should be .bmp) to OpenGL texture
  bg_image = pygame.image.load(imname).convert()
  bg_data = pygame.image.tostring(bg_image,"RGBX",1)

  glMatrixMode(GL_MODELVIEW)
  glLoadIdentity()
```

```
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)

  # bind the texture
  glEnable(GL_TEXTURE_2D)
  glBindTexture(GL_TEXTURE_2D,glGenTextures(1))
  glTexImage2D(GL_TEXTURE_2D,0,GL_RGBA,width,height,0,GL_RGBA,GL_UNSIGNED_BYTE,bg_data)
  glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST)
  glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST)

  # create quad to fill the whole window
  glBegin(GL_QUADS)
  glTexCoord2f(0.0,0.0); glVertex3f(-1.0,-1.0,-1.0)
  glTexCoord2f(1.0,0.0); glVertex3f( 1.0,-1.0,-1.0)
  glTexCoord2f(1.0,1.0); glVertex3f( 1.0, 1.0,-1.0)
  glTexCoord2f(0.0,1.0); glVertex3f(-1.0, 1.0,-1.0)
  glEnd()

  # clear the texture
  glDeleteTextures(1)
```

This function first uses some PyGame functions to load an image and serialize it to a raw string representation that can be used by PyOpenGL. Then we reset the model view and clear the color and depth buffer. Next we bind the texture so that we can use it for the quad and specify interpolation. The quad is defined with corners at -1 and 1 in both dimensions. Note that the coordinates in the texture image goes from 0 to 1. Finally, we clear the texture so it doesn't interfere with what we want to draw later.

Now we are ready to place objects in the scene. We will use the "hello world" computer graphics example, the Utah teapot `http://en.wikipedia.org/wiki/Utah_teapot`. This teapot has a rich history and is available as one of the standard shapes in GLUT:

```
from OpenGL.GLUT import *
glutSolidTeapot(size)
```

This generates a solid teapot model of relative size *size*.

The following function will set up the color and properties to make a pretty red teapot.

```
def draw_teapot(size):
  """ Draw a red teapot at the origin. """
  glEnable(GL_LIGHTING)
  glEnable(GL_LIGHT0)
  glEnable(GL_DEPTH_TEST)
  glClear(GL_DEPTH_BUFFER_BIT)

  # draw red teapot
```

```
  glMaterialfv(GL_FRONT,GL_AMBIENT,[0,0,0,0])
  glMaterialfv(GL_FRONT,GL_DIFFUSE,[0.5,0.0,0.0,0.0])
  glMaterialfv(GL_FRONT,GL_SPECULAR,[0.7,0.6,0.6,0.0])
  glMaterialf(GL_FRONT,GL_SHININESS,0.25*128.0)
  glutSolidTeapot(size)
```

The first two lines enable lighting and a light. Lights are numbered as GL_LIGHT0, GL_LIGHT1, etc. We will only use one light in this example. The `glEnable()` function is used to turn on OpenGL features. These are defined with uppercase constants. Turning off a feature is done with the corresponding function `glDisable()`. Next depth testing is turned on so that objects are rendered according to their depth (so that far away objects are not drawn in front of near objects) and the depth buffer is cleared. Next the material properties of the object, such as the diffuse and specular colors, are specified. The last line adds a solid Utah teapot with the specified material properties.

Tying it all together

The full script for generating an image like the one in Figure 4.5 looks like this (assuming that you also have the functions introduced above in the same file).

```
from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
import pygame, pygame.image
from pygame.locals import *
import pickle

width,height = 1000,747

def setup():
  """ Setup window and pygame environment. """
  pygame.init()
  pygame.display.set_mode((width,height),OPENGL | DOUBLEBUF)
  pygame.display.set_caption('OpenGL AR demo')

# load camera data
with open('ar_camera.pkl','r') as f:
  K = pickle.load(f)
  Rt = pickle.load(f)

setup()
draw_background('book_perspective.bmp')
set_projection_from_camera(K)
set_modelview_from_camera(Rt)
draw_teapot(0.02)
```

```
while True:
  event = pygame.event.poll()
  if event.type in (QUIT,KEYDOWN):
    break
  pygame.display.flip()
```

First this script loads the camera calibration matrix and the rotation and translation part of the camera matrix using Pickle. This assumes that you saved them as described on page 114. The `setup()` function initializes PyGame, sets the window to the size of the image and makes the drawing area a double buffer OpenGL window. Next the background image is loaded and placed to fit the window. The camera and model view matrices are set and finally the teapot is drawn at the correct position.

Events in PyGame are handled using infinite loops with regular polling for any changes. These can be keyboard, mouse or other events. In this case we check if the application was quit or if a key was pressed and exit the loop. The command `pygame.display.flip()` draws the objects on the screen.

The result should look like Figure 4.5. As you can see, the orientation is correct (the teapot is aligned with the sides of the cube in Figure 4.4). To check that the placement is correct, you can try to make the teapot really small by passing a smaller value for the *size* variable. The teapot should be placed close to the $[0, 0, 0]$ corner of the cube in Figure 4.4. An example is shown in Figure 4.5.

Loading models

Before we end this chapter, we will touch upon one last detail; loading 3D models and displaying them. The PyGame cookbook has a script for loading models in .obj format available at `http://www.pygame.org/wiki/OBJFileLoader`. You can learn more about the .obj format and the corresponding material file format at `http://en.wikipedia.org/wiki/Wavefront_.obj_file`.

Let's see how to use that with a basic example. We will use a freely available toy plane model from `http://www.oyonale.com/modeles.php`[2]. Download the .obj version and save it as *toyplane.obj*. You can of course replace this model with any model of your choice, the code below will be the same.

Assuming that you downloaded the file as *objloader.py*, add the following function to the file you used for the teapot example above.

```
def load_and_draw_model(filename):
  """ Loads a model from an .obj file using objloader.py.
    Assumes there is a .mtl material file with the same name. """
  glEnable(GL_LIGHTING)
```

[2]Models courtesy of Gilles Tran (Creative Commons License By Attribution).

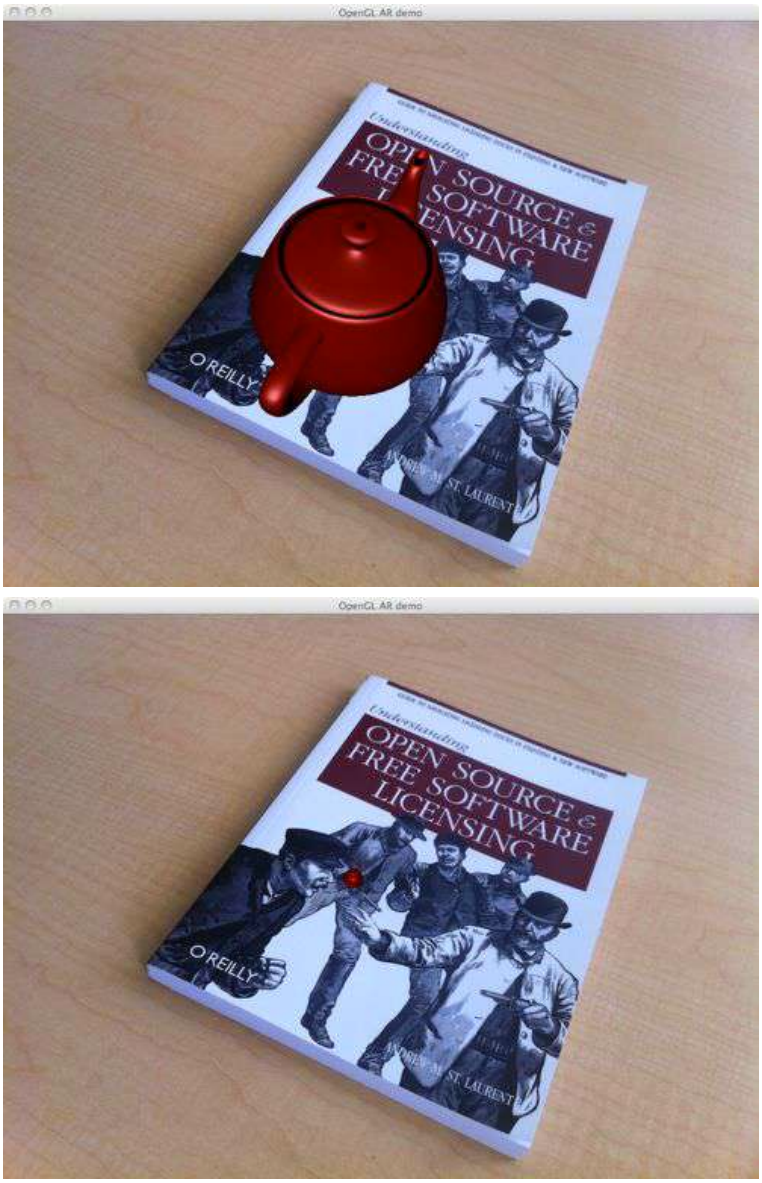


Figure 4.5: Augmented reality. Placing a computer graphics model an a book in a scene using camera parameters computed from feature matches. (top) the Utah teapot rendered in place aligned with the coordinate axis. (bottom) sanity check to see the position of the origin.

```
  glEnable(GL_LIGHT0)
  glEnable(GL_DEPTH_TEST)
  glClear(GL_DEPTH_BUFFER_BIT)

  # set model color
  glMaterialfv(GL_FRONT,GL_AMBIENT,[0,0,0,0])
  glMaterialfv(GL_FRONT,GL_DIFFUSE,[0.5,0.75,1.0,0.0])
  glMaterialf(GL_FRONT,GL_SHININESS,0.25*128.0)

  # load from a file
  import objloader
  obj = objloader.OBJ(filename,swapyz=True)
  glCallList(obj.gl_list)
```

Same as before, we set the lighting and the color properties of the model. Next we load a model file into an `OBJ` object and execute the OpenGL calls from the file.

You can set the texture and material properties in a corresponding .mtl file. The `objloader` module actually requires a material file. Rather than modifying the loading script, we take the pragmatic approach of just creating a tiny material file. In this case we'll just specify the color.

Create a file *toyplane.mtl* with the following lines.

```
newmtl lightblue
Kd  0.5  0.75  1.0
illum 1
```

This sets the diffuse color of the object to a light grayish blue. Now, make sure to replace the "usemtl" tag in your .obj file to

```
usemtl lightblue
```

Adding textures we leave to the exercises. Replacing the call to `draw_teapot()` in the example above with

```
load_and_draw_model('toyplane.obj')
```

should generate a window like the one shown in Figure 4.6.

This is as deep as we will go into augmented reality and OpenGL in this book. With the recipe for calibrating cameras, computing camera pose, translating the cameras into OpenGL format and rendering models in the scene, the groundwork is laid for you to continue exploring augmented reality. In the next chapter we will continue with the camera model and compute 3D structure and camera pose without the use of markers.

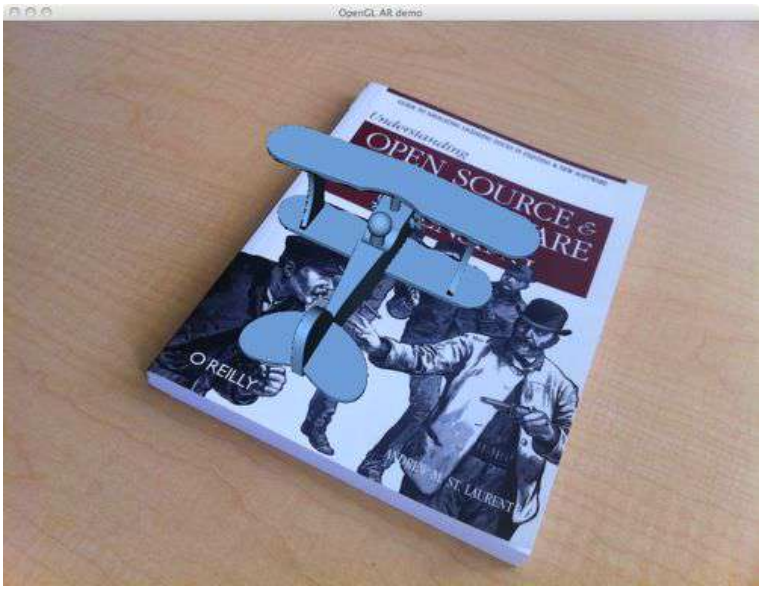


Figure 4.6: Loading a 3D model from an .obj file and placing it on a book in a scene using camera parameters computed from feature matches.

Exercises

1. Modify the example code for the motion in Figure 4.2 to transform the points instead of the camera. You should get the same plot. Experiment with different transformations and plot the results.

2. Some of the Oxford multi-view datasets have camera matrices given. Compute the camera positions for one of the sets an plot the camera path. Does it match with what you are seeing in the images?

3. Take some images of a scene with a planar marker or object. Match features to a full frontal image to compute the pose of each image's camera location. Plot the camera trajectory and the plane of the marker. Add the feature points if you like.

4. In our augmented reality example we assumed the object to be placed at the origin and applied only the cameras position to the model view matrix. Modify the example to place several objects at different locations by adding the object transformation to the matrix. For example, place a grid of teapots on the marker.

5. Take a look at the online documentation for .obj model files and see how to use textured models. Find a model (or create your own) and add it to the scene.

Chapter 5

Multiple View Geometry

This chapter will show you how to handle multiple views and how to use the geometric relationships between them to recover camera positions and 3D structure. With images taken at different view points it is possible to compute 3D scene points as well as camera locations from feature matches. We introduce the necessary tools and show a complete 3D reconstruction example. The last part of the chapter shows how to compute dense depth reconstructions from stereo images.

5.1 Epipolar Geometry

Multiple view geometry is the field studying the relationship between cameras and features when there are correspondences between many images that are taken from varying viewpoints. The image features are usually interest points and we will focus on that case throughout this chapter. The most important constellation is two-view geometry.

With two views of a scene and corresponding points in these views there are geometric constraints on the image points as a result of the relative orientation of the cameras, the properties of the cameras, and the position of the 3D points. These geometric relationships are described by what is called *epipolar geometry*. This section will give a very short description of the basic components we need. For more details on the subject see [13].

Without any prior knowledge of the cameras, there is an inherent ambiguity in that a 3D point, $\mathbf{X}$, transformed with an arbitrary (4×4) homography H as $H\mathbf{X}$ will have the same image point in a camera PH^{-1} as the original point in the camera P. Expressed with the camera equation, this is

$$\lambda\mathbf{x} = P\mathbf{X} = PH^{-1}H\mathbf{X} = \hat{P}\hat{\mathbf{X}} \ .$$

Because of this ambiguity, when analyzing two view geometry we can always transform the cameras with a homography to simplify matters. Often this homography is just a rigid transformation to change the coordinate system. A good choice is to set the origin and coordinate axis to align with the first camera so that

$$P_1 = K_1[I \mid 0] \text{ and } P_2 = K_2[R \mid \mathbf{t}] \ .$$

Here we use the same notation as in Chapter 4; K_1 and K_2 are the calibration matrices, R is the rotation of the second camera, and $\mathbf{t}$ is the translation of the second camera. Using these camera matrices one can derive a condition for the projection of a point $\mathbf{X}$ to image points $\mathbf{x}_1$ and $\mathbf{x}_2$ (with P_1 and P_2 respectively). This condition is what makes it possible to recover the camera matrices from corresponding image points.

The following equation must be satisfied

$$\mathbf{x}_2^T \ F \ \mathbf{x}_1 = 0 \ , \tag{5.1}$$

where

$$F = K_2^{-T} \ S_{\mathbf{t}} R \ K_1^{-1}$$

and the matrix $S_{\mathbf{t}}$ is the skew symmetric matrix

$$S_{\mathbf{t}} = \begin{bmatrix} 0 & -t_3 & t_2 \\ t_3 & 0 & -t_1 \\ -t_2 & t_1 & 0 \end{bmatrix} \ . \tag{5.2}$$

Equation (5.1) is called the *epipolar constraint*. The matrix F in the epipolar constraint is called the *fundamental matrix* and as you can see, it is expressed in components of the two camera matrices (their relative rotation R and translation $\mathbf{t}$). The fundamental matrix has rank 2 and $\det(F) = 0$. This will be used in algorithms for estimating F. The fundamental matrix makes it possible to compute the camera matrices and then a 3D reconstruction.

The equations above mean that the camera matrices can be recovered from F, which in turn can be computed from point correspondences as we will see later. Without knowing the internal calibration (K_1 and K_2) the camera matrices are only recoverable up to a projective transformation. With known calibration, the reconstruction will be metric. A *metric reconstruction* is a 3D reconstruction that correctly represents distances and angles[1].

There is one final piece of geometry needed before we can proceed to actually using this theory on some image data. Given a point in one of the images, for example $\mathbf{x}_2$ in the second view, equation (5.1) defines a line in the first image since

$$\mathbf{x}_2^T \ F \ \mathbf{x}_1 = \mathbf{l}_1^T \mathbf{x}_1 = 0 \ .$$

[1]The absolute scale of the reconstruction cannot be recovered but that is rarely a problem.

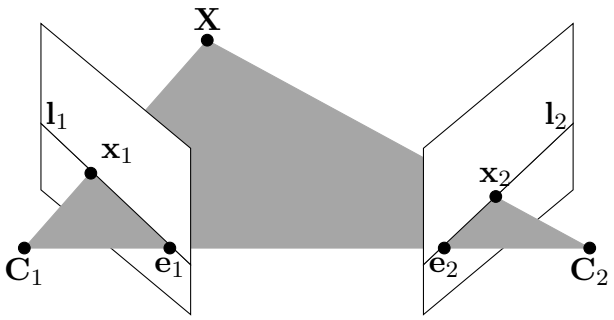


Figure 5.1: An illustration of epipolar geometry. A 3D point $\mathbf{X}$ is projected to $\mathbf{x}_1$ and $\mathbf{x}_2$, in the two views respectively. The baseline between the two camera centers, $\mathbf{C}_1$ and $\mathbf{C}_2$, intersect the image planes in the epipoles, $\mathbf{e}_1$ and $\mathbf{e}_2$. The lines $\mathbf{l}_1$ and $\mathbf{l}_2$ are called epipolar lines.

The equation $\mathbf{l}_1^T\mathbf{x}_1 = 0$ determines a line with all points $\mathbf{x}_1$ in the first image satisfying the equation belonging to the line. This line is called an *epipolar line* corresponding to the point $\mathbf{x}_2$. This means that a corresponding point to $\mathbf{x}_2$ must lie on this line. The fundamental matrix can therefore help the search for correspondences by restricting the search to this line.

The epipolar lines all meet in a point, $\mathbf{e}$, called the *epipole*. The epipole is actually the image point corresponding to the projection of the other camera center. This point can be outside the actual image, depending on the relative orientation of the cameras. Since the epipole lies on all epipolar lines it must satisfy $F\mathbf{e}_1 = 0$. It can therefore be computed as the null vector of F as we will see later. The other epipole can be computed from the relation $\mathbf{e}_2^T F = 0$.

A sample data set

In the coming sections we will need a data set with image points, 3D points and camera matrices to experiment with and illustrate the algorithms. We will use one of the sets from the Oxford multi-view datasets available at `http://www.robots.ox.ac.uk/~vgg/data/data-mview.html`. Download the zipped file for the Merton1 data. The following script will load all the data for you.

```
import camera

# load some images
im1 = array(Image.open('images/001.jpg'))
im2 = array(Image.open('images/002.jpg'))
```

```
# load 2D points for each view to a list
points2D = [loadtxt('2D/00'+str(i+1)+'.corners').T for i in range(3)]

# load 3D points
points3D = loadtxt('3D/p3d').T

# load correspondences
corr = genfromtxt('2D/nview-corners',dtype='int',missing='*')

# load cameras to a list of Camera objects
P = [camera.Camera(loadtxt('2D/00'+str(i+1)+'.P')) for i in range(3)]
```

This will load the first two images (out of three), all the image feature points[2] for the three views, the reconstructed 3D points corresponding to the image points, which points correspond across views and finally the camera matrices (where we used the Camera class from the previous chapter). Here we used `loadtxt()` to read the text files to `NumPy` arrays. The correspondences contain missing data since not all points are visible or successfully matched in all views. The correspondences need to be loaded with this taken into account. The function `genfromtxt()` solves this by replacing the missing values (denoted with '*' in this file) with -1.

A convenient way of running this script and getting all the data is to save the code above in a file, for example *load_vggdata.py*, and use the command `execfile()` like this

```
execfile('load_vggdata.py')
```

at the beginning of your scripts or experiments.

Let's see what this data looks like. Try to project the 3D points into one view and compare the results with the observed image points.

```
# make 3D points homogeneous and project
X = vstack( (points3D,ones(points3D.shape[1])) )
x = P[0].project(X)

# plotting the points in view 1
figure()
imshow(im1)
plot(points2D[0][0],points2D[0][1],'*')
axis('off')

figure()
imshow(im1)
plot(x[0],x[1],'r.')
axis('off')
```

[2] Actually Harris corner points, see Section 2.1.

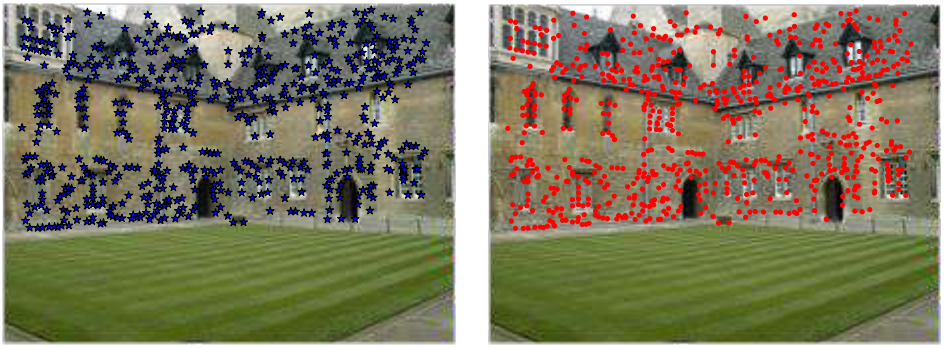


Figure 5.2: The Merton1 data set from the Oxford multi-view datasets. (left) view 1 with image points shown. (right) view 1 with projected 3D points.

```
show()
```

This creates a plot with the first view and image points in that view, for comparison the projected points are shown in a separate figure. Figure 5.2 shows the resulting plots. If you look closely, you will see that the second plot with the projected 3D points contains more points than the first. These are image feature points reconstructed from view 2 and 3 but not detected in view 1.

Plotting 3D data with `Matplotlib`

To visualize our 3D reconstructions, we need to be able to plot in 3D. The `mplot3d` toolkit in `Matplotlib` provides 3D plotting of points, lines, contours, surfaces and most other basic plotting components as well as 3D rotation and scaling from the controls of the figure window.

Making a plot in 3D is done by adding the *projection="3d"* keyword to the axes object like this:

```
from mpl_toolkits.mplot3d import axes3d

fig = figure()
ax = fig.gca(projection="3d")

# generate 3D sample data
X,Y,Z = axes3d.get_test_data(0.25)

# plot the points in 3D
ax.plot(X.flatten(),Y.flatten(),Z.flatten(),'o')
```

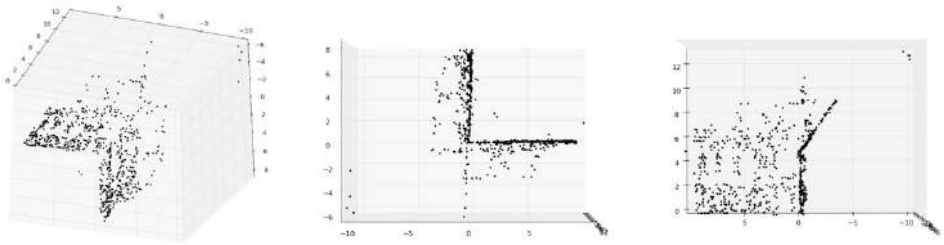


Figure 5.3: The 3D points of the Merton1 data set from the Oxford multi-view datasets shown using `Matplotlib`. (left) view from above and to the side. (middle) view from the top showing the building walls and points on the roof. (right) side view showing the profile of one of the walls and a frontal view of points on the other wall.

```
show()
```

The function `get_test_data()` generates sample points on a regular x, y grid with the parameter determining the spacing. Flattening these grids gives three lists of points that can be sent to `plot()`. This should plot 3D points on what looks like a surface. Try it out and see for yourself.

Now we can plot the Merton sample data to see what the 3D points look like.

```
# plotting 3D points
from mpl_toolkits.mplot3d import axes3d
fig = figure()
ax = fig.gca(projection='3d')
ax.plot(points3D[0],points3D[1],points3D[2],'k.')
```

Figure 5.3 shows the 3D points from three different views. The figure window and controls look like the standard plot windows for images and 2D data with an additional 3D rotation tool.

Computing F - The eight point algorithm

The *eight point algorithm* is an algorithm for computing the fundamental matrix from point correspondences. Here's a brief description, the details can be found in [14] and [13].

The epipolar constraint (5.1) can be written as a linear system like

$$\begin{bmatrix} x_2^1x_1^1 & x_2^1y_1^1 & x_2^1w_1^1 & \dots & w_2^1w_1^1 \\ x_2^2x_1^2 & x_2^2y_1^2 & x_2^2w_1^2 & \dots & w_2^2w_1^2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ x_2^nx_1^n & x_2^ny_1^n & x_2^nw_1^n & \dots & w_2^nw_1^n \end{bmatrix} \begin{bmatrix} F_{11} \\ F_{12} \\ F_{13} \\ \vdots \\ F_{33} \end{bmatrix} = A\mathbf{f} = 0 \ ,$$

where $\mathbf{f}$ contains the elements of F, $\mathbf{x}_1^i = [x_1^i, y_1^i, w_1^i]$ and $\mathbf{x}_2^i = [x_2^i, y_2^i, w_2^i]$ is a correspondence pair and there are n point correspondences in total. The fundamental matrix has nine elements but since the scale is arbitrary, only eight equations are needed. Eight point correspondences are therefore needed to compute F, hence the name of the algorithm.

Create a file *sfm.py*, and add the following function for the eight point algorithm that minimizes $||A\mathbf{f}||$.

```
def compute_fundamental(x1,x2):
  """ Computes the fundamental matrix from corresponding points
    (x1,x2 3*n arrays) using the normalized 8 point algorithm.
    each row is constructed as
    [x'*x, x'*y, x', y'*x, y'*y, y', x, y, 1] """

  n = x1.shape[1]
  if x2.shape[1] != n:
    raise ValueError("Number of points don't match.")

  # build matrix for equations
  A = zeros((n,9))
  for i in range(n):
    A[i] = [x1[0,i]*x2[0,i], x1[0,i]*x2[1,i], x1[0,i]*x2[2,i],
        x1[1,i]*x2[0,i], x1[1,i]*x2[1,i], x1[1,i]*x2[2,i],
        x1[2,i]*x2[0,i], x1[2,i]*x2[1,i], x1[2,i]*x2[2,i] ]

  # compute linear least square solution
  U,S,V = linalg.svd(A)
  F = V[-1].reshape(3,3)

  # constrain F
  # make rank 2 by zeroing out last singular value
  U,S,V = linalg.svd(F)
  S[2] = 0
  F = dot(U,dot(diag(S),V))

  return F
```

As usual we compute the least squares solution using SVD. Since the resulting solution might not have rank 2 as a proper fundamental matrix should, we replace the result with the closest rank 2 approximation by zeroing out the last singular value. This is a standard trick and a useful one to know. The function ignores the important step of normalizing the image coordinates. Ignoring normalization could give numerical problems. Let's leave that for later.

The epipole and epipolar lines

As mentioned at the start of this section, the epipole satisfies $F\mathbf{e}_1 = 0$ and can be computed from the null space of F. Add this function to *sfm.py*.

```
def compute_epipole(F):
  """ Computes the (right) epipole from a
    fundamental matrix F.
    (Use with F.T for left epipole.) """

  # return null space of F (Fx=0)
  U,S,V = linalg.svd(F)
  e = V[-1]
  return e/e[2]
```

If you want the epipole corresponding to the left null vector (corresponding to the epipole in the other image), just transpose F before passing it as input.

We can try these two functions on the first two views of our sample data set like this:

```
import sfm

# index for points in first two views
ndx = (corr[:,0]>=0) & (corr[:,1]>=0)

# get coordinates and make homogeneous
x1 = points2D[0][:,corr[ndx,0]]
x1 = vstack( (x1,ones(x1.shape[1])) )
x2 = points2D[1][:,corr[ndx,1]]
x2 = vstack( (x2,ones(x2.shape[1])) )

# compute F
F = sfm.compute_fundamental(x1,x2)

# compute the epipole
e = sfm.compute_epipole(F)

# plotting
figure()
```

```
imshow(im1)
# plot each line individually, this gives nice colors
for i in range(5):
  sfm.plot_epipolar_line(im1,F,x2[:,i],e,False)
axis('off')

figure()
imshow(im2)
# plot each point individually, this gives same colors as the lines
for i in range(5):
  plot(x2[0,i],x2[1,i],'o')
axis('off')

show()
```

First the points that are in correspondence between the two images are selected and made into homogeneous coordinates. Here we just read them from a text file, in reality these would be the result of extracting features and matching them like we did in Chapter 2. The missing values in the correspondence list *corr* are -1 so picking indices greater or equal to zero gives the points visible in each view. The two conditions are combined with the array operator &.

Lastly, the first five of the epipolar lines are shown in the first view and the corresponding matching points in view 2. Here we used the helper plot function.

```
def plot_epipolar_line(im,F,x,epipole=None,show_epipole=True):
  """ Plot the epipole and epipolar line F*x=0
    in an image. F is the fundamental matrix
    and x a point in the other image."""

  m,n = im.shape[:2]
  line = dot(F,x)

  # epipolar line parameter and values
  t = linspace(0,n,100)
  lt = array([(line[2]+line[0]*tt)/(-line[1]) for tt in t])

  # take only line points inside the image
  ndx = (lt>=0) & (lt<m)
  plot(t[ndx],lt[ndx],linewidth=2)

  if show_epipole:
    if epipole is None:
      epipole = compute_epipole(F)
    plot(epipole[0]/epipole[2],epipole[1]/epipole[2],'r*')
```

This function parameterizes the line with the range of the x axis and removes parts of lines above and below the image border. If the last parameter *show_epipole* is true,

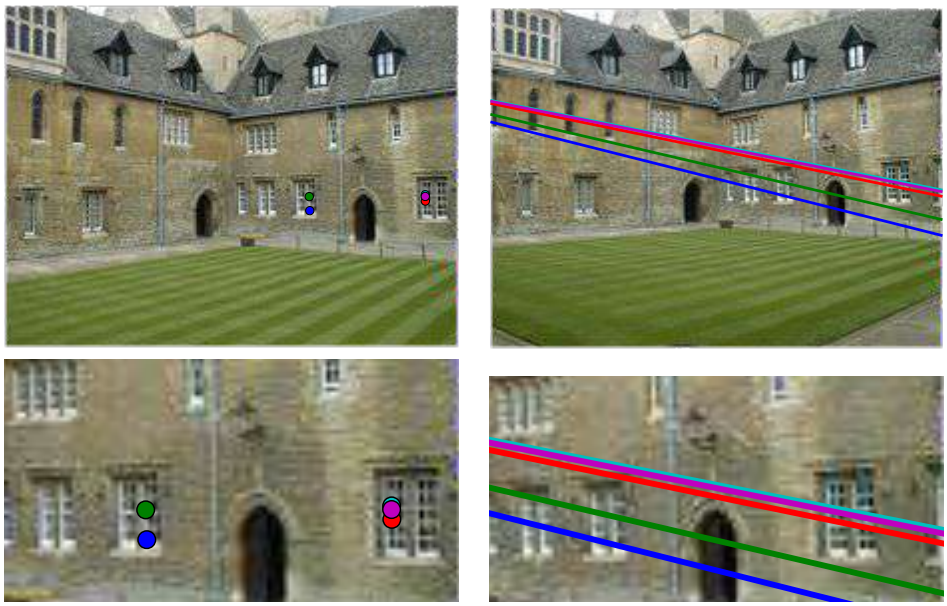


Figure 5.4: Epipolar lines in view 1 shown for five points in view 2 of the Merton1 data. The bottom row shows a close up of the area around the points. The lines can be seen to converge on a point outside the image to the left. The lines show where point correspondences can be found in the other image (the color coding matches between lines and points).

the epipole will be plotted as well (and computed if not passed as input). The plots are shown in Figure 5.4. The color coding matches between the plots so you can see that the corresponding point in one image lies somewhere along the same-color line as a point in the other image.

5.2 Computing with Cameras and 3D Structure

The previous section covered relationships between views and how to compute the fundamental matrix and epipolar lines. Here we briefly explain the tools we need for computing with cameras and 3D structure.

Triangulation

Given known camera matrices, a set of point correspondences can be triangulated to recover the 3D positions of these points. The basic algorithm is fairly simple.

For two views with camera matrices P_1 and P_2, each with a projection $\mathbf{x}_1$ and $\mathbf{x}_2$ of the same 3D point $\mathbf{X}$ (all in homogeneous coordinates), the camera equation (4.1) gives the following relation

$$\begin{bmatrix} P_1 & -\mathbf{x}_1 & 0 \\ P_2 & 0 & -\mathbf{x}_2 \end{bmatrix} \begin{bmatrix} \mathbf{X} \\ \lambda_1 \\ \lambda_2 \end{bmatrix} = 0 \ .$$

There might not be an exact solution to these equations due to image noise, errors in the camera matrices or other sources of errors. Using SVD, we can get a least squares estimate of the 3D point.

Add the following function that computes the *least squares triangulation* of a point pair to *sfm.py*.

```
def triangulate_point(x1,x2,P1,P2):
  """ Point pair triangulation from
    least squares solution. """

  M = zeros((6,6))
  M[:3,:4] = P1
  M[3:,:4] = P2
  M[:3,4] = -x1
  M[3:,5] = -x2

  U,S,V = linalg.svd(M)
  X = V[-1,:4]

  return X / X[3]
```

The first four values in the last eigenvector are the 3D coordinates in homogeneous coordinates. To triangulate many points, we can add the following convenience function.

```
def triangulate(x1,x2,P1,P2):
  """ Two-view triangulation of points in
    x1,x2 (3*n homog. coordinates). """

  n = x1.shape[1]
  if x2.shape[1] != n:
    raise ValueError("Number of points don't match.")

  X = [ triangulate_point(x1[:,i],x2[:,i],P1,P2) for i in range(n)]
  return array(X).T
```

This function takes two arrays of points and returns an array of 3D coordinates.

Try the triangulation on the Merton1 data like this.

```
import sfm

# index for points in first two views
ndx = (corr[:,0]>=0) & (corr[:,1]>=0)

# get coordinates and make homogeneous
x1 = points2D[0][:,corr[ndx,0]]
x1 = vstack( (x1,ones(x1.shape[1])) )
x2 = points2D[1][:,corr[ndx,1]]
x2 = vstack( (x2,ones(x2.shape[1])) )

Xtrue = points3D[:,ndx]
Xtrue = vstack( (Xtrue,ones(Xtrue.shape[1])) )

# check first 3 points
Xest = sfm.triangulate(x1,x2,P[0].P,P[1].P)
print Xest[:,:3]
print Xtrue[:,:3]

# plotting
from mpl_toolkits.mplot3d import axes3d
fig = figure()
ax = fig.gca(projection='3d')
ax.plot(Xest[0],Xest[1],Xest[2],'ko')
ax.plot(Xtrue[0],Xtrue[1],Xtrue[2],'r.')
axis('equal')

show()
```

This will triangulate the points in correspondence from the first two views and print out the coordinates of the first three points to the console before plotting the recovered 3D points next to the true values. The printout looks like this:

```
[[ 1.03743725  1.56125273  1.40720017]
 [-0.57574987 -0.55504127 -0.46523952]
 [ 3.44173797  3.44249282  7.53176488]
 [ 1.          1.          1.        ]]
[[ 1.0378863   1.5606923   1.4071907 ]
 [-0.54627892 -0.5211711  -0.46371818]
 [ 3.4601538   3.4636809   7.5323397 ]
 [ 1.          1.          1.        ]]
```

The estimated points are close enough. The plot looks like Figure 5.5, as you can see the points match fairly well.

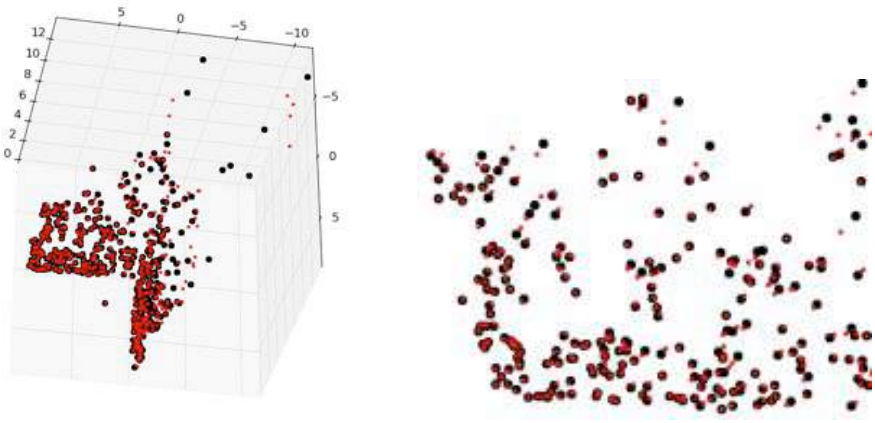


Figure 5.5: Triangulated points using camera matrices and point correspondences. The estimated points are shown with black circles and the true points with red dots. (left) view from above and to the side. (right) close up of the points from one of the building walls.

Computing the camera matrix from 3D points

With known 3D points and their image projections, the camera matrix, P, can be computed using a direct linear transform approach. This is essentially the inverse problem to triangulation and is sometimes called *camera resectioning*. This way to recover the camera matrix is again a least squares approach.

From the camera equation (4.1), each visible 3D point $\mathbf{X}_i$ (in homogeneous coordinates) is projected to an image point $\mathbf{x}_i = [x_i, y_i, 1]$ as $\lambda_i \mathbf{x}_i = P\mathbf{X}_i$ and the corresponding points satisfy the relation

$$\begin{bmatrix} \mathbf{X}_1^T & 0 & 0 & -x_1 & 0 & 0 & \dots \\ 0 & \mathbf{X}_1^T & 0 & -y_1 & 0 & 0 & \dots \\ 0 & 0 & \mathbf{X}_1^T & -1 & 0 & 0 & \dots \\ \mathbf{X}_2^T & 0 & 0 & 0 & -x_2 & 0 & \dots \\ 0 & \mathbf{X}_2^T & 0 & 0 & -y_2 & 0 & \dots \\ 0 & 0 & \mathbf{X}_2^T & 0 & -1 & 0 & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \end{bmatrix} \begin{bmatrix} \mathbf{p}_1^T \\ \mathbf{p}_2^T \\ \mathbf{p}_3^T \\ \lambda_1 \\ \lambda_2 \\ \vdots \end{bmatrix} = 0 \ ,$$

where $\mathbf{p}_1$, $\mathbf{p}_2$ and $\mathbf{p}_3$ are the three rows of P. This can be written more compactly as

$$M\mathbf{v} = 0 \ .$$

The estimation of the camera matrix is then obtained using SVD. With the matrices described above, the code is straight-forward. Add the function below to *sfm.py*.

```
def compute_P(x,X):
  """ Compute camera matrix from pairs of
    2D-3D correspondences (in homog. coordinates). """

  n = x.shape[1]
  if X.shape[1] != n:
    raise ValueError("Number of points don't match.")

  # create matrix for DLT solution
  M = zeros((3*n,12+n))
  for i in range(n):
    M[3*i,0:4] = X[:,i]
    M[3*i+1,4:8] = X[:,i]
    M[3*i+2,8:12] = X[:,i]
    M[3*i:3*i+3,i+12] = -x[:,i]

  U,S,V = linalg.svd(M)

  return V[-1,:12].reshape((3,4))
```

This function takes the image points and 3D points and builds up the matrix M above. The first 12 values of the last eigenvector are the elements of the camera matrix and are returned after a reshaping operation.

Again, let's try this on our sample data set. The following script will pick out the points that are visible in the first view (using the missing values from the correspondence list), make them into homogeneous coordinates and estimate the camera matrix.

```
import sfm, camera

corr = corr[:,0] # view 1
ndx3D = where(corr>=0)[0] # missing values are -1
ndx2D = corr[ndx3D]

# select visible points and make homogeneous
x = points2D[0][:,ndx2D] # view 1
x = vstack( (x,ones(x.shape[1])) )
X = points3D[:,ndx3D]
X = vstack( (X,ones(X.shape[1])) )

# estimate P
Pest = camera.Camera(sfm.compute_P(x,X))

# compare!
print Pest.P / Pest.P[2,3]
print P[0].P / P[0].P[2,3]
```

Figure 5.6: Projected points in view 1 computed using an estimated camera matrix.

```
xest = Pest.project(X)

# plotting
figure()
imshow(im1)
plot(x[0],x[1],'b.')
plot(xest[0],xest[1],'r.')
axis('off')

show()
```

To check the camera matrices they are printed to the console in normalized form (by dividing with the last element). The printout looks like this.

```
[[  1.06520794e+00  -5.23431275e+01   2.06902749e+01   5.08729305e+02]
 [ -5.05773115e+01  -1.33243276e+01  -1.47388537e+01   4.79178838e+02]
 [  3.05121915e-03  -3.19264684e-02  -3.43703738e-02   1.00000000e+00]]
[[  1.06774679e+00  -5.23448212e+01   2.06926980e+01   5.08764487e+02]
 [ -5.05834364e+01  -1.33201976e+01  -1.47406641e+01   4.79228998e+02]
 [  3.06792659e-03  -3.19008054e-02  -3.43665129e-02   1.00000000e+00]]
```

The top is the estimated camera matrix and below the one computed by the creators of the data set. As you can see, they are almost identical. Lastly, the 3D points are projected using the estimated camera and plotted. The result looks like Figure 5.6 with the true points in blue and the estimated camera projection in red.

Computing the camera matrix from a fundamental matrix

In a two view scenario, the camera matrices can be recovered from the fundamental matrix. Assuming the first camera matrix is normalized to $P_1 = [I \mid 0]$, the problem is

to find the second camera matrix P_2. There are two different cases, the uncalibrated case and the calibrated case.

The uncalibrated case - projective reconstruction Without any knowledge of the camera's intrinsic parameters the camera matrix can only be retrieved up to a projective transformation. This means that if the camera pair is used to reconstruct 3D points, the reconstruction is only accurate up to a projective transformation (you can get any solution out of the whole range of projective scene distortions). This means that angles and distances are not respected.

This means that in the uncalibrated case the second camera matrix can be chosen up to a (3×3) projective transformation. A simple choice is

$$P_2 = [S_{\mathbf{e}}F \mid \mathbf{e}] \ ,$$

where $\mathbf{e}$ is the left epipole, $\mathbf{e}^T F = 0$ and $S_{\mathbf{e}}$ a skew matrix as in equation (5.2). Remember, a triangulation with this matrix will most likely give distortions, for example in the form of skewed reconstructions.

Here is what it looks like in code:

```
def compute_P_from_fundamental(F):
  """ Computes the second camera matrix (assuming P1 = [I 0])
    from a fundamental matrix. """

  e = compute_epipole(F.T) # left epipole
  Te = skew(e)
  return vstack((dot(Te,F.T).T,e)).T
```

We used the helper function `skew()` defined as.

```
def skew(a):
  """ Skew matrix A such that a x v = Av for any v. """

  return array([[0,-a[2],a[1]],[a[2],0,-a[0]],[-a[1],a[0],0]])
```

Add both these functions to the file *sfm.py*.

The calibrated case - metric reconstruction With known calibration the reconstruction will be metric and preserve properties of Euclidean space (except for a global scale parameter). In terms of reconstructing a 3D scene, this calibrated case is the interesting one.

With known calibration K, we can apply its inverse K^{-1} to the image points $\mathbf{x}_K = K^{-1}\mathbf{x}$ so that the camera equation becomes

$$\mathbf{x}_K = K^{-1}K[R \mid \mathbf{t}]\mathbf{X} = [R \mid \mathbf{t}]\mathbf{X} \ ,$$

in the new image coordinates. The points in these new image coordinates satisfy the same fundamental equation as before

$$\mathbf{x}_{K_2}^T F \mathbf{x}_{K_1} = 0 \ .$$

The fundamental matrix for calibration normalized coordinates is called the *essential matrix* and is usually denoted E instead of F to make it clear that this is the calibrated case and the image coordinates are normalized.

The camera matrices recovered from an essential matrix respect metric relationships but there are four possible solutions. Only one of them has the scene in front of both cameras so it is easy to pick the right one.

Here is an algorithm for computing the four solutions (see [13] for the details). Add this function to *sfm.py*.

```
def compute_P_from_essential(E):
  """ Computes the second camera matrix (assuming P1 = [I 0])
    from an essential matrix. Output is a list of four
    possible camera matrices. """

  # make sure E is rank 2
  U,S,V = svd(E)
  if det(dot(U,V))<0:
    V = -V
  E = dot(U,dot(diag([1,1,0]),V))

  # create matrices (Hartley p 258)
  Z = skew([0,0,-1])
  W = array([[0,-1,0],[1,0,0],[0,0,1]])

  # return all four solutions
  P2 = [vstack((dot(U,dot(W,V)).T,U[:,2])).T,
      vstack((dot(U,dot(W,V)).T,-U[:,2])).T,
      vstack((dot(U,dot(W.T,V)).T,U[:,2])).T,
      vstack((dot(U,dot(W.T,V)).T,-U[:,2])).T]

  return P2
```

First this function makes sure the essential matrix is rank 2 (with two equal non-zero singular values), then the four solutions are created according to the recipe in [13]. A list with four camera matrices is returned. How to pick the right one, we leave to the example later.

This concludes all the theory needed to compute 3D reconstructions from a collection of images.

5.3 Multiple View Reconstruction

Let's look at how to use the concepts above to compute an actual 3D reconstruction from a pair of images. Computing a 3D reconstruction like this is usually referred to as *structure from motion* (*SfM*) since the motion of a camera (or cameras) give you 3D structure.

Assuming the camera has been calibrated, the steps are as follows:

1. Detect feature points and match them between the two images.
2. Compute the fundamental matrix from the matches.
3. Compute the camera matrices from the fundamental matrix.
4. Triangulate the 3D points.

We have all the tools to do this but we need a robust way to compute a fundamental matrix when the point correspondences between the images contain incorrect matches.

Robust fundamental matrix estimation

Similar to when we needed a robust way to compute homographies (Section 3.3), we also need to be able to estimate a fundamental matrix when there is noise and incorrect matches. As before we will use RANSAC, this time combined with the eight point algorithm. It should be mentioned that the eight point algorithm breaks down for planar scenes so you cannot use it for scenes where the scene points are all on a plane.

Add this class to *sfm.py*.

```
class RansacModel(object):
  """ Class for fundmental matrix fit with ransac.py from
    http://www.scipy.org/Cookbook/RANSAC"""

  def __init__(self,debug=False):
    self.debug = debug

  def fit(self,data):
    """ Estimate fundamental matrix using eight
      selected correspondences. """

    # transpose and split data into the two point sets
    data = data.T
    x1 = data[:3,:8]
    x2 = data[3:,:8]
```

```
    # estimate fundamental matrix and return
    F = compute_fundamental_normalized(x1,x2)
    return F

  def get_error(self,data,F):
    """ Compute x^T F x for all correspondences,
      return error for each transformed point. """

    # transpose and split data into the two point
    data = data.T
    x1 = data[:3]
    x2 = data[3:]

    # Sampson distance as error measure
    Fx1 = dot(F,x1)
    Fx2 = dot(F,x2)
    denom = Fx1[0]**2 + Fx1[1]**2 + Fx2[0]**2 + Fx2[1]**2
    err = ( diag(dot(x1.T,dot(F,x2))) )**2 / denom

    # return error per point
    return err
```

As before, we need `fit()` and `get_error()` methods. The error measure chosen here is the Sampson distance (see [13]). The `fit()` method now selects eight points and uses a normalized version of the eight point algorithm.

```
def compute_fundamental_normalized(x1,x2):
  """ Computes the fundamental matrix from corresponding points
    (x1,x2 3*n arrays) using the normalized 8 point algorithm. """

  n = x1.shape[1]
  if x2.shape[1] != n:
    raise ValueError("Number of points don't match.")

  # normalize image coordinates
  x1 = x1 / x1[2]
  mean_1 = mean(x1[:2],axis=1)
  S1 = sqrt(2) / std(x1[:2])
  T1 = array([[S1,0,-S1*mean_1[0]],[0,S1,-S1*mean_1[1]],[0,0,1]])
  x1 = dot(T1,x1)

  x2 = x2 / x2[2]
  mean_2 = mean(x2[:2],axis=1)
  S2 = sqrt(2) / std(x2[:2])
  T2 = array([[S2,0,-S2*mean_2[0]],[0,S2,-S2*mean_2[1]],[0,0,1]])
  x2 = dot(T2,x2)

  # compute F with the normalized coordinates
```

```
  F = compute_fundamental(x1,x2)

  # reverse normalization
  F = dot(T1.T,dot(F,T2))

  return F/F[2,2]
```

This function normalizes the image points to zero mean and fixed variance.

Now we can use this class in a function. Add the following function to *sfm.py*.

```
def F_from_ransac(x1,x2,model,maxiter=5000,match_theshold=1e-6):
  """ Robust estimation of a fundamental matrix F from point
    correspondences using RANSAC (ransac.py from
    http://www.scipy.org/Cookbook/RANSAC).

    input: x1,x2 (3*n arrays) points in hom. coordinates. """

  import ransac

  data = vstack((x1,x2))

  # compute F and return with inlier index
  F,ransac_data = ransac.ransac(data.T,model,8,maxiter,match_theshold,20,return_all=True)
  return F, ransac_data['inliers']
```

Here we return the best fundamental matrix F together with the inlier index (so that we know what matches were consistent with F). Compared to the homography estimation, we increased the default max iterations and changed the matching threshold which was in pixels before and is in Sampson distance now.

3D reconstruction example

In this section we will see a complete example of reconstructing a 3D scene from start to finish. We will use two images taken with a camera with known calibration. The images are of the famous Alcatraz prison and are shown in Figure 5.7[3].

Let's split the code up in a few chunks so that it is easier to follow. First we extract features, match them and estimate a fundamental matrix and camera matrices.

```
import homography
import sfm
import sift

# calibration
K = array([[2394,0,932],[0,2398,628],[0,0,1]])
```

[3]Images courtesy of Carl Olsson `http://www.maths.lth.se/matematiklth/personal/calle/`.

Figure 5.7: Example image pair of a scene where the images are taken at different viewpoints.

```
# load images and compute features
im1 = array(Image.open('alcatraz1.jpg'))
sift.process_image('alcatraz1.jpg','im1.sift')
l1,d1 = sift.read_features_from_file('im1.sift')

im2 = array(Image.open('alcatraz2.jpg'))
sift.process_image('alcatraz2.jpg','im2.sift')
l2,d2 = sift.read_features_from_file('im2.sift')


# match features
matches = sift.match_twosided(d1,d2)
ndx = matches.nonzero()[0]

# make homogeneous and normalize with inv(K)
x1 = homography.make_homog(l1[ndx,:2].T)
ndx2 = [int(matches[i]) for i in ndx]
x2 = homography.make_homog(l2[ndx2,:2].T)

x1n = dot(inv(K),x1)
x2n = dot(inv(K),x2)


# estimate E with RANSAC
model = sfm.RansacModel()
E,inliers = sfm.F_from_ransac(x1n,x2n,model)

# compute camera matrices (P2 will be list of four solutions)
P1 = array([[1,0,0,0],[0,1,0,0],[0,0,1,0]])
P2 = sfm.compute_P_from_essential(E)
```

The calibration is known so here we just hardcode the K matrix at the beginning. As in earlier examples, we pick out the points that belong to matches. After that we normalize them with K^{-1} and run the RANSAC estimation with the normalized eight point algorithm. Since the points are normalized, this gives us an essential matrix. We make sure to keep the index of the inliers, we will need them. From the essential matrix we compute the four possible solutions of the second camera matrix.

From the list of camera matrices, we pick the one that has the most scene points in front of both cameras after triangulation.

```
# pick the solution with points in front of cameras
ind = 0
maxres = 0
for i in range(4):
  # triangulate inliers and compute depth for each camera
  X = sfm.triangulate(x1n[:,inliers],x2n[:,inliers],P1,P2[i])
  d1 = dot(P1,X)[2]
  d2 = dot(P2[i],X)[2]
  if sum(d1>0)+sum(d2>0) > maxres:
    maxres = sum(d1>0)+sum(d2>0)
    ind = i
    infront = (d1>0) & (d2>0)


# triangulate inliers and remove points not in front of both cameras
X = sfm.triangulate(x1n[:,inliers],x2n[:,inliers],P1,P2[ind])
X = X[:,infront]
```

We loop through the four solutions and each time triangulate the 3D points corresponding to the inliers. The sign of the depth is given by the third value of each image point after projecting the triangulated X back to the images. We keep the index with the most positive depths and also store a boolean for each point in the best solution so that we can pick only the ones that actually are in front. Due to noise and errors in all of the estimations done, there is a risk that some points still are behind one camera, even with the correct camera matrices. Once we have the right solution, we triangulate the inliers and keep the points in front of the cameras.

Now we can plot the reconstruction.

```
# 3D plot
from mpl_toolkits.mplot3d import axes3d

fig = figure()
ax = fig.gca(projection='3d')
ax.plot(-X[0],X[1],X[2],'k.')
axis('off')
```

The 3D plots with `mplot3d` have the first axis reversed compared to our coordinate system so we change the sign.

We can then plot the reprojection in each view.

```
# plot the projection of X
import camera

# project 3D points
cam1 = camera.Camera(P1)
cam2 = camera.Camera(P2[ind])
x1p = cam1.project(X)
x2p = cam2.project(X)

# reverse K normalization
x1p = dot(K,x1p)
x2p = dot(K,x2p)


figure()
imshow(im1)
gray()
plot(x1p[0],x1p[1],'o')
plot(x1[0],x1[1],'r.')
axis('off')

figure()
imshow(im2)
gray()
plot(x2p[0],x2p[1],'o')
plot(x2[0],x2[1],'r.')
axis('off')
show()
```

After projecting the 3D points we need to reverse the initial normalization by multiplying with the calibration matrix.

The result looks like Figure 5.8. As you can see, the reprojected points (blue) don't exactly match the original feature locations (red) but they are reasonably close. It is possible to further refine the camera matrices to improve the reconstruction and reprojection but that is outside the scope of this simple example.

Extensions and more than two views

There are some steps and further extensions to multiple view reconstructions that we cannot cover in a book like this. Here are some of them with references for further

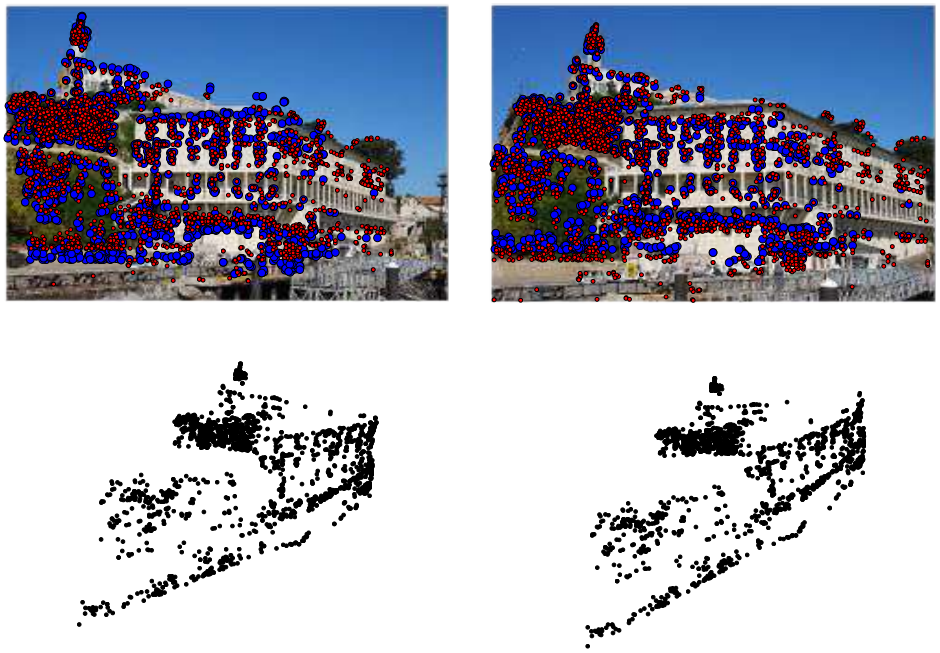


Figure 5.8: Example of computing a 3D reconstruction from a pair of images using image matches. (top) the two images with feature points shown in red and reprojected reconstructed 3D points shown in blue. (bottom) the 3D reconstruction.

reading.

More views With more than two views of the same scene the 3D reconstruction will usually be more accurate and more detailed. Since the fundamental matrix only relates a pair of views, the process is a little different with many images.

For video sequences, one can use the temporal aspect and match features in consecutive frame pairs. The relative orientation needs to be added incrementally from each pair to the next (similar to how we added homographies in the panorama example in Figure 3.12). This approach usually works well and tracking can be used to effectively find correspondences (see Section 10.4 for more on tracking). One problem is that errors will accumulate the more views are added. This can be fixed with a final optimization step, see below.

With still images, one approach is to find a central reference view and compute all the other camera matrices relative to that one. Another method is to compute camera matrices and a 3D reconstruction for one image pair and then incrementally add new images and 3D points, see for example [34]. As a side note, there are ways to compute 3D and camera positions from three views at the same time (see for example [13]) but beyond that an incremental approach is needed.

Bundle adjustment From our simple 3D reconstruction example in Figure 5.8 it is clear that there will be errors in the position of the recovered points and the camera matrices computed from the estimated fundamental matrix. With more views the errors will accumulate. A final step in multiple view reconstructions is therefore often to try to minimize the reprojection errors by optimizing the position of the 3D points and the camera parameters. This process is called *bundle adustment*. Details can be found in [13] and [35] and a short overview at `http://en.wikipedia.org/wiki/Bundle_adjustment`.

Self calibration In the case of uncalibrated cameras, it is sometimes possible to compute the calibration from image features. This process is called *self-calibration*. There are many different algorithms depending on what assumptions can be made on parameters of the camera calibration matrix and depending on what types of image data is available (feature matches, parallel lines, planes etc.). The interested reader can take a look at [13] and [26] (Chapter 6).

As a side note to calibration, there is a useful script *extract_focal.pl* as part of the Bundler SfM system `http://phototour.cs.washington.edu/bundler/`. This uses a lookup table for common cameras and estimates the focal length based on the image EXIF data.

5.4 Stereo Images

A special case of multi-view imaging is *stereo vision* (or *stereo imaging*) where two cameras are observing the same scene with only a horizontal (sideways) displacement between the cameras. When the cameras are configured so that the two images have the same image plane with the image rows vertically aligned, the image pair is said to be *rectified*. This is common in robotics and such a setup is often called a *stereo rig*.

Any stereo camera setup can be rectified by warping the images to a common plane so that the epipolar lines are image rows (a stereo rig is usually constructed to give such rectified image pairs). This is outside the scope of this section but the interested reader can find the details in [13] (page 303) or [3] (page 430).

Assuming that the two images are rectified, finding correspondences is constrained to searching along image rows. Once a corresponding point is found, its depth (Z coordinate) can be computed directly from the horizontal displacement as it is inversely proportional to the displacement,

$$Z = \frac{fb}{x_l - x_r} \ ,$$

where f is the rectified image focal length, b the distance between the camera centers, and x_l and x_r the x-coordinate of the corresponding point in the left and right image. The distance separating the camera centers is called the *baseline*. Figure 5.9 illustrates a rectified stereo camera setup.

Stereo reconstruction (sometimes called *dense depth reconstruction*) is the problem of recovering a depth map (or inversely a disparity map) where the depth (or disparity) for each pixel in the image is estimated. This is a classic problem in computer vision and there are many algorithms for solving it. The Middlebury Stereo Vision Page (`http://vision.middlebury.edu/stereo/`) contains a constantly updated evaluation of the best algorithms with code and descriptions of many implementations. In the next section we will implement a stereo reconstruction algorithm based on normalized cross correlation.

Computing disparity maps

In this stereo reconstruction algorithm we will try a range of displacements and record the best displacement for each pixel by selecting the one with the best score according to normalized cross correlation of the local image neighborhood. This is sometimes called plane sweeping since each displacement step corresponds to a plane at some depth. While not exactly state of the art in stereo reconstruction, this is a simple method that usually gives decent results.

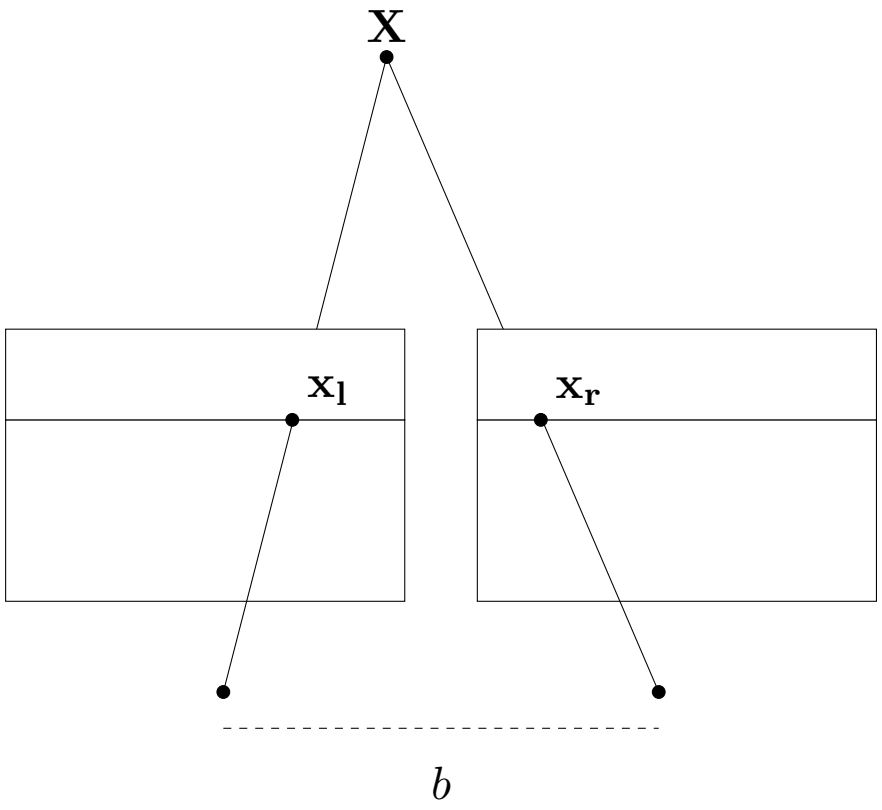


Figure 5.9: An illustration of a rectified stereo image setup where corresponding points are on the same rows in both images. (Images from the Middlebury Stereo Vision set "cones".)

Normalized cross correlation can be efficiently computed when applied densely across images. This is different from when we applied it between sparse point correspondences in Chapter 2. We want to evaluate normalized cross correlation on a patch (basically a local neighborhood) around each pixel. For this case we can rewrite the NCC around a pixel, equation (2.3), as

$$ncc(I_1, I_2) = \frac{\sum_{\mathbf{x}}(I_1(\mathbf{x}) - \mu_1)(I_2(\mathbf{x}) - \mu_2)}{\sqrt{\sum_{\mathbf{x}}(I_1(\mathbf{x}) - \mu_1)^2 \sum_{\mathbf{x}}(I_2(\mathbf{x}) - \mu_2)^2}}$$

where we skip the normalizing constant in front (it is not needed here) and the sums are taken over the pixels of a local patch around the pixel.

Now, we want this for every pixel in the image. The three sums are over a local patch region and can be computed efficiently using image filters, just like we did for blur and derivatives. The function `uniform_filter()` in the `ndimage.filters` module will compute the sums over a rectangular patch.

Here's the function that does the plane sweep and returns the best disparity for each pixel. Create a file *stereo.py* and add the following.

```
def plane_sweep_ncc(im_l,im_r,start,steps,wid):
  """ Find disparity image using normalized cross-correlation. """

  m,n = im_l.shape

  # arrays to hold the different sums
  mean_l = zeros((m,n))
  mean_r = zeros((m,n))
  s = zeros((m,n))
  s_l = zeros((m,n))
  s_r = zeros((m,n))

  # array to hold depth planes
  dmaps = zeros((m,n,steps))

  # compute mean of patch
  filters.uniform_filter(im_l,wid,mean_l)
  filters.uniform_filter(im_r,wid,mean_r)

  # normalized images
  norm_l = im_l - mean_l
  norm_r = im_r - mean_r

  # try different disparities
  for displ in range(steps):
    # move left image to the right, compute sums
    filters.uniform_filter(roll(norm_l,-displ-start)*norm_r,wid,s) # sum nominator
    filters.uniform_filter(roll(norm_l,-displ-start)*roll(norm_l,-displ-start),wid,s_l)
```

```
    filters.uniform_filter(norm_r*norm_r,wid,s_r) # sum denominator

    # store ncc scores
    dmaps[:,:,displ] = s/sqrt(s_l*s_r)

  # pick best depth for each pixel
  return argmax(dmaps,axis=2)
```

First we need to create some arrays to hold the filtering results as `uniform_filter()` takes them as input arguments. Then we create an array to hold each of the planes so that we can apply `argmax()` along the last dimension to find the best depth for each pixel. The function iterates over all *steps* displacements from *start*. One image is shifted using the `roll()` function and the three sums of the NCC are computed using filtering.

Here is a full example of loading images and computing the displacement map using this function.

```
import stereo

im_l = array(Image.open('scene1.row3.col3.ppm').convert('L'),'f')
im_r = array(Image.open('scene1.row3.col4.ppm').convert('L'),'f')

# starting displacement and steps
steps = 12
start = 4

# width for ncc
wid = 9

res = stereo.plane_sweep_ncc(im_l,im_r,start,steps,wid)

import scipy.misc
scipy.misc.imsave('depth.png',res)
```

Here we first load a pair of images from the classic "tsukuba" set and convert them to grayscale. Next we set the parameters needed for the plane sweep function, the number of displacements to try, the starting value and the width of the NCC patch. You will notice that this method is fairly fast, at least compared to matching features with NCC. This is because everything is computed using filters.

This approach also works for other filters. The uniform filter gives all pixels in a square patch equal weight but in some cases other filters for the NCC computation might be preferred. Here is one alternative using a Gaussian filter that produces smoother disparity maps, add this to *stereo.py*.

```
def plane_sweep_gauss(im_l,im_r,start,steps,wid):
```

```
""" Find disparity image using normalized cross-correlation
  with Gaussian weighted neigborhoods. """

m,n = im_l.shape

# arrays to hold the different sums
mean_l = zeros((m,n))
mean_r = zeros((m,n))
s = zeros((m,n))
s_l = zeros((m,n))
s_r = zeros((m,n))

# array to hold depth planes
dmaps = zeros((m,n,steps))

# compute mean
filters.gaussian_filter(im_l,wid,0,mean_l)
filters.gaussian_filter(im_r,wid,0,mean_r)

# normalized images
norm_l = im_l - mean_l
norm_r = im_r - mean_r

# try different disparities
for displ in range(steps):
  # move left image to the right, compute sums
  filters.gaussian_filter(roll(norm_l,-displ-start)*norm_r,wid,0,s) # sum nominator
  filters.gaussian_filter(roll(norm_l,-displ-start)*roll(norm_l,-displ-start),wid,0,s_l)
  filters.gaussian_filter(norm_r*norm_r,wid,0,s_r) # sum denominator

  # store ncc scores
  dmaps[:,:,displ] = s/sqrt(s_l*s_r)

# pick best depth for each pixel
return argmax(dmaps,axis=2)
```

The code is the same as for the uniform filter with the exception of the extra argument in the filtering. We need to pass a zero to `gaussian_filter()` to indicate that we want a standard Gaussian and not any derivatives (see page 1.4 for details).

Use this function the same way as the previous plane sweep function. Figures 5.10 and fig-stereo-cones show some results of these two plane sweep implementations on some standard stereo benchmark images. The images are from [29] and [30] and are available at `http://vision.middlebury.edu/stereo/data/`. Here we used the "tsukuba" and "cones" images and set *wid* to 9 in the standard version and 3 for the Gaussian version. The top row shows the image pair, bottom left is the standard NCC plane sweep, and bottom right is the Gaussian version. As you can see, the Gaussian

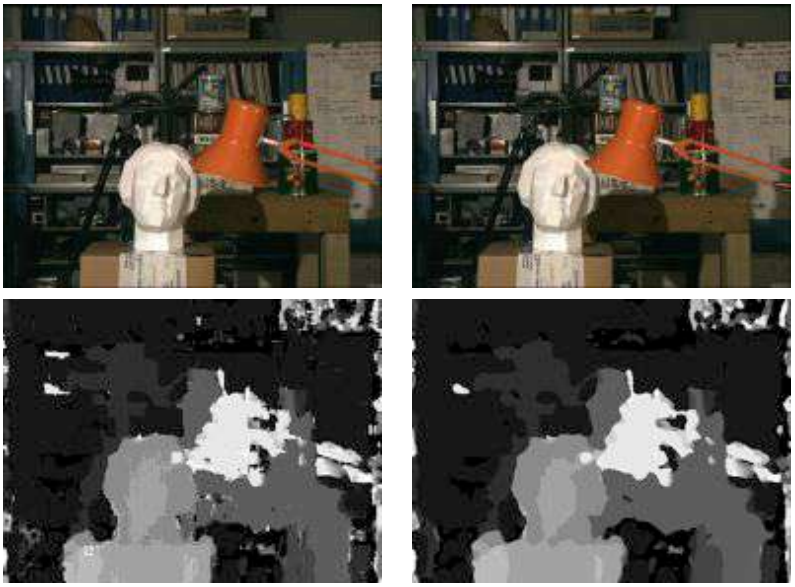


Figure 5.10: Example of computing disparity maps from a stereo image pair with normalized cross-correlation.

version is less noisy but also has less detail than the standard version.

Exercises

1. Use the techniques introduced in this chapter to verify matches in the White house example on page 66 (or even better, an example of your own) and see if you can improve on the results.

2. Compute feature matches for an image pair and estimate the fundamental matrix. Use the epipolar lines to do a second pass to find more matches by searching for the best match along the epipolar line for each feature.

3. Take a set with three or more images. Pick one pair and compute 3D points and camera matrices. Match features to the remaining images to get correspondences. Then take the 3D points for the correspondences and compute camera matrices for the other images using resection. Plot the 3D points and the camera positions. Use a set of your own or one of the Oxford multi-view sets.

4. Implement a stereo version that uses sum of squared differences (SSD) instead of NCC using filtering the same way as in the NCC example.

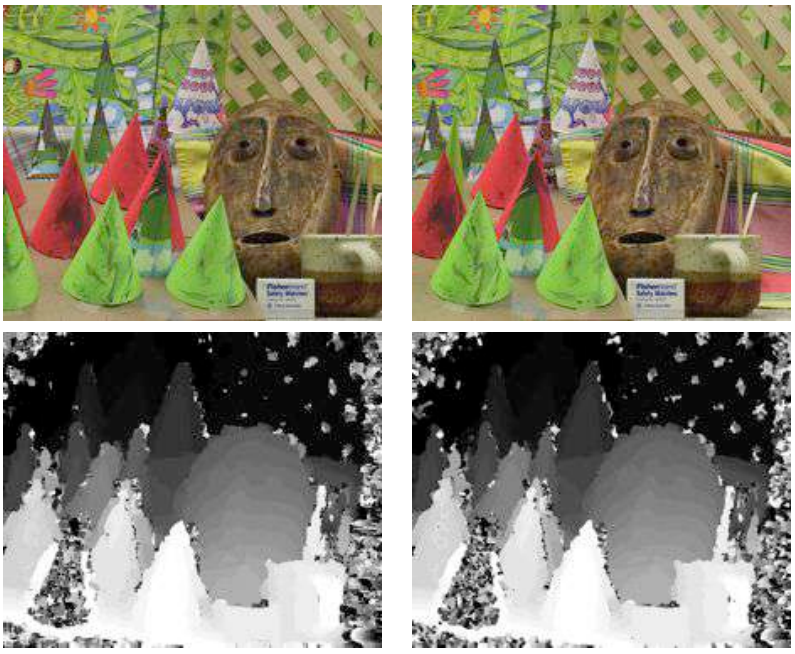


Figure 5.11: Example of computing disparity maps from a stereo image pair with normalized cross-correlation.

5. Try smoothing the stereo depth maps using the ROF de-noising from Section 1.5. Experiment with the size of the cross-correlation patches to get sharp edges with noise levels that can be removed with smoothing.

6. One way to improve the quality of the disparity maps is to compare the disparities from moving the left image to the right and the right image to the left and only keep the parts that are consistent. This will for example clean up the parts where there is occlusion. Implement this idea and compare the results to the one-directional plane sweeping.

7. The New York Public Library has many old historic stereo photographs. Browse the gallery at `http://stereo.nypl.org/gallery` and download some images you like (you can right click and save JPEGs). The images should be rectified already. Cut the image in two parts and try the dense depth reconstruction code.