CHAPTER 3

SciPy

With NumPy we can achieve fast solutions with simple coding. Where does SciPy come into the picture? It's a package that utilizes NumPy arrays and manipulations to take on standard problems that scientists and engineers commonly face: integration, determining a function's maxima or minima, finding eigenvectors for large sparse matrices, testing whether two distributions are the same, and much more. We will cover just the basics here, which will allow you to take advantage of the more complex features in SciPy by going through easy examples that are applicable to real-world problems.

We will start with optimization and data fitting, as these are some of the most common tasks, and then move through interpolation, integration, spatial analysis, clustering, signal and image processing, sparse matrices, and statistics.

3.1 Optimization and Minimization

The optimization package in SciPy allows us to solve minimization problems easily and quickly. But wait: what is minimization and how can it help you with your work? Some classic examples are performing linear regression, finding a function's minimum and maximum values, determining the root of a function, and finding where two functions intersect. Below we begin with a simple linear regression and then expand it to fitting non-linear data.

The optimization and minimization tools that NumPy and SciPy provide are great, but they do not have Markov Chain Monte Carlo (MCMC) capabilities—in other words, Bayesian analysis. There are several popular MCMC Python packages like PyMC,[1] a rich package with many options, and emcee,[2] an affine invariant MCMC ensemble sampler (meaning that large scales are not a problem for it).

[1] *http://pymc-devs.github.com/pymc/*

[2] *http://danfm.ca/emcee/*

3.1.1 Data Modeling and Fitting

There are several ways to fit data with a linear regression. In this section we will use `curve_fit`, which is a χ^2-based method (in other words, a best-fit method). In the example below, we generate data from a known function with noise, and then fit the noisy data with `curve_fit`. The function we will model in the example is a simple linear equation, $f(x) = ax + b$.

```
import numpy as np
from scipy.optimize import curve_fit

# Creating a function to model and create data
def func(x, a, b):
    return a * x + b

# Generating clean data
x = np.linspace(0, 10, 100)
y = func(x, 1, 2)

# Adding noise to the data
yn = y + 0.9 * np.random.normal(size=len(x))

# Executing curve_fit on noisy data
popt, pcov = curve_fit(func, x, yn)

# popt returns the best fit values for parameters of
# the given model (func).

print(popt)
```

The values from `popt`, if a good fit, should be close to the values for the `y` assignment. You can check the quality of the fit with `pcov`, where the diagonal elements are the variances for each parameter. Figure 3-1 gives a visual illustration of the fit.

Taking this a step further, we can do a least-squares fit to a Gaussian profile, a non-linear function:

$$a * \exp\left(\frac{-(x-\mu)^2}{2\,\sigma^2}\right),$$

where a is a scalar, μ is the mean, and σ is the standard deviation.

```
# Creating a function to model and create data
def func(x, a, b, c):
    return a*np.exp(-(x-b)**2/(2*c**2))

# Generating clean data
x = np.linspace(0, 10, 100)
y = func(x, 1, 5, 2)

# Adding noise to the data
yn = y + 0.2 * np.random.normal(size=len(x))

# Executing curve_fit on noisy data
popt, pcov = curve_fit(func, x, yn)
```

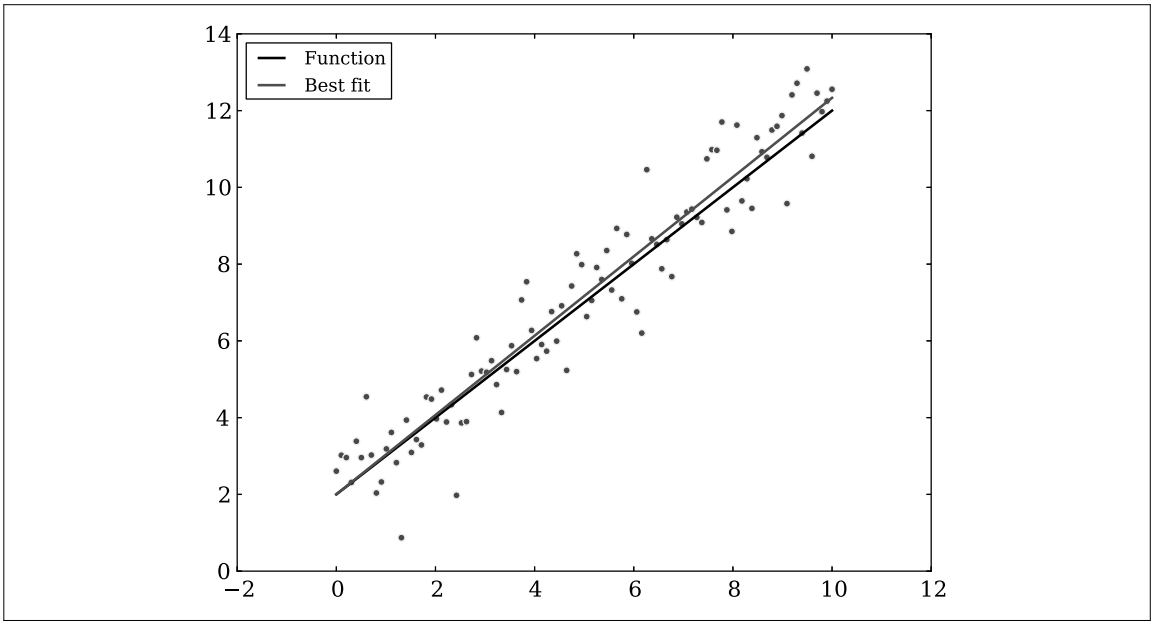


Figure 3-1. Fitting noisy data with a linear equation.

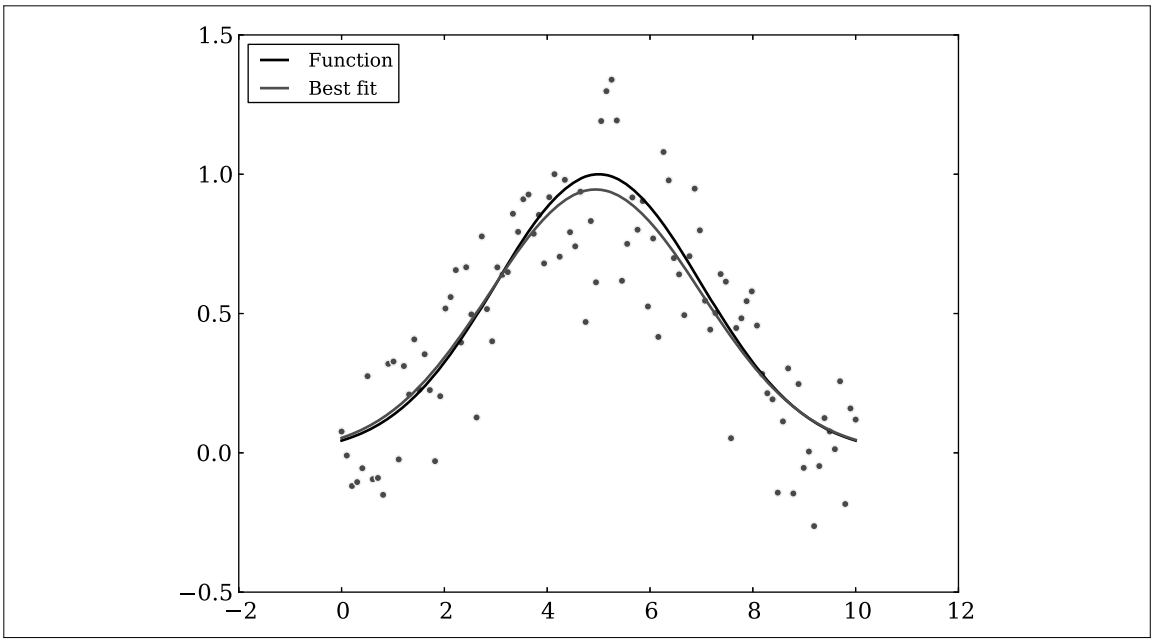


Figure 3-2. Fitting noisy data with a Gaussian equation.

```
# popt returns the best-fit values for parameters of the given model (func).
print(popt)
```

As we can see in Figure 3-2, the result from the Gaussian fit is acceptable.

Going one more step, we can fit a one-dimensional dataset with multiple Gaussian profiles. The `func` is now expanded to include two Gaussian equations with different input variables. This example would be the classic case of fitting line spectra (see Figure 3-3).

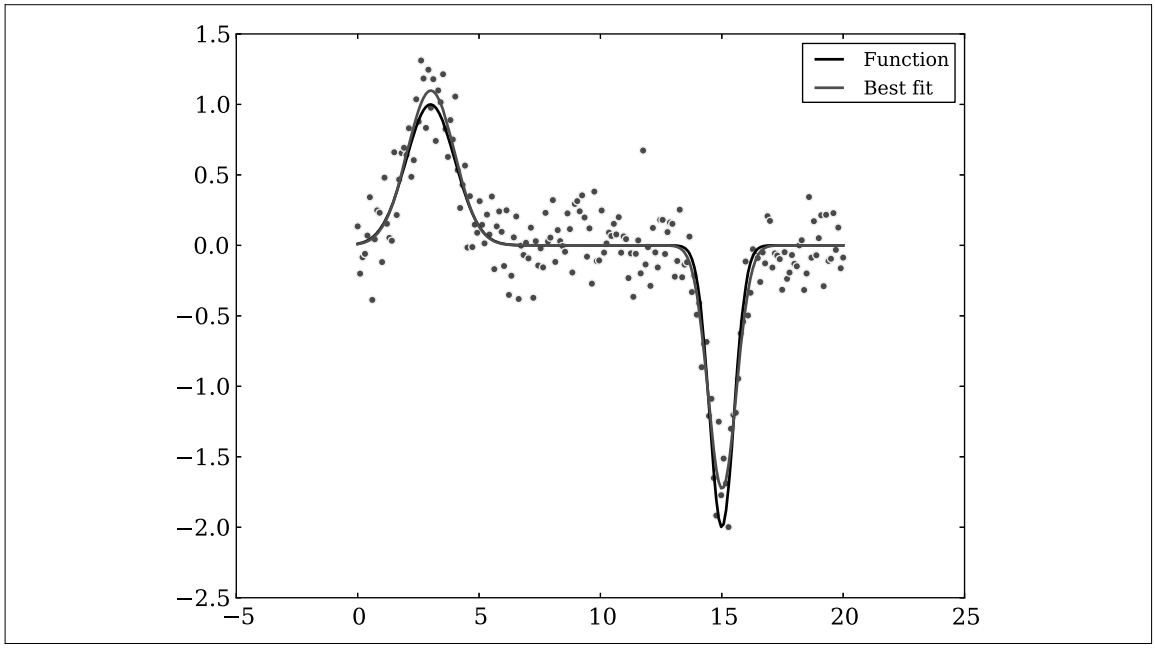


Figure 3-3. Fitting noisy data with multiple Gaussian equations.

```
# Two-Gaussian model
def func(x, a0, b0, c0, a1, b1,c1):
    return a0*np.exp(-(x - b0) ** 2/(2 * c0 ** 2))\
           + a1 * np.exp(-(x - b1) ** 2/(2 * c1 ** 2))

# Generating clean data
x = np.linspace(0, 20, 200)
y = func(x, 1, 3, 1, -2, 15, 0.5)

# Adding noise to the data
yn = y + 0.2 * np.random.normal(size=len(x))

# Since we are fitting a more complex function,
# providing guesses for the fitting will lead to
# better results.

guesses = [1, 3, 1, 1, 15, 1]
# Executing curve_fit on noisy data
popt, pcov = curve_fit(func, x, yn,
                       p0=guesses)
```

3.1.2 Solutions to Functions

With data modeling and fitting under our belts, we can move on to finding solutions, such as "What is the root of a function?" or "Where do two functions intersect?" SciPy provides an arsenal of tools to do this in the `optimize` module. We will run through the primary ones in this section.

Let's start simply, by solving for the root of an equation (see Figure 3-4). Here we will use `scipy.optimize.fsolve`.

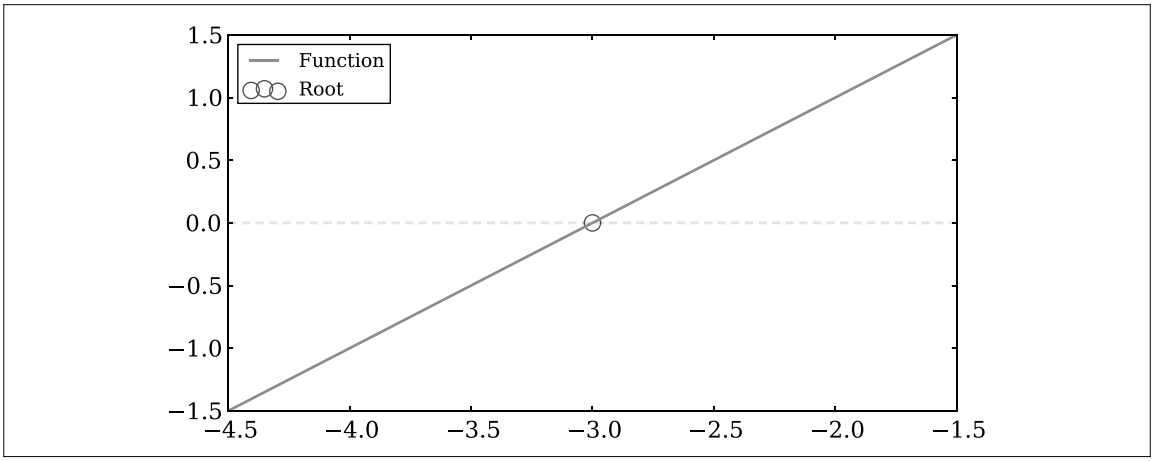


Figure 3-4. Approximate the root of a linear function at $y = 0$.

```
from scipy.optimize import fsolve
import numpy as np

line = lambda x: x + 3

solution = fsolve(line, -2)
print solution
```

Finding the intersection points between two equations is nearly as simple.[3]

```
from scipy.optimize import fsolve
import numpy as np

# Defining function to simplify intersection solution
def findIntersection(func1, func2, x0):
    return fsolve(lambda x : func1(x) - func2(x), x0)

# Defining functions that will intersect
funky = lambda x : np.cos(x / 5) * np.sin(x / 2)
line = lambda x : 0.01 * x - 0.5

# Defining range and getting solutions on intersection points
x = np.linspace(0,45,10000)
result = findIntersection(funky, line, [15, 20, 30, 35, 40, 45])

# Printing out results for x and y
print(result, line(result))
```

As we can see in Figure 3-5, the intersection points are well identified. Keep in mind that the assumptions about where the functions will intersect are important. If these are incorrect, you could get specious results.

[3] This is a modified example from *http://glowingpython.blogspot.de/2011/05/hot-to-find-intersection-of-two.html*.

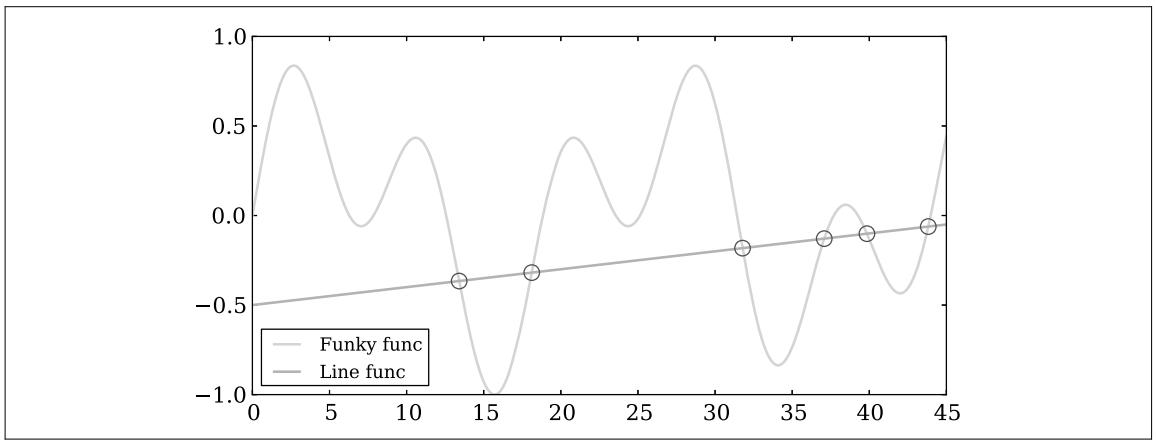


Figure 3-5. Finding the intersection points between two functions.

3.2 Interpolation

Data that contains information usually has a functional form, and as analysts we want to model it. Given a set of sample data, obtaining the intermediate values between the points is useful to understand and predict what the data will do in the non-sampled domain. SciPy offers well over a dozen different functions for interpolation, ranging from those for simple univariate cases to those for complex multivariate ones. Univariate interpolation is used when the sampled data is likely led by one independent variable, whereas multivariate interpolation assumes there is more than one independent variable.

There are two basic methods of interpolation: (1) Fit one function to an entire dataset or (2) fit different parts of the dataset with several functions where the joints of each function are joined smoothly. The second type is known as a spline interpolation, which can be a very powerful tool when the functional form of data is complex. We will first show how to interpolate a simple function, and then proceed to a more complex case. The example below interpolates a sinusoidal function (see Figure 3-6) using `scipy.interpolate.interp1d` with different fitting parameters. The first parameter is a "linear" fit and the second is a "quadratic" fit.

```
import numpy as np
from scipy.interpolate import interp1d

# Setting up fake data
x = np.linspace(0, 10 * np.pi, 20)
y = np.cos(x)

# Interpolating data
fl = interp1d(x, y, kind='linear')
fq = interp1d(x, y, kind='quadratic')

# x.min and x.max are used to make sure we do not
# go beyond the boundaries of the data for the
# interpolation.
xint = np.linspace(x.min(), x.max(), 1000)
yintl = fl(xint)
yintq = fq(xint)
```

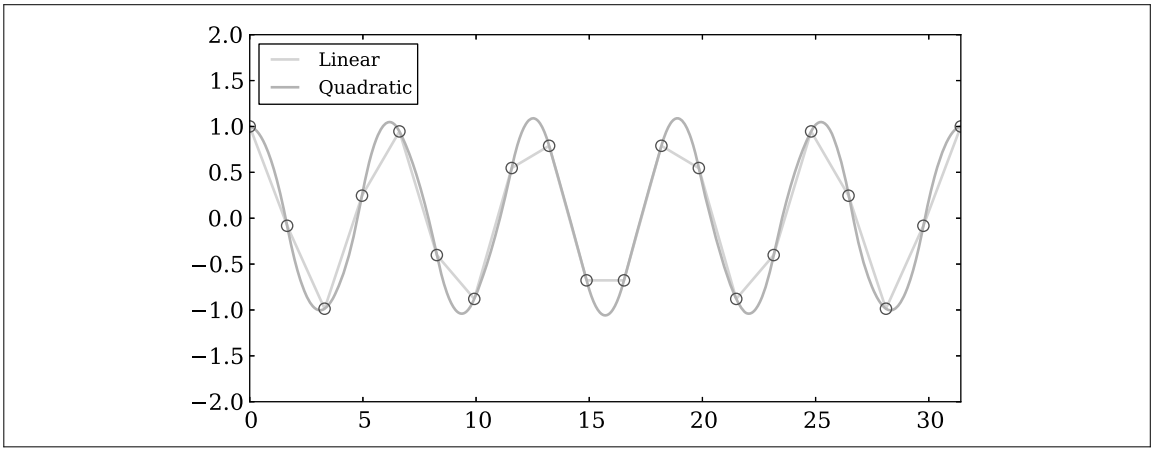


Figure 3-6. Synthetic data points (red dots) interpolated with linear and quadratic parameters.

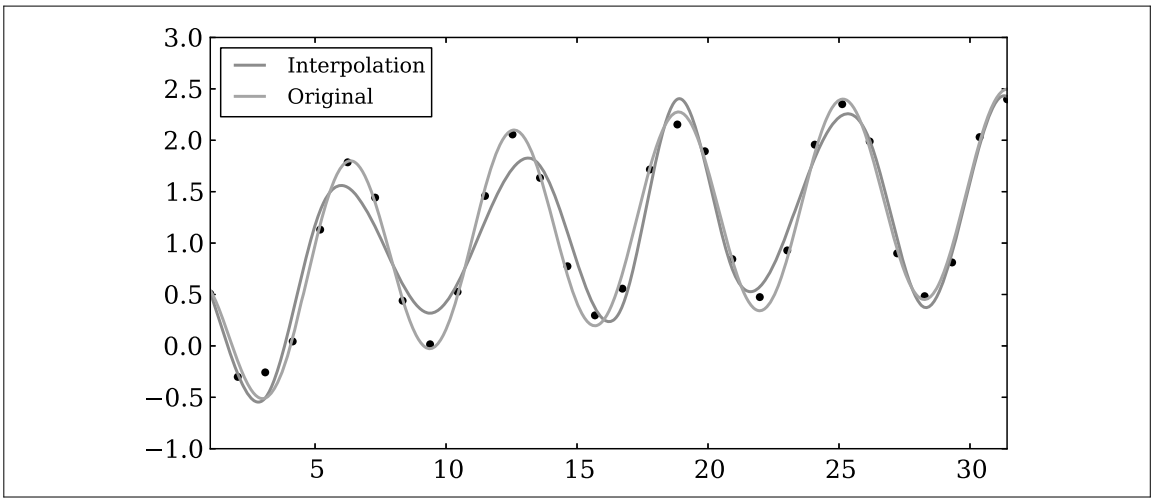


Figure 3-7. Interpolating noisy synthetic data.

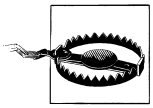


Figure 3-6 shows that in this case the quadratic fit is far better. This should demonstrate how important it is to choose the proper parameters when interpolating data.

Can we interpolate noisy data? Yes, and it is surprisingly easy, using a spline-fitting function called `scipy.interpolate.UnivariateSpline`. (The result is shown in Figure 3-7.)

```
import numpy as np
import matplotlib.pyplot as mpl
from scipy.interpolate import UnivariateSpline

# Setting up fake data with artificial noise
sample = 30
x = np.linspace(1, 10 * np.pi, sample)
y = np.cos(x) + np.log10(x) + np.random.randn(sample) / 10

# Interpolating the data
f = UnivariateSpline(x, y, s=1)
```

```
# x.min and x.max are used to make sure we do not
# go beyond the boundaries of the data for the
# interpolation.
xint = np.linspace(x.min(), x.max(), 1000)
yint = f(xint)
```

The option `s` is the smoothing factor, which should be used when fitting data with noise. If instead `s=0`, then the interpolation will go through all points while ignoring noise.

Last but not least, we go over a multivariate example—in this case, to reproduce an image. The `scipy.interpolate.griddata` function is used for its capacity to deal with unstructured *N*-dimensional data. For example, if you have a 1000 × 1000-pixel image, and then randomly selected 1000 points, how well could you reconstruct the image? Refer to Figure 3-8 to see how well *scipy.interpolate.griddata performs*.

```
import numpy as np
from scipy.interpolate import griddata

# Defining a function
ripple = lambda x, y: np.sqrt(x**2 + y**2)+np.sin(x**2 + y**2)

# Generating gridded data. The complex number defines
# how many steps the grid data should have. Without the
# complex number mgrid would only create a grid data structure
# with 5 steps.
grid_x, grid_y = np.mgrid[0:5:1000j, 0:5:1000j]

# Generating sample that interpolation function will see
xy = np.random.rand(1000, 2)
sample = ripple(xy[:,0] * 5 , xy[:,1] * 5)

# Interpolating data with a cubic
grid_z0 = griddata(xy * 5, sample, (grid_x, grid_y), method='cubic')
```

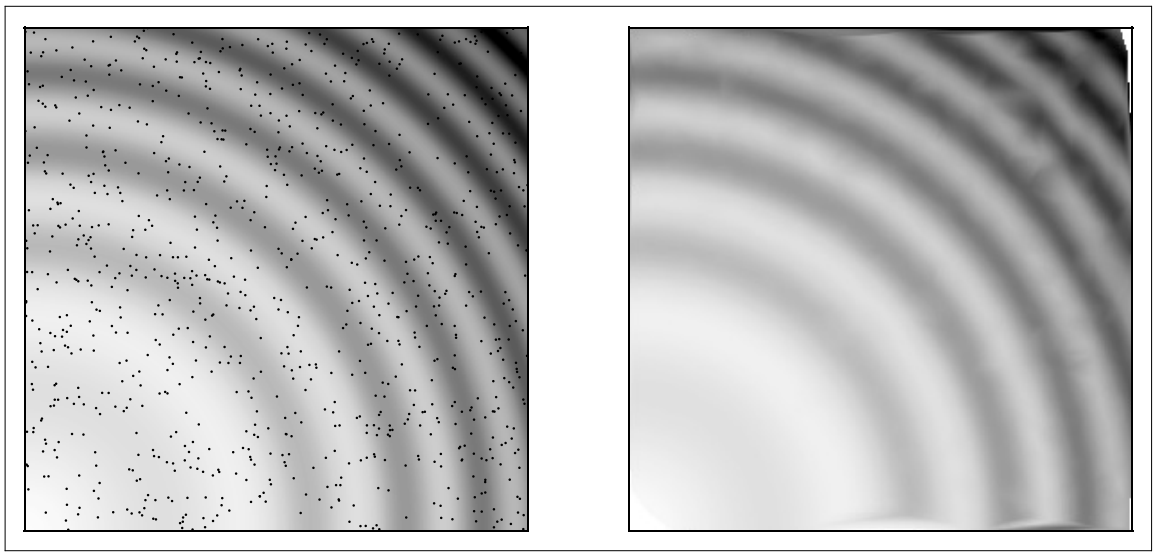


Figure 3-8. Original image with random sample (black points, left) and the interpolated image (right).

On the left-hand side of Figure 3-8 is the original image; the black points are the randomly sampled positions. On the right-hand side is the interpolated image. There are some slight glitches that come from the sample being too sparse for the finer structures. The only way to get a better interpolation is with a larger sample size. (Note that the `griddata` function has been recently added to SciPy and is only available for version 0.9 and beyond.)

If we employ another multivariate spline interpolation, how would its results compare? Here we use `scipy.interpolate.SmoothBivariateSpline`, where the code is quite similar to that in the previous example.

```
import numpy as np
from scipy.interpolate import SmoothBivariateSpline as SBS

# Defining a function
ripple = lambda x, y: np.sqrt(x**2 + y**2)+np.sin(x**2 + y**2)

# Generating sample that interpolation function will see
xy= np.random.rand(1000, 2)
x, y = xy[:,0], xy[:,1]
sample = ripple(xy[:,0] * 5 , xy[:,1] * 5)

# Interpolating data
fit = SBS(x * 5, y * 5, sample, s=0.01, kx=4, ky=4)
interp = fit(np.linspace(0, 5, 1000), np.linspace(0, 5, 1000))
```

We have a similar result to that in the last example (Figure 3-9). The left panel shows the original image with randomly sampled points, and in the right panel is the interpolated data. The `SmoothBivariateSpline` function appears to work a bit better than `griddata`, with an exception in the upper-right corner.

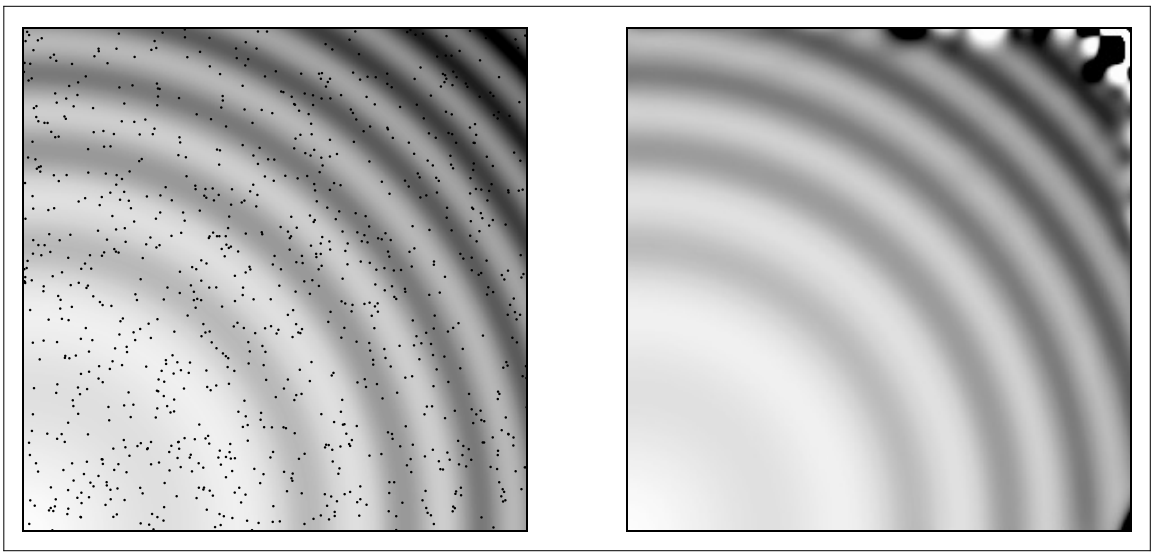
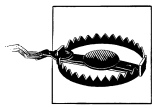


Figure 3-9. Original image with random sample (black points, left) and the interpolated image (right).



Although from the figure `SmoothBivariateSpline` does appear to work better, run the code several times to see what happens. `SmoothBivariate-Spline` is very sensitive to the data sample it is given, and interpolations can go way off the mark. `griddata` is more robust and can produce a reasonable interpolation regardless of the data sample it is given.

3.3 Integration

Integration is a crucial tool in math and science, as differentiation and integration are the two key components of calculus. Given a curve from a function or a dataset, we can calculate the area below it. In the traditional classroom setting we would integrate a function analytically, but data in the research setting is rarely given in this form, and we need to approximate its definite integral.

The main purpose of integration with SciPy is to obtain numerical solutions. If you need indefinite integral solutions, then you should look at SymPy.[4] It solves mathematical problems symbolically for many types of computation beyond calculus.

SciPy has a range of different functions to integrate equations and data. We will first go over these functions, and then move on to the data solutions. Afterward, we will employ the data-fitting tools we used earlier to compute definite integral solutions.

3.3.1 Analytic Integration

We will begin working with the function expressed below. It is straightforward to integrate and its solution's estimated error is small. See Figure 3-10 for the visual context of what is being calculated.

$$\int_0^3 \cos^2(e^x)\,\mathrm{d}x \tag{3.1}$$

```
import numpy as np
from scipy.integrate import quad

# Defining function to integrate
func = lambda x: np.cos(np.exp(x)) ** 2

# Integrating function with upper and lower
# limits of 0 and 3, respectively
solution = quad(func, 0, 3)
print solution

# The first element is the desired value
# and the second is the error.
# (1.296467785724373, 1.397797186265988e-09)
```

[4] *http://sympy.org/en/index.html*

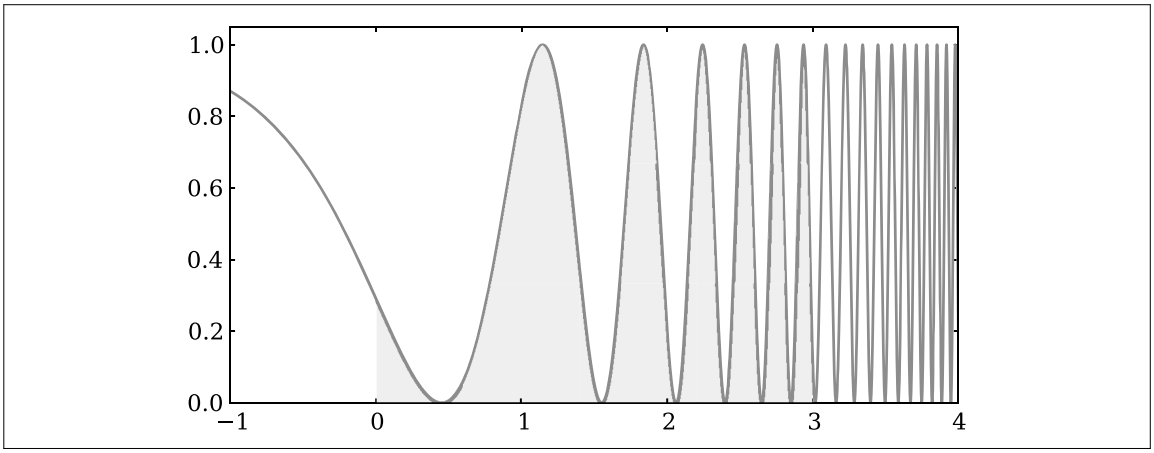


Figure 3-10. Definite integral (shaded region) of a function.

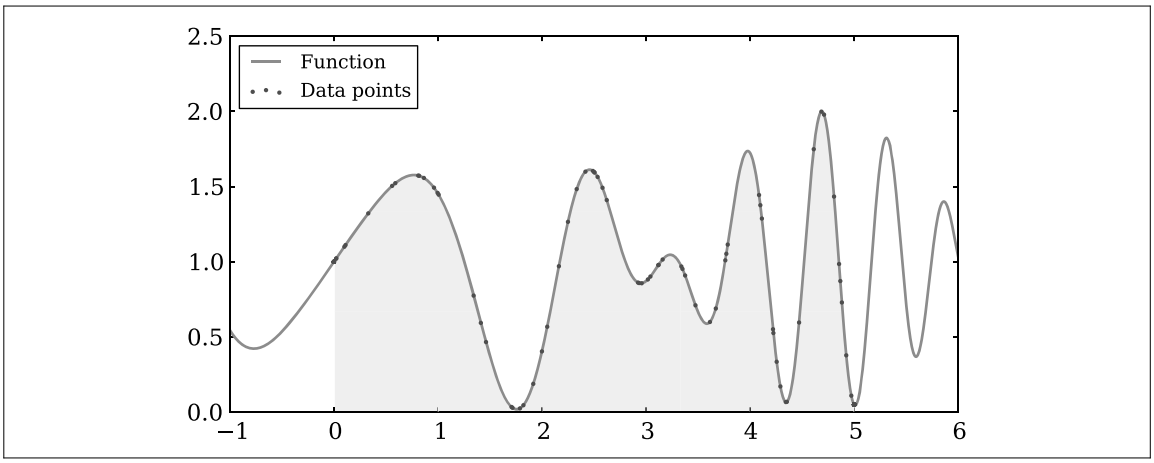


Figure 3-11. Definite integral (shaded region) of a function. The original function is the line and the randomly sampled data points are in red.

3.3.2 Numerical Integration

Let's move on to a problem where we are given data instead of some known equation and numerical integration is needed. Figure 3-11 illustrates what type of data sample can be used to approximate acceptable indefinite integrals.

```
import numpy as np
from scipy.integrate import quad, trapz

# Setting up fake data
x = np.sort(np.random.randn(150) * 4 + 4).clip(0,5)
func = lambda x: np.sin(x) * np.cos(x ** 2) + 1
y = func(x)

# Integrating function with upper and lower
# limits of 0 and 5, respectively
fsolution = quad(func, 0, 5)
dsolution = trapz(y, x=x)
```

```
print('fsolution = ' + str(fsolution[0]))
print('dsolution = ' + str(dsolution))
print('The difference is ' + str(np.abs(fsolution[0] - dsolution)))

# fsolution = 5.10034506754
# dsolution = 5.04201628314
# The difference is 0.0583287843989.
```

The `quad` integrator can only work with a callable function, whereas `trapz` is a numerical integrator that utilizes data points.

3.4 Statistics

In NumPy there are basic statistical functions like `mean`, `std`, `median`, `argmax`, and `argmin`. Moreover, the `numpy.arrays` have built-in methods that allow us to use most of the NumPy statistics easily.

```
import numpy as np

# Constructing a random array with 1000 elements
x = np.random.randn(1000)

# Calculating several of the built-in methods
# that numpy.array has
mean = x.mean()
std = x.std()
var = x.var()
```

For quick calculations these methods are useful, but more is usually needed for quantitative research. SciPy offers an extended collection of statistical tools such as distributions (continuous or discrete) and functions. We will first cover how to extrapolate the different types of distributions. Afterward, we will discuss the SciPy statistical functions used most often in various fields.

3.4.1 Continuous and Discrete Distributions

There are roughly 80 continuous distributions and over 10 discrete distributions. Twenty of the continuous functions are shown in Figure 3-12 as probability density functions (PDFs) to give a visual impression of what the `scipy.stats` package provides. These distributions are useful as random number generators, similar to the functions found in `numpy.random`. Yet the rich variety of functions SciPy provides stands in contrast to the `numpy.random` functions, which are limited to uniform and Gaussian-like distributions.

When we call a distribution from `scipy.stats`, we can extract its information in several ways: probability density functions (PDFs), cumulative distribution functions (CDFs), random variable samples (RVSs), percent point functions (PPFs), and more. So how do we set up SciPy to give us these distributions? Working with the classic normal function

$$\text{PDF} = e^{(-x^2/2)/\sqrt{2\,\pi}} \tag{3.2}$$

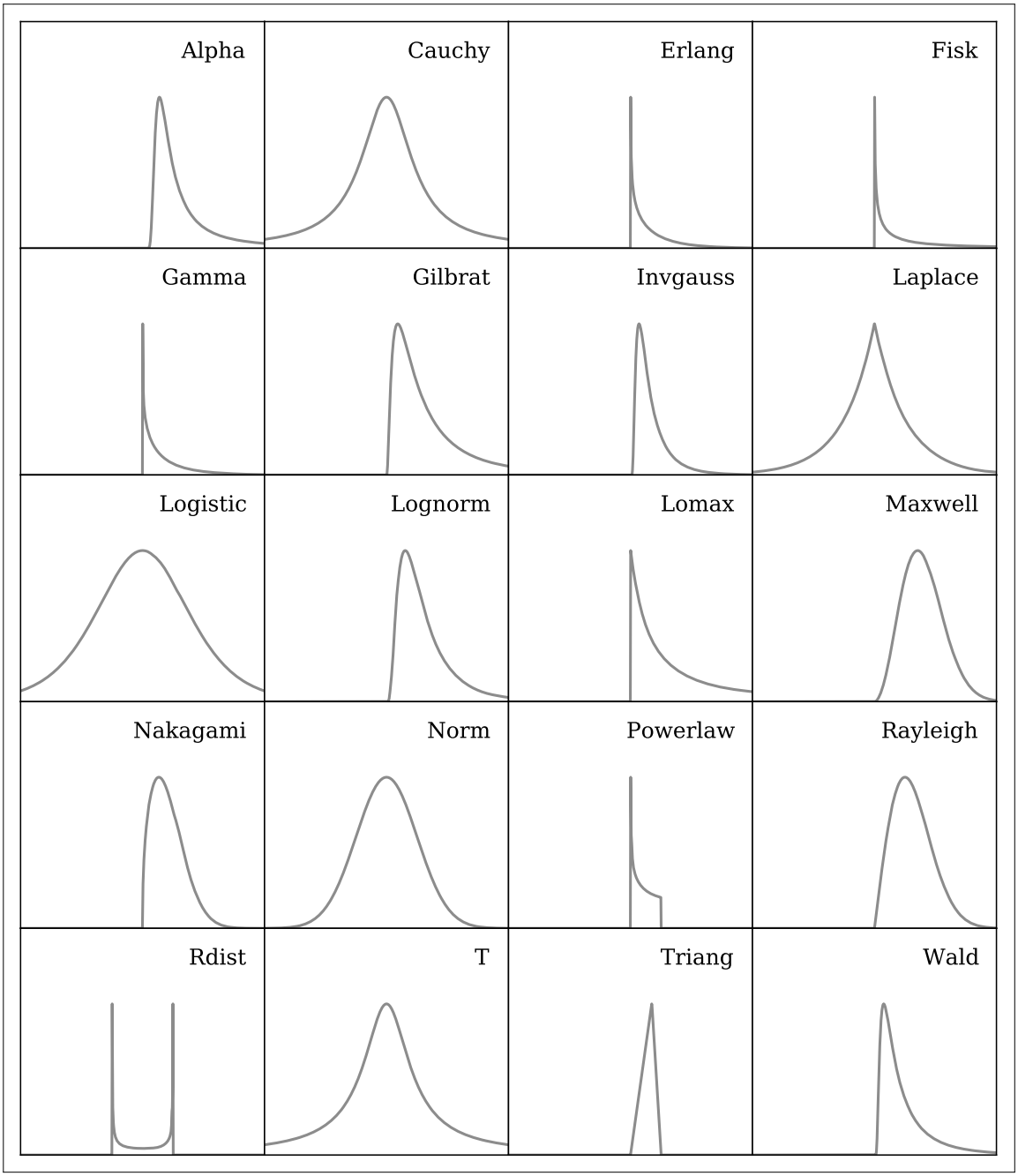


Figure 3-12. A sample of 20 continuous distributions in SciPy.

we demonstrate how to access the distribution.

```
import numpy as np
import scipy.stats import norm

# Set up the sample range
x = np.linspace(-5,5,1000)
```

```
# Here set up the parameters for the normal distribution,
# where loc is the mean and scale is the standard deviation.
dist = norm(loc=0, scale=1)

# Retrieving norm's PDF and CDF
pdf = dist.pdf(x)
cdf = dist.cdf(x)

# Here we draw out 500 random values from the norm.
sample = dist.rvs(500)
```

The distribution can be centered at a different point and scaled with the options `loc` and `scale` as shown in the example. This works as easily with all distributions because of their functional behavior, so it is important to read the documentation[5] when necessary.

In other cases one will need a discrete distribution like the Poisson, binomial, or geometric. Unlike continuous distributions, discrete distributions are useful for problems where a given number of events occur in a fixed interval of time/space, the events occur with a known average rate, and each event is independent of the prior event.

Equation 3.3 is the probability mass function (PMF) of the geometric distribution.

$$\text{PMF} = (1 - p)^{(k-1)}\, p \tag{3.3}$$

```
import numpy as np
from scipy.stats import geom

# Here set up the parameters for the geometric distribution.
p = 0.5
dist = geom(p)

# Set up the sample range.
x = np.linspace(0, 5, 1000)

# Retrieving geom's PMF and CDF
pmf = dist.pmf(x)
cdf = dist.cdf(x)

# Here we draw out 500 random values.
sample = dist.rvs(500)
```

3.4.2 Functions

There are more than 60 statistical functions in SciPy, which can be overwhelming to digest if you simply are curious about what is available. The best way to think of the statistics functions is that they either describe or test samples—for example, the frequency of certain values or the Kolmogorov-Smirnov test, respectively.

Since SciPy provides a large range of distributions, it would be great to take advantage of the ones we covered earlier. In the `stats` package, there are a number of functions

[5] *http://docs.scipy.org/doc/scipy/reference/stats.html*

such as `kstest` and `normaltest` that test samples. These distribution tests can be very helpful in determining whether a sample comes from some particular distribution or not. Before applying these, be sure you have a good understanding of your data, to avoid misinterpreting the functions' results.

```
import numpy as np
from scipy import stats

# Generating a normal distribution sample
# with 100 elements
sample = np.random.randn(100)

# normaltest tests the null hypothesis.
out = stats.normaltest(sample)
print('normaltest output')
print('Z-score = ' + str(out[0]))
print('P-value = ' + str(out[1]))

# kstest is the Kolmogorov-Smirnov test for goodness of fit.
# Here its sample is being tested against the normal distribution.
# D is the KS statistic and the closer it is to 0 the better.
out = stats.kstest(sample, 'norm')
print('\nkstest output for the Normal distribution')
print('D = ' + str(out[0]))
print('P-value = ' + str(out[1]))

# Similarly, this can be easily tested against other distributions,
# like the Wald distribution.
out = stats.kstest(sample, 'wald')
print('\nkstest output for the Wald distribution')
print('D = ' + str(out[0]))
print('P-value = ' + str(out[1]))
```

Researchers commonly use descriptive functions for statistics. Some descriptive functions that are available in the `stats` package include the geometric mean (`gmean`), the skewness of a sample (`skew`), and the frequency of values in a sample (`itemfreq`). Using these functions is simple and does not require much input. A few examples follow.

```
import numpy as np
from scipy import stats

# Generating a normal distribution sample
# with 100 elements
sample = np.random.randn(100)

# The harmonic mean:  Sample values have to
# be greater than 0.
out = stats.hmean(sample[sample > 0])
print('Harmonic mean = ' + str(out))

# The mean, where values below -1 and above 1 are
# removed for the mean calculation
out = stats.tmean(sample, limits=(-1, 1))
print('\nTrimmed mean = ' + str(out))
```

```
# Calculating the skewness of the sample
out = stats.skew(sample)
print('\nSkewness = ' + str(out))

# Additionally, there is a handy summary function called
# describe, which gives a quick look at the data.
out = stats.describe(sample)
print('\nSize = ' + str(out[0]))
print('Min = ' + str(out[1][0]))
print('Max = ' + str(out[1][1]))
print('Mean = ' + str(out[2]))
print('Variance = ' + str(out[3]))
print('Skewness = ' + str(out[4]))
print('Kurtosis = ' + str(out[5]))
```

There are many more functions available in the `stats` package, so the documentation is worth a look if you need more specific tools. If you need more statistical tools than are available here, try RPy.[6] R is a cornerstone package for statistical analysis, and RPy ports the tools available in that system to Python. If you're content with what is available in SciPy and NumPy but need more automated analysis, then take a look at Pandas.[7] It is a powerful package that can perform quick statistical analysis on big data. Its output is supplied in both numerical values and plots.

3.5 Spatial and Clustering Analysis

From biological to astrophysical sciences, spatial and clustering analysis are key to identifying patterns, groups, and clusters. In biology, for example, the spacing of different plant species hints at how seeds are dispersed, interact with the environment, and grow. In astrophysics, these analysis techniques are used to seek and identify star clusters, galaxy clusters, and large-scale filaments (composed of galaxy clusters). In the computer science domain, identifying and mapping complex networks of nodes and information is a vital study all on its own. With big data and data mining, identifying data clusters is becoming important, in order to organize discovered information, rather than being overwhelmed by it.

If you need a package that provides good graph theory capabilities, check out NetworkX.[8] It is an excellent Python package for creating, modulating, and studying the structure of complex networks (i.e., minimum spanning trees analysis).

SciPy provides a spatial analysis class (`scipy.spatial`) and a cluster analysis class (`scipy.cluster`). The spatial class includes functions to analyze distances between data points (e.g., k-d trees). The cluster class provides two overarching subclasses: vector quantization (`vq`) and hierarchical clustering (`hierarchy`). Vector quantization groups

[6] *http://rpy.sourceforge.net/*

[7] *http://pandas.pydata.org/*

[8] *http://networkx.lanl.gov/*

large sets of data points (vectors) where each group is represented by centroids. The `hierarchy` subclass contains functions to construct clusters and analyze their substructures.

3.5.1 Vector Quantization

Vector quantization is a general term that can be associated with signal processing, data compression, and clustering. Here we will focus on the clustering component, starting with how to feed data to the `vq` package in order to identify clusters.

```
import numpy as np
from scipy.cluster import vq

# Creating data
c1 = np.random.randn(100, 2) + 5
c2 = np.random.randn(30, 2) - 5
c3 = np.random.randn(50, 2)

# Pooling all the data into one 180 x 2 array
data = np.vstack([c1, c2, c3])

# Calculating the cluster centroids and variance
# from kmeans
centroids, variance = vq.kmeans(data, 3)

# The identified variable contains the information
# we need to separate the points in clusters
# based on the vq function.
identified, distance = vq.vq(data, centroids)

# Retrieving coordinates for points in each vq
# identified core
vqc1 = data[identified == 0]
vqc2 = data[identified == 1]
vqc3 = data[identified == 2]
```

The result of the identified clusters matches up quite well to the original data, as shown in Figure 3-13 (the generated cluster data is on the left and the `vq`-identified clusters are the on the right). But this was done only for data that had little noise. What happens if there is a randomly distributed set of points in the field? The algorithm fails with flying colors. See Figure 3-14 for a nice illustration of this.

3.5.2 Hierarchical Clustering

Hierarchical clustering is a powerful tool for identifying structures that are nested within larger structures. But working with the output can be tricky, as we do not get cleanly identified clusters like we do with the `kmeans` technique. Below is an example[9] wherein we generate a system of multiple clusters. To employ the hierarchy function,

[9] The original effort in using this can be found at *http://stackoverflow.com/questions/2982929/plotting-results-of-hierarchical-clustering-ontop-of-a-matrix-of-data-in-python*.

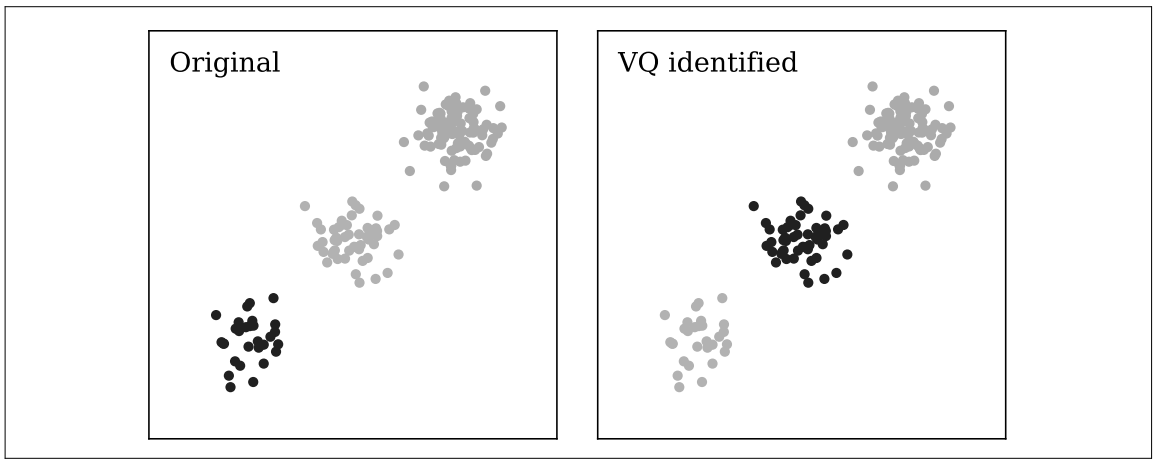


Figure 3-13. Original clusters (left) and `vq.kmeans`*-identified clusters (right). Points are associated to a cluster by color.*

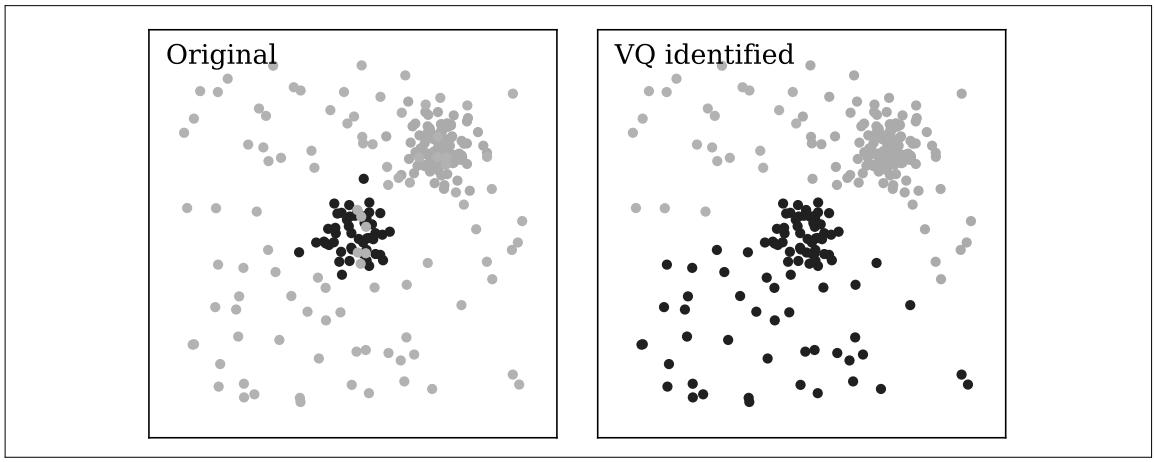


Figure 3-14. Original clusters (left) and `vq.kmeans`*-identified clusters (right). Points are associated to a cluster by color. The uniformly distributed data shows the weak point of the* `vq.kmeans` *function.*

we build a distance matrix, and the output is a dendrogram tree. See Figure 3-15 for a visual example of how hierarchical clustering works.

```
import numpy as np
import matplotlib.pyplot as mpl
from mpl_toolkits.mplot3d import Axes3D
from scipy.spatial.distance import pdist, squareform
import scipy.cluster.hierarchy as hy

# Creating a cluster of clusters function
def clusters(number = 20, cnumber = 5, csize = 10):
    # Note that the way the clusters are positioned is Gaussian randomness.
    rnum = np.random.rand(cnumber, 2)
    rn = rnum[:,0] * number
    rn = rn.astype(int)
    rn[np.where(rn < 5 )] = 5
    rn[np.where(rn > number/2. )] = round(number / 2., 0)
```

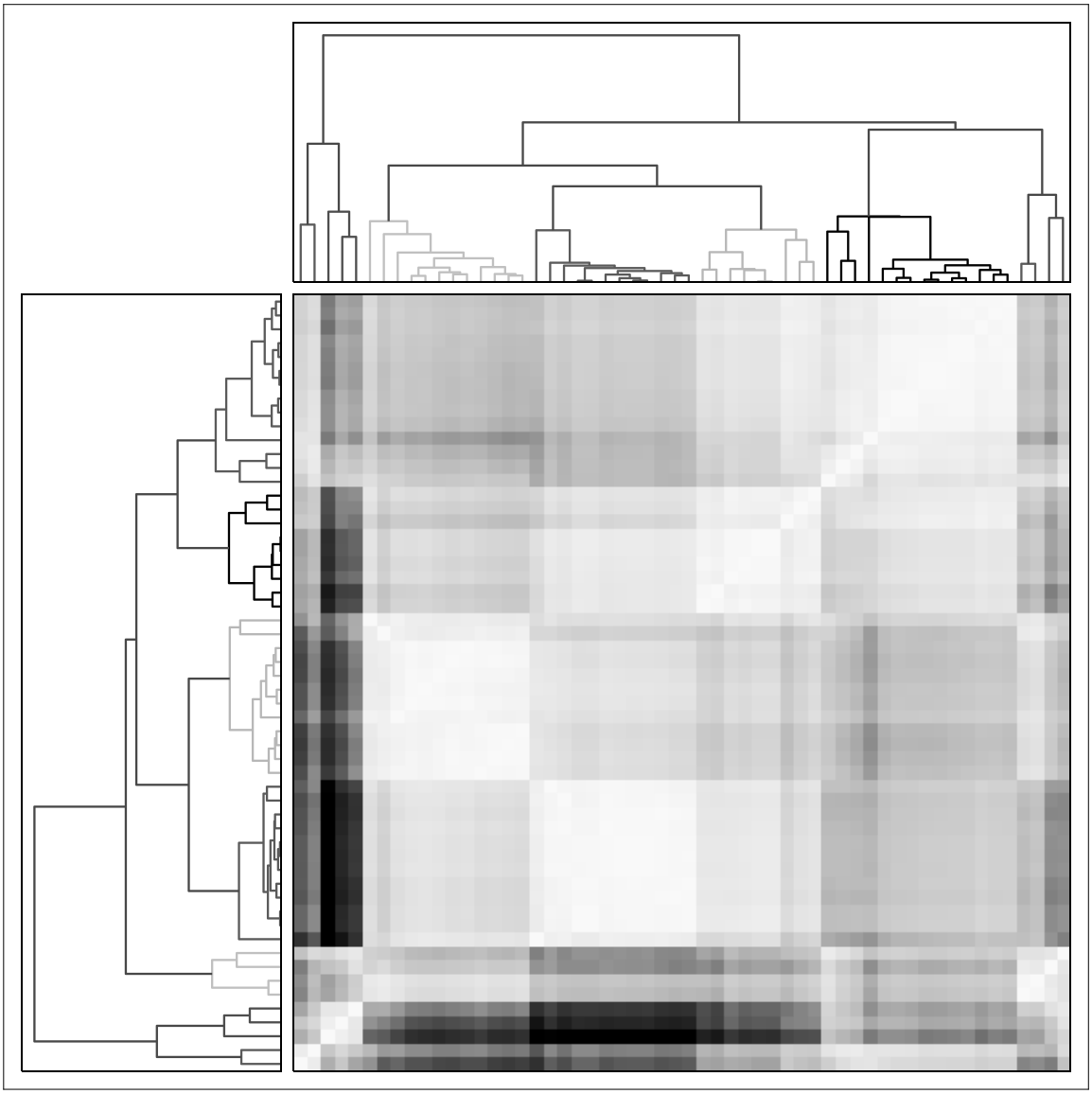


Figure 3-15. The pixelated subplot is the distance matrix, and the two dendrogram subplots show different types of dendrogram methods.

```
ra = rnum[:,1] * 2.9
ra[np.where(ra < 1.5)] = 1.5

cls = np.random.randn(number, 3) * csize

# Random multipliers for central point of cluster
rxyz = np.random.randn(cnumber-1, 3)
for i in xrange(cnumber-1):
    tmp = np.random.randn(rn[i+1], 3)
    x = tmp[:,0] + ( rxyz[i,0] * csize )
    y = tmp[:,1] + ( rxyz[i,1] * csize )
    z = tmp[:,2] + ( rxyz[i,2] * csize )
    tmp = np.column_stack([x,y,z])
    cls = np.vstack([cls,tmp])
return cls
```

```
# Generate a cluster of clusters and distance matrix.
cls = clusters()
D = pdist(cls[:,0:2])
D = squareform(D)

# Compute and plot first dendrogram.
fig = mpl.figure(figsize=(8,8))
ax1 = fig.add_axes([0.09,0.1,0.2,0.6])
Y1 = hy.linkage(D, method='complete')
cutoff = 0.3 * np.max(Y1[:, 2])
Z1 = hy.dendrogram(Y1, orientation='right', color_threshold=cutoff)
ax1.xaxis.set_visible(False)
ax1.yaxis.set_visible(False)

# Compute and plot second dendrogram.
ax2 = fig.add_axes([0.3,0.71,0.6,0.2])
Y2 = hy.linkage(D, method='average')
cutoff = 0.3 * np.max(Y2[:, 2])
Z2 = hy.dendrogram(Y2, color_threshold=cutoff)
ax2.xaxis.set_visible(False)
ax2.yaxis.set_visible(False)

# Plot distance matrix.
ax3 = fig.add_axes([0.3,0.1,0.6,0.6])
idx1 = Z1['leaves']
idx2 = Z2['leaves']
D = D[idx1,:]
D = D[:,idx2]
ax3.matshow(D, aspect='auto', origin='lower', cmap=mpl.cm.YlGnBu)
ax3.xaxis.set_visible(False)
ax3.yaxis.set_visible(False)

# Plot colorbar.
fig.savefig('cluster_hy_f01.pdf', bbox = 'tight')
```

Seeing the distance matrix in the figure with the dendrogram tree highlights how the large and small structures are identified. The question is, how do we distinguish the structures from one another? Here we use a function called `fcluster` that provides us with the indices to each of the clusters at some threshold. The output from `fcluster` will depend on the method you use when calculating the `linkage` function, such as *complete* or *single*. The cutoff value you assign to the cluster is given as the second input in the `fcluster` function. In the `dendrogram` function, the cutoff's default is `0.7 * np.max(Y[:, 2])`, but here we will use the same cutoff as in the previous example, with the scaler `0.3`.

```
# Same imports and cluster function from the previous example
# follow through here.

# Here we define a function to collect the coordinates of
# each point of the different clusters.
def group(data, index):
    number = np.unique(index)
    groups = []
    for i in number:
        groups.append(data[index == i])

    return groups
```

```
# Creating a cluster of clusters
cls = clusters()

# Calculating the linkage matrix
Y = hy.linkage(cls[:,0:2], method='complete')

# Here we use the fcluster function to pull out a
# collection of flat clusters from the hierarchical
# data structure. Note that we are using the same
# cutoff value as in the previous example for the dendrogram
# using the 'complete' method.
cutoff = 0.3 * np.max(Y[:, 2])
index = hy.fcluster(Y, cutoff, 'distance')

# Using the group function, we group points into their
# respective clusters.
groups = group(cls, index)

# Plotting clusters
fig = mpl.figure(figsize=(6, 6))
ax = fig.add_subplot(111)
colors = ['r', 'c', 'b', 'g', 'orange', 'k', 'y', 'gray']
for i, g in enumerate(groups):
    i = np.mod(i, len(colors))
    ax.scatter(g[:,0], g[:,1], c=colors[i], edgecolor='none', s=50)
    ax.xaxis.set_visible(False)
    ax.yaxis.set_visible(False)

fig.savefig('cluster_hy_f02.pdf', bbox = 'tight')
```

The hierarchically identified clusters are shown in Figure 3-16.

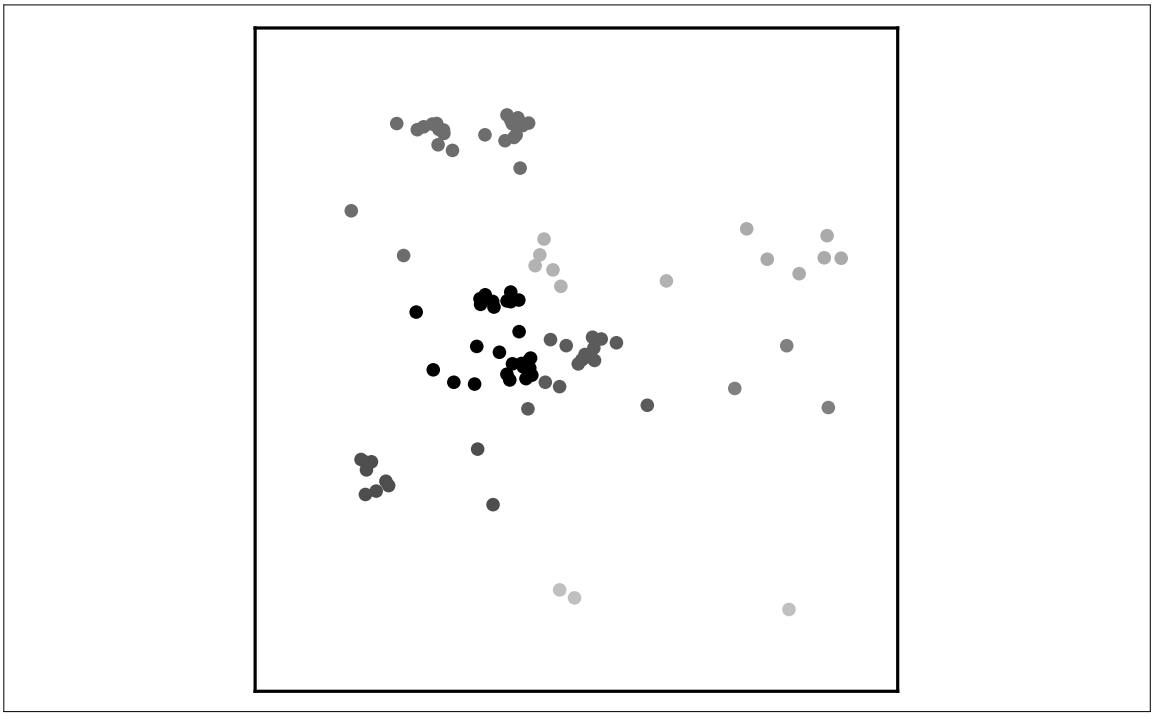


Figure 3-16. Hierarchically identified clusters.

Figure 3-17. A stacked image that is composed of hundreds of exposures from the International Space Station.

3.6 Signal and Image Processing

SciPy allows us to read and write image files like JPEG and PNG images without worrying too much about the file structure for color images. Below, we run through a simple illustration of working with image files to make a nice image[10] (see Figure 3-17) from the International Space Station (ISS).

```
import numpy as np
from scipy.misc import imread, imsave
from glob import glob

# Getting the list of files in the directory
files = glob('space/*.JPG')

# Opening up the first image for loop
im1 = imread(files[0]).astype(np.float32)

# Starting loop and continue co-adding new images
for i in xrange(1, len(files)):
    print i
    im1 += imread(files[i]).astype(np.float32)

# Saving img
imsave('stacked_image.jpg', im1)
```

[10] Original Pythonic effort can be found at *http://stackoverflow.com/questions/9251580/stacking-astronomy-images-with-python*.

Figure 3-18. A stacked image that is composed of hundreds of exposures from the International Space Station.

The JPG images in the Python environment are NumPy arrays with `(426, 640, 3)`, where the three layers are red, green, and blue, respectively.

In the original stacked image, seeing the star trails above Earth is nearly impossible. We modify the previous example to enhance the star trails as shown in Figure 3-18.

```
import numpy as np
from scipy.misc import imread, imsave
from glob import glob

# This function allows us to place in the
# brightest pixels per x and y position between
# two images. It is similar to PIL's
# ImageChop.Lighter function.
def chop_lighter(image1, image2):
    s1 = np.sum(image1, axis=2)
    s2 = np.sum(image2, axis=2)

    index = s1 < s2
    image1[index, 0] = image2[index, 0]
    image1[index, 1] = image2[index, 1]
    image1[index, 2] = image2[index, 2]
    return image1

# Getting the list of files in the directory
files = glob('space/*.JPG')

# Opening up the first image for looping
im1 = imread(files[0]).astype(np.float32)
im2 = np.copy(im1)
```

```
# Starting loop
for i in xrange(1, len(files)):
    print i
    im = imread(files[i]).astype(np.float32)
    # Same before
    im1 += im
    # im2 shows star trails better
    im2 = chop_lighter(im2, im)

# Saving image with slight tweaking on the combination
# of the two images to show star trails with the
# co-added image.
imsave('stacked_image.jpg', im1/im1.max() + im2/im2.max()*0.2)
```

When dealing with images without SciPy, you have to be more concerned about keeping the array values in the right format when saving them as image files. SciPy deals with that nicely and allows us to focus on processing the images and obtaining our desired effects.

3.7 Sparse Matrices

With NumPy we can operate with reasonable speeds on arrays containing 10^6 elements. Once we go up to 10^7 elements, operations can start to slow down and Python's memory will become limited, depending on the amount of RAM available. What's the best solution if you need to work with an array that is far larger—say, 10^{10} elements? If these massive arrays primarily contain zeros, then you're in luck, as this is the property of *sparse matrices*. If a sparse matrix is treated correctly, operation time and memory usage can go down drastically. The simple example below illustrates this.

You can determine the byte size of a `numpy.array` by calling its method `nbytes`. This can be especially useful when trying to determine what is hogging memory in your code. To do the same with `sparse` matrices, you can use `data.nbytes`.

```
import numpy as np
from scipy.sparse.linalg import eigsh
from scipy.linalg import eigh
import scipy.sparse
import time

N = 3000
# Creating a random sparse matrix
m = scipy.sparse.rand(N, N)

# Creating an array clone of it
a = m.toarray()

print('The numpy array data size: ' + str(a.nbytes) + ' bytes')
print('The sparse matrix data size: ' + str(m.data.nbytes) + ' bytes')

# Non-sparse
t0 = time.time()
```

```
res1 = eigh(a)
dt = str(np.round(time.time() - t0, 3)) + ' seconds'
print('Non-sparse operation takes ' + dt)

# Sparse
t0 = time.time()
res2 = eigsh(m)
dt = str(np.round(time.time() - t0, 3)) + ' seconds'
print('Sparse operation takes ' + dt)
```

The memory allotted to the NumPy array and sparse matrix were 68 MB and 0.68 MB, respectively. In the same order, the times taken to process the Eigen commands were 36.6 and 0.2 seconds on my computer. This means that the sparse matrix was 100 times more memory efficient and the Eigen operation was roughly 150 times faster than the non-sparse cases.

If you're unfamiliar with sparse matrices, I suggest reading *http://www.scipy.org/SciPyPackages/Sparse*, where the basics on sparse matrices and operations are discussed.

In 2D and 3D geometry, there are many sparse data structures used in fields like engineering, computational fluid dynamics, electromagnetism, thermodynamics, and acoustics. Non-geometric instances of sparse matrices are applicable to optimization, economic modeling, mathematics and statistics, and network/graph theories.

Using `scipy.io`, you can read and write common sparse matrix file formats such as Matrix Market and Harwell-Boeing, or load MatLab files. This is especially useful for collaborations with others who use these data formats. In the next section, we expand on these `scipy.io` capabilities.

3.8 Reading and Writing Files Beyond NumPy

NumPy provides a good set of input and output capabilities with ASCII files. Its binary support is great if you only need to share information to be read from one Python environment to another. But what about more universally used binary file formats? If you are using Matlab or collaborating with others who are using it, then as briefly mentioned in the previous section, it is not a problem for NumPy to read and write Matlab-supported files (using `scipy.io.loadmat` and `scipy.savemat`).

In fields like astronomy, geography, and medicine, there is a programming language called IDL. It saves files in a binary format and can be read by NumPy using a built-in package called `scipy.io.readsav`. It is a flexible and fast module, but it does not have writing capabilities.

Last but not least, you can query, read, and write Matrix Market files. These are very commonly used to share matrix data structures that are written in ASCII format. This format is well supported in other languages like C, Fortran, and Matlab, so it is a good format to use due to its universality and user readability. It is also suitable for sparse matrices.