§4 Scientific Programming with NumPy, Matplotlib, and SciPy

NumPy, **Matplotlib**, and **SciPy** have become the standard packages for performing scientific programming in Python. In Chapter 2, we have already worked with the NumPy array class `ndarray` which is frequently used for representing scientific data. Here we will first discuss other features of NumPy that are useful for scientific computing. Next, we will study Matplotlib which is the most popular Python package for plotting scientific data. Finally, we will discuss SciPy which is a library of Python modules for scientific computing that provides more specific functionality than the generic data structures and mathematical algorithms of NumPy.

4.1 NumPy

NumPy is the fundamental package for scientific computing in Python. Its core object is the object of a multidimensional array class called `ndarray` which describes a table of elements all of the same type, with an arbitrary number of dimensions as needed. Because of greater efficiency and mathematical convenience, the `ndarray` class is used frequently in scientific applications. NumPy also has other functionality that are useful for scientific computing. Here let us study the most common of these functionality one-by-one.

4.1.1 Reading and Writing an Array to a File

NumPy arrays can be saved in a platform-independent *binary* format. For instance, the command

```
np.save("myarray.npy", a)
```

will save the array `a` to the binary file `myarray.npy`. Note that the `.npy` extension is appended if it is not provided. The array can then be reloaded using NumPy on any other operating system with the command

```
a = np.load("myarray.npy")
```

where the `.npy` extension must be provided.

However, scientific data are frequently read in from a text file, which may contain comments, missing values, and blank lines. Columns of values may be either aligned in

a fixed width format or separated by one or more delimiting characters (such as spaces, tabs, or commas). Furthermore, there may be a descriptive header and even footnotes to the file, which make it hard to parse directly using Python's string methods.

NumPy provides several functions for reading data from a text file. A simple one is `loadtxt` which handles many common cases. It will return an `ndarray` class object which contains the data from the text file. The common syntax for calling `loadtxt` is

```
np.loadtxt(<fname>, dtype=float, comments="#", delimiter=None,
           skiprows=0, usecols=None, unpack=False)
```

The arguments of this function are as follows:

1. The <*fname*> argument is a filename, an open file, or a generator returning the lines of data to be read. If the filename has the extension `.gz` or `.bz2`, then the file is first decompressed. It is the only required argument of this function.
2. The `dtype` argument is the data type of the array. By default it is `float`. But it can be set explicitly by the `dtype` argument.
3. The `comments` argument is the characters or list of characters used to indicate the start of a comment. Its default value is `#`. Note that `None` implies no comments.
4. The `delimiter` argument is the string used to separate columns of data in the file. Its default value is `None`, meaning that any amount of whitespace delimits the data.
5. The `skiprows` argument is an integer giving the number of lines at the start of the file to skip over before reading the data. Its default value is `0`, i. e. no header.
6. The `usecols` argument is a sequence of column numbers indicating which columns of the file to read, with `0` being the first column. Its default value is `None`, meaning that all columns will be read.
7. The `unpack` argument is a boolean value indicating whether the returned array is transposed. Set `unpack=True` will transpose this array so that individual columns can be picked off and assigned to different variables. By default it is `False`.

Below are some examples illustrating the use of `loadtxt` for reading text files.

```
>>> import numpy as np
>>> from io import StringIO
>>> a = StringIO("1 2 \n3 4")
>>> x = np.loadtxt(a)
>>> x
```

```
array([[1., 2.],
       [3., 4.]])
>>> b = StringIO("1, 2, 3, 4\n5, 6, 7, 8")
>>> x, y, z = np.loadtxt(b, delimiter=", ", usecols=(0,1,3),
...                      unpack=True)
>>> x, y, z
(array([1., 5.]), array([2., 6.]), array([4., 8.]))
>>> c = StringIO("Header 1\n Header 2\n1 2 3\n4 5 6\n?7 8 9")
>>> x = np.loadtxt(c, dtype=int, comments="?", skiprows=2)
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> d = StringIO("F 1.53 54\nM 1.75 72")
>>> x = np.loadtxt(d, dtype={"names": ("gender", "height", "weight"),
...                          "formats": ("S1", "f4", "f4")})
>>> x
array([(b'F', 1.53, 54.), (b'M', 1.75, 72.)],
      dtype=[('gender', 'S1'), ('height', '<f4'), ('weight', '<f4')])
```

All these examples use the `loadtxt` function to read data from a `StringIO` object which behaves like a **`file`** object. In particular, the command `a = StringIO("1 2 \n 3 4")` tells Python to use `a` like a file object made up of the two lines `"1 2"` and `"3 4"` which is opened in text mode. In the last example, the `dtype` argument is set to be a dictionary item specifying that each row of data is a **structured array** in which each field has its own name and format. Moreover, the resultant output has some string literals with a prefix of `b` such as `b'F'`. Such prefix indicates that the string literal is of `byte` type rather than **`str`** type. Read NumPy documentation at **https://docs.scipy.org/doc/numpy/index.html** for the details about structured arrays and `byte` type.

A more sophisticated NumPy function for reading data from a text file is `genfromtxt`. It is similar to `loadtxt` but has a few more options and is able to cope with missing data and footers. Some of the arguments for this function are the same as for `loadtxt` including <*fname*> (the only required argument), `dtype`, `comments`, `usecols`, and `unpack`. But this function also has some arguments different from those for `loadtxt`.

Instead of the `skiprows` of `loadtxt`, the `genfromtxt` function has two optional arguments `skip_header` and `skip_footer` which gives the number of lines to skip at the

beginning and the end of the file, respectively. In addition, the delimiter argument works the same as for `loadtxt` but can also be provided as a sequence of integers giving the widths of each field to be read in where the data does not have any delimiters. For example, suppose the text file `data1.txt` is to be interpreted as consisting of four columns with widths 2, 1, 9 and 3 characters as follows:

```
 12  100.231.03
 11 1201.842.04
 11   99.324.02
```

So the first row is to be split as `" 1"`, `"2"`, `" 100.231"`,and `".03"`. Without any delimiter character, it is impossible to read this file with `loadtxt`. But this can done with `genfromtxt`:

```
>>> x = np.genfromtxt(fname="data1.txt", dtype="i4, i4, f8, f8",
...                   delimiter=[2,1,9,3])
>>> print(x)
[(1, 2,  100.231, 0.03) (1, 1, 1201.842, 0.04) (1, 1,   99.324, 0.02)]
```

If a data set is incomplete, then `loadtxt` will be unable to parse the fields with missing data into valid values for the array and will raise an exception. However, `genfromtxt` sets missing or invalid entries equal to the default values given in Table 4.1.

Table 4.1: Default filling values for missing data used by NumPy function `genfromtxt`

Data Type	Dafault Value
`bool`	`False`
`complex`	`np.nan + 0.j`
`float`	`np.nan`
`int`	`-1`
`string`	`"???"`

To illustrate how `genfromtxt` deals with missing data, consider the comma-separated file `data2.txt`:

```
10.1,4,—0.1,2
10.2,4,,0
10.3,???,,4
10.4,2,0.,
10.5,—1,???,3
```

where the missing data are indicated by empty fields and entries with `???`. Accordingly, `genfromtxt` sets the missing fields to its defaults:

```
>>> x = np.genfromtxt(fname="data2.txt", dtype="f8, i4, f8, i4",
...                   delimiter=",")
>>> print(x)
[(10.1,  4, -0.1,  2) (10.2,  4,  nan,  0) (10.3, -1,  nan,  4)
 (10.4,  2,  0. , -1) (10.5, -1,  nan,  3)]
```

The `missing_values` and `filling_values` arguments of `genfromtxt` allow closer control over which default values to be used for individual columns. If `missing_values` is given as a sequence of strings, then each string is associated with a column in the data file, in order; if it is given as a dictionary of string values, then the keys denote either column indexes (if they are integers) or column names (if they are strings). The corresponding argument `filling_values` maps these column indexes or names to default values. If `filling_values` is provided as a single value, then this value is used for missing data in all columns. For instance, the following Python code replaces the missing or invalid values in column 1 and 2 (indicated by `???`) with 999 and -99, respectively, and the missing values in column 3 with 0:

```
>>> x = np.genfromtxt(fname="data2.txt", dtype="f8, i4, f8, i4",
...                   delimiter=",",
...                   missing_values={1: "???", 2: "???"},
...                   filling_values={1: 999, 2: -99, 3: 0})
>>> print(x)
[(10.1,   4,  -0.1, 2) (10.2,   4, -99. , 0) (10.3, 999, -99. , 4)
 (10.4,   2,   0. , 0) (10.5,  -1, -99. , 3)]
```

Here the missing entry in the second column has been replaced by the value `999` instead of the default value -1. This would be particularly helpful if -1 is a valid value for this column.

Moreover, the `names` argument of `genfromtxt` allows one to set names for the columns of data read in. If it is the boolean value `True`, the names are read from the first valid line after the number of lines skipped over specified by the `skip_header` argument. But if `names` is a comma separated string of names or a sequence of strings, then those strings will be used as names. By default, `names` is `None` and the field names are taken from the `dtype` if it is given.

Let us study a realistic example to gain a clearer understanding about how the

`genfromtxt` function works. Suppose the time taken for a group of students to read out 25 randomly ordered color names, first in black ink and then in a color other than the one they name (e. g. the word "blue" in red ink), was measured in an experiment to study the Stroop effect. Below is the text file which records the experimental results in which the missing data are indicated by the character `X`.

```
Subject Number, Gender, Time (words in black), Time (words in color)
1,F,18.72,31.11
2,F,21.14,52.47
3,F,19.38,33.92
      ⋮
25,M,33.21,61.59
```

Here is the Python program `stroop.py` which reads in this data with `genfromtxt` and summarize the results.

```
# stroop.py
# This program reads in the data of the experiment on the study of
# the stroop effect from the file stroop.txt with missing values
# identified and replaced by NaN. Then it produce the statistics of
# the experimental results.
# Last Update on 6 Jan 2022 by F K Chow

import numpy as np

# Read the data from stroop.txt, identifying missing values and
# replacing them with NaN
data = np.genfromtxt("stroop.txt", skip_header=1,
                     dtype=[("student","u8"), ("gender","S1"),
                            ("black","f8"), ("color","f8")],
                     delimiter=",", missing_values="X")
nwords = 25

# Remove invalid data from the data set
filtered_data = data[np.isfinite(data["black"]) &
                     np.isfinite(data["color"])]
```

```
# Extract rows by gender (M/F) and word color (black/color) and
# normalize to time taken per word
fb = filtered_data["black"][filtered_data["gender"]==b"F"]/nwords
mb = filtered_data["black"][filtered_data["gender"]==b"M"]/nwords
fc = filtered_data["color"][filtered_data["gender"]==b"F"]/nwords
mc = filtered_data["color"][filtered_data["gender"]==b"M"]/nwords

# Produce statistics: mean and standard deviation by gender and
# word color
mu_fb, sig_fb = np.mean(fb), np.std(fb)
mu_fc, sig_fc = np.mean(fc), np.std(fc)
mu_mb, sig_mb = np.mean(mb), np.std(mb)
mu_mc, sig_mc = np.mean(mc), np.std(mc)
print("Mean and standard deviation of times per word (sec)")
print("{:^7s}|{:^15s}|{:^15s}|{:^12s}".format("gender", "black",
                                              "color", "difference"))
print("{:^7s}| {:5.3f}+/-{:5.3f} | {:5.3f}+/-{:5.3f} | {:^10.3f}"
      .format("F", mu_fb, sig_fb, mu_fc, sig_fc, mu_fc - mu_fb))
print("{:^7s}| {:5.3f}+/-{:5.3f} | {:5.3f}+/-{:5.3f} | {:^10.3f}"
      .format("M", mu_mb, sig_mb, mu_mc, sig_mc, mu_mc - mu_mb))
```

Its output shows a significantly slower per-word speed for the false-colored words than for the words in black:

```
Mean and standard deviation of times per word (sec)
gender |      black      |      color      | difference
   F   | 0.770+/-0.137 | 1.632+/-0.306 |    0.862
   M   | 0.849+/-0.186 | 1.679+/-0.394 |    0.830
```

4.1.2 Polynomials

NumPy provides a powerful set of classes for representing polynomials, including methods for evaluation, algebra, root-finding, and fitting of several kinds of polynomial basis functions. In this subsection, the power series, which is the simplest and most familiar basis, will be first introduced. After that will be a discussion of a few other classical orthogonal polynomial basis functions. Finally, we will discuss the tools provided by NumPy for

fitting and approximating data series by polynomial expansions.

A simple polynomial power series of degree n has the powers of x, i. e. $1(= x^0)$, x, x^2, x^3, ..., x^n, as its basis with coefficients c_i:

$$p(x) = c_0 + c_1 x + c_2 x^2 + c_3 x^3 + \ldots + c_n x^n$$

Here we study the `Polynomial` class provided by the NumPy `polynomial` package for representing and manipulating such series.

To import the `Polynomial` class from the `polynomial` package directly, we use the command

```
>>> from numpy.polynomial import Polynomial
```

Alternatively, if the whole NumPy library is already imported as `np`, then rather than constantly refer to this class as `np.polynomial.Polynomial`, it is convenient to define a variable:

```
>>> import numpy as np
>>> Polynomial = np.polynomial.Polynomial
```

This is the way that we will refer to the `Polynomial` class in this section.

To define a `Polynomial` object, we pass a sequence of coefficients in increasing powers of x, starting with c_0, to the `Polynomial` constructor. For instance, the polynomial

$$p(x) = 5 + 8x - 3x^2$$

is represented by the `Polynomial` object

```
>>> p = Polynomial([5, 8, -3])
```

We can inspect the coefficients of a `Polynomial` object with `print` or by referring to its `coef` attribute.

```
>>> print(p)
poly([ 5.  8. -3.])
>>> p.coef
array([ 5.,  8., -3.])
```

Notice that the integer coefficients used to define the polynomial have been automatically cast to `float`. It is also possible to use `complex` coefficients.

To evaluate a `Polynomial` object for a given value of x, we "call" it as follows:

```
>>> p(3)
2.0
```

```
>>> x = np.linspace(0, 5, 11)
>>> print(p(x))
[  5.     8.25  10.    10.25   9.     6.25   2.    -3.75 -11.   -19.75
 -30.  ]
```

The `Polynomial` class implements the familiar Python numeric operators `+`, `-`, `*`, `//`, `%`, `**`, and **`divmod`** (which returns the quotient and remainder of a division operation as a tuple) on `Polynomial` objects. To give you an idea, consider the following examples using the polynomials

$$p(x) = 12 + x - x^2 \qquad \text{and} \qquad q(x) = 4 - 2x$$

```
>>> p = Polynomial([12, 1, -1])
>>> q = Polynomial([4, -2])
>>> print(p + q)
poly([16. -1. -1.])
>>> print(p - q)
poly([ 8.  3. -1.])
>>> print(p*q)
poly([ 48. -20.  -6.   2.])
>>> print(p//q)
poly([0.5 0.5])
>>> print(p%q)
poly([10.])
>>> print(q**3)
poly([ 64. -96.  48.  -8.])
```

There are a few things to note here. First, `Polynomial` objects can only be raised to a non-negative integer power using the `**` operator. Moreover, division of `Polynomial` objects is analogous to integer division and uses the same `//` operator. That is to say, the result is the quotient polynomial which is another `Polynomial` object (with no reciprocal powers of x), possibly leaving a remainder. Thus `p//q` returns the quotient polynomial $(\frac{1}{2} + \frac{1}{2}x)$ since $p = q(\frac{1}{2} + \frac{1}{2}x) + 10$. On the other hand, the remainder of division of `Polynomial` objects (which will be another `Polynomial` object in general) is returned by the modulus operator `%`. Indeed, the built-in function **`divmod`** returns both quotient and remainder of such division in a tuple:

```
>>> quotient, remainder = divmod(p, q)
>>> print(quotient)
```

```
poly([0.5 0.5])
>>> print(remainder)
poly([10.])
```

It is not always convenient to create a new `Polynomial` object in order to use these numeric operators on one another. So many of the operators described here also work with scalars:

```
>>> p = Polynomial([12, 1, -1])
>>> print(p*3)
poly([36.  3. -3.])
>>> print(p/4)
poly([ 3.    0.25 -0.25])
```

These operators even work for tuples, lists and arrays of polynomial coefficients. For instance, we can use the following code to multiply $p(x)$ by $2x - 3x^3$:

```
>>> print(p*[0, 2, 0, -3])
poly([  0.  24.   2. -38.  -3.   3.])
```

Recall that one polynomial can be substituted into another. We simply use `p(q)` to evaluate $p(q(x))$:

```
>>> p = Polynomial([12, 1, -1])
>>> q = Polynomial([4, -2])
>>> print(p(q))
poly([ 0. 14. -4.])
```

where `p` and `q` are defined as above.

The `Polynomial` class provides a number of methods useful for dealing with polynomial power series. For example, the method `roots` returns the roots of a polynomial power series. Repeated roots are simply repeated in the returned array. As an illustration, if `p` and `q` are `Polynomial` objects defined as above, then

```
>>> p.roots()
array([-3.,  4.])
>>> (q*q).roots()
array([2., 2.])
>>> Polynomial([29, 10, 1]).roots()
array([-5.-2.j, -5.+2.j])
```

We can also use the `fromroots` method of this class to create `Polynomial` objects from the roots of a polynomial power series:

```
>>> print(Polynomial.fromroots([-3, 2, 4]))
poly([ 24. -10.  -3.   1.])
```

It implies that $(3+x)(2-x)(4-x) = 24 - 10x - 3x^2 + x^3$. Note that the coefficient of the highest power of x in a polynomial created in this way must be 1.

Polynomial power series can be differentiated with the `deriv` method of the `Polynomial` class. By default, this function returns the first-order derivative. But the optional argument `m` can be set to return the mth-order derivative:

```
>>> p = Polynomial([12, 1, -1])
>>> print(p.deriv())
poly([ 1. -2.])
>>> print(p.deriv(2))
poly([-2.])
```

Polynomial power series can also be integrated with an optional lower bound L and a constant of integration k using the `integ` method of the `Polynomial` class. To illustrate how these parameters are treated, consider the following example:

$$\int_L^x (4-2x)\,dx = \left[4x - x^2\right]_L^x = L^2 - 4L + 4x - x^2$$

$$\int (4-2x)\,dx = \left[4x - x^2\right]_L^x = k + 4x - x^2$$

By default, both L and k are set to zero. However, they can be specified by passing the arguments `lbnd` and `k` to the `integ` method:

```
>>> q = Polynomial([4, -2])
>>> print(q.integ())
poly([ 0.  4. -1.])
>>> print(q.integ(lbnd=1))
poly([-3.  4. -1.])
>>> print(q.integ(lbnd=2))
poly([-4.  4. -1.])
>>> print(q.integ(k=1))
poly([ 1.  4. -1.])
```

Polynomial power series can be integrated repeatedly by passing a value to `m` which gives the number of integrations to perform.

In addition to the `Polynomial` class representing simple power series in the form of $c_0+c_1x+c_2x^2+\ldots+c_nx^n$, NumPy `polynomial` package provides other classes to represent a series composed of any numbers of classical orthogonal polynomials. These polynomials and their linear combinations are widely used in physics, statistics, and mathematics. As of NumPy version 1.9, the polynomial classes provided for representing these polynomials are `Chebyshev`, `Legendre`, `Laguerre`, `Hermite` ("physicists' version"), and `HertmiteE` ("probabilists' version"). To illustrate the use of these polynomial classes, we consider the Legendre polynomials denoted as $P_n(x)$ which are the solutions to the Legendre differential equation

$$\frac{d}{dx}\left[(1-x^2)\frac{d}{dx}P_n(x)\right]+n(n+1)P_n(x)=0.$$

The first few Legendre polynomials are

$$P_0(x)=1, \qquad P_1(x)=x, \qquad P_2(x)=\frac{1}{2}(3x^2-1),$$
$$P_3(x)=\frac{1}{2}(5x^3-3x), \qquad P_4(x)=\frac{1}{8}(35x^4-30x^2+3),$$

and they are plotted in Figure 4.1. A useful property of the Legendre polynomials is their orthogonality on the interval $[-1,1]$:

$$\int_{-1}^{1}P_n(x)P_m(x)\,dx=\left(\frac{2}{2n+1}\right)\delta_{mn}$$

which is important in their use as a basis for representing suitable functions.

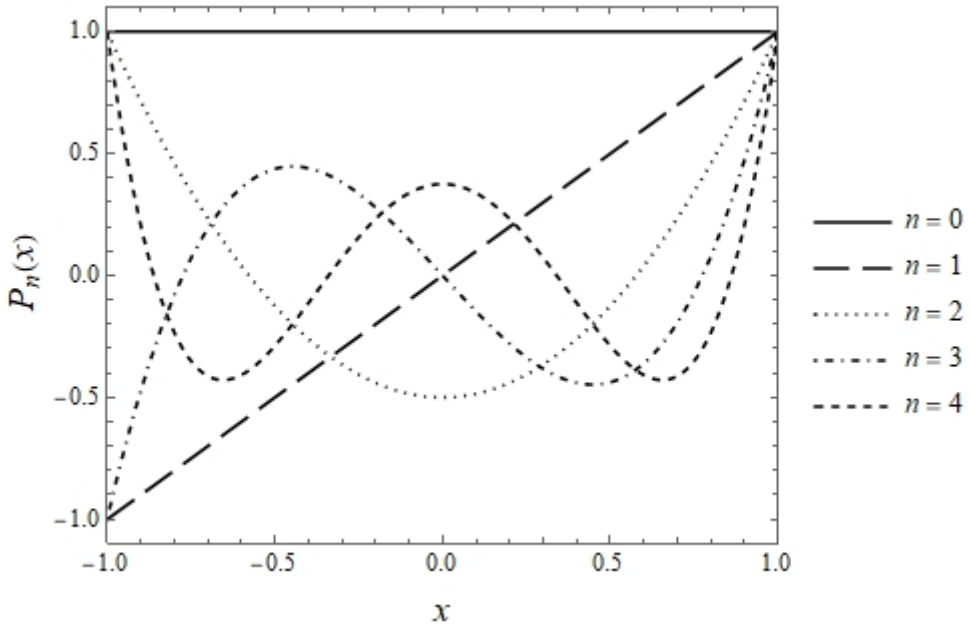

Figure 4.1: The first five Legendre Polynomials, $P_n(x)$ for $n=0,1,2,3,4$.

To create a linear combination of Legendre polynomials, pass the coefficients to the `Legendre` constructor, just like that for `Polynomial`. For example, we can use the following code to construct the polynomial expansion $3P_1(x) + 4P_2(x)$:

```
>>> Legendre = np.polynomial.Legendre
>>> A = Legendre([0, 3, 4])
```

An existing `Polynomial` object can be converted into a `Legendre` object using the `cast` method:

```
>>> A = Polynomial([1, 3, 3])
>>> B = Legendre.cast(A)
>>> print(B)
leg([2. 3. 2.])
```

In other words, $1 + 3x + 3x^2 = 2P_0 + 3P_1 + 2P_2$.

Indeed, an instance of a single Legendre polynomial can be created with the `basis` method. For instance, the input

```
>>> L4 = Legendre.basis(4)
```

results in the creation of a `Legendre` object representing $P_4(x)$, and is equivalent to calling `Legendre([0, 0, 0, 0, 1])`. To obtain a regular power series, we can cast it back to a `Polynomial`:

```
>>> print(Polynomial.cast(L4))
poly([ 0.375  0.    −3.75   0.     4.375])
```

Other than the functions just described for `Polynomial`, including differentiation and integration of polynomial series, the classes for the classical orthogonal polynomials have several useful methods. In particular, the method `convert` converts between different kinds of polynomials. For example, the linear combination $A(x) = 3P_1(x) + 4P_2(x) = 3x + 4[\frac{1}{2}(3x^2 - 1)] = -2 + 3x + 6x^2$ is represented by an instance of the `Polynomial` class as:

```
>>> A = Legendre([0, 3, 4])
>>> B = A.convert(kind=Polynomial)
>>> print(B)
poly([−2.  3.  6.])
```

Because the objects `A` and `B` represent the same underlying polynomial (just expanded in different basis sets), they evaluate to the same value when given the same x and have the same roots:

```
>>> A(-3) == B(-3)
True
>>> print(A.roots(), B.roots(), sep="\n")
[-0.87915287  0.37915287]
[-0.87915287  0.37915287]
```

A common use of polynomial expansions is in fitting and approximating data series. NumPy `Polynomial` package provide methods for the least-squares fitting of functions. Here we study the `fit` function of the NumPy polynomial classes.

Typically, a one-dimensional fitting problem requires the best-fit polynomial to a finite, continuous function over some finite region of the x-axis, i. e. the **domain**. However, polynomials themselves can differ from each other wildly and diverge as $x \to \pm\infty$. This makes any attempt to blindly find the least squares fit on the domain of the function itself potentially risky: the fitted polynomial is frequently subject to numerical instability, overflow, underflow, and other types of ill-conditioning. To give you an idea, consider the function

$$f(x) = e^{-\sin 40x}$$

in the interval $(100, 100.1)$. This function is well-behaved everywhere and takes very moderate values between e^{-1} and e^{1}. Yet a straightforward least squares fit to a fourth-order polynomial on this domain gives:

$$-7.38143 \times 10^9 + 1.47551 \times 10^8 x + 26.0634x^2 - 14740.0x^3 + 73.6611x^4$$

which is built up from difference between vast monomial terms. Obviously, it has the potential for numerical instability and loss of accuracy with even moderate values of x.

Each of the NumPy polynomial classes has a default **window** over which it is optimal to take a linear combination in fitting a function. For example, the Legendre polynomials window is the region $[-1, 1]$ plotted above, on which $P_n(x)$ are orthogonal and everywhere $|P_n(x)| \leq 1$. In fact, it is rather unlikely that the function to be fitted falls within the chosen polynomials' window. So it is necessary to relate the domain of the function to the window. This is done by shifting and scaling the x-axis, i. e. by mapping points in the function's domain to points in the fitting polynomials' window. The `fit` method of the NumPy polynomial classes does this automatically. Thus the fourth-order least squares fit to the earlier mentioned function yields

```
>>> x = np.linspace(100, 100.1, 1001)
>>> y = np.exp(-np.sin(40*x))
```

```
>>> p = Polynomial.fit(x, y, 4)
>>> print(p)
poly([ 1.49422551 −2.54641449  0.63284641  1.84246463 −1.02821956])
```

The domain and window of a polynomial can be inspected as the attributes `domain` and `window` respectively:

```
>>> p.domain
array([100. , 100.1])
>>> p.window
array([−1.,  1.])
```

It is important to note that the argument `x` is mapped from the domain to the window whenever a polynomial is evaluated. As a result, two polynomials with different domains and/or windows may evaluate to different values *even if they have the same coefficients*. As an example, we create a `Polynomial` object with the same coefficients as the fitted polynomial `p` above:

```
>>> q = Polynomial([1.49422551, −2.54641449, 0.63284641,
...                 1.84246463, −1.02821956])
```

It is created with the default domain and window, which are both $(-1, 1)$:

```
>>> print(q.domain, q.window)
[−1  1] [−1  1]
```

So evaluating `q` at 100.05, say, maps 100.05 in the domain to 100.05 in the window and gives a very different answer from the evaluation of `p` at the same point in the domain (which maps to 0 in the window):

```
>>> p(100.05), q(100.05)
(1.4942255113760086, −101176442.96772559)
```

It's easy to show that the mapping function from x in a domain (a, b) to x' in a window (a', b') is

$$x' = m(x) = \chi + \mu x, \qquad \text{where } \mu = \frac{b' - a'}{b - a},\ \chi = b' - b\left(\frac{b' - a'}{b - a}\right).$$

These parameters are returned by the NumPy polynomial classes `mapparms` method:

```
>>> chi, mu = p.mapparms()
>>> print(chi, mu)
−2001.0000000001137 20.000000000001137
>>> print(q(chi + mu*100.05))
```

```
1.49422551
```

We can even change `domain` and `window` of a polynomial by direct assignment:

```
>>> q.domain = np.array([100., 100.1])
>>> q(100.05)
1.49422551
>>> q.window = np.array([0, 2])
>>> q(100.05)
0.39490249999999993
```

To evaluate a polynomial on a set of evenly distributed points in its domain, we can use the `linspace` method of the NumPy polynomial classes. For example, if `p` is the polynomial defined above, then

```
>>> p.linspace(5)
(array([100.   , 100.025, 100.05 , 100.075, 100.1  ]),
array([1.80280222, 2.63107256, 1.49422551, 0.54527422, 0.39490249]))
```

The command `p.linspace` returns two arrays with the specified number of samples on the polynomial's domain representing the x points and the values the polynomial takes at those points, $p(x)$.

As we have seen above, the NumPy polynomial classes `fit` method returns a least-squares fitted polynomial to data `y` sampled at values `x`. In its simplest use, we only need to pass array-like objects `x` and `y` plus a value for `deg`, the degree of polynomial to fit, to `fit`. The `fit` method returns the polynomial which minimizes the sum of the squared errors

$$E = \sum_i [y_i - p(x_i)]^2$$

For example, the Python code

```
>>> x = np.linspace(300, 600, 1001)
>>> y = 1/x**3
>>> p = Polynomial.fit(x, y, 3)
```

produces the best-fit cubic polynomial to the function x^{-3} on the interval $(300, 600)$.

In addition, weighted least-squares fitting is achieved by setting the argument `w` of `fit` to a sequence of weighting values that has the same length as `x` and `y`. The polynomial returned is the one which minimizes the sum of the weighted squared errors

$$E = \sum_i w_i^2 [y_i - p(x_i)]^2$$

Remember that the domain and window of the fitted polynomial can be specified with the arguments `domain` and `window`. By default, a minimal domain covering the points `x` is used.

It is a good idea to check the *quality* of the fit before using the fitted polynomial. Setting the argument `full=True` causes the `fit` method to return two objects — the fitted polynomial and a list of various statistics about the fit itself. To illustrate how to use this argument, we consider the following example with the arrays `x` and `y` defined above:

```
>>> p, [resid, rank, sing_val, rcond] = Polynomial.fit(x, y, 3,
...                                                    full=True)
>>> p
Polynomial([ 1.07505624e-08, -1.06806897e-08,  9.41698401e-09,
-5.34787918e-09], domain=[300., 600.], window=[-1.,  1.])
>>> resid
array([4.66790796e-17])
>>> rank
4
>>> sing_val
array([1.3843828 , 1.32111941, 0.50462215, 0.28893641])
>>> rcond
2.2226664952995634e-13
```

This list can be analyzed to study how well the polynomial function fits the data. First, `resid` is the sum of the squared residual

$$resid = \sum_i [y_i - p(x_i)]^2$$

which has a smaller value for a better fit. Next, `rank` and `sing_val` are the rank and singular values of the matrix inverted in the least squares algorithm to find the coefficients of the fitted polynomial. Ill-conditioning of this matrix can lead to poor fit (in particular if the fitted polynomial degree is too high). Lastly, `rcond` is the cutoff ratio for small singular values within this matrix so that values smaller than this value are set to zero in the fit (to protect the fit from false output introduced by round-off error) and a `RankWarning` exception would be raised. If this happens, then the data may be too noisy or not well described by the polynomial of the specified degree. Note that least-squares fitting should always be carried out at double precision and beware of "over-fitting" the data, i. e. attempting to fit a function with too high degree.

4.1.3 Linear Algebra

NumPy has a `matrix` class for matrix operations which is getting deprecated. Users should use the `ndarray` class instead to carry out all the matrix operations on a regular two-dimensional NumPy array. These include scalar multiplication, matrix product, elementwise multiplication, and transpose:

```
>>> A = np.array([[1, 3], [4, 2]])
>>> A*4
array([[ 4, 12],
       [16,  8]])
>>> B = np.array([[2, 1], [3, 0]])
>>> A.dot(B)
array([[11,  1],
       [14,  4]])
>>> A*B
array([[ 2,  3],
       [12,  0]])
>>> A.transpose()
array([[1, 4],
       [3, 2]])
```

Note that the transpose returns a view of the original matrix.

We can also create an identity matrix by simply passing the two dimensions of the matrix to the NumPy method `eye`:

```
>>> np.eye(4, 4)
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

In addition, NumPy provides further methods for vector and matrix products. For instance, the NumPy methods `inner` and `outer` compute the inner and outer products of two vectors:

```
>>> u = np.array([1, 2, 3])
>>> v = np.array([2, 1, 0])
>>> np.inner(u, v)
```

```
4
>>> np.outer(u, v)
array([[2, 1, 0],
       [4, 2, 0],
       [6, 3, 0]])
```

However, raising a matrix to an (integer) power requires the method `matrix_power` from the `np.linalg` module:

```
>>> A = np.array([[1, 3], [4, 2]])
>>> np.linalg.matrix_power(A, 2)
array([[13,  9],
       [12, 16]])
```

In contrast, the $**$ operator performs elementwise exponentiation:

```
>>> A**2
array([[ 1,  9],
       [16,  4]], dtype=int32)
```

The `np.linalg` module contains a number of methods for finding the properties of a matrix or vector. For example, the method `norm` gives the norm of a matrix or vector. It is possible to calculate several different norms (see NumPy documentation for details). But the ones used by default are the Frobenius norm for two-dimensional arrays:

$$\|A\| = \sqrt{\sum_{i,j} |a_{ij}|^2}$$

and the Euclidean norm for one-dimensional arrays:

$$\|u\| = \sqrt{\sum_{i} |u_i|^2}$$

Therefore,

```
>>> A = np.array([[1, 3], [4, 2]])
>>> np.linalg.norm(A)
5.477225575051661
>>> u = np.array([1, 3j, 2+1j])
>>> np.linalg.norm(u)
3.872983346207417
```

Moreover, the `np.linalg` method `det` returns the determinant of a matrix while the regular NumPy method `trace` returns its trace:

```
>>> A = np.array([[1, 3], [4, 2]])
>>> np.linalg.det(A)
-10.000000000000002
>>> np.trace(A)
3
```

And the `np.linalg` method `matrix_rank` gives the rank of a matrix:

```
>>> np.linalg.matrix_rank(A)
2
>>> np.linalg.matrix_rank(np.array([[1, 1], [2, 2]]))
1
```

On the other hand, we can find the inverse of a square matrix by using the `inv` method from the `np.linalg` module. A `LinAlgError` exception would be raised if the matrix inversion fails:

```
>>> A = np.array([[1, 3], [4, 2]])
>>> np.linalg.inv(A)
array([[-0.2,  0.3],
       [ 0.4, -0.1]])
>>> np.linalg.inv(np.array([[1, 1], [2, 2]]))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "C:\Users\Judy.ODIN\AppData\Local\Programs\Python\Python37\lib
\site-packages\numpy\linalg\linalg.py", line 551, in inv
    ainv = _umath_linalg.inv(a, signature=signature, extobj=extobj)
  File "C:\Users\Judy.ODIN\AppData\Local\Programs\Python\Python37\lib
\site-packages\numpy\linalg\linalg.py", line 97, in _raise_linalgerro
r_singular
    raise LinAlgError("Singular matrix")
numpy.linalg.LinAlgError: Singular matrix
```

In addition, we can use the `np.linalg` method `eig` to compute the eigenvalues and eigenvectors of a general square array with shape `(n,n)`. This method returns the eigenvalues `w` as an array of shape `(n,)` and the normalized eigenvectors `v` as a complex array of shape `(n,n)`. The eigenvalues are not returned in any particular order, but the eigenvalue `w[i]` corresponds to the eigenvector `v[:,i]`. Note that the eigenvectors are arranged in columns. If the eigenvalue calculation does not converge for some reason, a `LinAlgError`

would be raised. For instance, if `A` is the matrix defined above, then

```
>>> eigvals, eigvecs = np.linalg.eig(A)
>>> eigvals
array([-2.,  5.])
>>> eigvecs
array([[-0.70710678, -0.6       ],
       [ 0.70710678, -0.8       ]])
```

If the matrix is Hermitian or real-symmetric, then the `np.linalg` method `eigh` may be used instead to find the eigenvalues and eigenvectors. This method takes an additional argument `UPLO` which can be `'L'` or `'U'` to indicate whether the lower or upper triangular part of the matrix is used. The default value is `'L'`. Besides, the `np.linalg` module provides two additional methods `eigvals` and `linalg.eigvalsh` for finding only the eigenvalues of a general and Hermitian matrix, respectively.

NumPy also provides an efficient and numerically stable method for solving systems of linear equations. The system of linear equations

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n} &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n} &= b_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn} &= b_n \end{aligned}$$

can be expressed as the matrix equation $\mathbf{Ax} = \mathbf{b}$:

$$\begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \ldots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

The solution of this system of equations (i. e. the vector `x`) is returned by the `solve` method from the `np.linalg` module. For example, the three simultaneous equations

$$\begin{aligned} 3x_1 - 2x_2 + 2x_3 &= 9 \\ x_1 - 2x_2 + x_3 &= 5 \\ 2x_1 - x_2 - 2x_3 &= -1 \end{aligned}$$

can be represented as the matrix equation $\mathbf{Ax} = \mathbf{b}$:

$$\begin{bmatrix} 3 & -2 & 2 \\ 1 & -2 & 1 \\ 2 & -1 & -2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 9 \\ 5 \\ -1 \end{bmatrix}$$

It can be solved by passing the arrays corresponding to matrix **A** and vector **b** to `np.linalg.solve`:

```
>>> A = np.array([[3, −2, 2], [1, −2, 1], [2, −1, −2]])
>>> b = np.array([9, 5, −1])
>>> np.linalg.solve(A, b)
array([ 1., −1.,  2.])
```

That is to say, the solution is $x_1 = 1$, $x_2 = -1$, $x_3 = 2$. If no unique solution exists (for nonsquare or singular matrix **A**), then a `LinAlgError` would be raised.

When a system of equations $\mathbf{Ax} = \mathbf{b}$ does not have a unique solution, a least-squares solution that minimizes the L^2 norm, i. e. $\|b - Ax\|^2$ (sum of squared residuals), may be sought by using the `np.linalg` method `lstsq`. Passed `A` and `b`, `np.linalg.lstsq` returns the solution array `x`, the sum of squared residuals, the rank of `A`, and the singular values of `A`.

A typical use of this method is to find the "line of best-fit" $y = mx + c$ for some data thought to be linearly related as in the following example. The Beer-Lambert Law relates the concentration c of a substance in a solution sample to the intensity I_t of light transmitted through the sample across a given path length l at a given wavelength λ:

$$I_t = I_0 e^{-\alpha c l}$$

where I_0 is the incident light intensity and α is the absorption coefficient at λ.

If a series of measurements of the fraction of light transmitted I_t/I_0 is given, then α may be determined through a least squares fit to the straight line:

$$y = \ln\left(\frac{I_t}{I_0}\right) = -\alpha c l.$$

This line passes through the origin as $y = 0$ for $c = 0$. However, we will fit the more general linear relationship:

$$y = mc + k$$

where $m = -\alpha l$, and verify that k is close to zero.

Given a sample with path length $l = 0.8\,\text{cm}$, the following data were measured for I_t/I_0 at five different concentrations:

c/M	0.4	0.6	0.8	1.0	1.2
I_t/I_0	0.886	0.833	0.784	0.738	0.694

where M is the molar defined by $1\,\text{M} = 1\,\text{mol/L}$.

The matrix form of the least squares equation to be solved is

$$\begin{bmatrix} c_1 & 1 \\ c_2 & 1 \\ c_3 & 1 \\ c_4 & 1 \\ c_5 & 1 \end{bmatrix} \begin{bmatrix} m \\ k \end{bmatrix} = \begin{bmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \\ T_5 \end{bmatrix}$$

where $T = \ln(I_t/I_0)$. Below is the Python program `lambertlstsq.py` which determines m and hence α from the given data using `np.linalg.lstsq`:

```
# lambertlstsq.py
# This program determines the absorption coefficient alpha from the
# least-square fitting of the given data of absorbance versus
# concentration with the Beer-Lambert Law.
# Last Update on 19 Nov 2020 by F K Chow

import matplotlib.pyplot as plt
import numpy as np

pathlen = 0.8 # Path length in cm
# Given data: concentration c in M and absorbance It/I0
c = np.array([0.4, 0.6, 0.8, 1.0, 1.2])
It_over_I0 = np.array([0.886, 0.833, 0.784, 0.738, 0.694])

# Set up the matrices for the least squares fitting
n = len(c)
A = np.vstack((c, np.ones(n))).transpose()
T = np.log(It_over_I0)

# Perform least squares fitting on the data and output the result
x, resid = np.linalg.lstsq(A, T, rcond=None)[0:2]
m, k = x
alpha = -m/pathlen
print("alpha = {:.3f} M^-1 cm^-1".format(alpha))
print("k =", k)
print("rms residual =", np.sqrt(resid[0]))
```

```
# Generate the graph of the data and the fitted line
fig, ax = plt.subplots()
ax.plot(c, T, "o")
ax.plot(c, m*c + k)
ax.set_xlabel("$c\;/\mathrm{M}$")
ax.set_ylabel("$\ln(I_\mathrm{t}/I_0)$")
plt.show()
```

Here is the output of this program:

```
alpha = 0.381 M^—1 cm^—1
k = 0.0005917197689537256
rms residual = 0.0006721659086752617
```

Note that the least squares best fit gives a value of $\alpha = 0.381\,\mathrm{M}^{-1}\,\mathrm{cm}^{-1}$ and a value of k close to zero. Figure 4.2 is the graph of the data and the fitted line generated by this program.

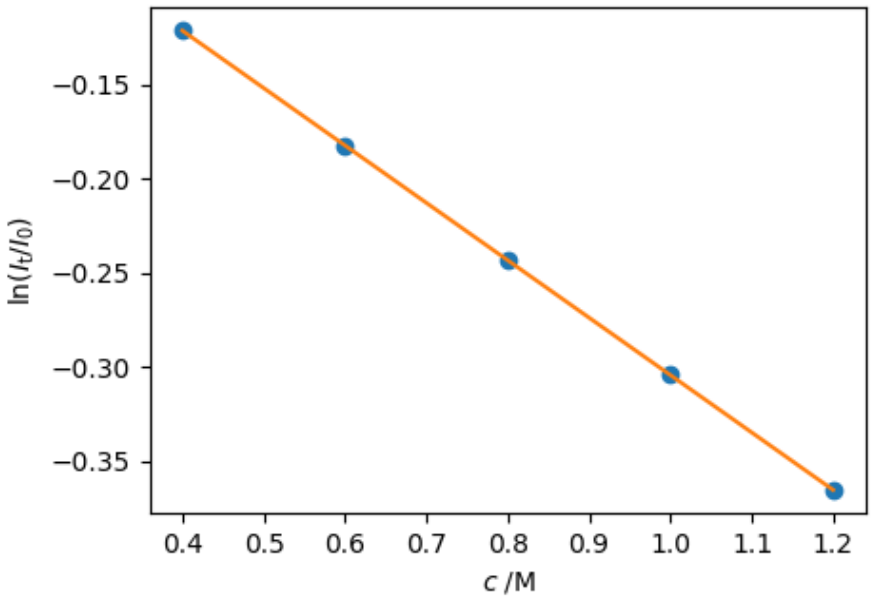

Figure 4.2: Line of least-squares best fit to absorbance $\ln(I_t/I_0)$ as a function of concentration c for path length $l = 0.8\,\mathrm{cm}$.

4.1.4 Random Sampling

NumPy `random` module provides methods for generating random numbers from several probability distributions as well as convenient tools to choose random entries from an

array and randomly shuffle the contents of an array. Just like the `random` module of the core library (see subsection 2.2.6), `np.random` uses a Mersenne Twister PRNG to generate random numbers. The way it seeds itself is operating-system dependent, but it can be reseeded with any hashable object (e. g. an immutable object such as an integer) by calling `np.random.seed`. To give you an idea, consider the `randint` method:

```
>>> np.random.seed(36)
>>> np.random.randint(1, 10, 10)
array([6, 3, 9, 2, 6, 8, 7, 5, 1, 7])
>>> np.random.randint(1, 10, 10)
array([7, 7, 2, 7, 5, 5, 4, 4, 6, 6])
>>> np.random.randint(1, 10, 10)
array([7, 9, 4, 9, 6, 6, 1, 8, 4, 9])
>>> np.random.seed(36)
>>> np.random.randint(1, 10, 10)
array([6, 3, 9, 2, 6, 8, 7, 5, 1, 7])
```

NumPy `random` module contains several methods for the generation of uniformly distributed random numbers. One of these method is `random_sample` which takes the shape of an array as its argument and creates an array of the corresponding shape filled with numbers chosen randomly from the uniform distribution over the interval $[0, 1)$, i. e. the interval between 0 and 1 inclusive of 0 but exclusive of 1:

```
>>> np.random.random_sample((2, 4))
array([[0.80533342, 0.91054136, 0.75058165, 0.04738617],
       [0.59184877, 0.76431157, 0.46489744, 0.79493874]])
```

Notice that a single random number would be returned if this method is called without an argument. If you want the numbers chosen from the uniform distribution over $[a, b)$, then you just need to do a bit of extra work:

```
>>> a, b = 20, 50
>>> a + (b - a)*np.random.random_sample((2, 4))
array([[39.44831857, 36.94426837, 34.91648232, 43.99539108],
       [23.42282576, 38.05801883, 27.23782915, 37.74752049]])
```

In a uniform distribution, each number has the same probability of being sampled which can be observed from a histogram of a large number of samples. Figure 4.3 shows a histogram of 10000 samples of uniformly distributed random number on the interval $[0, 1)$ generated by using the method `np.random.random_sample`.

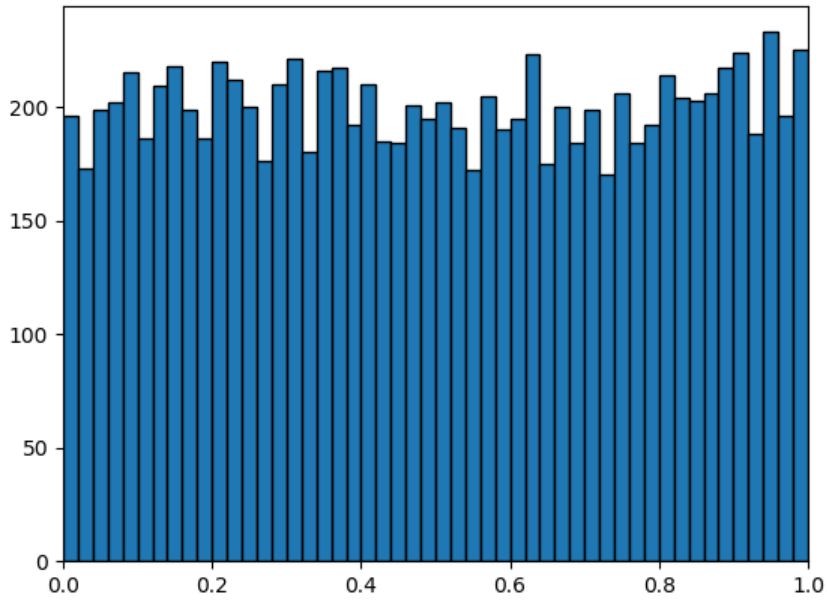

Figure 4.3: A histogram of 10000 samples drawn from the uniform distribution on $[0, 1)$ using the method `np.random.random_sample`.

The NumPy `random` module's method `rand` is similar to `random_sample`; but the dimensions of the desired array is passed as separate arguments. For example,

```
>>> np.random.rand(2, 4)
array([[0.18870802, 0.25208778, 0.62060801, 0.4020674 ],
       [0.29316198, 0.43278195, 0.75207199, 0.1528521 ]])
```

Moreover, sampling random integers is supported through a couple of methods of NumPy `random` module. The `randint` method takes up to three arguments `low`, `high`, and `size`:

1. If both `low` and `high` are supplied, then the random numbers are sampled from the discrete half-open interval [`low`, `high`).
2. If `low` is supplied but `high` is not, then the sampled interval is [`0`, `low`).
3. The argument `size` is the shape of the array of random integers desired. If it is omitted, then a single random integer is returned just like `np.random.rand` .

Here is an interactive Python session demonstrating how this method works:

```
>>> np.random.randint(5)
2
>>> np.random.randint(5, size=10)
array([3, 2, 3, 3, 4, 3, 0, 1, 3, 1])
>>> np.random.randint(5, size=[2, 5])
```

```
array([[2, 3, 2, 2, 1],
       [0, 1, 2, 0, 0]])
>>> np.random.randint(1, 5, (2, 5))
array([[3, 4, 3, 4, 4],
       [3, 2, 1, 1, 3]])
```

The `randint` method can be useful for selecting random elements (with replacement) from an array by picking random indexes:

```
>>> a = np.array([1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4])
>>> a[np.random.randint(len(a), size=5)]
array([4, 2, 1, 2, 3])
```

The method `random_integers` is another method provided by the NumPy `random` module for sampling random integers. It has the same syntax as `randint` but returns integers sampled from the uniform distribution over the closed interval [`low`, `high`] (if `high` is supplied) or [`0`, `low`] (if `high` is not supplied).

In addition, NumPy `random` module supports the sampling of a number of random distributions. The full range of these distributions is described in the official documentation. Here we describe only the normal, binomial, and Poisson distributions in details.

The normal probability distribution is described by the Gaussian function:

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right]$$

where μ is the mean and σ is the standard deviation. The NumPy `random` module's method `normal` selects random samples from the normal distribution. The mean and standard deviation are specified by the first two arguments `loc` and `scale` respectively, whose default values are 0 and 1. The shape of the returned array is specified by the third argument `size`. For example,

```
>>> np.random.normal()
1.2235400916576593
>>> np.random.normal(scale=5, size=4)
array([ 4.21458827,  1.60573179, -2.27969382, -6.96268209])
>>> np.random.normal(50, 10, size=(3, 2))
array([[48.42404291, 41.91441663],
       [54.61930328, 33.0432395 ],
       [38.63801667, 60.67570464]])
```

It is also possible to draw numbers from the standard normal distribution (which has mean $\mu = 0$ and standard deviation $\sigma = 1$) using the `np.random.randn` method. Like `np.random.rand`, it takes the dimensions of an array as its arguments:

```
>>> np.random.randn(2, 3)
array([[ 0.23518807,  0.28604119, -1.15012469],
       [-0.95799193, -0.55818598, -0.20996614]])
```

Although this method does not provide a way to set the mean and standard deviation explicitly, the standard normal distribution can be rescaled easily:

```
>>> mu, sigma = 50, 10
>>> mu + sigma*np.random.randn(3, 2)
array([[54.20233124, 53.03781867],
       [43.01191247, 44.43029967],
       [61.64222558, 48.32033123]])
```

The binomial probability distribution describes the number of particular outcomes in a sequence of n **Bernoulli trials**, i. e. n independent experiments, each of which can yield exactly two possible outcomes such as heads/tails or success/failure. If the probability of a single particular outcome (say success) is p, the probability that such a sequence of trials yields exactly k such outcomes is

$$\binom{n}{k} p^k (1-p)^{n-k}, \qquad \text{where } \binom{n}{k} = \frac{n!}{k!(n-k)!}$$

To sample from the binomial distribution described by parameters n and p, we can use the method `np.random.binomial` with n and p passed as the first and second arguments `n` and `p`. The shape of the array of samples can be again specified with the third argument `size`. For example, the probability of a fair coin coming up heads each time is $\frac{1}{2}$ when it is tossed. The probability of getting exactly three heads out of four tosses, is thus $4(\frac{1}{2})(\frac{1}{2})^3 = \frac{1}{4}$ where the factor of $\binom{4}{3} = 4$ accounts for the four possible equivalent outcomes: THHH, HTHH, HHTH, HHHT. We can obtain the expected number of heads in four tosses of the coin for ten times as follows:

```
>>> np.random.binomial(4, 0.5, 10)
array([3, 3, 1, 1, 3, 4, 1, 1, 2, 2])
```

The Poisson distribution describes the probability of a particular number of independent events occurring in a given interval of time if these events occur at a known average rate. It is also used for occurrences in specified intervals over other domains such as

distance or volume. The Poison probability distribution of the number of events k is

$$P(k) = \frac{\lambda^k e^{-\lambda}}{k!}$$

where the parameter λ is the expected average number of events occurring within the considered interval. The NumPy `random` module implements the method `poisson` to draw samples from the Poisson distribution. This method takes λ as its first argument `lam` (which defaults to 1) and the shape of the array of samples can be specified with a second argument `size` as before. For example, if I receive an average of 3.5 emails an hour, then a sample of the number of emails I receive each hour over the next 9 hours could be obtained as:

```
>>> np.random.poisson(3.5, 9)
array([ 3,  2,  4,  4,  3,  2,  5, 10,  3])
```

Besides, sometimes you may want to pick one or more values at random (with or without replacement) from a given array of values. This is the purpose of the NumPy `random` module's method `choice`. Given a single argument which is an one-dimensional sequence, it returns a random element drawn from the sequence:

```
>>> np.random.choice([−2, 3, 2, 4, −1, 4, −5, 3])
−1
>>> np.random.choice(np.arange(20))
11
```

The second argument `size` controls the shape of the array of random samples returned as before. By default, the elements of the sequence are drawn randomly with a uniform distribution and *with replacement*. To draw the sample without replacement, set the argument `replace=False`:

```
>>> a = np.array([1, 0, 4, 2, 1])
>>> np.random.choice(a, 6)
array([2, 4, 2, 1, 2, 2])
>>> np.random.choice(a, (2, 2), replace=False)
array([[1, 2],
       [4, 1]])
>>> np.random.choice(a, (2, 3), replace=False)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "mtrand.pyx", line 1165, in mtrand.RandomState.choice
```

```
ValueError: Cannot take a larger sample than population when 'replace
=False'
```

The last example shows that it's impossible to draw a larger number of elements than that in the original population if you are sampling without replacement.

To specify the probability of each element being selected, pass a sequence with the same length as the population to be sampled as the argument `p` for `choice`. Note that the probabilities should have a sum of 1. For example,

```
>>> a = np.array([1, 3, 5, 7, 9])
>>> np.random.choice(a, 5, p=[0.2, 0.1, 0.6, 0., 0.1])
array([3, 5, 5, 5, 9])
>>> np.random.choice(a, 2, False, p=[0.2, 0.1, 0.6, 0., 0.1])
array([5, 1])
```

NumPy `random` module also provides two methods for permuting the contents of an array: `shuffle` randomly rearranges the order of the elements in place whereas `permutation` makes a copy of the array first, leaving the original unchanged. Here is an interaction with the Python interpreter demonstrating how to use these methods:

```
>>> a = np.arange(8)
>>> np.random.permutation(a)
array([1, 0, 3, 7, 5, 4, 2, 6])
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7])
>>> np.random.shuffle(a)
>>> a
array([2, 5, 3, 0, 1, 7, 4, 6])
```

Note that these methods only act on the first dimension of the array:

```
>>> a = np.arange(8).reshape(2, 4)
>>> a
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
>>> np.random.permutation(a)
array([[4, 5, 6, 7],
       [0, 1, 2, 3]])
```

We can observe that only the rows of the array are permuted.

4.2 Matplotlib

Matplotlib is a large plotting package organized in a hierarchy. At the top of the hierarchy is the `pyplot` module which provides a "state-machine" interface similar to MATLAB and allows the user to add plot elements (e. g. data points, lines, and annotations) through simple function calls. In addition, `pylab` is a module that combines the functions and classes from `pyplot` and `numpy` into a single namespace. This module gets installed together with Matplotlib, but its use is now discouraged.

At a lower level of the hierarchy, Matplotlib has an object-oriented interface that allows one to create a **figure** object to which one or more **axes** objects are attached. These axes objects are then used for most plotting, annotation, and customization. This is the approach we adopt in this section. To use Matplotlib in this way, we are recommended to use the following imports:

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

Here we assume that Matplotlib has been imported in this way whenever it is used.

4.2.1 Basics of Matplotlib

The `Figure` is the top-level object of a Matplotlib plot which contains all the elements of the plot. To create a `Figure` object, call the function `figure` of the `pyplot` module. Although no arguments are necessary for this function, optional arguments can be set to customize the plot. The common arguments for `figure` are shown in Table 4.2.

Table 4.2: Common arguments to the `pyplot` function `figure`

Argument	Description
`num`	An identifier for the figure. If none is provided, then an integer starting at 1 is used and incremented with each figure created. Alternatively using a string will set the window title to that string when the figure is displayed with `pyplot` function `show`.
`figsize`	A tuple of figure width and height in inches
`dpi`	Figure resolution in dots-per-inch
`facecolor`	Figure background color
`edgecolor`	Figure border color

For instance, the following Python code

```
>>> fig = plt.figure()
>>> fig = plt.figure("Population density", figsize=(4.5, 2),
...                  facecolor="red")
```

create a default-size figure with title "Figure 1" and a small figure with title "Population density" and red background, respectively.

To really plot some data, we need to create an `Axes` object — a region of the figure containing the axes, tick-marks, labels, plot lines, markers, and so on. The simplest figure consisting of a single `Axes` object is created and returned with the input

```
>>> ax = fig.add_subplot(111)
```

The argument `111` here is a commonly used abbreviation for the tuple `(1,1,1)` specifying subplot `1` of a figure with `1` row and `1` column of subplots. The `Axes` object `ax` is the one on which we can actually plot the data by invoking various methods on this object.

In Matplotlib, we can create a line plot by calling the `plot` method on an `Axes` object. Typically, this method takes two arguments which are the lists specifying the x- and y-coordinates for the points to be plotted. The `plot` method creates a line plot with the given data in the memory of the computer, but it doesn't actually display it on the computer screen. We need to call the function `show` from the `pyplot` module in order to take the plot in the memory and display it on the screen. Here is the Python code for making a simple line plot using the `plot` method:

```
>>> x = [0.5, 1.0, 2.0, 4.0, 7.0, 10.0]
>>> y = [1.0, 2.4, 1.7, 0.3, 0.6, 1.8]
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
>>> ax.plot(x, y)
[<matplotlib.lines.Line2D object at 0x000001B7986A7A58>]
>>> plt.show()
```

It produces a new window on the screen with the graph depicted in Fig. 4.4. The computer plots the points specified by the lists on the graph and joins them up with straight lines. (Note that if only a single list is provided as the argument for `plot`, it will be taken as a sequence of y-values which are plotted at unit intervals along the x-axis.) Besides, we can see that a set of buttons appear at the bottom of the graph window (not shown in the figures here, but you will see them if you run the code). These buttons allow you to

do a number of things with the graph such as zooming in portions of the graph, moving your view around the graph, and saving the graph as an image file on your computer.

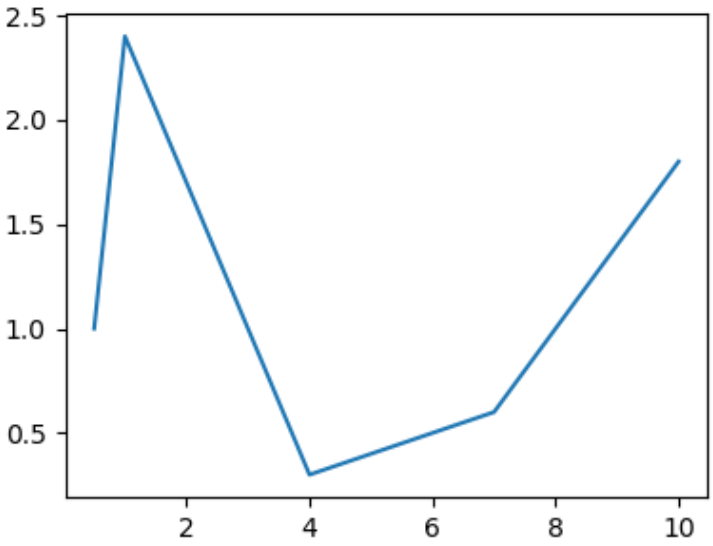

Figure 4.4: A simple line plot generated by the `Axes` class method `plot`.

Why do we need two commands `plot` and `show` to make a graph? In the above example, it would be fine to combine them into a single command that both creates a graph and shows it on the screen. However, there are more complicated situations where it is useful to have separate commands. In particular, in the cases in which two or more different curves are plotted on the same graph, we can add them by calling the `plot` method on a single `Axes` object multiple times, once for each curve. Then we call the `pyplot` function `show` once to make a graph with all these curves.

Next, we apply the `plot` method to produce a slightly more complicated graph — a graph of the function $\cos x$ from $x = 0$ to $x = 10$. To do this, we first create an array of the x-values and then take the cosines of these values to get the y-values of the points as follows:

```
>>> x = np.linspace(0, 10, 100)
>>> y = np.cos(x)
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
>>> ax.plot(x, y)
[<matplotlib.lines.Line2D object at 0x000001B798511320>]
>>> plt.show()
```

Figure 4.5 shows the graph produced by the above Python code. Beware that we have not really drawn a curve here. Our plot consists of a finite sets of points and the computer

drawn straight lines to join these points. So the end result is not actually a curve but is a set of straight-line segments. However, it looks like a cosine wave because our eyes are not sharp enough to see the slightly kinks where the segments meet. This is a useful and widely used trick for making curves in computer graphics: choose a set of sufficiently closed points that when joined together with straight lines the result looks like a curve even though it isn't.

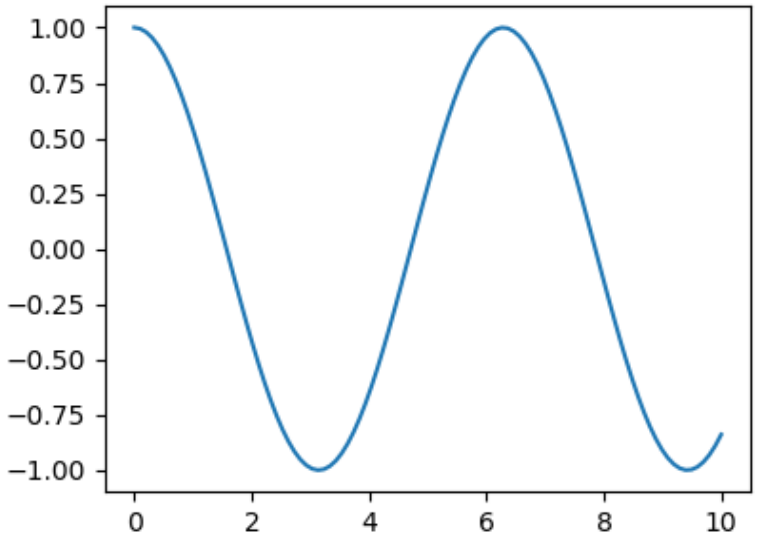

Figure 4.5: A graph of the cosine function generated by the `Axes` class method `plot`.

As another example of using the `plot` method, let's consider how to plot the experimental data in a computer file `values.txt` which is stored in two columns like this:

```
1       105.7529
2       101.7831
3       100
4       99.4528
5       98.452
...
```

We can make a graph of the data in this file (see Fig. 4.6) using the following code:

```
>>> infile = open("values.txt", "r")
>>> data = []
>>> for line in infile.readlines():
...     data.append(line.strip().split())
...
>>> x = [float(t[0]) for t in data]
>>> y = [float(t[1]) for t in data]
```

```
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
>>> ax.plot(x, y)
[<matplotlib.lines.Line2D object at 0x0000026DABF222B0>]
>>> plt.show()
```

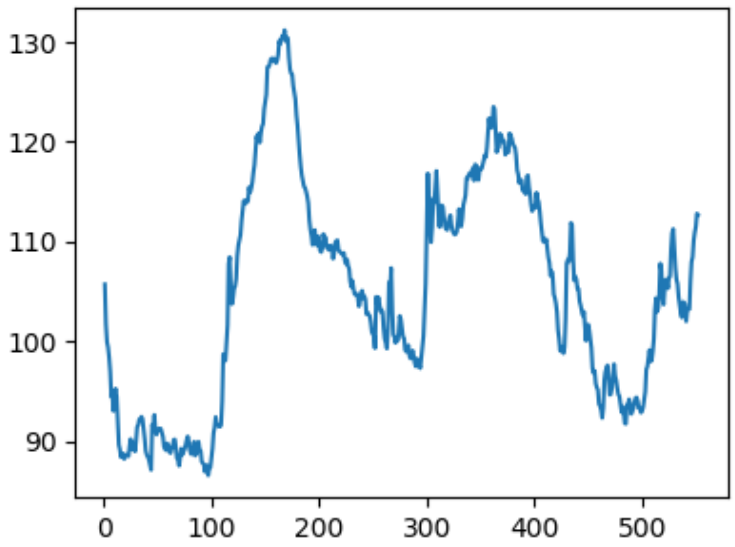

Figure 4.6: A graph of data from a file generated by the `Axes` class's method `plot`.

By default, Matplotlib runs in non-interactive mode. In such mode, the `pyplot` function `show` would block the code execution until you close all the windows with the plotted graphs. Only once you close the windows, the computer would proceed with the next line of your code. We would say that the `show` method is executed as a **blocking function** — it blocks the progress of the code until the function is done with its job. We have seen another example of a blocking function previously: the function **`input`** which collects input from the user at the keyboard. It also halts the code execution until its job is done.

Beware that the `plot` method of the `Axes` class actually returns a **`list`** of `Line2D` objects representing the plotted lines. In its simplest usage, only a single line is plotted. So this list consists of only one `Line2D` object that can be assigned to a variable. As an illustration, the following Python routine plots the function $y = \sinh(x)$ and its cubic approximation $y = x + x^3/6$ on the same graph as shown in Figure 4.7.

```
>>> x = np.linspace(-2, 2, 1000)
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
>>> line_sinh, = ax.plot(x, np.sinh(x))
```

```
>>> line_cub, = ax.plot(x, x + x**3/6)
>>> plt.show()
```

Here the syntax `line_sinh, = ...` is used to assign the returned `Line2D` object to the variable `line_sinh` instead of the `list` containing this object.

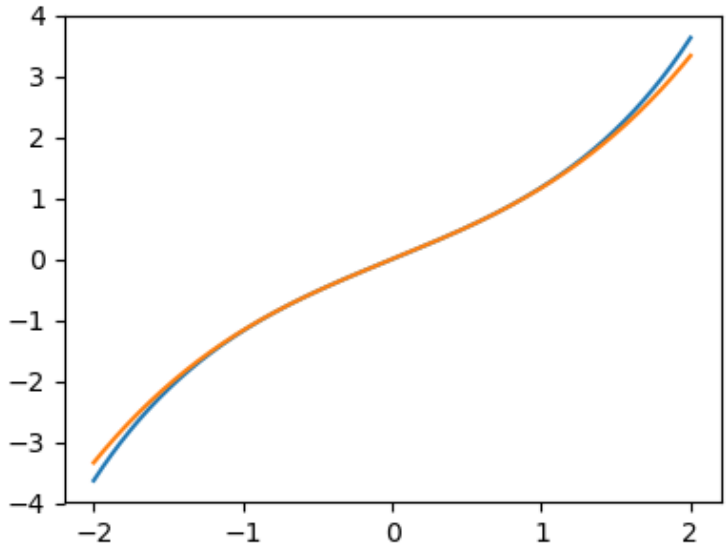

Figure 4.7: A plot of two lines produced by calling `plot` on a single `Axes` object.

Table 4.3: Common Matplotlib marker styles

Code	Marker	Description
`.`	.	Point
`o`	○	Circle
`+`	+	Plus
`x`	×	Cross
`D`	◇	Diamond
`v`	▽	Downward Triangle
`^`	△	Upward Triangle
`s`	□	Square
`*`	★	Star

We can change the style of a line plot generated by the `Axes` class method `plot` by passing extra arguments to `plot`. By default, `plot` produces a line plot without any markers — symbols printed at each point on the plot. To specify the markers, we add an extra argument `marker` to `plot`. A number of markers are available in Matplotlib and are documented online at **https://matplotlib.org**. Table 4.3 lists the single character string codes for some common marker styles. For example, the command

```
ax.plot(x, y, marker="s")
```

plots a line with squares as the markers. In addition, other marker properties can be set with the arguments listed in Table 4.4. Read the official documentation for the details on how to further customize the markers in Matplotlib graphics.

Table 4.4: Matplotlib marker properties

Argument	**Abbreviation**	**Description**
`markersize`	`ms`	Marker size in points
`markerfacecolor`	`mfc`	Fill color of the marker
`markeredgecolor`	`mec`	Edge color of the marker
`markeredgewidth`	`mew`	Edge width of the marker in points
`markevery`		Set to a positive integer N to print a marker every N points; default is `None` which prints a marker for each point

Table 4.5: Common Matplotlib line styles

Code	**Line style**
	No line
`-`	Solid
`--`	Dashed
`:`	Dotted
`-.`	Dash-dot

For the line plots generated by `plot`, the default plot style is a solid line of thickness 1.0 pt in a color determined by the order in which it is added to the plot. We can select other styles for the line from the predefined options with the `linestyle` argument. Possible string values to pass to this argument are listed in Table 4.5. Note that no line would be drawn if one set `linestyle = ""` for an empty string. Further customization of the line style is possible by setting the `dashes` argument to a sequence of values describing the repeated dash pattern in points. For example, `dashes=[2, 4, 8, 4, 2, 4]` represents a repeated pattern of dot (2 pts), space (4 pts), dash (8 pts), space (4 pts), dot (2 pts), space (4 pts) as the line style. Alternatively, one can call the `set_dashes` method on a plotted line as follows:

```
>>> x = np.linspace(0, 2*np.pi, 1000)
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
>>> line, = ax.plot(x, np.cos(x))
>>> line.set_dashes([2, 4, 8, 4, 2, 4])
```

```
>>> plt.show()
```

Table 4.6: Code letters and names for Matplotlib basic colors

Code	Name
`b`	blue
`g`	green
`r`	red
`c`	cyan
`m`	magenta
`y`	yellow
`k`	black
`w`	white

We can also specify the thickness of a plotted line in points by passing a **`float`** to the `linewidth` argument of `plot`. Moreover, line colors are specified with the `color` argument. Here are the common formats for specifying colors with this argument:

1. string: a code letter representing a basic color.
2. string: a X11/CSS4 color name (case-insensitive), e. g. "blueviolet" and "darkorange" (see Matplotlib documentation for the full list of these color names).
3. string: a HTML 6-digit hex-string preceded by `#` giving the red, green, blue (RGB) components of the color, e. g. **"`#ff00ff`"** is magenta.
4. string: a string representation of a `float` between 0 (black) and 1 (white) inclusive for gray level, e. g. `0.5` gives a gray level half between black and white.
5. tuple: a tuple of three `float` values in the range of 0 to 1 giving the RGB components, e. g. `(0., 0., 0.5)` is a dark blue color.

Table 4.6 lists the code letters and names for the basic colors in Matplotlib. For example, the following Python code produces the graph as shown in Fig. 4.8.

```
>>> x = np.linspace(0.1, 1, 100)
>>> yi, ye = 1./x, 10.*np.exp(-2.*x)
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
>>> line1, = ax.plot(x, yi, color="r", linestyle=":", linewidth=4.)
>>> line2, = ax.plot(x, ye, color="m", linestyle="-.", linewidth=2.)
>>> plt.show()
```

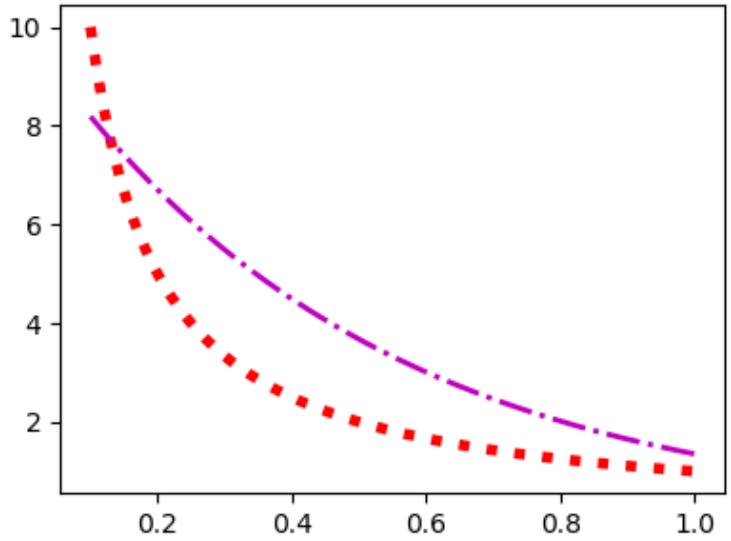

Figure 4.8: Two different line styles on the same graph produced by the Axes class method `plot`.

For convenience, we can use the abbreviations `c` for `color`, `ls` for `linestyle`, and `lw` for `linewidth` to specify the properties of plotted lines. For instance, the Python statement

```
ax.plot(x, y, c="g", ls="-.", lw=2)
```

draws a green dash-dot line of thickness 2 pt in the line plot. It is also possible to specify the color, linestyle and marker style in a single string such as

```
ax.plot(x, y, "r:^")
```

which specifies the plot style to be a red dotted line with triangle markers. In addition, multiple lines can be plotted using a sequence of `x`, `y`, format arguments. For example,

```
ax.plot(x, y1, "r:", x, y2, "k-.")
```

plots a red dotted line for `(x, y1)` and a black dash-dot line for `(x, y2)`.

By default, Matplotlib sets the axis limits in the generated plot to include all the given data. To set the axis limits to specific values, use the `Axes` class methods `set_xlim` and `set_ylim`. Either both limits or an individual limit can be set with the arguments `left`, `right` (or `xmin`, `xmax`) for `set_xlim` and `bottom` and `top` (or `ymin`, `ymax`) for `set_ylim`. Unspecified limits remain unchanged. Note that `left` precedes `right` while `bottom` precedes `top`. To give you an idea, consider the Python code:

```
>>> x = np.linspace(-3, 3, 1000)
>>> y = x**3 - 3*x**2 + 3*x - 1
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
```

```
>>> ax.plot(x, y)
[<matplotlib.lines.Line2D object at 0x0000026DABF222B0>]
>>> ax.set_xlim(-2, 2)
(-2, 2)
>>> ax.set_ylim(bottom=0)
(0, 11.6)
>>> plt.show()
```

It plots the function $y = x^3 - 3x^2 + 3x - 1$ for $-2 \leq x \leq 2$ with the lower limit of y set to 0. If `bottom` is greater than `top` or `right` is less than `left`, then the corresponding axis will be reversed, i. e. the values on this axis will *decrease* from `bottom` to `top` or from `left` to `right`.

It's a good practice to label the axes of your graphs so that anyone knows what they represent. You can add labels to the x- and y-axes of a subplot by calling `set_xlabel` and `set_ylabel` on the corresponding `Axes` object. In addition, you can add legend label to a plotted line by passing the `label` argument to the `plot` method call on your `Axes` object. However, the legend itself would not appear on the plot unless `legend` is called on this object (e. g. with `ax.legend()`). The appearance of the legend itself can be customized extensively by passing additional arguments to `legend`. But the most common one that you may want to pass is `loc` which defines the location of the legend on the plot. Table 4.7 lists the legend location specifier for this argument.

Table 4.7: Legend location specifiers for the `loc` argument of the method `legend`

String	Integer
`"best"`	0
`"upper right"`	1
`"upper left"`	2
`"lower right"`	3
`"lower left"`	4
`"right"`	5
`"center left"`	6
`"center right"`	7
`"lower center"`	8
`"upper center"`	9
`"center"`	10

There are two types of title you may want to give to your figure: the `Figure` class method `suptitle` adds a centered title to the entire figure which may contain more than

one subplot whereas the `Axes` class method `set_title` adds a title to a single subplot. As an illustration, below is the Python program `marriageages.py` which plots lines in which legend labels are added for the lines and custom markers are used for the data points themselves as depicted in Figure 4.9.

```
# marriageages.py
# This program plots median age at first marriage in the United
# States during 1890–2010.
# Last Update on 19 Nov 2020 by F K Chow

import matplotlib.pyplot as plt
import numpy as np

# Years
year = np.linspace(1890, 2010, 13, dtype=int)
# Median ages of men at first marriage
age_m = [26.1, 25.9, 25.1, 24.6, 24.3, 24.3, 22.8, 22.8, 23.2, 24.7,
         26.1, 26.8, 28.2]
# Median ages of women at first marriage
age_f = [22.0, 21.9, 21.6, 21.2, 21.3, 21.5, 20.3, 20.3, 20.8, 22.0,
         23.9, 25.1, 26.1]

# Plot median ages with male or female symbols as markers
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(year, age_m, marker="$\u2642$", markersize=14, c="b", lw=2,
        mfc="b", mec="b", label="Male")
ax.plot(year, age_f, marker="$\u2640$", markersize=14, c="m", lw=2,
        mfc="m", mec="m", label="Female")
ax.set_xlim(1880,2020)
ax.set_xlabel("Year")
ax.set_ylabel("Age")
ax.set_title("Median Age at First Marriage in the US during 1890–2010")
ax.legend(loc = "upper left")
plt.show()
```

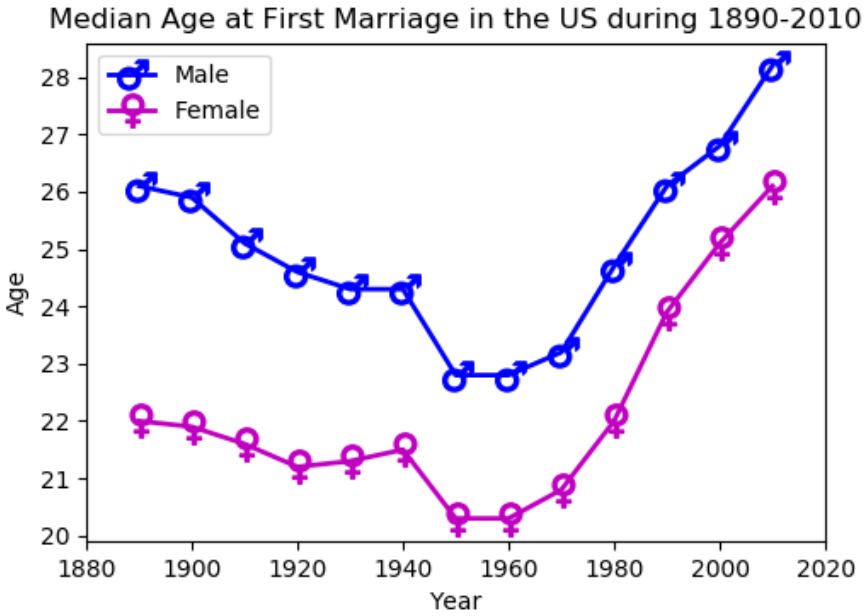

Figure 4.9: Median age at first marriage in the US for 1890-2010.

Each text element of a Matplotlib plot like titles and axis labels can be customized with the arguments given in Table 4.8. For example, we can customize the title by

```
ax.set_title("Title", fontsize=18, fontstyle="italic", color="b")
```

To use the same font properties for all text elements, it is easier to set the setting of the Matplotlib function `rc` using a dictionary of values. This involves a separate import from `matplotlib` first. For instance, the following Python code specifies that all text elements will be rendered in 22-point, bold serif in the subsequent plots.

```
>>> from matplotlib import rc
>>> font_prop = {"family" : "serif", "weight" : "bold", "size" : 22}
>>> rc("font", **font_prop)
```

Table 4.8: Common font property arguments for text elements of a Matplotlib plot

Argument	Description
`fontfamily`	The font family (e. g. `"sans-serif"` and `"monospace"`)
`fontsize`	The font size which is either an absolute size in points (e. g. `12`) or a size string relative to the default size (e. g. `"small"` and `"medium"`)
`fontstyle`	The font style (e. g. `"normal"` and `"italic"`)
`fontweight`	The font weight (e. g. `"normal"` and `"bold'`)
`color` or `c`	The font color which is any Matplotlib color specifier (e. g. `"m"` and `"#ff00ff"`)

By default, Matplotlib plots data on a linear scale. To set a logarithmic scale, call one or both of the followings on your `Axes` object:

```
ax.set_xscale("log")
ax.set_yscale("log")
```

Base-10 logarithms are used by default. But the base can be set with the optional arguments `basex` or `basey`. Nonpositive values in the data will be masked as invalid by default. If you want negative values to be handled "symmetrically" with positive ones such that $\log(-|x|) = -\log(|x|)$, then you should use `symlog` instead of `log`.

Grid lines are vertical lines (for the x-axis) and horizontal lines (for the y-axis) lines running across the plot to aid with locating the numerical values of data points. By default, no grid lines are drawn by Matplotlib. But grid lines may be turned on by calling the `grid` method on an `Axes` object to add both horizontal and vertical grid lines or on the `xaxis` or `yaxis` objects of an `Axes` object to select the grid lines to be added. For example, the Python statements

```
ax.xaxis.grid(True)
ax.yaxis.grid(True)
```

turn on vertical and horizontal grid lines, respectively. The line properties of the grid lines are set with arguments such as `linestyle`, `linewidth`, `color`, and so on just like that for plotted lines. Notice that there exist two kinds of grid lines corresponding to the major and minor tick marks. These can be selected with the `which` argument which takes the values `"major"`, `"minor"` and `"both"`. If it is not specified, then `which` takes the default value `"major"`. For example, the command

```
ax.yaxis.grid(True, which="minor", c="g")
```

selects minor green grid lines for the y-axis.

Matplotlib does a good job in labeling **tick marks** on each axis appropriately. But sometimes you want to customize them such as labeling the tick marks differently. For instance, one may want to set the tick mark values to a given sequence of values in a subplot. This is accomplished by calling the methods `set_xticks` and `set_yticks` on the `Axes` object of the subplot like

```
ax.set_xticks([0, 1, 2.5, 5.5, 12])
ax.set_yticks([10, 20, 50, 100, 200])
```

Notice that the ticks need not be evenly spaced. As mentioned before, there are two kinds of ticks in a Matplotlib plot: major ticks and minor ticks. Only major ticks are turned

on by default. The smaller and more frequent minor ticks can be easily enabled with

`ax.minorticks_on()`

The `xaxis` and `yaxis` attributes of an `Axes` objects also have a method `set_ticks_position` which takes a single argument to determine where the ticks appear: `"top"`, `"bottom"`, `"both"` (the default) or `"none"` for `xaxis`; `"left"`, `"right"`, `"both"` (the default) or `"none"` for `yaxis`.

We can also pass a sequence of strings of a suitable length to the `Axes` class methods `set_xticklabels` and `set_yticklabels` to replace the actual numbered labels. Beware that the length of the sequence of tick labels must be same as that of the list of tick values required. Besides, to remove all the tick labels, we just set them to an empty list using the command `ax.set_yticklabels([])`. This retains the tick marks themselves. However, if you want neither tick marks nor tick labels on the axis, then you should use the command `ax.set_yticks([])`. Below is the Python program `expdecay.py` which demonstrates how to customize tick marks and tick labels. The resultant plot is shown in Figure 4.10.

```
# expdecay.py
# This program plots the exponential decay labeled by lifetimes such
# that after each lifetime its value falls by a factor of e.
# Last Update on 19 Nov 2020 by F K Chow

import matplotlib.pyplot as plt
import numpy as np

# Initial value of y at t=0 and lifetime in s
N, tau = 10000, 28
# Maximum time to consider (s)
tmax = 100
# A suitable grid of time point and the exponential decay itself
t = np.linspace(0, tmax, 1000)
y = N*np.exp(-t/tau)

# Plot the exponential decay with customized tick labels
fig = plt.figure()
ax = fig.add_subplot(111)
```

```
ax.plot(t, y)
ax.set_xlabel(r"$t\;/\mathrm{s}$")
ax.set_ylabel(r"$y$")
ntau = tmax // tau + 1
xticks = [i*tau for i in range(ntau)]
yticks = [N*np.exp(-i) for i in range(ntau)]
ax.set_xticks(xticks)
ax.set_yticks(yticks)
xtick_labels = [r"$0$", r"$\tau$"] + [r"${}\tau$".format(k) for k in
                                      range(2,ntau)]
ax.set_xticklabels(xtick_labels)
ytick_labels = [r"$N$", r"$N/e$"] + [r"$N/{}e$".format(k) for k in
                                     range(2,ntau)]
ax.set_yticklabels(ytick_labels)
ax.grid()
plt.show()
```

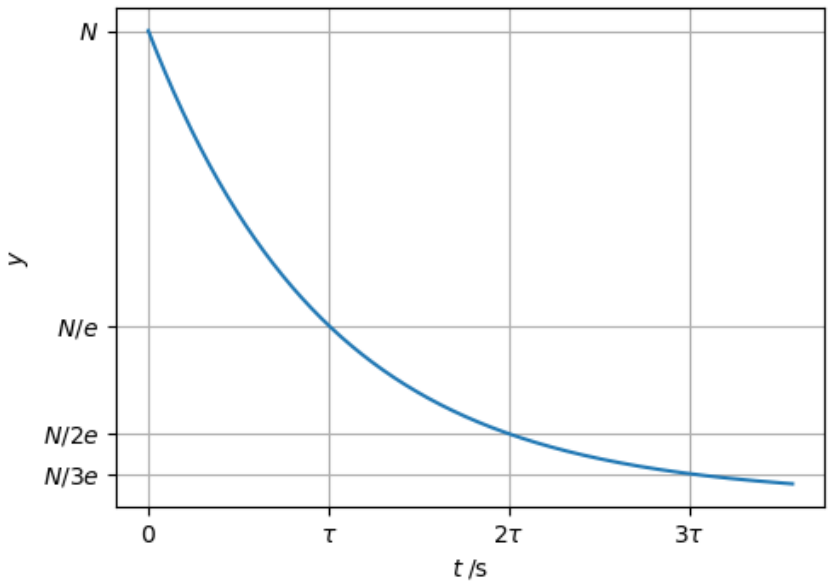

Figure 4.10: An exponential decay with customized tick labels.

To produce a plotted line with error bars, use the `Axes` class method `errorbars` instead of `plot`. In addition to the usual arguments of the `plot` method, `errorbars` allows the specification of errors in the x- and y-coordinates by passing the following types of values to the arguments `xerr` and `yerr`:

1. `None`: No error bars for this coordinate.

2. A scalar (e. g. `xerr=0.2`): all values are associated with symmetric error bars at plus and minus this value.
3. An array-type object of shape `(n, 1)` (e. g. `yerr=[0.1, 0.15, 0.1]`): the symmetric error bars are plotted at plus and minus the values in this sequence for each of the `n` data points.
4. An array-type object of shape `(2, n)` (i. e. two rows for each of `n` data points): error bars, which may be asymmetric, are plotted using minus-values from the first row and plus-values from the second row.

The appearance of the error bars can be customized using the arguments summarized in Table 4.9.

Table 4.9: Common arguments to `Axes` class method `errorbar`

Argument	Description
`x`, `y`	The data to plot (scalar or array-like)
`xerr`, `yerr`	Errors on the x and y data coordinates (scalar or array-like)
`fmt`	The plot format symbol (marker for the data point); set to `None` or the empty string `""` to display only the error bars
`ecolor`	A Matplotlib color specifier for the error bars; the default `None` uses the same color as the connecting line between data markers
`elinewidth`	The width of the error bar lines in points; use `None` to use the same linewidth as the plotted data
`capsize`	The length of the error bar caps in points

For instance, the following Python code creates a plot with asymmetric horizontal and vertical error bars as shown in Figure 4.11.

```
>>> x = np.array([ 0.3, 0.5, 0.7, 0.9])
>>> y = np.array([ 1. , 2. , 2.5, 3.9])
>>> xerr = 0.05
>>> yerr = np.array([[0.1, 0.25, 0.5, 0.4],
...                  [0.1, 0.15, 0.2, 0.]])
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
>>> ax.errorbar(x, y, yerr, xerr, fmt="o", ls="")
<ErrorbarContainer object of 3 artists>
>>> plt.show()
```

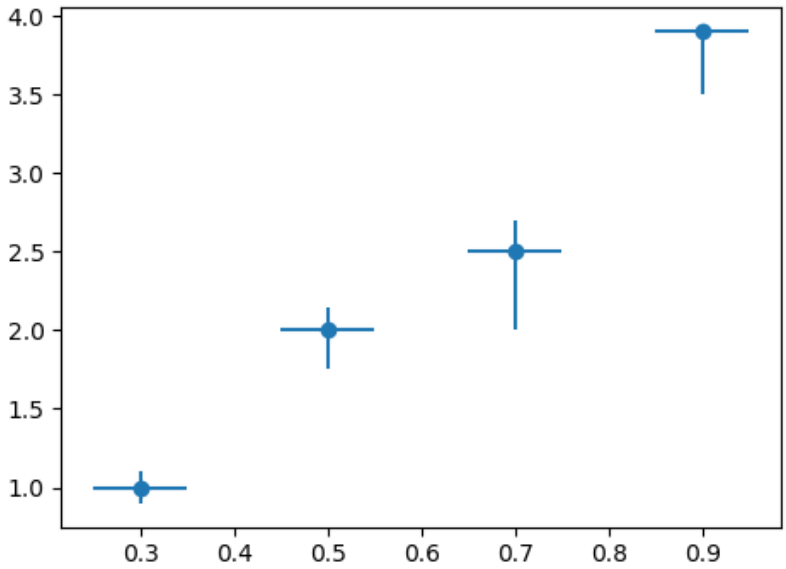

Figure 4.11: A simple plot with asymmetric horizontal and vertical error bars.

4.2.2 Scatter Plots and Contour Plots

A typical two-dimensional scatter plot depicts the data as points on a Cartesian axes system. Sometimes there is no meaningful ordering to the data and thus no need to join data points by lines. A scatter plot can be created by calling the method `scatter` on an `Axes` object. In addition to one-dimensional sequences of x and y data like that for `plot`, we can set the data point marker colors and sizes by passing a sequence of appropriate values of the same length as the data to the argument `s`. The sizes of the markers are in units of points squared so that their *area* is proportional to the values passed to the argument `s`. Indeed, a common way of indicating a third dimension to the data is manipulating the marker sizes. It is illustrated in the following Python program `scatterplot.py` with the resultant plot shown in Figure 4.12.

```
# scatterplot.py
# This program generates the scatter plot of demographic data for
# eight countries.
# Last Update on 19 Nov 2020 by F K Chow

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

```
# Name of countries
countries = ["Brazil", "Madagascar", "S Korea", "United States",
             "Ethiopia", "Pakistan", "China", "Belize"]
# Birth rate per 1000 population
birth_rate = [16.4, 33.5, 9.5, 14.2, 38.6, 30.2, 13.5, 23.0]
# Life expectancy at birth (years)
life_expectancy = [73.7, 64.3, 81.3, 78.8, 63.0, 66.4, 75.2, 73.7]
# Per person income fixed to US Dollars in 2000
GDP = np.array([4800, 240, 16700, 37700, 230, 670, 2640, 3490])

# Create the scatter plot of demographic data for the countries
fig = plt.figure()
ax = fig.add_subplot(111)
colors = range(len(countries)) # Some arbitrary colors
ax.scatter(birth_rate, life_expectancy, c=colors, s=GDP/15)
ax.set_xlim(5, 40)
ax.set_ylim(60, 85)
ax.set_xlabel("Birth rate per 1000 population")
ax.set_ylabel("Life expectancy at birth (years)")
ax.set_title("Demographic Data for Eight Countries")
plt.show()
```

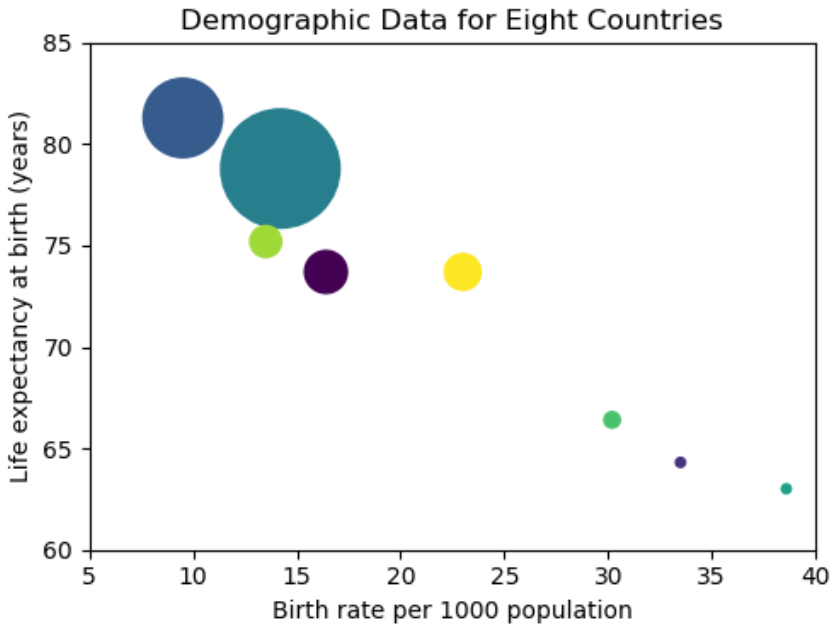

Figure 4.12: A scatter plot of demographic data for eight countries.

A contour plot is a way for representing a three-dimensional surface in two dimensions by plotting a number of curves known as **contour lines**. Each contour line connects the points on the x-y plane which have a constant value of a third variable z. These constant values are called the **contour levels** which determine the number and location of the contour lines. To make a contour plot, we can use the `Axes` class method `contour`. Its simplest invocation is `contour(Z)` where the two-dimensional array `Z` gives the height values over which the contour lines are drawn. The x and y values are the row and column indexes of `Z` and the contour levels are selected automatically. To explicitly specify (x, y) coordinates, pass them as `contour(X, Y, Z)`. The arrays `X` and `Y` must be either two-dimensional with the same shape as `Z` or one-dimensional so that `X` and `Y` have the same length as the number of columns and rows in `Z`, respectively. Notice that the contour levels can be controlled by an optional argument `level` of the `contour` method whose value can be either a scalar `N` giving the total number of contour levels or a sequence `V` listing the values of `Z` at which to draw contours.

By default, Matplotlib use the default colormap to color the contours. In this process, the data are normalized linearly onto the interval $[0, 1]$ which is then mapped onto a list of colors that are used to style the contours at the corresponding values. To use other colormap for the contours, set the optional argument `cmap` of the `contour` method. The `matplotlib.cm` module provides a number of colormaps: some of the common ones are `cm.hot`, `cm.bone`, `cm.viridis`, `cm.jet`, `cm.gray`, and `cm.hsv`. If you want to use a colormap with its colors reversed, append `_r` at the end of its name (e. g. `cm.jet_r`).

As an alternative, the `Axes` class method `contour` supports the `colors` argument which takes either a single Matplotlib color specifier or a sequence of such specifiers. For single-color contour plots, contours corresponding to negative values are plotted in dashed lines. The widths of the contour lines can be styled individually or all together with the argument `linewidths`. For example, below is the Python program `elecdipoleplot.py` with the resultant plot shown in Figure 4.13.

```
# elecdipoleplot.py
# This program produces the contour plot of the electric potential of
# an electric dipole.
# Last Update on 19 Nov 2020 by F K Chow

import matplotlib.pyplot as plt
```

```
import numpy as np

# Dipole charge (C) and permittivity of free space (F/m )
q, eps0 = 1.602e-19, 8.854e-12
# Distance between dipole (m) and a convenient parameter combination
d = 1.e-12
k = (q*d)/(4*np.pi*eps0)
# Cartesian axis system with origin at the dipole's center (m)
X = np.linspace(-5e-11, 5e-11, 1000)
Y = X.copy()
X, Y = np.meshgrid(X, Y)
# Dipole electric potential (V), using point dipole approximation
Phi = k*X/np.hypot(X, Y)**3

# Create the contour plot of the electric potentila of the dipole
fig = plt.figure()
ax = fig.add_subplot(111)
lev = np.array([10**pw for pw in np.linspace(0, 5, 20)])
lev = list((-1)*lev[::-1]) + list(lev)
ax.contour(X, Y, Phi, levels=lev, colors="k", linewidths=2)
plt.show()
```

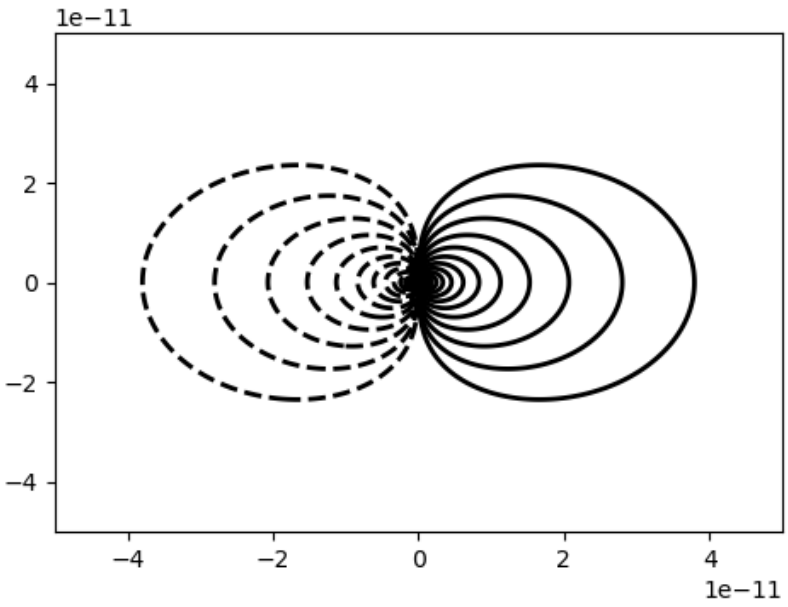

Figure 4.13: A contour plot of the electrostatic potential of a point dipole.

In the above program, the numpy method `meshgrid` is used to create the 2D arrays `X` and `Y` for the contour plot. It builds a rectangular grid from two 1D arrays of x and y values. Its output are two 2D arrays where the first one consists of identical columns of x values increasing from left to right and the second one consists of identical rows of y values increasing from top to bottom.

Sometimes we may want to add labels to the contours in a contour plot. In order to do so, store the `ContourSet` object returned by the call to the method `contour` on an `Axes` object and then pass it to the method `clabel` of this `Axes` object (probably with some additional parameters to specify the font properties). In some situation, it's desired to draw filled contours in a contour plot. This can be done by using the `Axes` class method `contourf` which takes the same arguments as `contour`. Note that `contour` and `contourf` can be used together. As an example, the following Python program `filledcontours.py` produces the filled contour plot as shown in Figure 4.14.

```
# filledcontours.py
# This program produces a contour plot with filled and style contours.
# Last Update on 19 Nov 2020 by F K Chow

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

X = np.linspace(0,1,100)
Y = X.copy()
X, Y = np.meshgrid(X, Y)
alpha = np.radians(25)
cX, cY = 0.5, 0.5
sigX, sigY = 0.2, 0.3
rX = np.cos(alpha)*(X - cX) - np.sin(alpha)*(Y - cY) + cX
rY = np.sin(alpha)*(X - cX) + np.cos(alpha)*(Y - cY) + cY
Z = (rX - cX)*np.exp(-((rX - cX)/sigX)**2) \
    *np.exp(-((rY - cY)/sigY)**2)

# Create the contour plot of the data with filled and style contours
fig = plt.figure()
```

```
ax = fig.add_subplot(111)
cpf = ax.contourf(X, Y, Z, 20, cmap=cm.gray)
cols = ["w" if level < 0 else "k" for level in cpf.levels]
cp = ax.contour(X, Y, Z, 20, colors=cols)
ax.clabel(cp, fontsize=12, colors=cols)
plt.show()
```

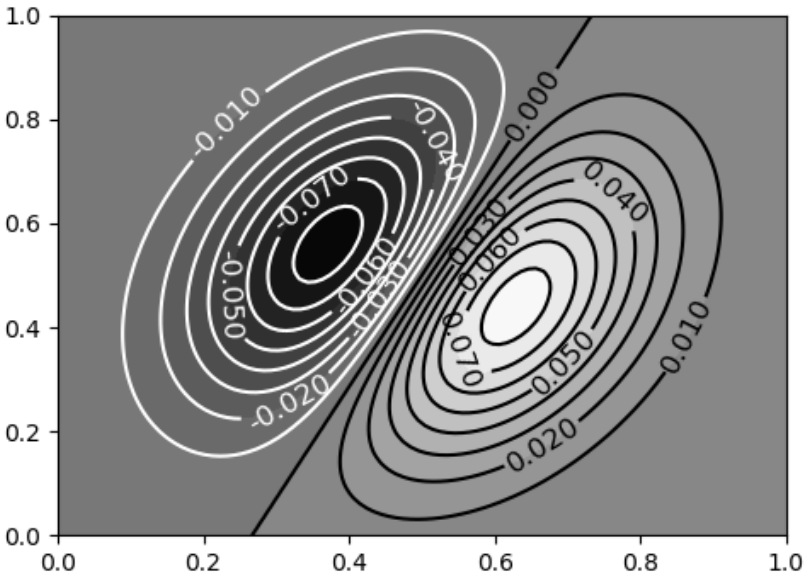

Figure 4.14: A filled contour plot with labeled contours.

4.2.3 Bar Charts and Pie Charts

In Matplotlib, `Axes` class provides the method `bar` for plotting a bar chart. This method makes a plot of rectangular bars defined by their position and height. For example, the command

```
ax.bar([0, 1, 2], [40, 80, 20])
```

creates a bar chart with bars of height 40, 80, 20 at the position $x = 0, 1, 2$, respectively. By default, the bars are centered on the x positions and their bottom lies on the line $y = 0$. Moreover, the default width of the bars is 0.8 but can be set with the (third) argument `width`. If you want to align the bars on the left edges, set the argument `align` to `"edge"` as follows:

```
w = 0.5
x, y = np.array([0, 1, 2]), np.array([40, 80, 20])
ax.bar(x, y, w, align="edge")
```

Additional arguments including the provision of error bars are given in Table 4.10. By default, the `bar` method produces a vertical bar chart. Horizontal bar charts are produced by using the analogous `Axes` class method `barh`.

Table 4.10: Common arguments to `Axes` class methods `bar` and `barh`

Argument	Description
`width`	Width of the bars; a scalar if all bars have the same width and can be array-like for variable widths
`height`	A sequence of heights for the bars
`left`	The x-coordinates of the left edges of the bars (scalar or array-like; only valid for `barh`)
`bottom`	The y-coordinates of the bottom of the bars (scalar or array-like; only valid for `bar`)
`align`	The default `"center"` align the bars to their center; `"edge"` aligns the bars by their left edges (for vertical bars) or bottom edges (for horizontal bars)
`color`	Colors of the bar faces (scalar or array-like)
`edgecolor`	Colors of the bar edges (scalar or array-like)
`linewidth`	Line widths of the bar edges in points (scalar or array-like)
`xerr`, `yerr`	Error bar limits as for `errorbar` (scalar or array-like)
`error_kw`	A dictionary of keyword arguments for the customization of the appearance of the error bars (see Table 4.9)

For instance, below is the Python program `charfreq.py` which generates a bar chart of letter frequencies in the text of the novel *Moby-Dick*. The vertical bars in the chart are centered and labeled by letters as shown in Figure 4.15.

```
# charfreq.py
# This program produces a bar chart of letter frequencies in the text
# of the novel Moby—Dick.
# Last Update on 19 Nov 2020 by F K Chow

import matplotlib.pyplot as plt
import numpy as np

# Initialize the dictionary of letter counts and then read in the text
# and count the letter occurrences
letters = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

```
lcount = dict([(l, 0) for l in letters])
for l in open("moby-dick.txt").read():
    if ((l >= "a" and l <= "z") or (l >= "A" and l <= "Z")):
        lcount[l.upper()] += 1
norm = sum(lcount.values())

# Produce the bar chart of letter frequencies
fig = plt.figure()
ax = fig.add_subplot(111)
x = np.arange(26)
ax.bar(x, [lcount[l]/norm*100 for l in letters], width=0.8,
       color="g", alpha=0.5, align="center")
ax.set_xticks(x)
ax.set_xticklabels(letters)
ax.tick_params(axis="x", direction="out")
ax.set_xlim(-0.5, 25.5)
ax.yaxis.grid(True)
ax.set_ylabel("Letter frequency, %")
plt.show()
```

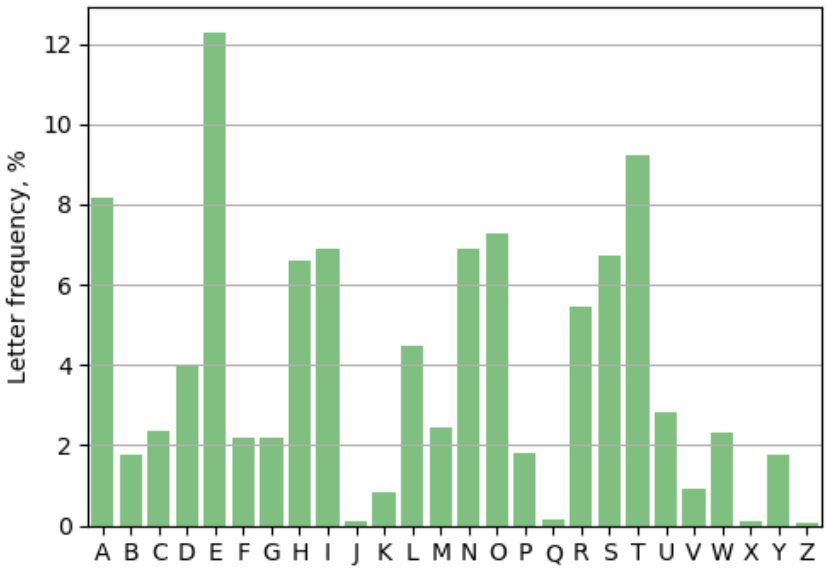

Figure 4.15: A bar chart of letter frequencies in the novel *Moby-Dick*.

It is straightforward to draw a pie chart in Matplotlib by passing an array of values to `Axes` class method `pie`. The values will be normalized by their sum if this sum is

greater than 1, or otherwise treated directly as fractions. Labels, percentages, "exploded" segments, and other effects are handled as described in Table 4.11.

Table 4.11: Common arguments to `Axes` class method `pie`

Argument	Description
`colors`	A sequence of Matplotlib color specifiers for coloring the segments
`labels`	A sequence of strings for labeling the segments
`explode`	A sequence of values specifying the fraction of the pie chart radius to offset each wedge by (0 for no explode effect)
`shadow`	A boolean value specifies whether to draw an attractive shadow under the pie
`startangle`	Rotate the "start" of the pie chart by this number of degrees counterclockwise from the horizontal axis
`autopct`	A format string to label the segments by their numeric values, or a function for generating this string from the data
`pctdistance`	The radial position of the `autopct` text relative to the pie radius; the default is 0.6 (i. e. within the pie)
`labeldistance`	The radial position of the `label` text relative to the pie radius; the default is 1.1 (i. e. just outside the pie)
`radius`	The radius of the pie (the default is 1); this is useful when creating overlapping pie charts with different radii

To illustrate how to work with the `pie` method, here is the Python program `pie.py` which depicts the emissions of greenhouse gases by mass of "carbon equivalent" (data from the 2007 IPCC report).

```
# pie.py
# This program produces a pie chart of greenhouse gas emissions by
# mass of "carbon equivalent".
# Last Update on 19 Nov 2020 by F K Chow

import matplotlib.pyplot as plt
import numpy as np

# Annual greenhouse gas emissions, billion tons carbon equivalent
gas_emis = np.array([(r"$\mathrm{CO_2}$-d", 2.2),
                     (r"$\mathrm{CO_2}$-f", 8.0),
```

```
                    ("Nitrous\nOxide", 1.0),
                    ("Methane", 2.3), ("Halocarbons", 0.1)],
                   dtype=[("source", "U17"), ("emission", "f4")])

# 5 colors beige
cols = ["#C7B299", "#A67C52", "#C69C6E", "#754C24", "#534741"]

# Produce the pie chart of greenhouse gas emissions
fig = plt.figure()
ax = fig.add_subplot(111)
ax.axis("equal")    # So our pie looks round!
ax.pie(gas_emis["emission"], colors=cols, shadow=True,
       startangle=90, explode=[0, 0, 0.1, 0, 0],
       labels=gas_emis["source"], autopct="%.1f%%", pctdistance=1.15,
       labeldistance=1.3)
plt.show()
```

Here the argument `explode` is set to a list so that the segment corresponding to nitrous oxide has been "exploded" by 10%. Moreover, the argument `autopct="%.1f%%"` specifies that the percentage values are formatted to one decimal place. The resultant pie chart is depicted in Figure 4.16

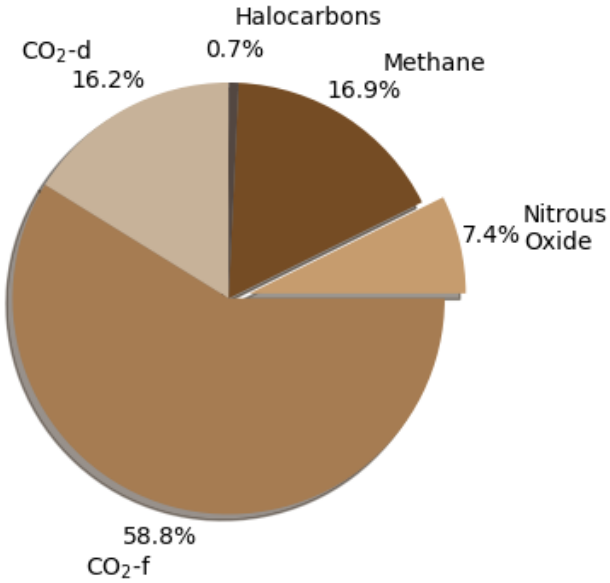

Figure 4.16: A pie chart of greenhouse gas emissions from five different sources. Note that CO_2-d denotes CO_2 emissions from deforestation and CO_2-f denotes CO_2 emissions from fossil fuel burning.

4.2.4 Multiple Subplots

Sometimes we may want to create a figure with more than one subplot (i. e. `Axes` object). To do this, simply call `add_subplot` on your `Figure` object, setting its argument to indicate where the subplot should be placed. Each such call returns an `Axes` object. As single figures with more than 10 subplots are uncommon, the usual argument is a three-digit number where the digits indicate the number of rows, number of columns, and subplot number in order. The subplot number increases along the columns in each row and then down the rows. For example, a figure consisting of three rows of two columns of subplots can be constructed by adding `Axes` objects as follows:

```
>>> fig = plt.figure()
>>> ax1 = fig.add_subplot(321)  # top left subplot
>>> ax2 = fig.add_subplot(322)  # top right subplot
>>> ax3 = fig.add_subplot(323)  # middle left subplot
      ⋮
>>> ax6 = fig.add_subplot(326)  # bottom right subplot
```

Alternatively, to create a figure and add all its subplots to it at the same time, call the `pyplot` function `subplots`. This function takes arguments `nrows` and `ncols` (in addition to those listed in Table 4.2) and returns a `Figure` object and an array of `Axes` objects, which can be indexed for each individual axis:

```
>>> fig, axes = plt.subplots(nrows=3, ncols=2)
>>> axes.shape
(3, 2)
>>> ax1 = axes[0, 0]    # top left subplot
>>> ax2 = axes[2, 1]    # bottom right subplot
```

In fact, a useful syntax to create a plot with a single `Axes` object is to call `subplots` with no arguments:

```
>>> fig, axes = plt.subplots()
>>> axes.plot(x, y)
[<matplotlib.lines.Line2D object at 0x0000027255033780>]
```

Notice that there is no need to index the single `Axes` object created.

Plots with subplots have the risk of their labels, titles and ticks overlapping each other. When this happens, call the method `tight_layout` on the `Figure` object and Matplotlib

will do its best to arrange them so that there is sufficient space between them. As an example, below is the Python program `1dboxwavefunc.py` which plots the wave functions of a quantum particle in a one-dimensional box in a single figure with the resultant plot given in Figure 4.17.

```
# 1dboxwavefunc.py
# This program plots the wave functions for the six lowest energy
# states of a quantum particle in a one-dimensional box in a single
# figure.
# Last Update on 6 Jan 2022 by F K Chow

import matplotlib.pyplot as plt
import numpy as np

# Generate the wave functions for the six lowest energy states
x = np.linspace(0, 1, 101)
f = np.zeros((6, 101))
for n in np.arange(1, 7):
    f[n - 1] = np.sqrt(2)*np.sin(n*np.pi*x)

# Plot the wave functions in a figure with 3x2 subplots
fig, axes = plt.subplots(nrows=3, ncols=2)
for i in np.arange(3):
    for j in np.arange(2):
        ax = axes[i, j]
        n = i*2 + j + 1
        ax.plot(x, f[n - 1], "k")
        ax.set_title("$n = {}$".format(n), fontfamily="monospace")
        ax.set_xlim(0, 1)
        ax.set_ylim(-2, 2)
        ax.set_xlabel("$x/L$")
        ax.set_ylabel(r"$\psi_n(x)/L^{-1/2}$")
fig.tight_layout()
plt.show()
```

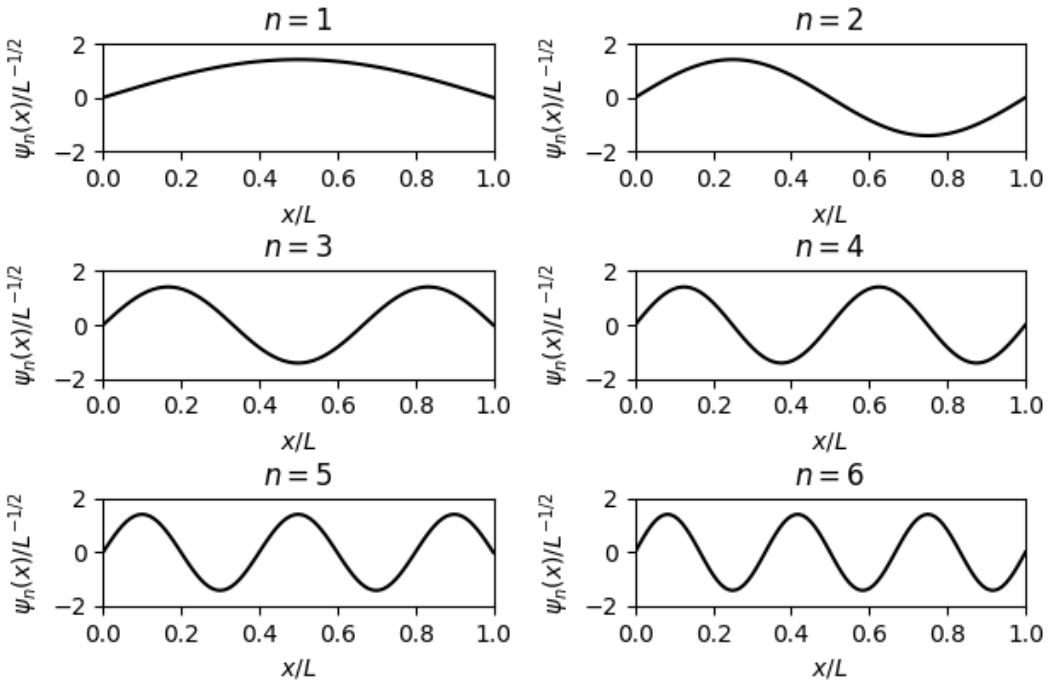

Figure 4.17: The wave functions for the six lowest energy states of a quantum particle in a one-dimensional box.

To further customize the spacing between subplots, call the method `subplots_adjust` on the `Figure` object. This method takes any of the keywords `left`, `right`, `bottom`, `top`, `hspace`, and `wspace`, which can be set to fractional values of the figure's height and width as appropriate to determine the positions of the subplots' left side, right side, bottom, top, vertical spacing, and horizontal spacing. A practical use of this function is to create "ganged" subplots sharing a common axis like that in the following Python program `subplots.py` (see Figure 4.18 for the plot).

```
# subplots.py
# This program creates five subplots with zero vertical spacing in a
# single figure.
# Last Update on 19 Nov 2020 by F K Chow

import matplotlib.pyplot as plt
import numpy as np

nrows = 5
fig, axes = plt.subplots(nrows, 1)
# Zero vertical spacing between the subplots
```

```
fig.subplots_adjust(hspace=0)

x = np.linspace(0,1,1000)
for i in range(nrows):
    # n = nrows for the top subplot, n = 0 for the bottom subplot
    n = nrows - i
    axes[i].plot(x, np.cos(n*np.pi*x), "k", lw=2)
    # We only want ticks on the bottom of each subplot
    axes[i].xaxis.set_ticks_position("bottom")
    if i < nrows - 1:
        # Set ticks at the nodes (zeros) of our cosine functions
        axes[i].set_xticks(np.arange(1/(2*n), 1, 1/n))
        # We only want labels on the bottom subplot xaxis
        axes[i].set_xticklabels("")
    axes[i].set_xlim(0, 1)
    axes[i].set_ylim(-1, 1)
    axes[i].set_yticklabels("")
    axes[i].tick_params(direction="in")
plt.show()
```

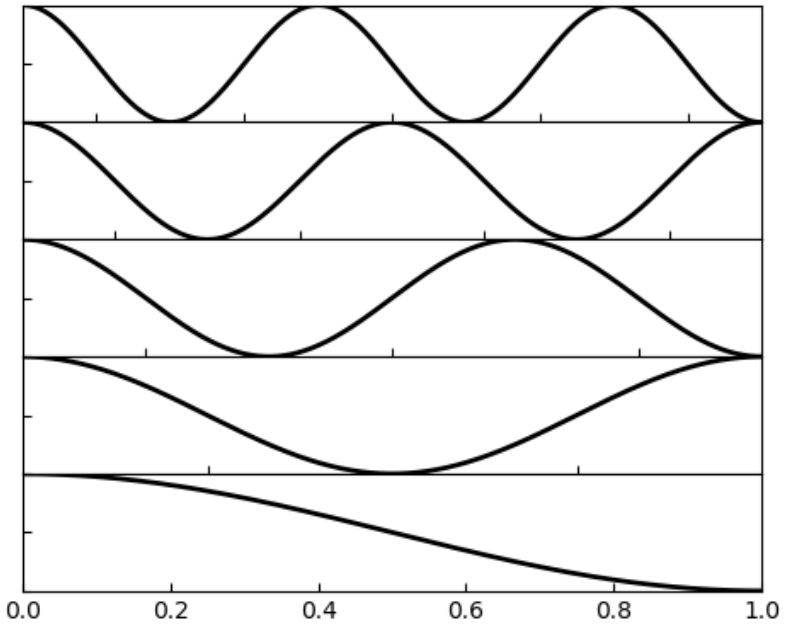

Figure 4.18: Five subplots of $\cos(n\pi x)$ for $n = 0, 1, 2, 3, 4$ adjusted to remove vertical space between them.

4.2.5 Heatmaps

Another way to depict two-dimensional data is plotting a **heatmap** (also known as **density plot**): an image in which the color of each pixel is determined by the corresponding value in the array of data. In Matplotlib, `Axes` class provide the method `imshow` for displaying an image (i. e. data on a 2D regular grid) on the axes as a heatmap. In its basic usage, `imshow` takes a two-dimensional array and maps its values to the pixels on an image according to some interpolation scheme and normalization. If the array data are taken from an image read in with the Matplotlib function `image.imread`, then all that required is usually the following Python code:

```
>>> import matplotlib.pyplot as plt
>>> import matplotlib.image as mpimg
>>> im = mpimg.imread("image.png")
>>> fig, ax = plt.subplots()
>>> ax.imshow(im)
<matplotlib.image.AxesImage object at 0x0000025490364DD8>
>>> plt.show()
```

Here `im` is a three-dimensional array of shape `(n, m, 3)` in which the "depth" coordinate corresponds to the red, green, and blue components of each pixel in the `n`-by-`m` image.

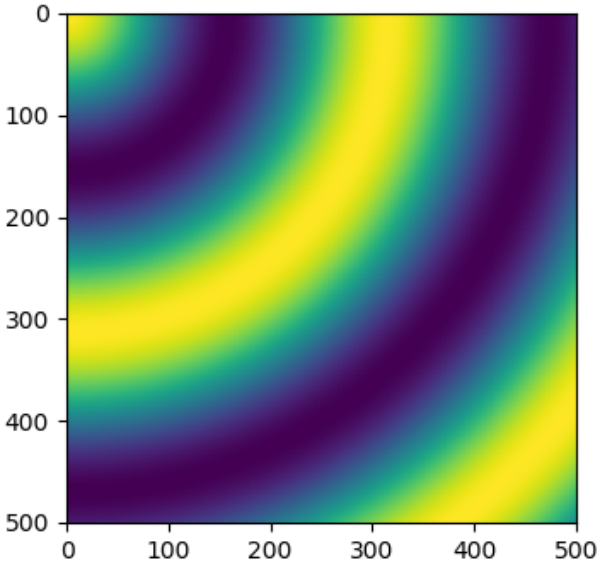

Figure 4.19: A basic density plot.

The method `imshow` is used frequently to visualize matrices or other two-dimensional arrays of data with a density plot. Below is the Python routine for making the density plot as shown in Figure 4.19.

```
>>> infile = open("circular.txt", "r")
>>> data = []
>>> for line in infile.readlines():
...     row = []
...     for i in line.strip().split():
...         row.append(float(i))
...     data.append(row)
...
>>> fig, ax = plt.subplots()
>>> ax.imshow(data)
<matplotlib.image.AxesImage object at 0x00000254901432E8>
>>> plt.show()
```

This routine reads the data from the file `circular.txt`, which contains a simple 2D array of values, and put it into the 2D list `data`. Then it creates the density plot with the `imshow` method and displays it with the `pyplot` function `show`. The computer automatically adjusts the color scale so that the picture uses the full range of available shades. It also adds numbered axes along the sides of the plot which measure the rows and columns of the array. Indeed, it is possible to change the calibration of the axes to use different units and we'll see how to do this later. The image produced is a direct picture of the list, laid out in the usual fashion for matrices, i. e. starting at the top and working downwards row by row. So the top left corner in Fig. 4.19 represents the value stored in the list element `data[0][0]`, followed to the right by `data[0][1]`, `data[0][2]`, and so on. Similarly, the next row is `data[1][0]`, `data[1][1]`, `data[1][2]`, and so on.

In a density plot produced by `imshow`, the numerical labels on the axes reflect the list indexes with the origin of the figure being at the top left and the vertical axis increasing downwards. While this is natural from the point of view of matrices, it is a little odd bit for a graph. Most of us are accustomed to graphs whose vertical axes increase upwards. Moreover, just like the standard matrices, the list elements `data[i][j]` are written with the row or vertical index first and the column or horizontal index second. It is the opposite of the convention normally used with graphs where we list the coordinates of a point in x, y order, i. e. horizontal first and then vertical. These conflicts between the conventions for matrices and for plotting graphs can be confusing.

Actually, `imshow` provides a way to deal with the problem that the origin is at the top in a density plot. You can do so by adding an additional argument `origin` to the `imshow`

method using the command:

```
ax.imshow(data, origin="lower")
```

which creates the plot as shown in Figure 4.20(a). It flips the density plot top-to-bottom, putting the list element `data[0][0]` in the lower left corner and changing the labeling of the vertical axis accordingly so that it increases in the upward direction. Note that this does not fix our other problem: indexes `i` and `j` for the element `data[i][j]` still correspond to vertical and horizontal positions respectively, not the reverse. You should keep this in mind when making density plots to avoid getting the axes swapped by mistake.

As mentioned in section 4.2.2, the `matplotlib.cm` module provides a number of colormaps. Density plots produced by the `imshow` method use `viridis` as the default colormap. But it's not always the best choice. Indeed, a simple gray-scale from black to white is easier to read for most purposes. We can change the colormap used by `imshow` to `gray` simply by adding an additional argument `cmap` to this method in the same way like that for the `contourf` method:

```
import matplotlib.cm as cm
ax.imshow(data, origin="lower", cmap=cm.gray)
```

The resultant plot is depicted in Figure 4.20(b). Even in black-and-white it looks somewhat different from the viridis-map version in Fig. 4.20(a); and it looks entirely different on the screen. A complete list of Matplotlib colormaps with illustration can be found in the online Matplotlib documentation. In addition, we can even define our own colormaps although the definitions involve some tricky programming.

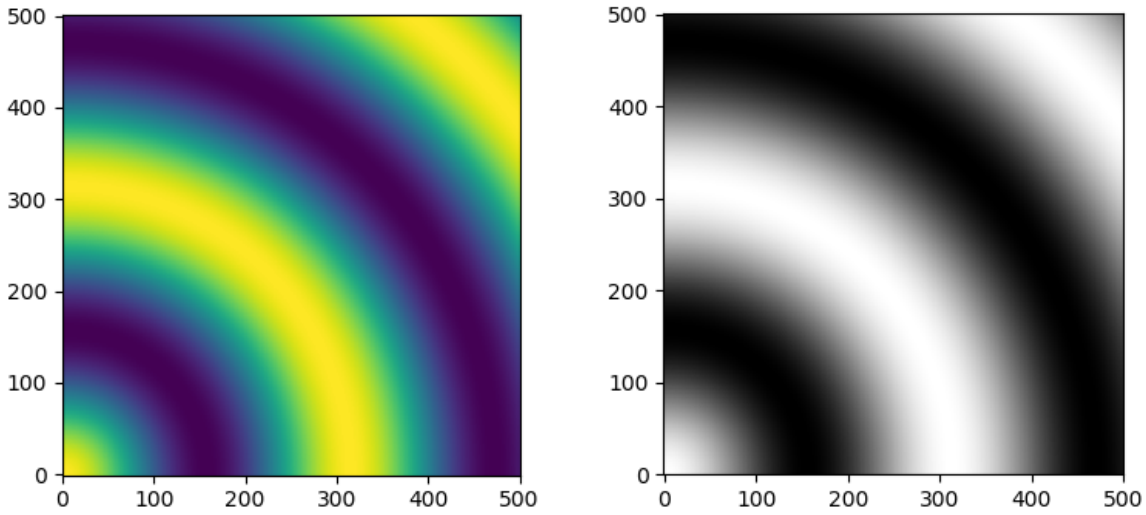

Figure 4.20: Two different versions of the same density plot. (a) A plot using the default `viridis` colormap. (b) A plot using the `gray` colormap.

Just like line plots and scatter plots, you can modify the appearance of density plots in various ways. For instance, you can change the scale marked on the axes of a density plot. By default, the scale corresponds to the elements of the list or array holding the data values. But sometimes you may want to calibrate your plot to a different scale. You can do so by adding an extra argument to the `imshow` method like this:

```
ax.imshow(data, origin="lower", cmap=cm.gray, extent=[0,10,0,5])
```

which results in a modified plot as shown in Figure 4.21. The argument consists of `extent=` followed by a list of four values which give the start and end of the horizontal scale and the start and end of the vertical scale in order. The computer will use these numbers to mark the axes. But the actual content displayed in the body of the density plot remains the same, i. e. the `extent` argument only affects how the plot is labeled. This trick is very useful if you want to calibrate your plot in "real" units.

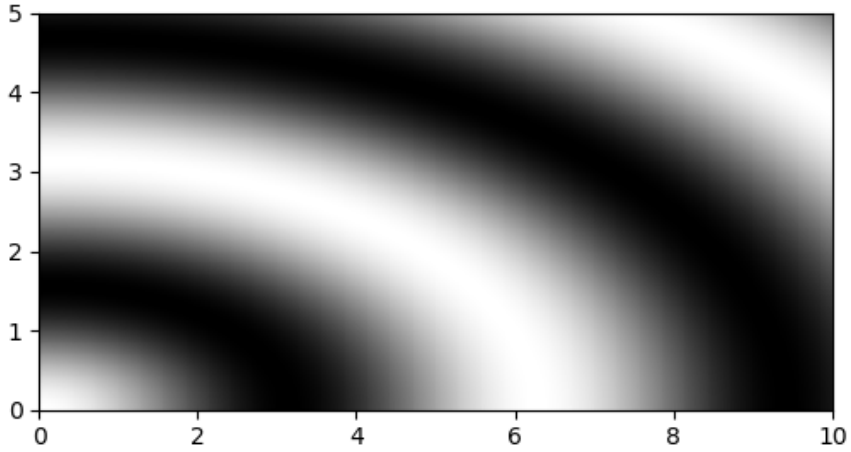

Figure 4.21: The same plot as in Fig. 4.20(b) but with the calibration of the axes changed.

We can also observe that the computer has changed the shape of the plot in Fig. 4.21, i. e. its aspect ratio, to accommodate the fact that the horizontal and vertical axes have different ranges. The `imshow` method always try to make unit distances equal along the horizontal and vertical directions whenever possible. However, sometimes this is not what we want. In such case, we can tell computer to use a different aspect ratio. For instance, if we want the plot in Fig. 4.21 to remain square, then we can add an additional argument `aspect` to the `imshow` method as follows:

```
ax.imshow(data, origin="lower", cmap=cm.gray, extent=[0,10,0,5],
          aspect=2.0)
```

It tells the computer to use unit distances twice as large along the vertical axis as along the horizontal one so that the plot would be made square again. Note that we are free to

use any or all of the `origin`, `extent`, and `aspect` arguments together in the same `imshow`. We don't have to use them all if we don't want and they can come in any order.

We should beware that the `axes` methods `plot` and `scatter` can be used to superimpose line plots or scatter plots of data on the same axes for a density plot. In fact, we can use any combination of `imshow`, `plot`, and `scatter` in sequence, followed by `show` to create a single graph with density data, line data, or scatter data on the same set of axes.

Sometimes we may want to visualize a small array as an image with a much larger number of pixels. In such case, a lot of intermediate points need to be approximated based on the values of the neighboring points. This process is known as **interpolation** and it is performed according to the **interpolation scheme**. In general, `imshow` displays the image using the default interpolation scheme whenever the display resolution is different from the image resolution. To display the image without interpolation, set the argument `interpolation` to `"none"` or `"nearest"`. To demonstrate how interpolation works for `imshow`, here is the Python program `matrixshow.py` which compares interpolation schemes for a small matrix visualized with `imshow` (see Fig. 4.22 for the visualization).

```
# matrixshow.py
# This program generates the visualization of a small matrix using
# imshow with two different interpolation schemes.
# Last Update on 19 Nov 2020 by F K Chow

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

# Make a matrix with ones in the shape of a cross
a = np.eye(10, 10)
a += a[::-1, :]

# Visualize the matrix with different interpolation schemes
fig = plt.figure()
ax1 = fig.add_subplot(121)
ax1.imshow(a, interpolation="nearest", cmap=cm.gray)
ax2 = fig.add_subplot(122)
```

```
ax2.imshow(a, interpolation="bilinear", cmap=cm.gray)
plt.show()
```

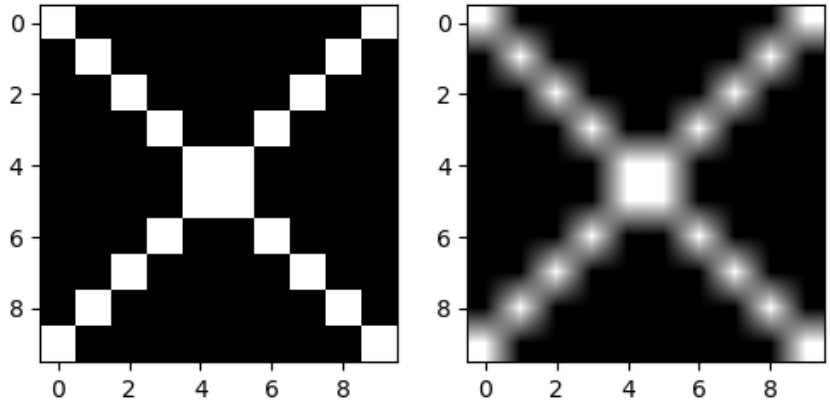

Figure 4.22: A small matrix visualized using `imshow` with different interpolation schemes.

It is often useful to have a legend indicating how are the colors in a heatmap related to the values of the array used to derive it. This is added with the `colorbar` method of the `Figure` object. In its most simple usage, simply call `colorbar(mappable)` where `mappable` is the `Image`, `ContourSet`, or other suitable object to which the colorbar applies and a new `Axes` object holding the colorbar will be created (and room is made in the figure to accommodate it). The `Axes` object can be further customized and labeled as illustrated in the following Python program `heatmap.py`. Figure 4.23 shows the output of this program.

```
# heatmap.py
# This program plots the heatmap of maximum daily temperatures in
# Boston during 2012.
# Last Update on 19 Nov 2020 by F K Chow

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

# Read in the relevant data from our input file
dt = np.dtype([("month", np.int), ("day", np.int), ("T", np.float)])
data = np.genfromtxt("boston2012.dat", dtype=dt, usecols=(1, 2, 3),
                     delimiter=(4, 2, 2, 6))
# In our heatmap, nan will mean "no such date", e.g., 30 Feb
```

```
heatmap = np.empty((12, 31))
heatmap[:] = np.nan
for month, day, T in data:
    # NumPy arrays are zero-indexed; days and months are not!
    heatmap[month-1, day-1] = T

# Plot the heatmap, customize and label the ticks
fig, axes = plt.subplots()
im = axes.imshow(heatmap, interpolation="nearest", cmap=cm.hot)
axes.set_yticks(range(12))
axes.set_yticklabels(["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul",
                      "Aug", "Sep", "Oct", "Nov", "Dec"])
days = np.array(range(0, 31, 2))
axes.set_xticks(days)
axes.set_xticklabels(["{:d}".format(day+1) for day in days])
axes.set_xlabel("Day of month")
axes.set_title("Maximum Daily Temperatures in Boston, 2012")
cbar = fig.colorbar(ax=axes, mappable=im, orientation="horizontal")
cbar.set_label("Temperature, $^\circ\mathrm{C}$")
plt.show()
```

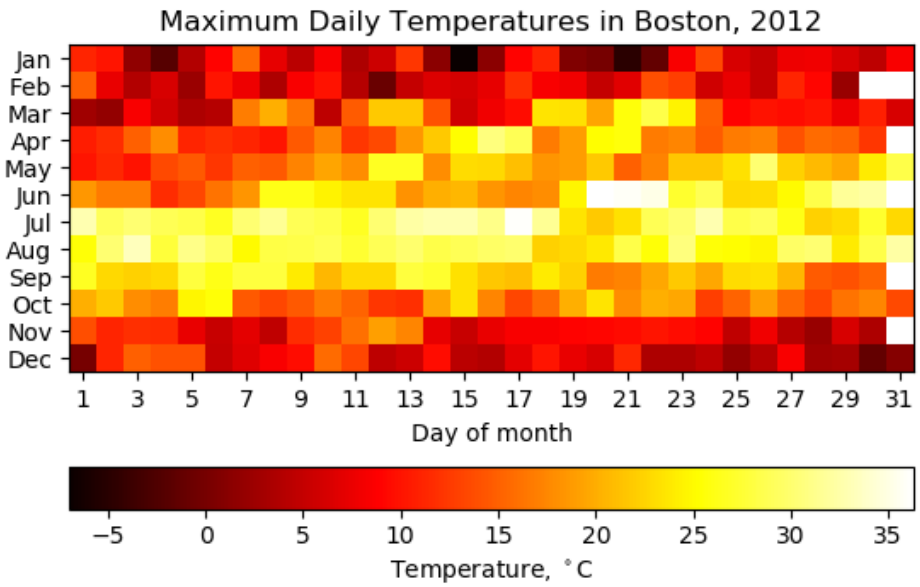

Figure 4.23: A heatmap of maximum daily temperatures in Boston during 2012.

In this program, the data file of the maximum daily temperatures in Boston for 2012 is

read in and then the data is plotted on a heatmap with a labeled colorbar legend. Note that the "mappable"" object passed to the `colorbar` method is the `AxesImage` object returned by `imshow`.

4.2.6 Other Common 2D Plotting

A polar plot is a plot of a function expressed in polar coordinates with radius r being a function of angle θ. To produce a polar plot, we can use the `polar` function from the `pyplot` module which takes on the arguments `theta` (which is usually the independent variable) and `r`. For example, the following Python code creates a polar plot of the rose curve defined by the polar equation $r = 2\cos 3\theta$ as shown in Fig. 4.24:

```
>>> theta = np.linspace(0, 2*np.pi, 1000)
>>> r = 2*np.cos(3*theta)
>>> plt.polar(theta, r)
[<matplotlib.lines.Line2D object at 0x00000209072B3208>]
>>> plt.ylim(0, 2.2)
(0, 2.2)
>>> plt.show()
```

Note that the limits for the r-coordinates is set manually to $(0, 2.2)$ by using the function `ylim` from `pyplot` to exclude those points with negative radius values.

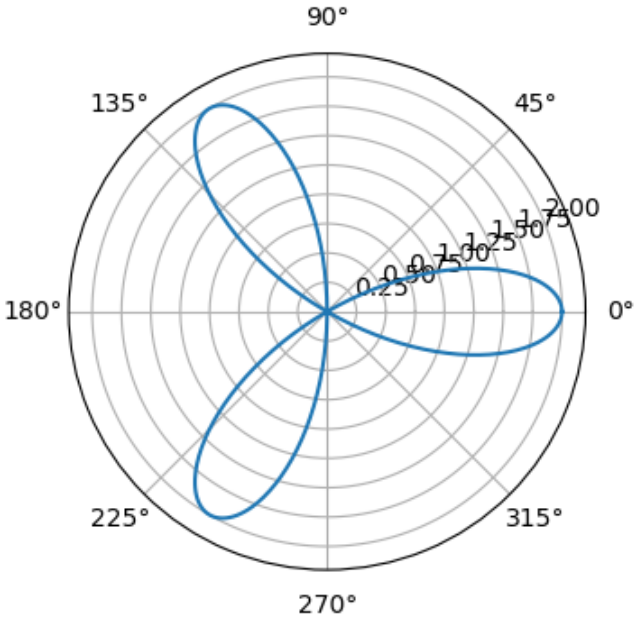

Figure 4.24: A polar plot of the rose curve defined by the polar equation $r = 2\cos 3\theta$.

A histogram represents the distribution of data as a series of bars with lengths proportional to the number of data items falling into predefined ranges known as bins. In other words, the range of data values is divided into intervals and the histogram is constructed by counting the number of data values in each interval. The `Axes` class method `hist` produces a histogram based on a set of data values. The number of bins can be passed as the optional argument `bins` and its default value is 10. Also by default the height of histogram bars are absolute counts of the data in individual bins. We can normalize the histogram so that its area is one by setting the optional argument `density` to `True`. In addition, the default edge color of the bars is `None` which makes the edge invisible. We can specify the edge color using the optional argument `edgecolor`. For example, below Python routine generates a histogram for 5000 random values taken from the normal distribution with mean 0 and standard deviation 1 (see Fig. 4.25 for the histogram).

```
>>> import matplotlib.pyplot as plt
>>> import random
>>> data = []
>>> for i in range(5000):
...     data.append(random.normalvariate(0, 1))
...
>>> fig, ax = plt.subplots()
>>> A = ax.hist(data, bins=20, edgecolor="k", facecolor="b")
>>> plt.show()
```

Note that the return object of the `hist` function is assigned to a variable in the above routine to suppress the output.

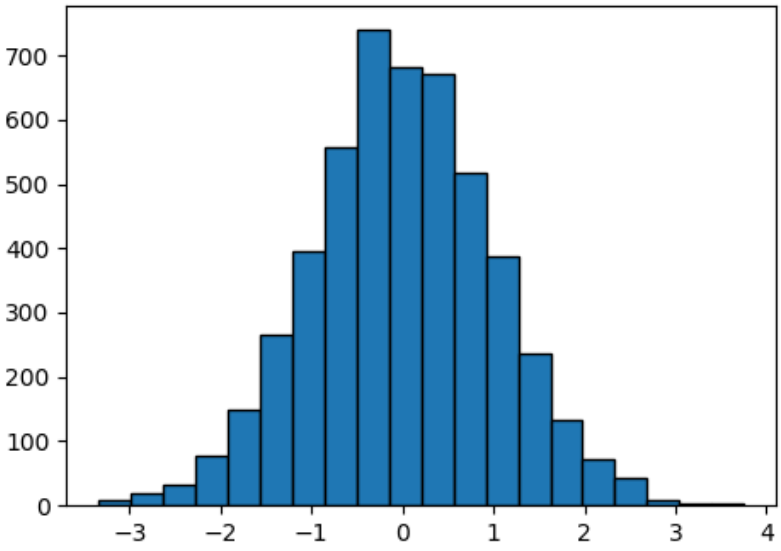

Figure 4.25: A histogram of normally distributed random data.

Sometimes we may want to plot two or or more data series which share the x-axis but with y values having large difference in magnitude or different units. We can use the method `twinx` of the `Axes` class to do so. This method creates a new `Axes` object with the same x-axis as the original one but with an independent y-axis positioned opposite to the original one. Below is the program `margDivorce.py` which illustrates how to use this method.

```
# margdivorce.py
# This program studies the correlation between the divorce rate in
# the US state of Maine and the per capita consumption of margarine
# in that country.
# Last Update on 19 Nov 2020 by F K Chow

import matplotlib.pyplot as plt

years = range(2000, 2010)
divorce_rate = [5.0, 4.7, 4.6, 4.4, 4.3, 4.1, 4.2, 4.2, 4.2, 4.1]
marg_consump = [8.2, 7, 6.5, 5.3, 5.2, 4, 4.6, 4.5, 4.2, 3.7]

fig, ax_d = plt.subplots()
ax_m = ax_d.twinx()
line1, = ax_d.plot(years, divorce_rate, 'b-o',
                   label='Divorce rate in Maine')
line2, = ax_m.plot(years, marg_consump, 'r-v',
                   label='Margarine consumption')
ax_d.set_ylabel('Divorces per 1000 people')
ax_m.set_ylabel('Pounds of margarine per capita')
plt.legend(handles=[line1, line2])
plt.show()
```

In this program, the call of the `plot` method on each `Axes` object returns a list of the `Line2D` object representing the plotted line and these objects are saved as `line1` and `line2`. Then we call the `legend` method from the `pyplot` module with the list of these `Line2D` objects passed as the argument `handles` to display the labels of the lines on the graph. The output of this program is the graph depicted in Fig. 4.26.

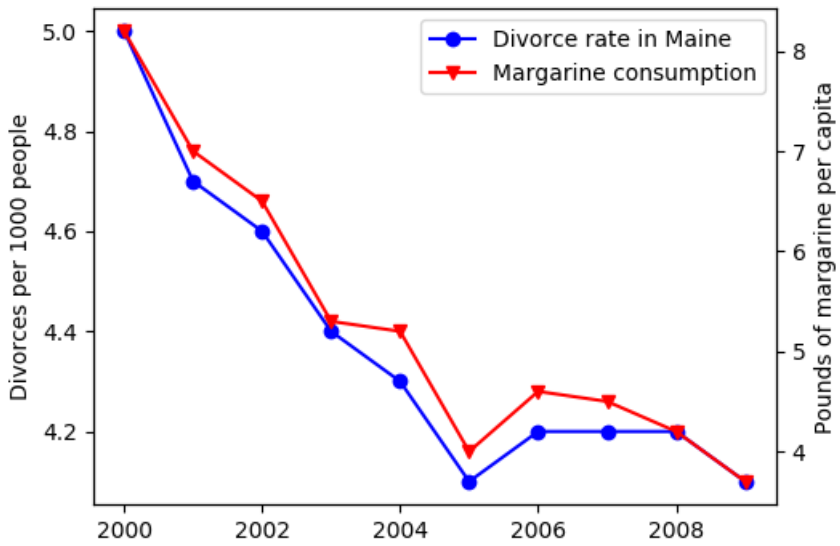

Figure 4.26: The correlation between the divorce rate in Maine and the per capita margarine consumption in the United States.

4.2.7 3D Plots

Matplotlib is primarily a 2D plotting library. But it also supports 3D plotting functionality that is good enough for many purposes. The easiest way to set up a 3D plot is importing the `Axes3D` class from the `mpl_toolkits.mplot3d` module and setting the subplot's `projection` argument to `"3d"`:

```
>>> import matplotlib.pyplot as plt
>>> from mpl_toolkits.mplot3d import Axes3D
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111, projection="3d")
```

The corresponding `Axes` object can then depicts data in three dimensions as a line plot, scatter plot, or surface plot.

The simplest kind of 3D surface plot is a wireframe plot. It can be created by the `Axes` class method `plot_wireframe` that draws lines in 3D perspective by joining the provided two-dimensional array of points `Z` on a grid of data values provided as two-dimensional arrays `X` and `Y` (like that for `contour`). By default, wires are drawn for every point in the array. If there are too many points, we can set the arguments `rstride` and `cstride` to specify the array row step size and column step size.

The `Axes` class also provides a similar method `plot_surface` which produces a 3D surface plot of filled patches. The patch colors can be set to a single color with the

`color` argument or styled to a specified color map with the `cmap` argument. The default value of `rstride` and `cstride` are 10 for the `plot_surface` method. These methods are demonstrated in the following Python program `3dsurfaceplots.py`. In this program, the `subplots` method's argument `subplot_kw` is set to {"projection": "3d"} to tell the computer plotting all the subplots as 3D plots. The resultant plots are given in Figure 4.27.

```
# 3dsurfaceplots.py
# This program produces four 3D surface plots of a simple
# two—dimensional Gaussian function.
# Last Update on 19 Nov 2020 by F K Chow

import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.cm as cm

# Generate the data from a 2D Gaussian function
L, n = 2, 400
x = np.linspace(—L, L, n)
y = x.copy()
X, Y = np.meshgrid(x, y)
Z = np.exp(—(X**2 + Y**2))

# Produce four 3D surface plots for the data
fig, ax = plt.subplots(nrows=2, ncols=2,
                       subplot_kw={"projection": "3d"})
ax[0,0].plot_wireframe(X, Y, Z, rstride=40, cstride=40)
ax[0,1].plot_surface(X, Y, Z, rstride=40, cstride=40, color="m")
ax[1,0].plot_surface(X, Y, Z, rstride=12, cstride=12, color="m")
ax[1,1].plot_surface(X, Y, Z, rstride=20, cstride=20, cmap=cm.hot)
for axes in ax.flatten():
    axes.set_xticks([—2, —1, 0, 1, 2])
    axes.set_yticks([—2, —1, 0, 1, 2])
    axes.set_zticks([0, 0.5, 1])
fig.tight_layout()
```

```
plt.show()
```

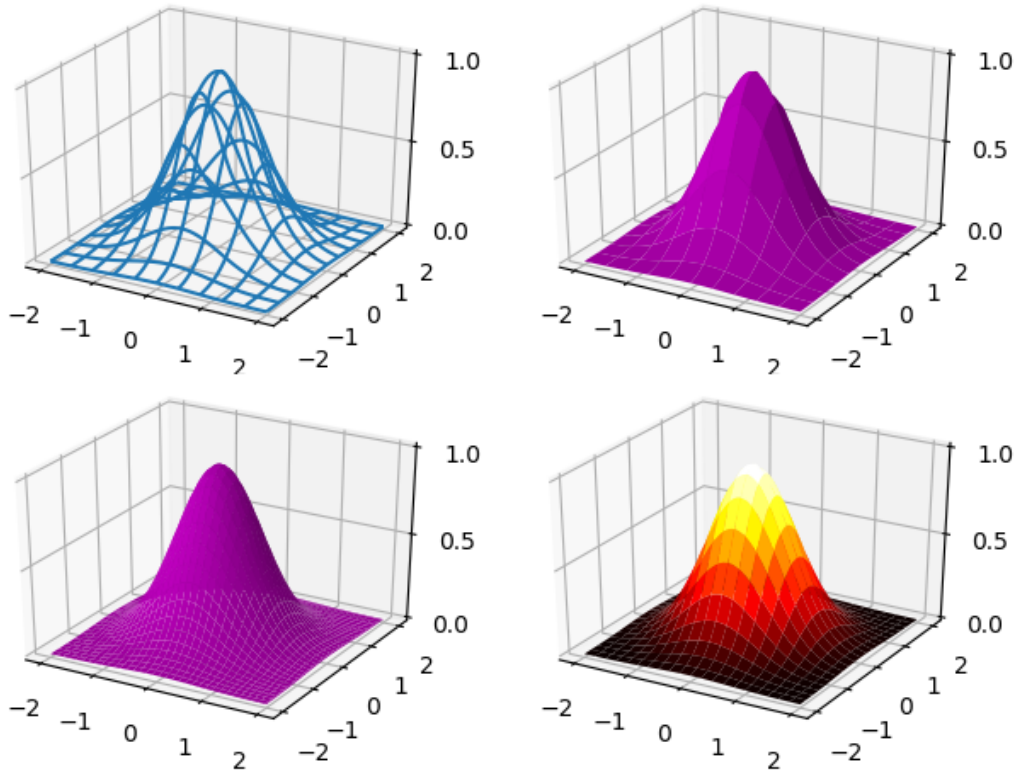

Figure 4.27: Four different 3D surface plots of the same function.

In an interactive plot, the viewing direction can be changed by clicking and dragging on the plot. To fix a particular viewing direction for a static plot image, pass the required elevation and azimuthal angles (in degrees, in that order) to the `Axes` class method `view_init`. As an example, below is the Python program `torussurface.py` which generates a 3D surface plot of a torus as depicted in Figure 4.28.

```
# torussurface.py
# This program generates a 3D surface plot of a torus.
# Last Update on 19 Nov 2020 by F K Chow

import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

# Generate the coordinates on the surface of a torus
n = 100
```

```
theta = np.linspace(0, 2.*np.pi, n)
phi = np.linspace(0, 2.*np.pi, n)
theta, phi = np.meshgrid(theta, phi)
c, a = 2, 1
x = (c + a*np.cos(theta))*np.cos(phi)
y = (c + a*np.cos(theta))*np.sin(phi)
z = a*np.sin(theta)

# Generate the 3D surface plot of the torus
fig = plt.figure()
ax1 = fig.add_subplot(121, projection="3d")
ax1.set_zlim(-3, 3)
ax1.plot_surface(x, y, z, rstride=5, cstride=5, color="k",
                 edgecolors="w")
ax1.view_init(24, 24)
ax2 = fig.add_subplot(122, projection="3d")
ax2.set_zlim(-3, 3)
ax2.plot_surface(x, y, z, rstride=5, cstride=5, color="k",
                 edgecolors="w")
ax2.view_init(0, 0)
ax2.set_xticks([])
plt.show()
```

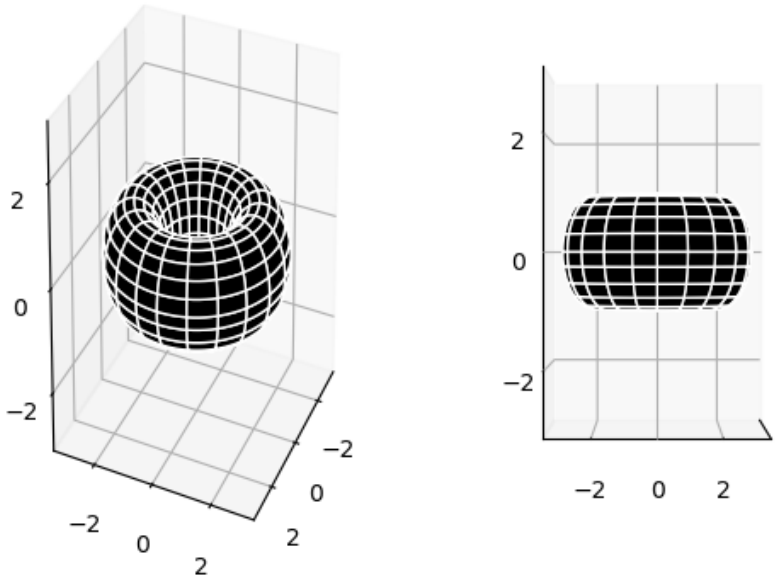

Figure 4.28: Two views of the same torus from different elevation and azimuthal angles.

Line plots and scatter plots work in 3D in a way similar to how they work in 2D. The basic method call is `plot(x, y, z)` and `scatter(x, y, z)` on the `Axes` object, where `x`, `y`, and `z` are one-dimensional arrays with the same length. To give you an idea, the following Python program `helixplot.py` plots a circularly polarized light as a helix on a three-dimensional plot (see Figure 4.29 for the plot).

```
# helixplot.py
# This program produce a 3D line plot of a helix.
# Last Update on 19 Nov 2020 by F K Chow

import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

# Generate the coordinates of the points on a helix
n = 1000
theta_max = 8*np.pi
theta = np.linspace(0, theta_max, n)
x = theta
z = np.sin(theta)
y = np.cos(theta)

# Generate the 3D line plot of the helix with a line through its
# center toegther with its sin/cos components
fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")
ax.plot(x, y, z, "b", lw=2)
ax.plot((-theta_max*0.2, theta_max*1.2), (0,0), (0,0), color="k",
        lw=2)
ax.plot(x, y, 0, color="r", lw=1, alpha=0.5)
ax.plot(x, [0]*n, z, color="m", lw=1, alpha=0.5)
ax.set_axis_off()   # Remove axis planes, ticks and labels
plt.show()
```

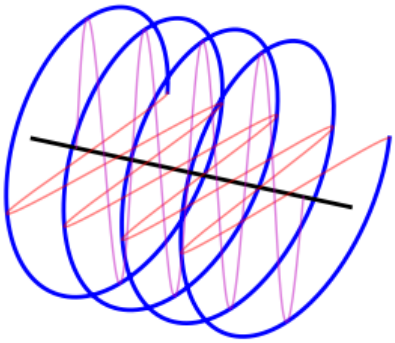

Figure 4.29: A depiction of circularly polarized light as a helix on a three-dimensional plot.

4.3 SciPy

SciPy is a Python library used for scientific computing that provides more specific functionality than NumPy. For example, it contains modules for data-fitting, integration, interpolation, special functions, and other common tasks in science and engineering. Just like the NumPy library, many of the underlying algorithms in SciPy are executed as compiled C code and so they are fast. Also like NumPy and Python itself, SciPy is a free software. As there are not much new syntax to learn for using the SciPy routines, here we will focus on the applications of the library in science and engineering.

4.3.1 Physical Constants and Special Functions

The useful `scipy.constants` package provides the internationally agreed standard values and uncertainties for physical constants. SciPy 1.5.4 contains the 2018 CODATA internationally recommended values of many physical constants. They are held with their units and uncertainties in a dictionary `scipy.constants.physical_constants` keyed by an identifying string. For example,

```
>>> import scipy.constants as pc
>>> pc.physical_constants['Boltzmann constant']
(1.380649e-23, 'J K^-1', 7.9e-30)
```

This package also provides the convenience methods `value`, `unit`, and `precision` to retrieve the corresponding properties on their own:

```
>>> pc.value('electron mass')
9.1093837015e-31
>>> pc.unit('electron mass')
```

```
'kg'
>>> pc.precision('electron mass')
3.0737534961217373e-10
```

To save typing, it is usual to assign the value to a variable name at the start of a program, for example,

```
>>> a0 = pc.value('Bohr radius')
```

A full list of the constants and their names is given in the SciPy documentation; but the more important ones are listed in Table 4.12. Some particularly important constants have a direct variable assignment within `scipy.constants` (in SI units) and so can be imported directly:

```
>>> from scipy.constants import e, h, c
>>> e, h, c      # Electron mass, Planck constant, speed of light
(1.602176634e-19, 6.62607015e-34, 299792458.0)
```

Whenever this is the case, the variable name is given in the table. You will probably find it convenient to use the `scipy.constants` values. However, we should be aware that the package may be updated if newer values are released — this means that your code may produce slightly different results for different versions of SciPy.

Table 4.12: Important physical constants in `scipy.constants`

Constant string	Variable	Value	Unit
'atomic mass constant'		1.6605390666e-27	kg
'Avogadro constant'	N_A	6.02214076e+23	mol^{-1}
'Bohr magneton'		9.2740100783e-24	$\mathrm{J\,T}^{-1}$
'Bohr radius'		5.29177210903e-11	m
'Boltzmann constant'	k	1.380649e-23	$\mathrm{J\,K}^{-1}$
'electron mass'	m_e	9.1093837015e-31	kg
'elementary charge'	e	1.602176634e-19	C
'fine-structure constant'	alpha	0.0072973525693	
'molar gas constant'	R	8.314462618	$\mathrm{J\,mol}^{-1}\,\mathrm{K}^{-1}$
'neutron mass'	m_n	1.67492749804e-27	kg
'Newtonian constant of gravitation'	G	6.6743e-11	$\mathrm{m}^3\,\mathrm{kg}^{-1}\,\mathrm{s}^{-2}$
'Planck constant'	h	6.62607015e-34	$\mathrm{J\,Hz}^{-1}$
'proton mass'	m_p	1.67262192369e-27	kg
'reduced Planck constant'	hbar	1.054571817e-34	J s
'Rydberg constant'	Rydberg	10973731.56816	m^{-1}
'speed of light in vacuum'	c	299792458.0	$\mathrm{m\,s}^{-1}$

Some useful conversion factors and methods as well as SI prefixes are also defined in the `scipy.constants` package. For instance,

```
>>> import scipy.constants as pc
>>> pc.pound    # 1 lb in kg
0.45359236999999997
>>> pc.ounce    # 1 oz in kg
0.028349523124999998
>>> pc.degree   # Degree in radians
0.017453292519943295
>>> pc.nano
1e-09
```

On the other hand, the `scipy.special` package supplies a large number of algorithms for evaluating special functions that are often encountered in science, mathematical analysis, and engineering, including:

- Bessel functions with their zeros, derivatives, and integrals
- Spherical Bessel functions
- Legendre and associated Legendre functions
- Spherical Harmonics
- Gamma and beta functions
- Hermite polynomials
- Laguerre polynomials
- Chebyshev polynomials
- The error function and related integrals

You can find the full list of the special functions with their details in SciPy documentation. (Indeed, all the special functions discussed in the theoretical physics course can be found in SciPy.) Most of these special functions are implemented in SciPy as universal functions. In other words, they support broadcasting and vectorization (automatic array-looping), and so work as expected with NumPy arrays. Here we discuss a few representative examples of these functions.

Bessel functions is an important class of function with many applications to physics and engineering. SciPy provides several functions for evaluating them, their derivatives, and their zeros:

- `jv(v, x)` returns the Bessel function of the first kind at `x` for order `v`, $J_v(x)$, where `v` can be a real number or an integer.
- `yn(n, x)` and `yv(v, x)` return the Bessel function of the second kind at `x` for integer order `n` ($Y_n(x)$) and real order `v` ($Y_v(x)$), respectively.
- `iv(v, x)` return the modified Bessel function of the first kind at `x` for real order `v`, $I_v(x)$.
- `kn(n, x)` and `kv(v, x)` return the modified Bessel function of the second kind at `x` for integer order `n` ($K_n(x)$) and real order `v` ($K_v(x)$), respectively.
- The functions `jvp(v, x)`, `yvp(v, x)`, `ivp(v, x)`, and `kvp(v, x)` return the derivatives of the earlier mentioned functions. By default, the first order derivative is returned; set the optional argument to `n` to return the nth-order derivative.
- Several functions can be used to obtain the zeros of the Bessel functions. Probably the most useful are `jn_zeros(n, nt)`, `jnp_zeros(n, nt)`, `yn_zeros(n, nt)`, `ynp_zeros(n, nt)`, which return the first `nt` zeros of $J_n(x)$, $J_n'(x)$, $Y_n(x)$, and $Y_n'(x)$.

As an example, consider the vibrations of a thin circular membrane stretched across a rigid circular frame such as a drum head. Physicists revealed that such vibration can be described as normal modes written in terms of Bessel functions:

$$z(r, \theta; t) = AJ_n(kr) \sin n\theta \cos kvt$$

where (r, θ) describes a position in polar coordinates with the origin at the center of the membrane, t is time, and v is a constant depending on the tension and surface density of the drum. The modes are labeled by integers $n = 0, 1, 2, \ldots$ and $m = 1, 2, 3, \ldots$ where k is the mth zero of J_n. The following Python program `drumnormalmode.py` produces a plot of the displacement of the membrane in the $n = 3$, $m = 2$ normal mode at time $t = 0$ as shown in Figure 4.30.

```
# drumnormalmode.py
# This program produces a plot of the displacement of a vibrating
# circular membrane in the normal mode described by integers n >= 0
# and 0 < m <= mmax at time t = 0.
# Last Update on 19 Nov 2020 by F K Chow

import matplotlib.pyplot as plt
import numpy as np
from scipy.special import jv, jn_zeros
```

```
n, m, mmax = 3, 2, 5
def displacement(n, m, r, theta):
    """ Calculate the displacement of the membrane at (r, theta; t=0)
        in the normal mode for integers n >= 0 and 0 < m <= mmax. """
    # Pick off the mth zero of the Bessel function Jv
    k = jn_zeros(n, mmax)[m-1]
    return np.sin(n*theta)*jv(n, r*k) # Assume A = 1

# Create arrays of cartesian coordinates (x, y) and vertical
# displacement z for the required normal mode at time t = 0
r = np.linspace(0, 1, 100)
theta = np.linspace(0, 2*np.pi, 100)
x = np.array([rr*np.cos(theta) for rr in r])
y = np.array([rr*np.sin(theta) for rr in r])
z = np.array([displacement(n, m, rr, theta) for rr in r])

# Generate the contour plot of the displacement
fig, ax = plt.subplots()
ax.contour(x, y, z)
ax.set_xlim(-1, 1)
ax.set_ylim(-1, 1)
plt.show()
```

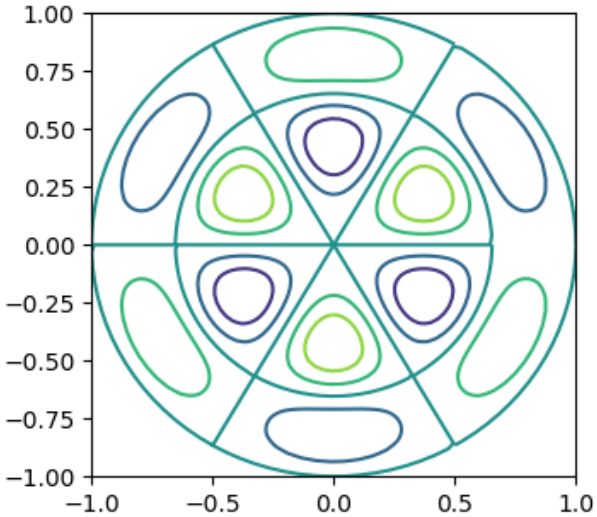

Figure 4.30: The $n = 3$, $m = 2$ normal mode of a vibrating circular membrane.

Hermite polynomials are set of orthogonal polynomials which have applications in various science such as physics and statistics. The physicist's Hermite polynomial of degree n is given by

$$H_n(x) = (-1)^n e^{x^2} \frac{d^n}{dx^n} e^{-x^2}$$

It is evaluated by

```
scipy.special.eval_hermite(n, x)
```

where **n** is the degree of the polynomial and **x** is an array-like object containing the points at which to evaluate the polynomial. To demonstrate how to use this function, consider the wave functions for a quantum harmonic oscillator:

$$\psi_n(x) = \frac{1}{\sqrt{2^n n!}} \left(\frac{m\omega}{\pi\hbar}\right)^{1/4} H_n\left(\sqrt{\frac{m\omega}{\hbar}}x\right) e^{-m\omega x^2/2\hbar}$$

where x is the position of the oscillator, m and ω are its mass and angular frequency, $\hbar = h/2\pi$ is the reduced Planck's constant, and $n = 0, 1, 2, \ldots$ is the quantum number of the state. Below is the listing of the Python program **qho.py** which plots the wave function of the oscillator (see Figure 4.31 for the plot).

```
# qho.py
# This program plots the wave function of a quantum harmonic
# oscillator in the state n >= 0. Set m*omega/hbar = 1 for
# convenience
# Last Update on 6 Jan 2022 by F K Chow

import matplotlib.pyplot as plt
import numpy as np
#from math import exp, factorial, sqrt
from scipy.special import eval_hermite

def waveFunc(n, x):
    """ Compute the wave function of the oscillator at x for the
        state n >= 0. """
    return (1.0/np.sqrt(np.sqrt(np.pi)*2**n*np.math.factorial(n)))\
           *eval_hermite(n, x)*np.exp(-x**2/2)

# Evaluate the wave function for -5 <= x <= 5
```

```
n = 3
x = np.linspace(-5, 5, 1000)
y = waveFunc(n, x)

# Generate the plot of the wave function
fig, ax = plt.subplots()
ax.plot(x, y)
ax.set_xlabel(r"$x$")
ax.set_ylabel(r"$\psi_3(x)$")
plt.show()
```

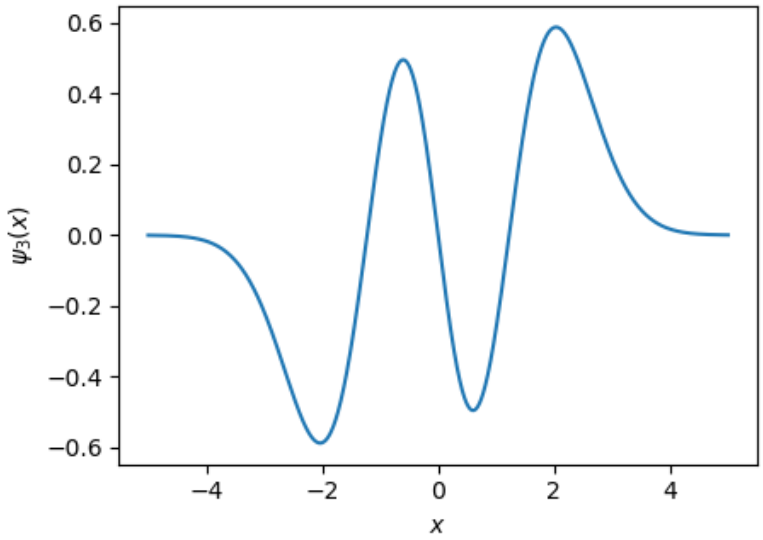

Figure 4.31: The wave function of a quantum harmonic oscillator in the state $n = 3$.

The **spherical harmonics** are a complete set of orthogonal functions on the sphere which are often encountered in solving partial differential equations in many scientific fields. The spherical harmonics used in SciPy are defined by the formula

$$Y_n^m(\phi, \theta) = \sqrt{\frac{(2n+1)}{4\pi}\frac{(n-m)!}{(n+m)!}} P_n^m(\cos\phi) e^{im\theta}$$

where $n = 0, 1, 2, \ldots$ is the **degree** of the harmonic, $m = -n, -n+1, \ldots, -1, 0, 1, \ldots n$ is its **order**, and the functions $P_n^m(x)$ are the associated Legendre polynomials. Notice that there are different conventions for the use of the symbols θ and ϕ. In SciPy, θ is the azimuthal angle taking values between 0 and 2π and ϕ is the polar angle taking values between 0 and π. The spherical harmonics is computed by calling

```
scipy.special.sph_harm(m, n, theta, phi)
```

where `n` and `m` are the degree and order of the harmonic, `theta` is an array-like object containing the azimuthal angles, and `phi` is an array-like object containing the polar angles. Visualizing the spherical harmonics is a little tricky because they are complex and defined in terms of angular coordinates (θ, ϕ). One way is to plot only the real part on a unit sphere. To illustrate how to work on it, the following Python program `sphericalhm.py` generates the visualization of the spherical harmonics as shown in Figure 4.32.

```
# sphericalhm.py
# This program generates the visualization of the spherical
# harmonics with order n and degree m.
# Last Update on 6 Jan 2022 by F K Chow

import matplotlib.pyplot as plt
import numpy as np
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D
from scipy.special import sph_harm

# Generate the azimuthal angle theta and polar angle phi
theta = np.linspace(0, 2*np.pi, 100)
phi = np.linspace(0, np.pi, 100)
theta, phi = np.meshgrid(theta, phi)

# Generate the Cartesian coordinates of the unit sphere
x = np.sin(phi)*np.cos(theta)
y = np.sin(phi)*np.sin(theta)
z = np.cos(phi)

# Calculate the spherical harmonic Y(n,m) and normalize to [0,1]
m, n = 2, 3
fcolors = sph_harm(m, n, theta, phi).real
fmax, fmin = fcolors.max(), fcolors.min()
fcolors = (fcolors - fmin)/(fmax - fmin)

# Generate the visualization of the spherical harmonics
```

```
fig = plt.figure(figsize=plt.figaspect(1.))
ax = fig.add_subplot(111, projection="3d")
ax.plot_surface(x, y, z, rstride=1, cstride=1,
                facecolors=cm.jet(fcolors))
ax.set_axis_off()   # Turn off the axis planes
plt.show()
```

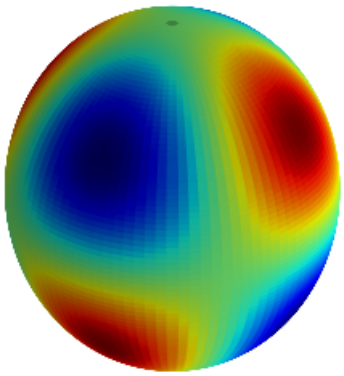

Figure 4.32: A visualization of the spherical harmonic with degree $n = 3$ and order $m = 2$.

4.3.2 Integration and Ordinary Differential Equations

The `scipy.integrate` package contains functions for computing definite integrals. It can evaluate both proper (with finite limits) and improper (with infinite limits) integrals. It can also perform numerical integration for systems of ordinary differential equations.

In `scipy.integrate`, the basic numerical integration routine is `quad`, which is based on the venerable FORTRAN 77 QUADPACK library. It uses **adaptive quadrature** to approximate the value of an integral by dividing its domain into subintervals that are chosen iteratively to meet a particular tolerance, i. e. estimated absolute or relative error. The function `quad` returns two values in a tuple — the value of the integral and an estimate of the absolute error in the result. In its simplest form, it takes three arguments: a Python function object `func` corresponding to the function to integrate and the limits of integration `a` and `b`. Note that `func` must take at least one argument; if it takes more than one, it is integrated along the coordinate corresponding to the first argument. For example, the integral $\int_1^4 x^{-2}\,dx = 3/4$ can be evaluated numerically by

```
>>> from scipy.integrate import quad
>>> def func(x):
```

```
...     return 1/x**2
...
>>> quad(func, 1, 4)
(0.7500000000000002, 1.9132345482589 93e-09)
```

To evaluate improper integrals, use `np.inf`:

```
>>> def func(x):
...     return np.exp(-x**2)
...
>>> quad(func, 0, np.inf)
(0.8862269254527579, 7.101318378329813e-09)
>>> np.sqrt(np.pi)/2    # The analytical result
0.8862269254527579
```

Functions with singularities or discontinuities can cause problems for the numerical quadrature routine even if the required integral is well-defined. For example, the sinc function $f(x) = \sin(x)/x$ has a removable singularity at $x = 0$, which causes the following simple application of `quad` to fail:

```
>>> def sinc(x):
...     return np.sin(x)/x
...
>>> quad(sinc, -2, 2)
__main__:2: RuntimeWarning: invalid value encountered in double_scalars
__main__:1: IntegrationWarning: The occurrence of roundoff error is det
ected, which prevents
  the requested tolerance from being achieved.  The error may be
  underestimated.
(nan, nan)
```

The solution is to configure `quad` by passing a list of such **break points** (the list need not to be ordered) to the `points` argument:

```
>>> quad(sinc, -2, 2, points=[0,])
(3.210825953605389, 3.5647329017567276e-14)
```

Note that break points cannot be specified with infinite limits.

The arguments `epsrel` and `epsabs` of the function `quad` allow the specification of a desired accuracy of the quadrature as a relative or absolute tolerance. The default

values are both `1.49e-8`. However, the integration can be done faster if a less-accurate answer is required. As an example, consider integrating the rapidly varying function $f(x) = e^{-|x|} \sin^2 x^2$:

```
>>> def func(x):
...     return np.exp(-np.abs(x))*np.sin(x**2)**2
...
>>> quad(func, -1, 2, epsabs=0.1)
(0.29551455828969975, 0.0015295718279094228)
>>> quad(func, -1, 2, epsabs=1.49e-8)
(0.29551455505239044, 4.449763314815072e-10)
```

Note that `epsabs` is only a requested upper bound: the actual estimated accuracy in the result may be much better. And the actual result may be more accurate than this estimate.

If a function takes one or more parameters in addition to its principal argument, these need to be passed to `quad` as a tuple in the argument `args`. For instance, the integral

$$I_{n,m} = \int_{-\pi/2}^{\pi/2} \sin^n x \cos^m x \, dx$$

can be evaluated numerically as follows:

```
>>> def func(x, n, m):
...     return np.sin(x)**n*np.cos(x)**m
...
>>> n, m = 2, 2
>>> quad(func, -np.pi/2, np.pi/2, args=(n, m))
(0.39269908169872414, 2.882349231820601e-10)
```

Note that the additional parameters (`n` and `m` here) must appear as arguments to our function *after* the variable to be integrated over (`x` here).

The `scipy.integrate` functions `dblquad`, `tplquad`, and `nquad` evaluate double, triple, and multiple integrals respectively. In general, the limits on one coordinate may depend on another coordinate for these integrals. Thus the syntax for calling these functions is a little more complicated.

The function `dblquad` evaluates the double integral:

$$\int_a^b \int_{g(x)}^{h(x)} f(x, y) \, dy \, dx.$$

Its first argument is a function of at least two variables, `func(y, x,...)`, which corresponds to the function $f(x, y)$. Note that `func` must take `y` as its first argument and `x` as its second argument. The integral limits are passed to `dblquad` in four further arguments. The first two arguments `a` and `b` specify the lower and upper limits on the x-integral respectively, as for `quad`. The next two arguments `gfun` and `hfun` are the lower and upper limits on the y-integral and they must be *callable objects* taking a single floating point argument which is the value of `x` at which the limit applies. If either of the y-integral limits does not depend on x, then `gfun` or `hfun` can return a constant value. As a simple example, the double integral

$$\int_1^4 \int_0^2 x^2 y \, dy \, dx$$

can be evaluated numerically as follows:

```
>>> from scipy.integrate import dblquad
>>> def func(y, x):
...     return x**2*y
...
>>> a, b = 1, 4
>>> def gfun(x):
...     return 0
...
>>> def hfun(x):
...     return 2
...
>>> dblquad(func, a, b, gfun, hfun)
(42.00000000000001, 4.662936703425658e-13)
```

Observe that `gfun` and `hfun` are each called with a value of `x`; but they return a constant (`0` and `2` respectively) no matter what this value is.

A double integral can be used to find the area of a two-dimensional shape bounded by one or more functions. For instance, consider the area inside the polar curve $r = 2+2\sin\theta$ but outside the circle defined by $r = 2$ for θ in $[0, 2\pi]$ (see Figure 4.33). Since these curves intersect at $\theta = 0$ and $\theta = \pi$, the area is given by the double integral

$$A = \int_0^{\pi} \int_2^{2+2\sin\theta} r \, dr \, d\theta$$

where $r \, dr \, d\theta$ is the infinitesimal area element in polar coordinates.

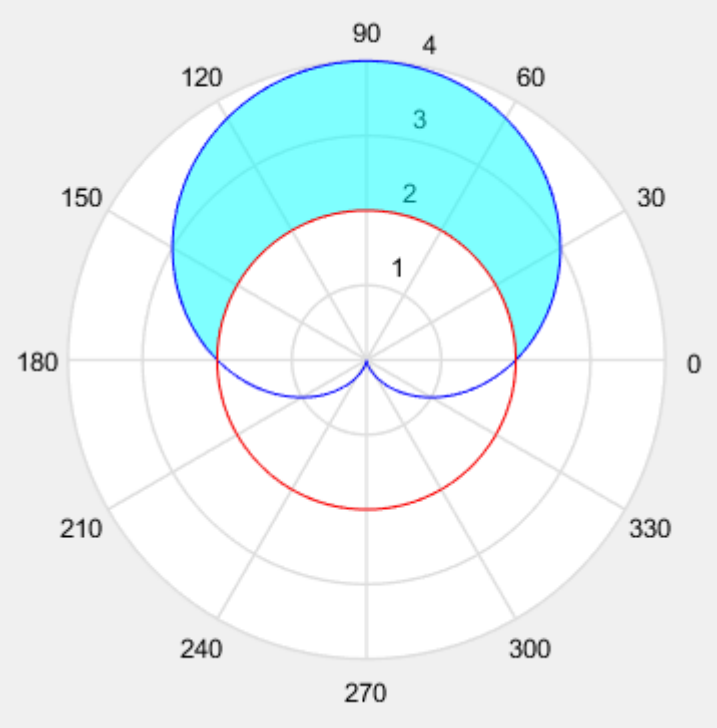

Figure 4.33: The region defined as the area inside the polar curve $r = 2 + 2\sin\theta$ but outside the circle $r = 2$.

It is fairly straightforward to show that this integral is equal to $A = 8 + \pi$. So it is easy to check with the numerical result:

```
>>> from scipy.integrate import dblquad
>>> def func(r, theta):
...     return r
...
>>> def gfun(theta):
...     return 2
...
>>> def hfun(theta):
...     return 2 + 2*np.sin(theta)
...
>>> dblquad(func, 0, np.pi, gfun, hfun)
(11.141592653589791, 1.2369652695009882e-13)
>>> 8 + np.pi
11.141592653589793
```

Here, the function to integrate is simply `r`. Moreover, the limits on r are 2 and $2 + 2\sin\theta$ for the inner integral while θ ranges from 0 to π for the outer integral.

The method `tplquad` evaluates triple integrals and takes a function of three variables `func(z, y, x)` and six further arguments. These arguments are constant x-limits `a` and `b`, y-limits `gfun(x)` and `hfun(x)` which are functions of `x` as for `dblquad`, and z-limits `qfun(x,y)` and `rfun(x,y)` which are functions of `x` and `y` in that order. To give you an idea, consider the volume of a unit sphere which can be expressed as a triple integral in spherical polar coordinates with constant limits:

$$\int_0^{2\pi} \int_0^{\pi} \int_0^{1} r^2 \sin\theta \, dr \, d\theta \, d\phi$$

This integral can be evaluated numerically by

```
>>> from scipy.integrate import tplquad
>>> def func(r, theta, phi):
...     return r**2*np.sin(theta)
...
>>> def qfun(phi, theta):
...     return 0
...
>>> def rfun(phi, theta):
...     return 1
...
>>> def gfun(phi):
...     return 0
...
>>> def hfun(phi):
...     return np.pi
...
>>> tplquad(func, 0, 2*np.pi, gfun, hfun, qfun, rfun)
(4.18879020478639, 4.650491330678174e-14)
```

The volume can be also expressed in Cartesian coordinates with limits as functions:

$$8 \int_0^{1} \int_0^{\sqrt{1-x^2}} \int_0^{\sqrt{1-x^2-y^2}} dz \, dy \, dx$$

which can be evaluated numerically as follows:

```
>>> def func(z, y, x):
...     return 8
...
```

```
>>> def qfun(x, y):
...     return 0
...
>>> def rfun(x, y):
...     return np.sqrt(1 - x**2 - y**2)
...
>>> def gfun(x):
...     return 0
...
>>> def hfun(x):
...     return np.sqrt(1 - x**2)
...
>>> tplquad(func, 0, 1, gfun, hfun, qfun, rfun)
(4.188790204786391, 7.067022522733168e-10)
```

Higher dimensional integrations are handled by the `scipy.integrate.nquad` function which will not be discussed here. You can find the documentation and examples for this method online.

Ordinary differential equations can be solved numerically with `scipy.integrate.odeint`. This function is based on the well-tested Fortran LSODA routine, which can automatically switch between **stiff** and **nonstiff** algorithms.[1] Notice that `odeint` only solves first-order differential equations. *If it's used to solve a higher-order equation, then it must be decomposed into a system of first-order equations first*, which will be explained later.

In its simplest use for the solution of a single first-order ordinary differential equation

$$\frac{dy}{dt} = f(y, t),$$

`odeint` takes three arguments: a function object returning dy/dt, an initial condition y_0, and a sequence of t values at which to calculate the solution $y(t)$. As an example, consider the first-order differential equation describing the rate of the first-order chemical reaction $\mathrm{A} \rightarrow \mathrm{P}$ in terms of the concentration of the reactant A:

$$\frac{d[\mathrm{A}]}{dt} = -k[\mathrm{A}].$$

[1] A differential equation is said to be stiff if a numerical method is required to take excessively small steps in its intervals of integration in relation to the smoothness of the exact underlying solution.

This equation has an easily obtainable analytical solution:

$$[\mathrm{A}] = [\mathrm{A}]_0 e^{-kt}.$$

where $[\mathrm{A}]_0$ is the initial concentration of A.

To solve the above equation numerically with `odeint`, write it in the form as shown above with a single dependent variable $y(t) = [\mathrm{A}]$ which is a function of the independent variable t (time). Then we have:

$$\frac{dy}{dt} = -ky.$$

We need to provide a function returning dy/dt as $f(y, t)$, an initial condition $y(0)$, and a sequence of time points upon which to calculate the solution. The derivative function is simply:

```
def dydt(y, t):
    return -k*y
```

which must take the dependent variable as its first argument and the independent variable as its second argument. Below is the Python program `firstorderrxn.py` which compares the numerical and exact solutions for a first-order chemical reaction with $k = 0.2\,\mathrm{s}^{-1}$ and $y(0) = [\mathrm{A}]_0 = 100\%$. The resultant plot is given in Figure 4.34.

```
# firstorderrxn.py
# This program compares the numerical and exact solutions for a
# first-order chemical reaction.
# Last Update on 20 Nov 2020 by F K Chow

import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import odeint

# Reaction rate constant in s^-1 and initial condition y(t=0) in %
k = 0.2
y0 = 100
# A suitable grid of time points for the reaction
t = np.linspace(0, 20, 20)

def dydt(y, t):
    """ Return dy/dt = f(y,t) at time t. """
```

```
    return -k*y

# Integrate the differential equation
y = odeint(dydt, y0, t)

# Plot and compare the numerical and exact solutions
fig, ax = plt.subplots()
ax.plot(t, y, "o", color="k", label="odeint")
ax.plot(t, y0*np.exp(-k*t), color="gray", label="Exact")
ax.set_xlabel(r"$t/\mathrm{s}$")
ax.set_ylabel("Remaining reactant (%)")
ax.legend()
plt.show()
```

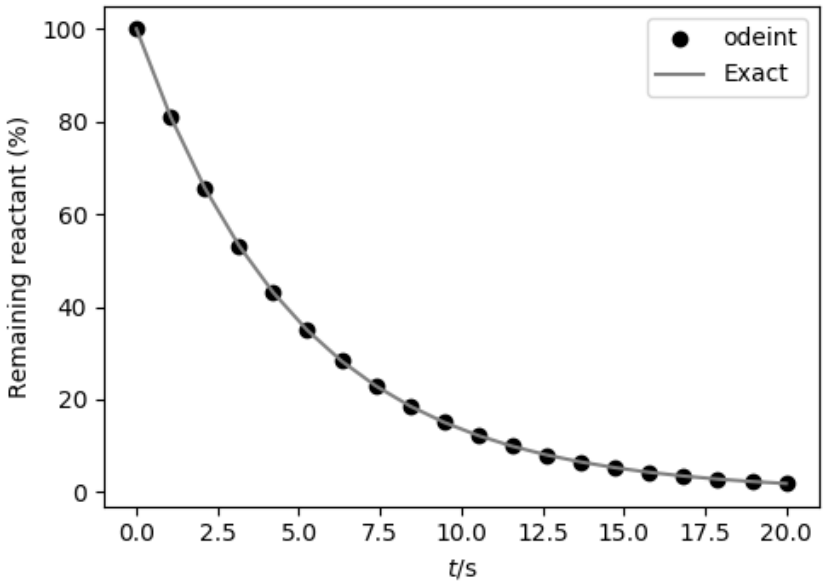

Figure 4.34: Numerical and exact solutions for the exponential decay of a reactant in a first-order chemical reaction.

Just like the `quad` family of routines, if the function returning the derivative requires further arguments, then they can be passed to `odeint` in the `args` parameter. In the previous example, `k` is resolved in global scope. But we could pass it with:

```
def dydt(y, t, k):
    return -k*y
```

Beware that the additional parameters must appear after the dependent and independent variables. The call to `odeint` would then be:

```
y = odeint(dydt, y0, t, args=(k,))
```

The `odeint` function can also solve a set of coupled first-order differential equations in more than one dependent variable $y_1(t)$, $y_2(t)$, ..., $y_n(t)$:

$$\begin{aligned}
\frac{dy_1}{dt} &= f_1(y_1, y_2, \ldots, y_n; t) \\
\frac{dy_2}{dt} &= f_2(y_1, y_2, \ldots, y_n; t) \\
&\vdots \\
\frac{dy_n}{dt} &= f_n(y_1, y_2, \ldots, y_n; t)
\end{aligned}$$

In this case, the function passed to `odeint` must return a sequence of derivatives dy_1/dt, dy_2/dt, ..., dy_n/dt for each of the dependent variables. In other words, it evaluates the functions $f_i(y_1, y_2, \ldots, y_n; t)$ for each of the y_i passed to it in a sequence y. The form of this function is:

```
def dydt(y, t):
    # y = [y1, y2, y3...] is a sequence of dependent variables
    dy1dt = f1(y, t) # calculate dy1/dt as f1(y1,y2,...,yn;t)
    dy2dt = f2(y, t) # calculate dy2/dt as f2(y1,y2,...,yn;t)
    dy3dt = f3(y, t) # calculate dy3/dt as f3(y1,y2,...,yn;t)
      ⋮
    # Return the derivatives in a sequence such as a tuple:
    return dy1dt, dy2dt, ..., dyndt
```

For a concrete example, consider a reaction proceeds via two first-order reaction steps: A $\rightarrow$ B $\rightarrow$ P with rate constants k_1 and k_2. The differential equations governing the rate of change of A and B are

$$\begin{aligned}
\frac{d[\mathrm{A}]}{dt} &= -k_1[\mathrm{A}], \\
\frac{d[\mathrm{B}]}{dt} &= k_1[\mathrm{A}] - k_2[\mathrm{B}].
\end{aligned}$$

Again, we can solve this pair of coupled equations analytically. In our numerical solution, we let $y_1 = [\mathrm{A}]$ and $y_2 = [\mathrm{B}]$ and obtain

$$\begin{aligned}
\frac{dy_1}{dt} &= -k_1 y_1, \\
\frac{dy_2}{dt} &= k_1 y_1 - k_2 y_2.
\end{aligned}$$

Below is the Python program **coupfirstorderrxns.py** which solves these equations numerically for $k_1 = 0.2\,\mathrm{s}^{-1}$, $k_2 = 0.8\,\mathrm{s}^{-1}$ and initial conditions $y_1(0) = 100$, $y_2(0) = 0$, and compares with the analytical result. The program output is shown in Fig. 4.35.

```
# coupfirstorderrxns.py
# This program compares the numerical and exact solutions for two
# coupled first—order chemical reactions.
# Last Update on 6 Jan 2022 by F K Chow

import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import odeint

# First—order reaction rate constant in s^—1
k1, k2 = 0.2, 0.8
# Initial condition on y1, y2: [A](t=0) = 100, [B](t=0) = 0
A0, B0 = 100, 0
# A suitable grid of time points for the reaction
t = np.linspace(0, 20, 20)

def dydt(y, t, k1, k2):
    """ Return dy_i/dt = f(y_i,t) at time t. """
    y1, y2 = y
    dy1dt = —k1*y1
    dy2dt = k1*y1 — k2*y2
    return dy1dt, dy2dt

# Integrate the DE to obtain the numerical solution
y0 = A0, B0
y1, y2 = odeint(dydt, y0, t, args=(k1, k2)).T
A, B = y1, y2
P = A0 + B0 — A — B  # [P] is determined by conservation

# Compute the exact solution
Aexact = A0*np.exp(—k1*t)
```

```
Bexact = A0*k1/(k2-k1)*(np.exp(-k1*t) - np.exp(-k2*t))
Pexact = A0 - Aexact - Bexact

# Plot and compare the numerical and exact solutions
fig, ax = plt.subplots()
ax.plot(t, A, "o", label="[A]")
ax.plot(t, B, "^", label="[B]")
ax.plot(t, P, "d", label="[P]")
ax.plot(t, Aexact)
ax.plot(t, Bexact)
ax.plot(t, Pexact)
ax.set_xlabel(r"$t/\mathrm{s}$")
ax.set_ylabel("Concentration (arbitrary units)")
ax.legend()
plt.show()
```

Note that `odeint` returns a 2D array with the values of each dependent variable in the *rows*. If we want to unpack this array to separate one-dimensional arrays `y1`, `y2`, and so on, then we need the transpose of this returned array.

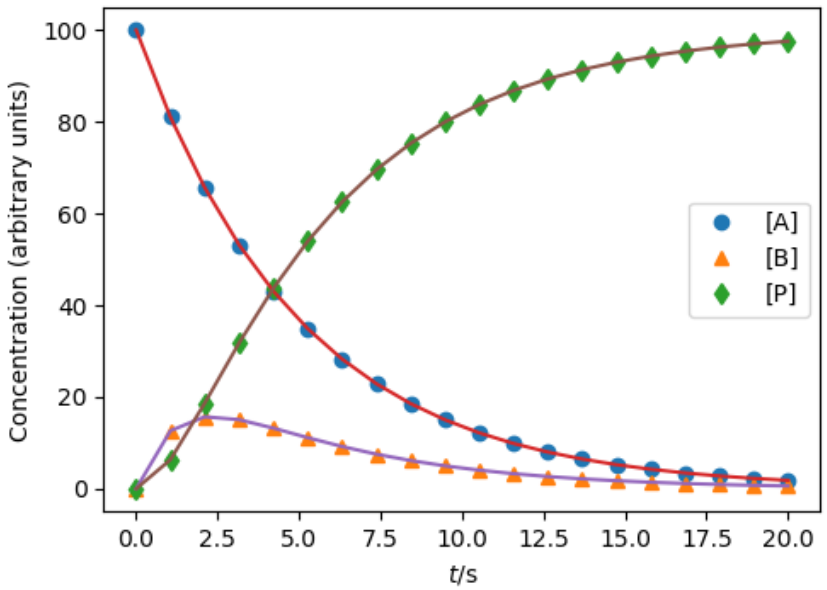

Figure 4.35: Numerical and exact solutions for two coupled first-order reactions.

To solve a higher-order ordinary differential equation with `odeint`, the equation must be first reduced into a system of first-order differential equations. In general, any differential equation with a single dependent variable of order n can be written as a system of

n first-order differential equations in n dependent variables. For instance, the equation of motion for a harmonic oscillator is a second-order differential equation:

$$\frac{d^2x}{dt^2} = -\omega^2 x,$$

where x is the displacement from equilibrium position and ω is the angular frequency. This equation may be decomposed into two first-order equations as follows:

$$\frac{dx_1}{dt} = x_2,$$
$$\frac{dx_2}{dt} = -\omega^2 x_1,$$

where x_1 represents x and x_2 represents dx/dt. This pair of coupled first order equations can be solved as before as in the Python program **harmonicmotion.py**. The plot produced by this program is depicted in Figure 4.36.

```
# harmonicmotion.py
# This program solves the equation of motion for a harmonic oscillator.
# Last Update on 20 Nov 2020 by F K Chow

import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import odeint

# Harmonic oscillator angular frequency s^-1
omega = 0.9
# Initial conditions on x1=x and x2=dx/dt at t=0 in cm and cm/s^-1
x0 = 3, 0
# A suitable grid of time points for the reaction
t = np.linspace(0, 20, 100)

def dxdt(x, t, omega):
    """ Return dx/dt = f(x,t) at time t """
    x1, x2 = x
    dx1dt = x2
    dx2dt = -omega**2*x1
    return dx1dt, dx2dt
```

```
# Integrate the DE to obtain the numerical solution
x1, x2 = odeint(dxdt, x0, t, args=(omega,)).T

# Plot and compare the numerical and exact solutions
fig, ax = plt.subplots()
ax.plot(t, x1, "o", color="k", label="odeint")
ax.plot(t, x0[0]*np.cos(omega*t), color="gray", label="Exact")
ax.set_xlabel(r"$t/\mathrm{s}$")
ax.set_ylabel(r"$x/\mathrm{cm}$")
ax.legend()
plt.show()
```

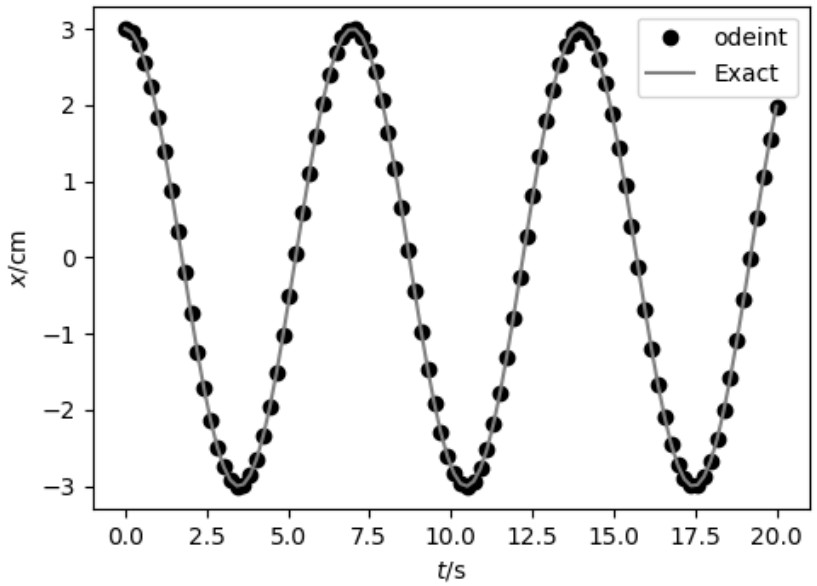

Figure 4.36: Numerical and exact solutions for the harmonic oscillator.

Indeed, the `odeint` function is a simplified interface to the more advanced method `scipy.integrate.ode` which provides a range of different numerical integrators, including Runge-Kutta algorithms and support for complex-valued variables.

4.3.3 Interpolation

Interpolation is the process of estimating intermediate values from a set of known data points. The `scipy.interpolate` package contains a large variety of functions and classes for interpolation in one and more dimensions. Some of the important ones are discussed in this subsection.

Table 4.13: Options of interpolation methods specified by the `kind` argument to the constructor of the class `scipy.interpolate.interp1d`

Option	Description
`'linear'`	The default option which uses only the values from the original data arrays bracketing the desired point
`'nearest'`	"Snap" to the nearest data point
`'zero'`	A zeroth-order spline: interpolates to the last value seen in its traversal of the data arrays
`'slinear'`	First-order spline interpolation (the same as `'linear'` in practice)
`'quadratic'`	Second-order spline interpolation
`'cubic'`	Cubic spline interpolation

The `scipy.interpolate.interp1d` class serves as a convenient method for performing one-dimensional interpolation. If the constructor of this class is called with arrays of data points `x` and `y` passed as arguments, then a function is returned which can be called to generate interpolated values of `y` at intermediate values of `x`. The default interpolation method used by `interp1d` is linear. But other interpolation methods can be used by specifying the `kind` argument (see Table 4.13). The following Python program `interp1ddemo.py` illustrates some of the interpolation methods available in `interp1d` with the results compared as shown in Figure 4.37.

```
# interp1ddemo.py
# This program illustrates some of the one-dimensional interpolation
# methods available in scipy.interpolate.interp1d.
# Last Update on 20 Nov 2020 by F K Chow

import matplotlib.pyplot as plt
import numpy as np
from scipy.interpolate import interp1d

# Generate the data to be interpolated
A, nu, k = 10, 4, 2
def f(x, A, nu, k):
    """ Return f(x) = A*e^(-k*x)*cos(2*pi*nu*x) """
    return A*np.exp(-k*x)*np.cos(2*np.pi*nu*x)
```

```
xmax, nx = 0.5, 8
x = np.linspace(0, xmax, nx)
y = f(x, A, nu, k)

# Perform the interpolation using different methods
f_nearest = interp1d(x, y, kind="nearest")
f_linear = interp1d(x, y)
f_cubic = interp1d(x, y, kind="cubic")

# Plot the data for comparison
fig, ax = plt.subplots()
x2 = np.linspace(0, xmax, 100)
ax.plot(x, y, "o", label="data points")
ax.plot(x2, f(x2, A, nu, k), label="exact")
ax.plot(x2, f_nearest(x2), linestyle="--", label="nearest")
ax.plot(x2, f_linear(x2), label="linear")
ax.plot(x2, f_cubic(x2), linestyle=":", label="cubic")
ax.legend()
plt.show()
```

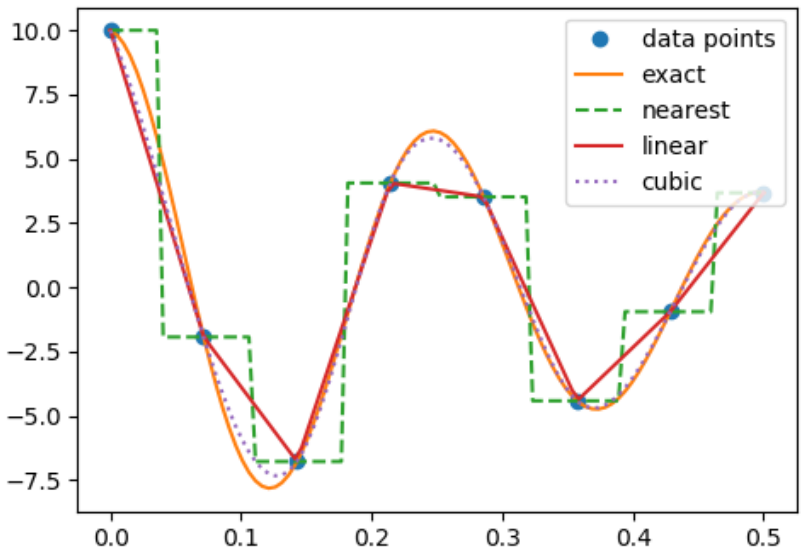

Figure 4.37: One-dimensional interpolation with `scipy.interpolate.interp1d` using different interpolation methods.

Next, we consider two kinds of multivariate interpolation corresponding to whether the

source data are structured (i. e. arranged on some kind of grid) or not. To interpolate over a two-dimensional grid of data, we can use the `scipy.interpolate.interp2d` class which is just the two-dimensional version of `interp1d`. However, it requires a two-dimensional array of values `z` and the two corresponding one-dimensional coordinate arrays `x` and `y` which need not have uniform spacing. Three kinds of spline interpolation methods are supported through the `kind` argument to `interp2d`: `'linear'` (the default), `'cubic'`, and `'quintic'`. Below is the Python program `interp2ddemo.py` demonstrating the use of this class for two-dimensional interpolation. Figure 4.38 is the output of this program

---

```
# interp2ddemo.py
# This program demonstrates two-dimensional interpolation with
# scipy.interpolate.interp2d.
# Last Update on 19 Nov 2020 by F K Chow

import matplotlib.pyplot as plt
import numpy as np
from scipy.interpolate import interp2d

# Generate the data to be interpolated
x = np.linspace(0, 4, 13)
y = np.array([0, 2, 3, 3.5, 3.75, 3.875, 3.9375, 4])
X, Y = np.meshgrid(x, y)
Z = np.sin(np.pi*X/2)*np.exp(Y/2)

# Interpolate the data onto a finer grid
f = interp2d(x, y, Z, kind="cubic")
x2 = np.linspace(0, 4, 65)
y2 = np.linspace(0, 4, 65)
X2, Y2 = np.meshgrid(x2, y2)
Z2 = f(x2, y2)

# Plot the data for comparison
fig, ax = plt.subplots(nrows=1, ncols=2)
ax[0].pcolormesh(X, Y, Z)
ax[1].pcolormesh(X2, Y2, Z2)
```

```
plt.show()
```

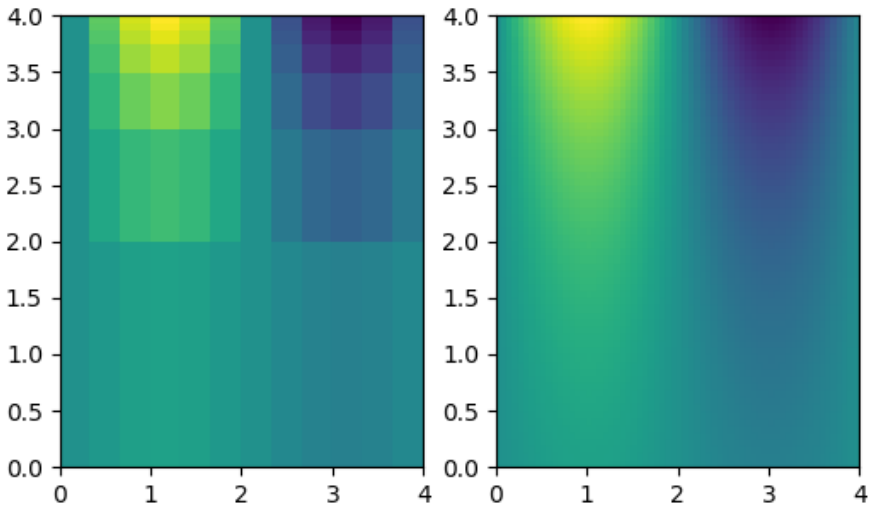

Figure 4.38: Two-dimensional interpolation from a coarse rectangular grid (left) to a finer one (right) with `scipy.interpolate.interp2d`.

If the mesh of (x, y) coordinates form a *regularly spaced grid*, then the fastest way to interpolate values from values of `z` is to use the `scipy.interpolate.RectBivariateSpline` class as illustrated in the following Python program `rbsplinedemo.py`. The resultant plot is depicted in Figure 4.39. Note that here for our two-dimensional array `Z` defined using the two-dimensional arrays `X` and `Y` generated from the `meshgrid`, the constructor method of `RectBivariateSpline` expects the corresponding one-dimensional arrays `y` and `x` to be passed in this order (which is opposite to that of `interp2d`).

```
# rbsplinedemo.py
# This program demonstrates two-dimensional interpolation with
# scipy.interpolate.RectBivariateSpline.
# Last Update on 20 Nov 2020 by F K Chow

import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
from scipy.interpolate import RectBivariateSpline

# Generate the data on a regularly spaced, coarse grid
xmax, ymax, dx, dy = 2, 4, 0.4, 0.4
```

```
x = np.arange(-xmax, xmax, dx)
y = np.arange(-ymax, ymax, dy)
X, Y = np.meshgrid(x, y)
Z = np.exp(-(2*X)**2-(Y/2)**2)

# Interpolate the data onto a regularly spaced, finer grid
interp_spline = RectBivariateSpline(y, x, Z)
dx2, dy2 = 0.16, 0.16
x2 = np.arange(-xmax, xmax, dx2)
y2 = np.arange(-ymax, ymax, dy2)
X2, Y2 = np.meshgrid(x2,y2)
Z2 = interp_spline(y2, x2)

# Plot the data for comparison
fig, ax = plt.subplots(nrows=1, ncols=2,
                       subplot_kw={"projection": "3d"})
ax[0].plot_wireframe(X, Y, Z, color="k")
ax[1].plot_wireframe(X2, Y2, Z2, color="k")
for axes in ax:
    axes.set_zlim(-0.2,1)
    axes.set_axis_off()
fig.tight_layout()
plt.show()
```

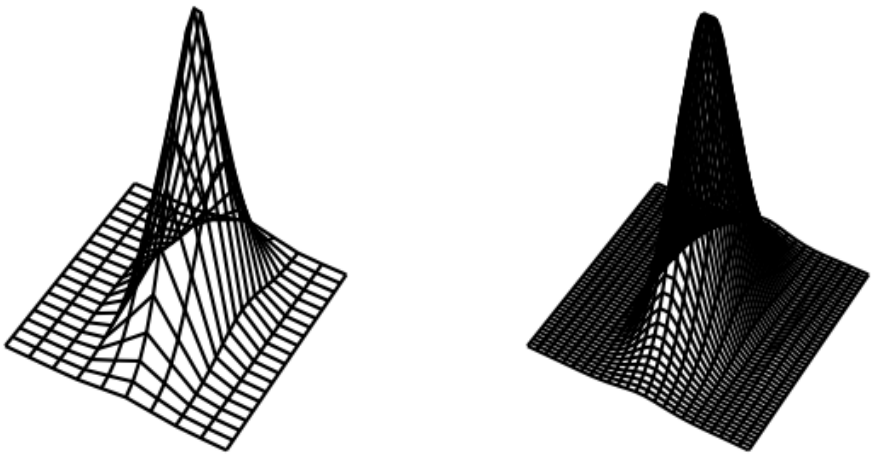

Figure 4.39: Two-dimensional interpolation from a coarse rectangular grid (left) to a finer one (right) with `scipy.interpolate.RectBivariateSpline`.

To interpolate unstructured data (i. e. data points provided at *arbitrary* coordinates (x, y)) onto a grid, the method `scipy.interpolate.griddata` can be used. Its basic usage for two dimensions is:

```
scipy.interpolate.griddata(points, values, xi, method="linear")
```

Note that the provided data are given as the one-dimensional array `values` at the coordinates `points` which is provided as a tuple of arrays `x` and `y` or as a single array of shape `(n,2)` where `n` is the length of the `values` array. Moreover, `xi` is an array of the coordinate grid over which to interpolate data. And the argument `method` specifies the method of interpolation which can be `'linear'` (the default), `'nearest'`, and `'cubic'`. Below is the Python program `griddatademo.py` illustrates different kinds of interpolation methods available in `griddata` with the results compared as shown in Figure 4.40.

```
# griddatademo.py
# This program illustrates different kinds of interpolation methods
# available in scipy.interpolate.griddata.
# Last Update on 20 Nov 2020 by F K Chow

import matplotlib.pyplot as plt
import numpy as np
from scipy.interpolate import griddata

# Generate the data to be interpolated
x = np.linspace(-1,1,100)
y = np.linspace(-1,1,100)
X, Y = np.meshgrid(x,y)
def f(x, y):
    """ Our model function """
    s = np.hypot(x, y)
    phi = np.arctan2(y, x)
    tau = s + s*(1-s)/5*np.sin(6*phi)
    return 5*(1-tau) + tau
T = f(X, Y)

# Choose npts point randomly from the domain of our model function
npts = 400
```

```
px, py = np.random.choice(x, npts), np.random.choice(y, npts)

# Plot the model function and the randomly selected sample points
fig, ax = plt.subplots(nrows=2, ncols=2)
ax[0,0].contourf(X, Y, T)
ax[0,0].scatter(px, py, c="k", alpha=0.2, marker=".")
ax[0,0].set_title("Sample points on f(X,Y)")

# Interpolate using three different methods and plot the results
for i, m in enumerate(("nearest", "linear", "cubic")):
    Ti = griddata((px, py), f(px,py), (X, Y), method=m)
    r, c = (i+1)//2, (i+1)%2
    ax[r,c].contourf(X, Y, Ti)
    ax[r,c].set_title("method = {}".format(m))
plt.show()
```

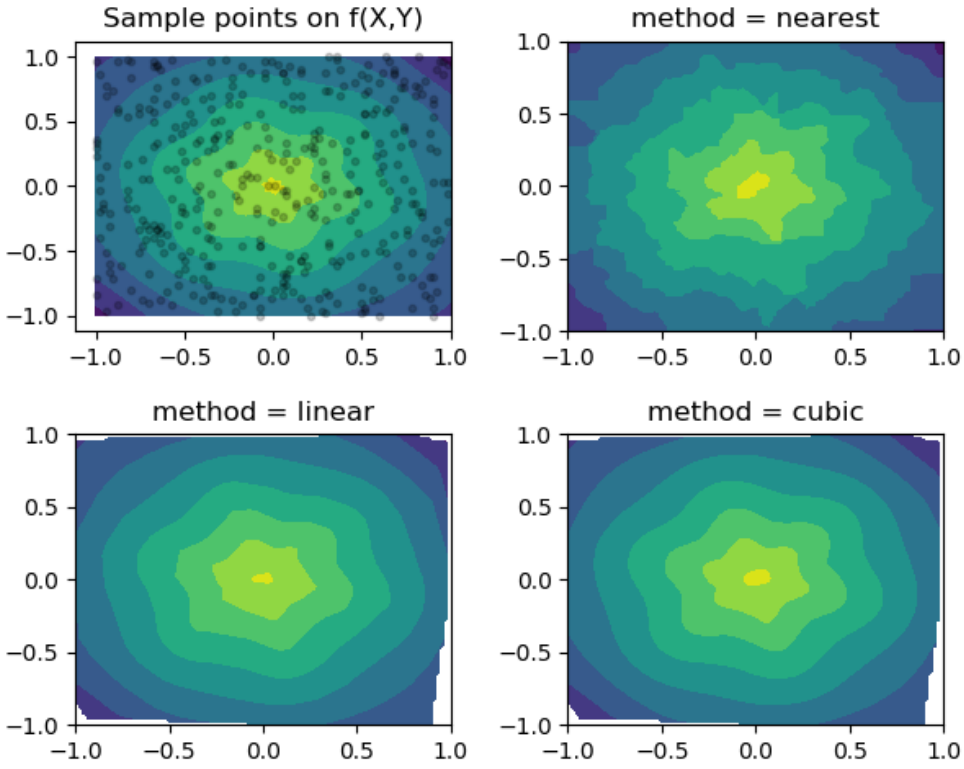

Figure 4.40: Two-dimensional interpolation with `scipy.interpolate.griddata` using different interpolation methods.

4.3.4 Data-fitting and Root-finding

The `scipy.optimize` package provides a number of popular algorithms for least squares data-fitting, multidimensional equation solving (i. e. root-finding), and minimization of multidimensional functions with or without additional constraints. Here we will take an overview on the important options available for data-fitting and root-finding. However, we should bear in mind that the best choice of algorithm will depend on the individual function being analyzed. For an arbitrary function, there is no guarantee that a particular method will converge on the desired root or that if it does so it will converge quickly. Some algorithms are better suited to certain functions than others, and the more you know about your function the better. SciPy can be configured to issue a warning message when a particular algorithm fails, and this message can usually help to analyze the problem. Furthermore, the result returned often depends on the initial guess provided to the algorithm. For example, we might expect (but cannot guarantee) that most numerical root-finders return the "nearest" root to the initial guess.

Least squares fitting is the method of finding the best fit curve to a set of data points by minimizing the sum of the squares of the **residuals** (i. e. deviations) of the points from the curve. In other words, the goal of this method is to find the values of the parameters $a_1, a_2, \ldots, a_m$ of a fitting function f that minimizes the sum

$$S = \sum_{i=1}^{N} [y_i - f(x_i; a_1, a_2, \ldots, a_m)]^2$$

where $\{(x_1, y_1), (x_2, y_2), \ldots (x_N, y_N)\}$ is the set of data points. If the fitting function f is a linear function in the parameters, then the method is called **linear least squares fitting**; otherwise, it is called **nonlinear least squares fitting**. Note that if uncertainties are given for the data points, the points can be weighted differently so that more weight is given to the points with less uncertainties. In SciPy, we can perform nonlinear least squares fitting with the function `scipy.optimize.leastsq`. The basic syntax for calling this function is:

```
scipy.optimize.leastsq(rfunc, p0, args=a)
```

where `rfunc` is the function that calculates the residuals, `p0` is the initial guess for the fitted parameters, and `a` is a tuple of any additional arguments to `rfunc` (which are typically arrays of the data and one or more independent variables). For example, consider fitting the artificial noisy decaying cosine function $f(t) = Ae^{-t/\tau}\cos(2\pi ft)$ which are generated as follows:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> A, freq, tau = 15, 5, 0.4
>>> def f(t, p):
...     A, freq, tau = p
...     return A*np.exp(-t/tau)*np.cos(2*np.pi*freq*t)
...
>>> tmax, dt = 1, 0.01
>>> t = np.arange(0, tmax, dt)
>>> yexact = f(t, (A, freq, tau))
>>> y = yexact + np.random.randn(len(yexact))*2
```

To fit this noisy data `y` to the parameters `A`, `freq`, and `tau` (pretending we don't know them), we first define the residuals function:

```
>>> def residuals(p, y, t):
...     A, freq, tau = p
...     return y - f(t, (A, freq, tau))
```

The first argument is the sequence of parameters `p` which we unpack into named variables for clarity. The additional arguments needed are the data itself `y` and the independent variable `t`. Now make some initial guesses for the parameters that aren't too wildly off and call `leastsq`:

```
>>> from scipy.optimize import leastsq
>>> p0 = 5, 5, 1
>>> plsq = leastsq(residuals, p0, args=(y, t))
>>> plsq[0]
array([13.73316814,  4.9855825 ,  0.44798887])
```

Like SciPy's other optimization routines, `leastsq` can be configured to return more information about its working. But here we report only the best fit parameters which is always the first item in the `plsq` tuple. Comparing with the true values of the parameters, we can see that the fitting is not bad. We can also compare the fitting with the data graphically by using the following code (see Figure 4.41):

```
>>> fig, ax = plt.subplots()
>>> ax.plot(t, y, "o", c="k", markersize=4, label="Data")
[<matplotlib.lines.Line2D object at 0x000001FB6C38B3C8>]
>>> ax.plot(t, yexact, c="b", label="Exact")
```

```
[<matplotlib.lines.Line2D object at 0x000001FB6C6C2240>]
>>> ax.plot(t,f(t, plsq[0]), c="r", label="Fit")
[<matplotlib.lines.Line2D object at 0x000001FB6C6C2630>]
>>> ax.legend()
<matplotlib.legend.Legend object at 0x000001FB6C6C2550>
>>> plt.show()
```

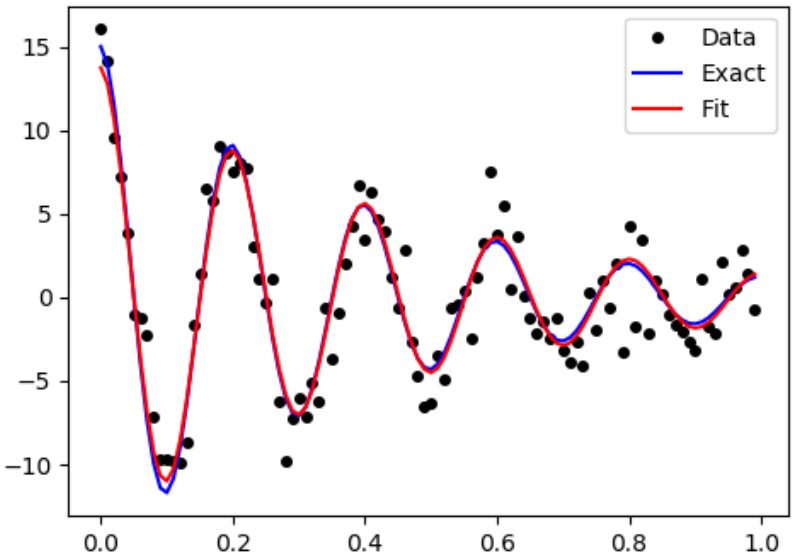

Figure 4.41: Nonlinear least squares fitting of data to a noisy decaying cosine function with `scipy.optimize.leastsq`.

SciPy also has a curve-fitting function `scipy.optimize.curve_fit` that fits data to a function directly (without the need for an additional function to calculate the residuals) and supports weighted nonlinear least squares fitting. This function can be called by the syntax:

```
curve_fit(f, xdata, ydata, p0, sigma, absolute_sigma)
```

where `f` is the function to be fitted to the data `(xdata, ydata)`, `p0` is the initial guess for the fitted parameters, `sigma` is an optional argument giving the uncertainty in `ydata`. If `absolute_sigma` is `True`, then the values in `sigma` are treated as the standard deviations and not just relative weights for the data points. However, these values are treated as relative weights for the default `absolute_sigma=False`. The `curve_fit` function returns the best-fit values of the parameters `popt` and the covariance matrix of the parameters `pcov`. As an demonstration of using `curve_fit` in unweighted and weighted least squares fitting, the following Python program `curvefitdemo.py` fits the Lorentzian line shape function

centered at x_0 with amplitude A and half width at half-maximum (HWHM) γ:

$$f(x; x_0, A, \gamma) = \frac{A\gamma^2}{\gamma^2 + (x - x_0)^2}$$

to some artificial noisy data. The noise is such that a region of the data close to the line center is much noisier than the rest.

```
# curvefitdemo.py
# This program demonstrates weighted and unweighted nonlinear least
# squares fitting with scipy.optimize.curve_fit.
# Last Update on 6 Jan 2022 by F K Chow

import matplotlib.pyplot as plt
import numpy as np
from scipy.optimize import curve_fit

def f(x, x0, A, gamma):
    """ Function to compute the Lorentzian line shape function
        centered at x0 with amplitude A and HWHM gamma """
    return A*gamma**2/(gamma**2 + (x-x0)**2)

def rms(y, yfit):
    """ Function to compute the root-mean-square error """
    return np.sqrt(np.sum((y-yfit)**2))

# Generate data from the Lorentzian line shape function
x0, A, gamma, n = 12, 3, 5, 200
x = np.linspace(1, 20, n)
yexact = f(x, x0, A, gamma)

# Add some noise with a sigma of 0.5 apart from a particularly noisy
# region near x0 where sigma is 3
sigma = np.ones(n)*0.5
sigma[np.abs(x-x0)<1] = 3
noise = np.random.randn(n)*sigma
y = yexact + noise
```

```
# Unweighted fit
p0 = 10, 4, 2
popt, pcov = curve_fit(f, x, y, p0)
yfit = f(x, *popt)
print("Unweighted fit parameters:", popt)
print("Covariance matrix:", pcov)
print("rms error in fit:", rms(yexact, yfit), "\n")

# Weighted fit
popt2, pcov2 = curve_fit(f, x, y, p0, sigma=sigma, absolute_sigma=True)
yfit2 = f(x, *popt2)
print("Weighted fit parameters:", popt2)
print("Covariance matrix:", pcov2)
print("rms error in fit:", rms(yexact, yfit2))

# Plot the graph to compare the uweighted and weighted fits
fig, ax = plt.subplots()
ax.plot(x, yexact, c="b", label="Exact")
ax.plot(x, y, "o", c="k", markersize=4, label="Noisy data")
ax.plot(x, yfit, c="g", ls="--", label="Unweighted fit")
ax.plot(x, yfit2, c="r", ls=":", label="Weighted fit")
ax.set_ylim(-1,4)
ax.legend(loc="lower center")
plt.show()
```

The results of the unweighted and weighted fits are depicted in Figure 4.42. We can observe that the unweighted fit is thrown off by the noisy region. Data in this region are given a lower weight in the weighted fit. So the parameters are closer to their true values and the fit is better. It is also confirmed by the output of the program:

```
Unweighted fit parameters: [11.77288361  3.26527304  4.61939211]
Covariance matrix: [[ 0.03335228 -0.00043134  0.00216912]
 [-0.00043134  0.01709573 -0.02758454]
 [ 0.00216912 -0.02758454  0.08716432]]
rms error in fit: 1.8667663199656352
```

```
Weighted fit parameters: [11.87920532  3.01194669  5.05727934]
Covariance matrix: [[ 0.01987397 −0.0030783   0.00668484]
 [−0.0030783   0.01157999 −0.02221154]
 [ 0.00668484 −0.02221154  0.06684026]]
rms error in fit: 0.5387095003360732
```

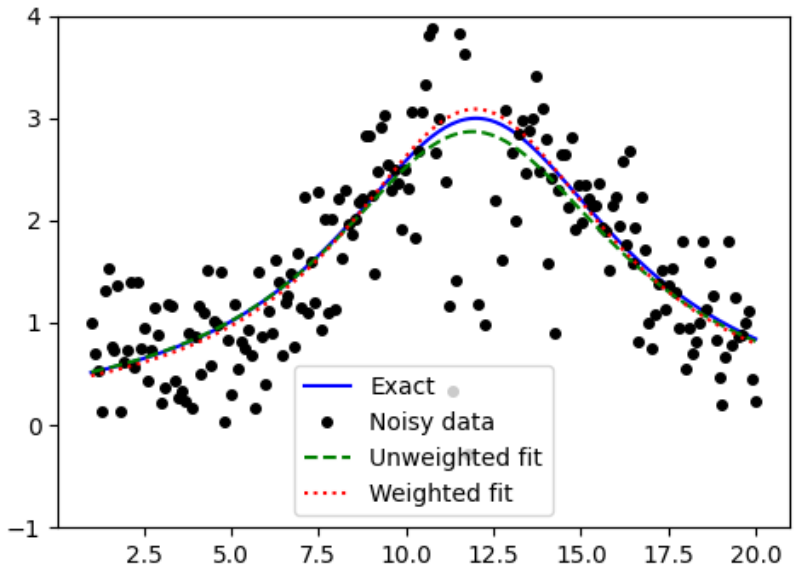

Figure 4.42: Demonstration of unweighted and weighted nonlinear least squares fitting with `scipy.optimize.curve_fit`.

The `scipy.optimize` package provides several functions for obtaining the roots of both univariate and multivariate functions. Here we describe only the algorithms relating to functions of a single variable: `brentq`, `brenth`, `ridder`, `bisect`, and `newton`. Except the method `newton`, each of these methods requires a continuous function $f(x)$ and a pair of numbers defining a **bracketing interval** for the root to find, i. e. values a and b such that the root lies in the interval $[a, b]$ and $f(a)$ and $f(b)$ have opposite signs. Details of the algorithms behind these root-finding methods can be found in standard textbooks on numerical analysis. (Some of these methods will be also discussed in Chapter 7.)

In general, we use the function `scipy.optimize.brentq` to find the root of a well-behaved function. It implements a version of Brent's method with inverse quadratic extrapolation (`scipy.optimize.brenth` is a similar algorithm but with hyperbolic extrapolation). As an example, consider the following function for $-1 \leq x \leq 1$:

$$f(x) = \frac{1}{10} + x \cos\left(\frac{2}{x}\right)$$

We can use the following Python code to plot this function as shown in Figure 4.43:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> x = np.linspace(-1, 1, 1000)
>>> def f(x):
...     return 0.1 + x*np.cos(2/x)
...
>>> fig, ax = plt.subplots()
>>> ax.plot(x, f(x))
[<matplotlib.lines.Line2D object at 0x00000248F0FE6278>]
>>> ax.axhline(0, color="k")
<matplotlib.lines.Line2D object at 0x00000248F2E75F28>
>>> ax.set_xlim(-1, 1)
(-1, 1)
>>> plt.show()
```

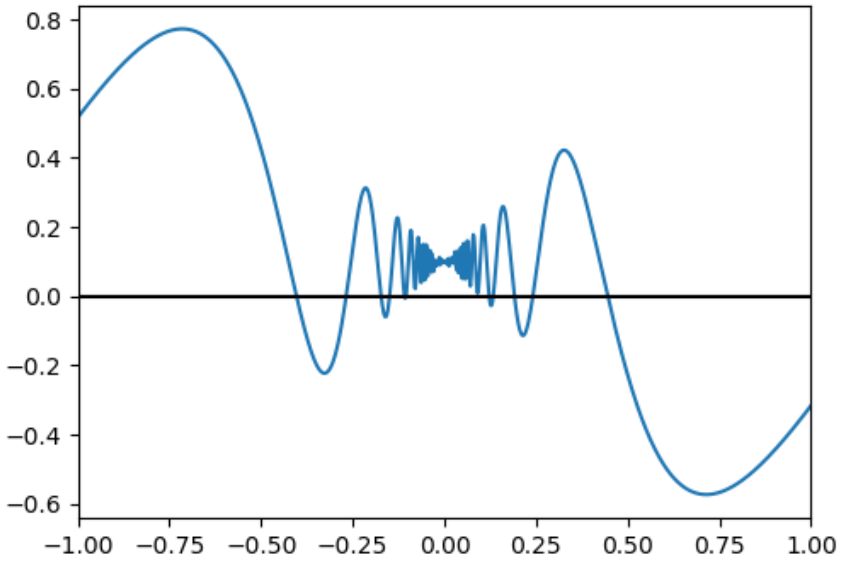

Figure 4.43: The function $f(x) = 1/10 + x\cos(2/x)$ and its root.

Obviously, there is a root between -0.5 and -0.3. We can use the function `brentq` to find this root as follows:

```
>>> from scipy.optimize import brentq
>>> brentq(f, -0.5, -0.3)
-0.4029684518832628
```

Moreover, the root-finding algorithm known as Ridder's method is implemented in the function `scipy.optimize.ridder` whereas the slower but very reliable (for continuous functions) method of bisection is implemented in the function `scipy.optimize.bisect`.

Root-finding by the Newton-Raphson method can be very fast for many continuous functions provided the first derivative $f'(x)$ can be calculated. For functions for which an analytical expression for $f'(x)$ can be coded, this is passed to the function `scipy.optimize.newton` as the argument `fprime` along with an initial guess `x0` in order to find the root using the Newton-Raphson method. However, if $f'(x)$ cannot be provided, then this function would instead use the secant method. Besides, if we can provide the second derivative $f''(x)$ as the argument `fprime2` in addition to the first derivative to the function `newton`, then the Halley's method (which converges even faster than the basic Newton-Raphson algorithm) would be used instead. Notice that the stopping condition within the iterative algorithm used by `newton` is the step size. So there is no guarantee that it has converged on the desired root. Thus the result should be verified by evaluating the function at the returned value to check that it is approximately zero.