8 SciPy

SciPy is a library of Python modules for scientific computing that provides more specific functionality than the generic data structures and mathematical algorithms of NumPy. For example, it contains modules for the evaluation of special functions frequently encountered in science and engineering, optimization, integration, interpolation and image manipulation. As with the NumPy library, many of SciPy's underlying algorithms are executed as compiled C code, so they are fast. Also like NumPy and Python itself, SciPy is free software.

There is little new syntax to learn in using the SciPy routines, so this chapter will focus on examples of the library's use in short programs of relevance to science and engineering.

8.1 Physical constants and special functions

The useful `scipy.constants` package provides the internationally agreed standard values and uncertainties for physical constants. The `scipy.special` package also supplies a large number of algorithms for calculating functions that appear in science, mathematical analysis and engineering, including:

- Airy functions
- Elliptic functions and integrals
- Bessel functions, their zeros, derivatives and integrals
- Spherical Bessel functions
- Struve functions
- A variety of statistical functions and distributions
- Gamma and beta functions
- The error function
- Fresnel integrals
- Legendre functions and associated Legendre functions
- A variety of orthogonal polynomials
- Hypergeometric functions
- Parabolic cylinder functions
- Matheiu functions
- Spheroidal functions
- Kelvin functions

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use , available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

They are described in detail in the documentation;[1] we focus in this section on a few representative examples.

Most of these special functions are implemented in SciPy as universal functions: that is, they support broadcasting and vectorization (automatic array-looping), and so work as expected with NumPy arrays.

8.1.1 Physical constants

SciPy contains the 2010 CODATA internationally recommended values[2] 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,

```
In [x]: import scipy.constants as pc
In [x]: pc.physical_constants['Avogadro constant']
Out[x]: (6.02214129e+23, 'mol^-1', 2.7e+16)
```

The convenience methods `value`, `unit` and `precision` retrieve the corresponding properties on their own:

```
In [x]: pc.value('elementary charge')
Out[x]: 1.602176565e-19
In [x]: pc.unit('elementary charge')
Out[x]: 'C'
In [x]: pc.precision('elementary charge')
2.1845282701410628e-08
```

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

```
In [x]: muB = pc.value('Bohr magneton')
```

A full list of the constants and their names is given in the SciPy documentation,[3] but Table 8.1 lists the more important ones. Some particularly important constants have a direct variable assignment within `scipy.constants` (in SI units) and so can be imported directly:

```
In [x]: from scipy.constants import c, R, k
In [x]: c, R, k     # speed of light, gas constant, Boltzmann constant
Out[x]: (299792458.0, 8.3144621, 1.3806488e-23)
```

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

There are one or two useful conversion factors and methods, and SI prefixes defined within the `scipy.constants` package, for example,

[1] http://docs.scipy.org/doc/scipy/reference/special.html.

[2] P. J. Mohr, B. N. Taylor, D. B. Newell, (2012). *Rev. Mod. Phys.*, **84**, 1527.

[3] http://docs.scipy.org/doc/scipy/reference/constants.html.

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

Table 8.1 Physical constants in `scipy.constants`

Constant string	Variable	Value	Units
`'atomic mass constant'`		`1.660538921e-27`	kg
`'Avogadro constant'`	`N_A`	`6.02214129e+23`	mol^{-1}
`'Bohr magneton'`		`9.27400968e-24`	$\mathrm{J\,T}^{-1}$
`'Bohr radius'`		`5.2917721092e-11`	m
`'Boltzmann constant'`	`k`	`1.3806488e-23`	$\mathrm{J\,K}^{-1}$
`'electron mass'`	`m_e`	`9.10938291e-31`	kg
`'elementary charge'`	`e`	`1.602176565e-19`	C
`'Faraday constant'`		`96485.3365`	$\mathrm{C\,mol}^{-1}$
`'fine-structure constant'`	`alpha`	`0.0072973525698`	
`'molar gas constant'`	`R`	`8.3144621`	$\mathrm{J\,K}^{-1}\,\mathrm{mol}^{-1}$
`'neutron mass'`	`m_n`	`1.674927351e-27`	kg
`'Newtonian constant of gravitation'`	`G`	`6.67384e-11`	$\mathrm{m}^3\,\mathrm{kg}^{-1}\,\mathrm{s}^{-2}$
`'Planck constant'`	`h`	6.62606957e-34	J s
`'Planck constant over 2 pi'`	`hbar`	1.054571726e-34	J s
`'proton mass'`	`m_p`	`1.672621777e-27`	kg
`'Rydberg constant'`	`Rydberg`	`10973731.5685`	m^{-1}
`'speed of light in vacuum'`	`c`	`299792458.0`	$\mathrm{m\,s}^{-1}$

```
In [x]: import scipy.constants as pc
In [x]: pc.atm
Out[x]: 101325.0              # 1 atm in Pa
In [x]: pc.bar
Out[x]: 100000.0              # 1 bar in Pa
In [x]: pc.torr
Out[x]: 133.32236842105263    # 1 torr in Pa
In [x]: pc.zero_Celsius
Out[x]: 273.15                # 0 degC in K
In [x]: pc.micro              # also nano, pico, mega, giga, etc.
Out[x]: 1e-06
```

Example E8.1 Let's use the `scipy.constants.physical_constants` dictionary to determine which are the least accurately known constants. To do this we need the *relative* uncertainties in the constants' values. The code mentioned here uses a structured array to calculate these and outputs the least well-determined constants.

Listing 8.1 Least well-defined physical constants

```
import numpy as np

from scipy.constants import physical_constants

def make_record(k, v):
    """
    Return the record for this constant from the key and value of its entry
    in the physical_constants dictionary.
```

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use , available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

```
    """
    name = k
    val, units, abs_unc = v
    # Calculate the relative uncertainty in ppm
    rel_unc = abs_unc / abs(val) * 1.e6
    return name, val, units, abs_unc, rel_unc

dtype = [('name', 'S50'), ('val', 'f8'), ('units', 'S20'),
         ('abs_unc', 'f8'), ('rel_unc', 'f8')]
constants = np.array([make_record(k, v) for k,v in physical_constants.items()],
                     dtype=dtype )
constants.sort(order='rel_unc')

# List the 10 constants with the largest relative uncertainties
for rec in constants[-10:]:
    print('{:.0f} ppm: {:s} = {:g} {:s}'.format(rec['rel_unc'],
              rec['name'].decode(), rec['val'], rec['units'].decode()))
```

The output is shown here. Note that G is not known to better than about 120 ppm (parts per million.)

```
91 ppm: proton-tau mass ratio = 0.528063
91 ppm: tau mass energy equivalent = 2.84678e-10 J
92 ppm: tau mass = 3.16747e-27 kg
119 ppm: Newtonian constant of gravitation over h-bar c = 6.70837e-39 (GeV/c^2)^-2
120 ppm: Newtonian constant of gravitation = 6.67384e-11 m^3 kg^-1 s^-2
545 ppm: proton mag. shielding correction = 2.5694e-05
545 ppm: proton magn. shielding correction = 2.5694e-05
980 ppm: deuteron rms charge radius = 2.1424e-15 m
5812 ppm: proton rms charge radius = 8.775e-16 m
9447 ppm: weak mixing angle = 0.2223
```

8.1.2 Airy and Bessel functions

The Airy functions Ai(x) and Bi(x) are the linearly independent solutions to the Airy equation, $y'' - xy = 0$, which occurs in quantum mechanics, optics, electrodynamics and other areas of physics. The functions (`Ai`, `Bi`) and their derivatives (`Aip`, `Bip`) are returned by the function `scipy.special.airy`. The only required argument is `x`, which could be complex and can be a NumPy array:

```
In [x]: Ai, Aip, Bi, Bip = airy(0)
In [x]: Ai, Aip, Bi, Bip
(0.35502805388781722, -0.25881940379280682, 0.61492662744600068, 0.44828835735382638)
```

The first `nt` zeros of the Airy functions and their derivatives are returned by the function `scipy.special.ai_zeros(nt)`:

```
In [x]: a, ap, ai, aip = ai_zeros(2)     # arrays for the first 2 zeros of Ai
In [x]: a[1], ap[1], ai[1], aip[1]       # look at the 2nd zero:
Out[x]: (-4.0879494441309721, -3.248197582179837, -0.41901547803256406,
         -0.80311136965486463)
In [x]: airy(a[1])[0]                    # Ai(a) should = 0
Out[x]: 1.2774882441379295e-15           # close enough
In [x]: airy(ap[1])[1]                   # Aip(ap) should = 0
```

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

```
Out[x]: -3.2322209157744908e-16          # close enough
In [x]: airy(ap[1])[0]                   # Ai(ap) is returned as ai above
Out[x]: -0.41901547803256395
In [x]: airy(a[1])[1]                    # Aip(a) is returned as aip above
Out[x]: -0.80311136965486396
```

♢ **Example E8.2** Consider a particle of mass m moving in a constant gravitational field such that its potential energy at a height z above a surface is mgz. If the particle bounces elastically on the surface, the classical probability density corresponding to its position is

$$P_{\rm cl}(z) = \frac{1}{\sqrt{z_{\rm max}(z_{\rm max} - z)}},$$

where $z_{\rm max}$ is the maximum height it reaches.

The quantum mechanical behaviour of this system may described by the solution to the time-independent Schrödinger equation,

$$-\frac{\hbar^2}{2m}\frac{{\rm d}^2\psi}{{\rm d}z^2} + mgz\psi = E\psi$$

which is simplified by the coordinate rescaling $q = z/\alpha$ where $\alpha = (\hbar^2/2m^2g)^{1/3}$:

$$\frac{{\rm d}^2\psi}{{\rm d}q^2} - (q - q_E)\psi = 0, \quad \text{where } q_E = \frac{E}{mg\alpha}.$$

The solutions to this differential equation are the Airy functions. The boundary condition $\psi(z) \to 0$ as $z \to \infty$ specifically gives:

$$\psi(q) = N_E {\rm Ai}(q - q_E),$$

where N_E is a normalization constant.

The second boundary condition, $\psi(q = 0) = 0$, leads to quantization in terms of a quantum number $n = 1, 2, 3, \cdots$ with scaled energy values q_E found from the zeros of the Airy function: ${\rm Ai}(-q_E) = 0$.

The following program plots the classical and quantum probability distributions, $P_{\rm cl}(z)$ and $|\psi(z)|^2$, for $n = 1$ and $n = 16$ (Figure 8.1).

Listing 8.2 Probability densities for a particle in a uniform gravitational field

```
# eg8-qm-gravfield.py
import numpy as np
from scipy.special import airy, ai_zeros
import pylab

nmax = 16

# Find the first nmax zeros of Ai(x)
a, _, _, _ = ai_zeros(nmax)                    ❶
# The actual boundary condition is Ai(-qE) = 0 at q=0, so:
qE = -a

def prob_qm(n):
```

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use , available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

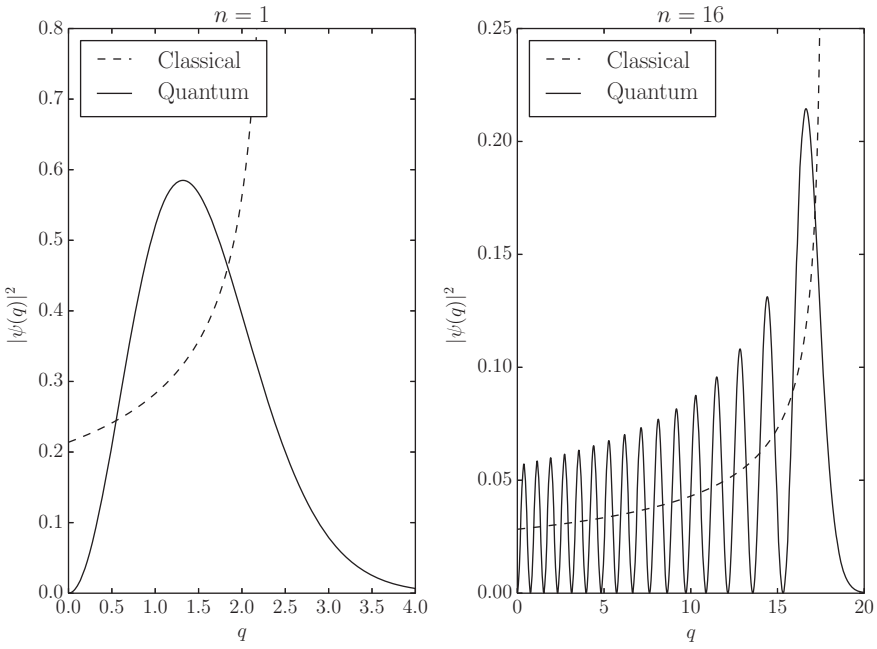


Figure 8.1 A comparison of classical and quantum probability distributions for a particle moving in a constant gravitationa field at two different energies.

```
    """
    Return the quantum mechanical probability density for a particle moving
    in a uniform gravitational field.

    """
    # The quantum mechanical wavefunction is proportional to Ai(q-qE) where
    # the qE corresponding to quantum number n is indexed at n-1
❷   psi, _, _, _ = airy(q-qE[n-1])
    # Return the probability density, after rough-and-ready normalization
    P = psi**2
❸   return P / (sum(P) * dq)

def prob_cl(n):
    """
    Return the classical probability density for a particle bouncing
    elastically in a uniform gravitational field.

    """
    # The classical probability density is already normalized
    return 0.5/np.sqrt(qE[n-1]*(qE[n-1]-q))

# The ground state, n=1
q, dq = np.linspace(0, 4, 1000, retstep=True)
pylab.plot(q, prob_cl(1), label='Classical')
pylab.plot(q, prob_qm(1), label='Quantum')
pylab.ylim(0,0.8)
pylab.legend()
pylab.show()
```

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use , available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

```
# An excited state, n=16
q, dq = np.linspace(0, 20, 1000, retstep=True)
pylab.plot(q, prob_cl(16), label='Classical')
pylab.plot(q, prob_qm(16), label='Quantum')
pylab.ylim(0,0.25)
pylab.legend(loc='upper left')
pylab.show()
```

❶ We use `scipy.special.ai_zeros` to retrieve the $n = 1$ and $n = 16$ eigenvalues.

❷ `scipy.special.airy` finds the corresponding wavefunctions and hence probability densities.

❸ For the sake of illustration, these are normalized approximately by a very simple numerical integration.

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

- `jn(v, x)` and `jv(v, x)` return the *Bessel function of the first kind* at `x` for order `v`, $J_v(x)$. `v` can be real or 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.
- `in(n, x)` and `iv(v, x)` return the *modified Bessel function of the first kind* at `x` for integer order `n` ($I_n(x)$) and real order `v` ($I_v(x)$), respectively.
- `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 derivative is returned; to return the nth derivative, set the optional argument, `n`.
- Several functions can be used to obtain the *zeros* of the Bessel functions. Probably the most useful are `jn_zeros(n, nt)`, `jn_zeros(n, nt)`, `jnp_zeros(n, nt)`, `yn_zeros(n, nt)` and `ynp_zeros(n, nt)`, which return the first `nt` zeros of $J_n(x)$, $J_n'(x)$, $Y_n(x)$ and $Y_n'(x)$.

Example E8.3 The vibrations of a thin circular membrane stretched across a rigid circular frame (such as a drum head) 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 $n = 0, 1, \cdots$ and $m = 1, 2, 3, \cdots$ where k is the mth zero of J_n.

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use , available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

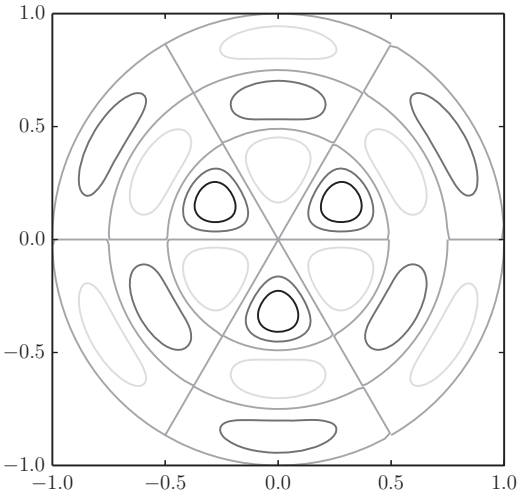


Figure 8.2 The $n = 3, m = 2$ normal mode of a vibrating circular drum.

The following program produces a plot of the displacement of the membrane in the $n = 3, m = 2$ normal mode at time $t = 0$ (Figure 8.2).

Listing 8.3 Normal modes of a vibrating circular drum

```
# eg8-drum-normal-modes.py
import numpy as np
from scipy.special import jn, jn_zeros
import pylab

# Allow calculations up to m = mmax
mmax = 5

def displacement(n, m, r, theta):
    """
    Calculate the displacement of the drum membrane at (r, theta; t=0)
    in the normal mode described by integers n >= 0, 0 < m <= mmax.

    """

    # Pick off the mth zero of Bessel function Jn
    k = jn_zeros(n, mmax+1)[m]
    return np.sin(n*theta) * jn(n, r*k)

# Positions on the drum surface are specified in polar coordinates
r = np.linspace(0, 1, 100)
theta = np.linspace(0, 2 * np.pi, 100)

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

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use , available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

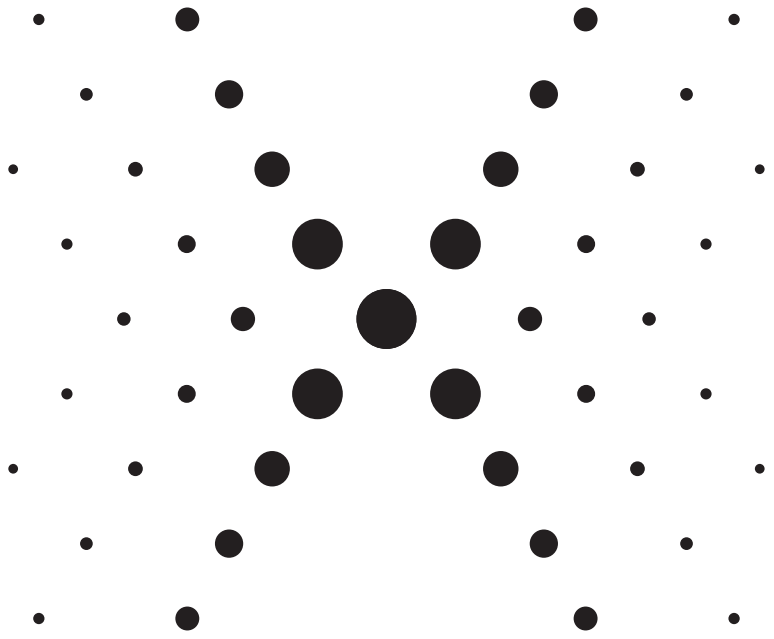


Figure 8.3 The diffraction pattern of a uniform, continuous helix.

```
n, m = 3, 2
z = np.array([displacement(n, m, rr, theta) for rr in r])

pylab.contour(x, y, z)
pylab.show()
```

Example E8.4 In an important paper in 1953.[4] Rosalind Franklin published the X-ray diffraction pattern of DNA from calf thymus, which displays a characteristic X shape of diffraction spots indicative of a helical structure.

The diffraction pattern of a uniform, continuous helix consists of a series of "layer lines" of spacing $1/p$ in reciprocal space where p is the helix pitch (the height of one complete turn of the helix, measured parallel to its axis). The intensity distribution along the nth layer line is proportional to the square of the nth Bessel function, $J_n(2\pi rR)$, where r is the radius of the helix and R is the radial coordinate in reciprocal space.

Consider the diffraction pattern of a helix with $p = 34$ Å and $r = 10$ Å. The code listing here produces an SVG image of the diffraction pattern of a helix (Figure 8.3).

[4] R. E. Franklin, R. G. Gosling, (1953). *Nature* **171**, 740.

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use , available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

Listing 8.4 Generating an image of the diffraction pattern of a uniform, continuous helix

```
# eg8-dna-diffraction.py

import numpy as np
from scipy.special import jn
import pylab

# Vertical range of the diffraction pattern: plot nlayer line layers above and
# below the center horizontal
nlayers = 5
ymin, ymax = -nlayers, nlayers

# Horizontal range of the diffraction pattern, x = 2pi.r.R
xmin, xmax = -10, 10
npts = 4000
x = np.linspace(xmin, xmax, npts)

# Diffraction pattern along each line layer: |Jn(x)|^2
# for n = 0, 1, ..., nlayers-1
❶ layers = np.array([jn(i, x)**2 for i in range(nlayers)])

# Obtain the indexes of the maxima in each layer
❷ maxi = [(np.diff(np.sign(np.diff(layers[i,:])))) < 0).nonzero()[0] + 1
                                                    for i in range(nlayers)]

# Create the SVG image, using circles of different radii for diffraction spots
svg_name='eg8-dna-diffraction.svg'
canvas_width = canvas_height = 500
fo = open(svg_name, 'w')
print("""<?xml version="1.0" encoding="utf-8"?>
         <svg xmlns="www.w3.org/2000/svg"
              xmlns:xlink="www.w3.org/1999/xlink"
              width="{}" height="{}" style="background: {}">""".format(
                canvas_width, canvas_height, '#ffffff'), file=fo)

def svg_circle(r, cx, cy):
    """ Return the SVG mark up for a circle of radius r centered at (cx,cy). """
    return r'<circle r="{}" cx="{}" cy="{}"/>'.format(r, cx, cy)

# For each spot in each layer, draw a circle on the canvas. The circle radius
# is the scaled value of the diffraction intensity maximum, with a ceiling
# value of spot_max_radius because the center spots are very intense
spot_scaling, spot_max_radius = 50, 20
for i in range(nlayers):
    for j in maxi[i]:
❸       sx = (x[j] - xmin)/(xmax-xmin) * canvas_width
        sy = (i - ymin)/(ymax - ymin) * canvas_height
        spot_radius = min(layers[i,j]*spot_scaling, spot_max_radius)
        print(svg_circle(spot_radius, sx, sy), file=fo)
        if i:
            # The pattern is symmetric about the center horizontal:
            # duplicate the layers with i > 0
            sy = canvas_height - sy
            print(svg_circle(spot_radius, sx, sy), file=fo)

print(r'</svg>', file=fo)
```

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

❶ The two-dimensional array, `layers`, holds the diffraction intensity in each line layer, calculated as the square of a Bessel function.

❷ For plotting the pattern, we need to find the indexes of the maxima in the `layers` array: this line of code finds these maxima by determining where the *differences* between neighboring items go from positive to negative.

❸ Map the `(x,y)` coordinates in the reciprocal space of the diffraction pattern onto the canvas coordinates, `(sx,sy)`.

8.1.3 The gamma and beta functions; elliptic integrals

The gamma function is defined by the improper integral

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t}\, \mathrm{d}t,$$

for real $x > 0$, and extended to negative x and complex numbers by analytic continuation. It occurs frequently in integration problems, combinatorics and in expressions for other special functions.

The gamma function and its natural logarithm are returned by the functions `gamma(x)` and `gammaln(x)`. There are also methods for the evaluation of the incomplete gamma functions (obtained by replacing the lower or upper limits in the integral above with the parameter `a`) and their inverses; these will not be described in detail here.

Example E8.5 The gamma function is related to the factorial by $\Gamma(x) = (x-1)!$ and both are plotted in the code mentioned later (see Figure 8.4). Note that $\Gamma(x)$ is not defined for negative integer x, which leads to discontinuities in the plot.

Listing 8.5 The Gamma function on the real line

```
# eg3-gamma.py
import numpy as np
from scipy.special import gamma
import pylab

# The Gamma function
ax = pylab.linspace(-5, 5, 1000)
pylab.plot(ax, gamma(ax), ls='-', c='k', label='$\Gamma(x)$')

# (x-1)! for x = 1, 2, ..., 6
ax2 = pylab.linspace(1,6,6)
xm1fac = np.array([1, 1, 2, 6, 24, 120])
pylab.plot(ax2, xm1fac, marker='*', markersize=12, markeredgecolor='r',
           ls='',c='r', label='$(x-1)!$')

pylab.ylim(-50,50)
pylab.xlim(-5, 5)
pylab.xlabel('$x$')
pylab.legend()
pylab.show()
```

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

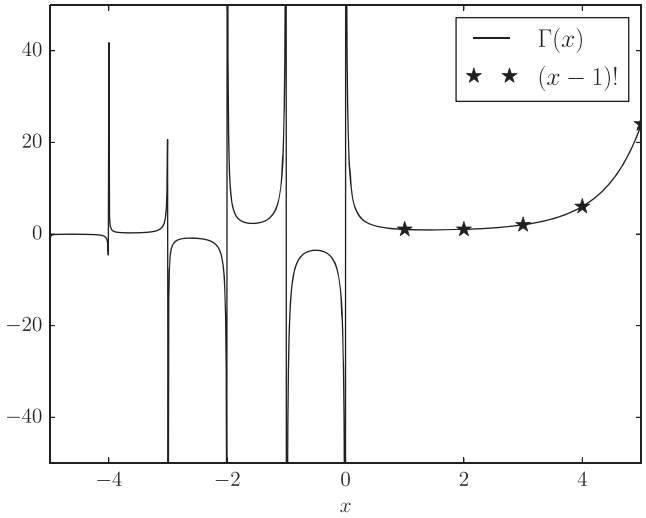


Figure 8.4 The gamma function on the real line, $\Gamma(x)$, and $(x-1)!$ for integer $x > 0$.

The beta function is defined by the definite integral

$$B(a,b) = \int_0^1 t^{a-1}(1-t)^{b-1}\,\mathrm{d}t, \quad a > 0, b > 0.$$

It is closely related to the gamma function: $B(a,b) = \Gamma(a)\Gamma(b)/\Gamma(a+b)$. The `scipy.special` functions `beta(a, b)` and `betaln(a,b)` return the beta function and its natural logarithm respectively. As with the gamma function, there is an *incomplete beta function*, $B(a,b;x)$, obtained by replacing the upper limit with x; the methods `betainc(a, b, x)` and `betaincinv(a, b, y)` return this function and its inverse.

Example E8.6 The exact classical mechanical description of a pendulum is quite complex, and the equations of motion usually only solved in introductory texts for small displacements about equilibrium. In this case, the period, $T \approx 2\pi\sqrt{L/g}$, and the motion is harmonic.

The general solution requires elliptic integrals, but the special case of a pendulum making 180° swings (i.e., ±90° about its equilibrium position) leads to the following expression for the period:

$$T = 2\sqrt{\frac{2l}{g}}\int_0^{\pi/2}\frac{\mathrm{d}\theta}{\sqrt{\cos\theta}}.$$

The substitution $x = \sin^2\theta$ transforms this integral into a beta function:

$$\int_0^{\pi/2}\frac{\mathrm{d}\theta}{\sqrt{\cos\theta}} = \tfrac{1}{2}\int_0^1 x^{-1/2}(1-x)^{-3/4}\,\mathrm{d}x = \tfrac{1}{2}B\left(\tfrac{1}{2},\tfrac{1}{4}\right).$$

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use , available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

Therefore,

$$T = \sqrt{2}B\left(\tfrac{1}{2}, \tfrac{1}{4}\right)\sqrt{\frac{l}{g}}.$$

To find the period of the pendulum in units of $\sqrt{l/g}$:

```
In [x]: import numpy as np
In [x]: from scipy.special import beta
In [x]: np.sqrt(2) * beta(0.5, 0.25)
7.4162987092054875
```

(Compare with the harmonic approximation, $2\pi = 6.283185$.)

The group of elliptic integrals and related functions form an important class of mathematical objects and have been widely studied. They find application in geometry, cryptography, analysis and many areas of physics. The complete elliptic integrals of the first and second kind, $K(m)$ and $E(m)$, are defined for $0 \leq m \leq 1$ by

$$K(m) = \int_0^{\pi/2} \frac{\mathrm{d}\theta}{\sqrt{1 - m\sin^2\theta}},$$
$$E(m) = \int_0^{\pi/2} \sqrt{1 - m\sin^2\theta}\,\mathrm{d}\theta.$$

Their values for the parameter m are returned by the functions `ellipk(m)` and `ellipe(m)`. The incomplete elliptic integrals (defined by replacing the upper limit of $\pi/2$ with the variable ϕ) are returned by `ellipkinc(phi, m)` and `ellipeinc(phi, m)` respectively:[5]

$$K(\phi, m) = \int_0^{\phi} \frac{\mathrm{d}\theta}{\sqrt{1 - m\sin^2\theta}},$$
$$E(\phi, m) = \int_0^{\phi} \sqrt{1 - m\sin^2\theta}\,\mathrm{d}\theta.$$

Example E8.7 The problem of finding an arc length of an ellipse is the origin of the name of the elliptic integrals. The equation of an ellipse with semi-major axis, a, and semi-minor axis, b, may be written in parametric form as

$$x = a\sin\phi$$
$$y = b\cos\phi$$

[5] It is necessary to be very careful with the notation of elliptic integrals; many sources use $F(\phi, m)$ instead of $K(\phi, m)$ for the first kind, define them with interchanged arguments (i.e., $F(m, \phi)$) or use the parameter k^2 instead m

$$F(\phi, k) = F(\phi|k^2) = \int_0^{\phi} \frac{\mathrm{d}\theta}{\sqrt{1 - k^2\sin^2\theta}}$$
$$E(\phi, k) = E(\phi|k^2) = \int_0^{\phi} \sqrt{1 - k^2\sin^2\theta}\,\mathrm{d}\theta.$$

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

The element of length along the ellipse's perimeter,

$$\begin{aligned} \mathrm{d}s &= \sqrt{\mathrm{d}x^2 + \mathrm{d}y^2} = \sqrt{a^2 \cos^2\phi + b^2 \sin^2\phi}\ \mathrm{d}\phi \\ &= a\sqrt{1 - e^2 \sin^2\phi}\ \mathrm{d}\phi, \end{aligned}$$

where $e = \sqrt{1 - b^2/a^2}$ is the *eccentricity*. The arc length may therefore be written in terms of incomplete elliptic integrals of the second kind:

$$\int \mathrm{d}s = a \int_{\phi_1}^{\phi_2} \sqrt{1 - e^2 \sin^2\phi}\ \mathrm{d}\phi = a[E(e;\phi_2) - E(e;\phi_1)].$$

Earth's orbit is an ellipse with semi-major axis 149,598,261 km and eccentricity 0.01671123. We will find the distance traveled by the Earth in one orbit, and compare it with that obtained assuming a circular orbit of radius 1 AU $\equiv$ 149597870.7 km.

The perimeter of an ellipse may be written using the earlier expression with $\phi_1 = 0, \phi_2 = 2\pi$:

$$P = a[E(e, 2\pi) - E(e, 0)] = 4aE(e),$$

since the entire perimeter is four times the quarter-perimeters, which may be written in terms of the *complete* elliptic integral of the second kind. We have

```
In [x]: import numpy as np
In [x]: from scipy.special import ellipe
In [x]: a, e  = 149598261, 0.01671123   # semi-major axis (km), eccentricity
In [x]: pe = 4 * a * ellipe(e)
In [x]: print(pe)
936014259.33                    # "exact" answer
In [x]: AU = 149597870.7        # mean orbit radius, km
In [x]: pc = 2 * np.pi * AU
In [x]: print(pc)
939951143.1675915               # assuming circular orbit
In [x]: (pc - pe) / pe * 100
0.42060084000247772
```

That is, the percentage error in the perimeter in treating the orbit as circular is about 0.42%.

8.1.4 The error function and related integrals

The error function, defined by:

$$\mathrm{erf}(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2}\ \mathrm{d}t$$

for real or complex z does not have a simple closed-form expression and so must be calculated numerically. `scipy.special` has several functions relating to the error function:

- `erf(z)`: the error function;
- `erfc(z)`: the complementary error function, $\mathrm{erfc}(z) = 1 - \mathrm{erf}(z)$. It is more accurate to use this function for large z than directly subtracting $\mathrm{erf}(z)$ from 1;

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

- `erfcx(z)`: the *scaled complementary error function*, $e^{z^2}\operatorname{erfc}(z)$;
- `erfinv(y)`: the inverse error function;
- `erfcinv(y)`: the inverse complementary error function;
- `wofz(z)`: the Faddeeva function, a scaled complementary error function with complex argument:

$$w(z) = e^{-z^2}\operatorname{erfc}(-iz) = \operatorname{erfcx}(-iz),$$

  which appears in problems related to plasma physics and radiative transfer;
- `dawsn(z)`: the related integral known as Dawson's integral:

$$D(z) = e^{-z^2}\int_0^z e^{t^2}\,\mathrm{d}t.$$

Example E8.8 The wavefunction corresponding to the ground state of the one-dimensional quantum harmonic oscillator may be written as follows in terms of a parameter $\alpha = \sqrt{mk}/\hbar$, where m is the mass and k the oscillator force constant.

$$\psi_0(x) = \left(\frac{\alpha}{\pi}\right)^{1/4}\exp\left(-\alpha x^2/2\right)$$

The probability density of the oscillator's position is given by $P_0(x) = |\psi_0(x)|^2$ and is nonzero outside the classical turning points, $\pm\alpha^{-1/2}$, a phenomenon known as tunneling. We will calculate the probability of tunneling for an oscillator in the state ψ_0.

The wavefunction is symmetric about $x = 0$, so the probability of tunneling is

$$\begin{aligned} P(x < -\alpha) + P(x > \alpha) = 2P(x > \alpha) &= 2\sqrt{\frac{\alpha}{\pi}}\int_{\alpha^{-1/2}}^{\infty}\exp\left(-\alpha x^2\right)\,\mathrm{d}x \\ &= \frac{2}{\sqrt{\pi}}\int_1^{\infty} e^{-y^2}\,\mathrm{d}y = \operatorname{erfc}(1). \end{aligned}$$

The complementary error function can be calculated directly:

```
In [x]: from scipy.special import erfc
In [x]: erfc(1)
0.15729920705028516
```

or about 16%.

Example E8.9 The *Voigt line profile* occurs in the modeling and analysis of radiative transfer in the atmosphere. It is the convolution of a Gaussian profile, $G(x;\sigma)$, and a Lorentzian profile, $L(x;\gamma)$:

$$V(x;\sigma,\gamma) = \int_{-\infty}^{\infty} G(x';\sigma)L(x-x';\gamma)\,\mathrm{d}x' \quad \text{where}$$

$$G(x;\sigma) = \frac{1}{\sigma\sqrt{2\pi}}\exp\left(\frac{-x^2}{2\sigma}\right) \quad \text{and} \quad L(x;\gamma) = \frac{\gamma/\pi}{x^2+\gamma^2}.$$

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use , available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

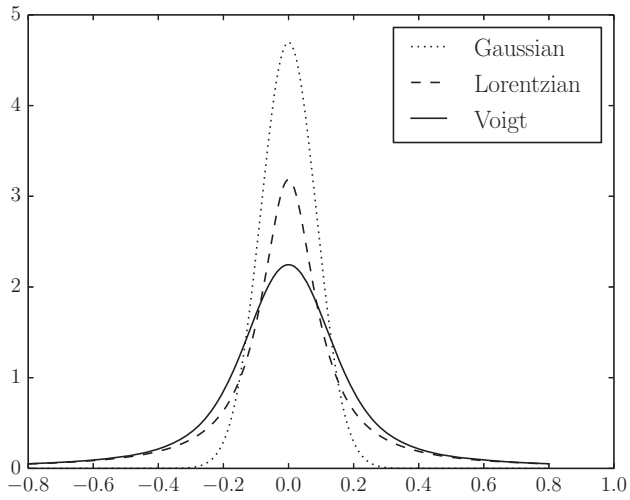


Figure 8.5 A comparison of the Lorentzian, Gaussian and Voigt line shapes with $\gamma = \alpha = 0.1$.

Here γ is the half-width at half-maximum (HWHM) of the Lorentzian profile and σ is the standard deviation of the Gaussian profile, related to its HWHM, α, by $\alpha = \sigma\sqrt{2\ln 2}$. In terms of frequency, ν, $x = \nu - \nu_0$ where ν_0 is the line center.

There is no closed form for the Voigt profile, but it is related to the real part of the Faddeeva function, $w(z)$ by

$$V(x;\sigma,\gamma) = \frac{\mathrm{Re}\,[w(z)]}{\sigma\sqrt{2\pi}}, \text{ where } z = \frac{x+i\gamma}{\sigma\sqrt{2}}.$$

The program mentioned here plots the Voigt profile for $\gamma = 0.1, \alpha = 0.1$ and compares it with the corresponding Gaussian and Lorentzian profiles (Figure 8.5). The equations mentioned earlier are implemented in the three functions, `G`, `L` and `V`, defined in the code here.

Listing 8.6 A comparison of the Lorentzian, Gaussian and Voigt line shapes

```
# eg8-voigt.py
import numpy as np
from scipy.special import wofz
import pylab

def G(x, alpha):
    """ Return Gaussian line shape at x with HWHM alpha """
    return np.sqrt(np.log(2) / np.pi) / alpha\
                           * np.exp(-(x / alpha)**2 * np.log(2))

def L(x, gamma):
    """ Return Lorentzian line shape at x with HWHM gamma """
    return gamma / np.pi / (x**2 + gamma**2)

def V(x, alpha, gamma):
    """
    Return the Voigt line shape at x with Lorentzian component HWHM gamma
    and Gaussian component HWHM alpha.
```

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use , available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

```
    """
    sigma = alpha / np.sqrt(2 * np.log(2))

    return np.real(wofz((x + 1j*gamma)/sigma/np.sqrt(2))) / sigma\
                                                          / np.sqrt(2*np.pi)

alpha, gamma = 0.1, 0.1
x = np.linspace(-0.8,0.8,1000)
pylab.plot(x, G(x, alpha), ls=':', c='k', label='Gaussian')
pylab.plot(x, L(x, gamma), ls='--', c='k', label='Lorentzian')
pylab.plot(x, V(x, alpha, gamma), c='k', label='Voigt')
pylab.legend()
pylab.show()
```

8.1.5 Fresnel integrals

The Fresnel integrals are encountered in optics and are defined by the equations

$$S(z) = \int_0^z \sin\left(\frac{\pi t^2}{2}\right) \mathrm{d}t, C(z) = \int_0^z \cos\left(\frac{\pi t^2}{2}\right) \mathrm{d}t.$$

Both are returned in a tuple for real or complex argument `z` by the `special.scipy` function `fresnel(z)`. The related function, `fresnel_zeros(nt)`, returns the first `nt` complex zeros of $S(z)$ and $C(z)$.

Example E8.10 As well as playing an important role in the description of diffraction effects in optics, the Fresnel integrals find an application in the design of motorway junctions (freeway intersections). The curve described by the parametric equations $(x, y) = (S(t), C(t))$ is called a *clothoid* (or Euler spiral) and has the property that its curvature is proportional to the distance along the path of the curve. Hence, a vehicle traveling at constant speed will experience a constant rate of angular acceleration as it travels around the curve – this means that the driver can turn the steering wheel at a constant rate, which makes the junction safer.

The following code plots the Euler spiral for $-10 \leq t \leq 10$ (Figure 8.6).

```
In [x]: import numpy as np
In [x]: from scipy.special import fresnel
In [x]: import pylab
In [x]: t = np.linspace(-10, 10, 1000)
In [x]: pylab.plot(*fresnel(t), c='k')
In [x]: pylab.show()
```

8.1.6 Binomial coefficients and exponential integrals

The binomial coefficient $\binom{n}{k} = {}^nC_k$ is returned by the `scipy.special` function `binom(n,k)`.

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use , available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

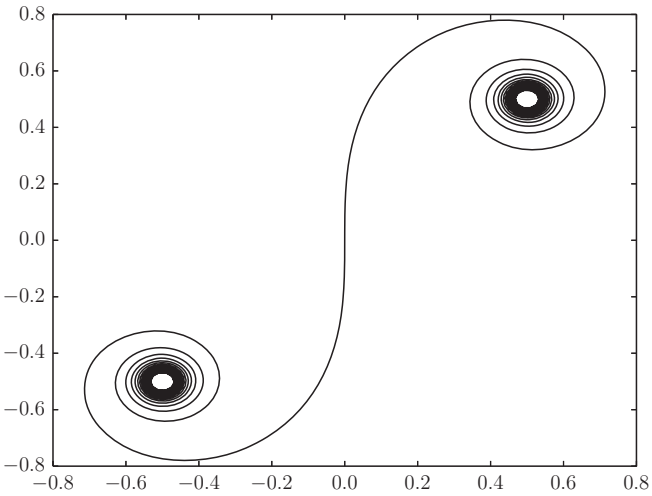


Figure 8.6 The Euler spiral.

Various functions are supplied for the evaluation of different forms of the exponential integral. The standard form is returned by `expi(z)`:

$$\mathrm{Ei}(z) = \int_{-\infty}^{z} \frac{e^t}{t}\,\mathrm{d}t, \quad |\arg(-z) < \pi|$$

`expn(n, x)` returns the value of

$$\int_1^{\infty} \frac{e^{-xt}}{t^n}\,\mathrm{d}t.$$

For $n = 1$, it is faster and more accurate to use `exp1(z)`:

$$\int_1^{\infty} \frac{e^{-zt}}{t}\,\mathrm{d}t.$$

Example E8.11 Any integral of the form

$$\int f(z)e^z\,\mathrm{d}z,$$

where $f(z) = P(z)/Q(z)$ is a rational function, can reduced to the form

$$\int R(z)e^z\,\mathrm{d}z + \sum_i \int \frac{e^z}{(z-a_i)^{n_i}}\,\mathrm{d}z,$$

where $R(z)$ is a polynomial (which may be zero) by expansion in partial fractions. The first integral here can be evaluated by standard methods (repeated integration by parts). Provided the path of integration does not pass through any singular points of the integrand, the second term can be written in terms of exponential integrals.

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use , available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

For example, consider the integral

$$I = \int_{-\infty}^{-2} \frac{e^z}{z^2(z-1)}\,\mathrm{d}z.$$

It can easily be shown that

$$\frac{1}{z^2(z-1)} = \frac{1}{z-1} - \frac{1}{z} - \frac{1}{z^2}$$

and so the integral may be written as the three terms

$$I = \int_{-\infty}^{-2} \frac{e^z}{z-1}\,\mathrm{d}z - \int_{-\infty}^{-2} \frac{e^z}{z}\,\mathrm{d}z - \int_{-\infty}^{-2} \frac{e^z}{z^2}\,\mathrm{d}z.$$

The second integral is simply $-\mathrm{Ei}(-2)$ and substitution $u = z - 1$ resolves the first integral to $e\mathrm{Ei}(-3)$. The last integral may be written in terms of $\mathrm{En}(z)$ or further reduced by integration by parts to

$$\int_{-\infty}^{-2} \frac{e^z}{z^2}\,\mathrm{d}z = -\frac{e^{-2}}{2} + \mathrm{Ei}(-2).$$

Therefore,

$$I = e\mathrm{Ei}(-3) - 2\mathrm{Ei}(-2) - \frac{e^{-2}}{2}.$$

In SciPy,

```
In [x]: import numpy as np
In [x]: from scipy.special import expi
In [x]: np.e * expi(-3) - 2*expi(-2) - np.exp(-2)/2
-0.0053357974213484663
```

8.1.7 Orthogonal polynomials and spherical harmonics

There are a large number of functions in `scipy.special` for the evaluation of different sorts of orthogonal polynomials, including the Legendre, Jacobi, Laguerre, Hermite and different flavors of Chebyshev polynomials. They take the general name `eval_poly(n, x)` where n is the order of the polynomial and `x` is an array-like sequence of values at which to evaluate the polynomial. Table 8.2 gives the names of some of these functions.

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 $n = 0, 1, 2, \cdots$ is called the *degree* and $m = -n, -n+1, \cdots n$ the *order* of the spherical harmonic. The functions $P_n^m(x)$ are the associated Legendre polynomials. As with so many special functions, different fields adopt different phase conventions and normalizations, so it is important to check these carefully and make the appropriate

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

Table 8.2 Some of the orthogonal polynomials in SciPy

Function	Description
`eval_legendre(n, x)`	Legendre polynomial, $P_n(x)$
`eval_chebyt(n, x)`	Chebyshev polynomial of the first kind, $T_n(x)$
`eval_chebyu(n, x)`	Chebyshev polynomial of the second kind, $U_n(x)$
`eval_hermite(n, x)`	(Physicists') Hermite polynomial, $H_n(x)$
`eval_jacobi(n, alpha, beta, x)`	Jacobi polynomial, $P_n^{(\alpha,\beta)}(x)$
`eval_laguerre(n, x)`	Laguerre polynomial of the first kind, $L_n(x)$
`eval_genlaguerre(n, alpha x)`	Generalized Laguerre polynomial of the first kind, $L_n^{\alpha}(x)$

modifications when using them. In particular, many other fields use l for the degree of the harmonic and reverse the definition of θ and ϕ. To be clear, in SciPy θ is the *azimuthal* (longitudinal) angle (taking values between 0 and 2π) and ϕ is the polar (colatitudinal) angle (between 0 and π).

The `scipy.special.sph_harm` method is called with the arguments:

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

where `theta` and `phi` can be array-like objects.

Example E8.12 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 the real part only on the unit sphere. Matplotlib provides a toolkit for such 3D plots, `mplot3d`, as illustrated by the following code which produces Figure 8.7.[6]

Listing 8.7 The spherical harmonic defined by $l = 3, m = 2$

```
# eg8-spherical-harmonics.py

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

phi = np.linspace(0, np.pi, 100)
theta = np.linspace(0, 2*np.pi, 100)
phi, theta = np.meshgrid(phi, theta)

# 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)

m, l = 2, 3
```

[6] See Section 7.2.3 and http://matplotlib.org/mpl_toolkits/mplot3d/.

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use , available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

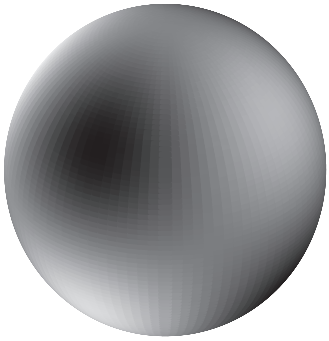


Figure 8.7 A depiction of the spherical harmonic defined by $l = 3, m = 2$.

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

# Set the aspect ratio to 1 so our sphere looks spherical
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))
# Turn off the axis planes
ax.set_axis_off()
plt.show()
```

8.1.8 Exercises

Questions

Q8.1.1 By changing a single line in the program of Example E8.1, output the 10 *most accurately* known constants (excluding those set to their values by definition).

Q8.1.2 Use SciPy's constants and conversion factors to calculate the number density, N/V, of ideal gas molecules at standard temperature and pressure ($T = 0$ °C, $p = 1$ atm). The ideal gas law is $pV = Nk_{\mathrm{B}}T$.

Problems

P8.1.1 Use `scipy.special.binom` to create a depiction of Pascal's triangle of binomial coefficients $\binom{n}{k}$ up to $n = 8$.

P8.1.2 The *Airy pattern* is the circular diffraction pattern of resulting from a uniformly illuminated circular aperture. It consists of a bright, central disc surrounded by fainter rings. Its mathematical description may be written in terms of the Bessel function of the

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use , available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

first kind,

$$I(\theta) = I_0 \left(\frac{2J_1(x)}{x} \right)^2,$$

where θ is the observation angle and $x = ka \sin\theta$. a is the aperture radius and $k = 2\pi/\lambda$ is the *angular wavenumber* of the light with wavelength λ.

Plot the Airy pattern as $I(x)/I_0$ for $-10 \leq x \leq 10$ and deduce from the position of the first minimum in this function the maximum resolving power (in arcsec) of the human eye (pupil diameter 3 mm) at a wavelength of 500 nm.

P8.1.3 Write a function, `get_wv`, which takes a molar bond dissociation energy, `D0`, in kJ mol^{-1} and returns the wavelength of a photon corresponding to that energy *per molecule*, in nm. The energy of a photon with wavelength λ is $E = hc/\lambda$.

For example,

```
In [x]: get_wv(497)
Out[x]: 240.69731528286377
```

P8.1.4 An *ellipsoid* is the three-dimensional figure bounded by the surface described by the equation

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1,$$

where a, b and c are the *semi-principal axes*. If $a = b = c$, the ellipsoid is a sphere. The volume of an ellipsoid has a simple form,

$$V = \frac{4}{3}\pi abc.$$

There is no closed formula for the surface area of a general ellipsoid, but it may be expressed in terms of incomplete elliptic integrals of the first and second kinds, $K(\phi, k)$ and $E(\phi, k)$:

$$S = 2\pi c^2 + \frac{2\pi ab}{\sin\phi} \left(K(\phi, k^2) \cos^2\phi + E(\phi, k^2) \sin^2\phi \right),$$

where

$$\cos\phi = \frac{c}{a}, \quad k = \frac{a\sqrt{b^2 - c^2}}{b\sqrt{a^2 - c^2}}$$

and the coordinate system has been chosen such that $a \geq b \geq c$.

Define a function, `ellipsoid_surface`, to calculate the surface area of a general ellipsoid, and compare the results for different-shaped ellipsoids with the following approximate formula:

$$S \approx 2\pi c^2 + 2\pi abr \left(1 - \frac{b^2 - c^2}{6b^2} r^2 \left(1 - \frac{3b^2 + 10c^2}{56b^2} r^2 \right) \right),$$

where $r = \dfrac{\phi}{\sin\phi}$.

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

P8.1.5 The *drawdown* or change in hydraulic head, s (a measure of the water pressure above some geodetic datum), a distance r from a well at time t, from which water is being pumped at a constant rate, Q, can be modeled using the *Theis* equation,

$$s(r,t) = H_0 - H(r,t) = \frac{Q}{2\pi T} W(u), \quad \text{where} \quad u = \frac{r^2 S}{4Tt}.$$

Here H_0 is the hydraulic head in the absence of the well, S is the aquifer storage coefficient (volume of water released per unit decrease in H per unit area) and T is the transmissivity (a measure of how much water is transported horizontally per unit time). The *Well Function*, $W(u)$ is simply the exponential integral, $E_1(u)$.

For a well being pumped at $Q = 1,000\ \mathrm{m^3\,day^{-1}}$ from an aquifer described by the parameters $H_0 = 20$ m, $S = 0.0003$, $T = 1,000\ \mathrm{m^2\,day^{-1}}$, determine the height of the hydraulic head as a function of r after $t = 1$ day of pumping.

Compare your answer with the approximate version of the Theis equation known as the Jacob equation, in which the well function is taken to be appoximately $W(u) \approx -\gamma - \ln u$ where $\gamma = 0.577215664\cdots$ is the Euler-Mascheroni constant.

P8.1.6 Some electronic components are cooled by annular fins (heatsinks) which conduct heat away from the component and provide a larger surface area for that heat to dissipate to the surroundings.

The cooling efficiency of an annular fin of width $2w$ and inner and outer radii r_0 and r_1 may be written in terms of modified Bessel functions of the first and second kinds:

$$\eta = \frac{2r_0}{\beta(r_1^2 - r_0^2)} \frac{K_1(u_0)I_1(u_1) - I_1(u_0)K_1(u_1)}{K_0(u_0)I_1(u_1) + I_0(u_0)K_1(u_1)},$$

where $u_0 = \beta r_0$, $u_1 = \beta r_1$ and

$$\beta = \sqrt{\frac{h_c}{\kappa w}}.$$

h_c is the heat transfer coefficient (which is taken to be constant over the fin's surface) and κ is the thermal conductivity of the fin material.

What is the cooling efficiency of an aluminium annular fin with dimensions $r_0 = 5$ mm, $r_1 = 10$ mm, $w = 0.1$ mm? Take $h_c = 10\ \mathrm{W\,m^{-2}\,K^{-1}}$ and $\kappa = 200\ \mathrm{W\,m^{-1}\,K^{-1}}$.

Calculate the heat dissipation, $\dot{Q}$ (the product of the efficiency, the fin area and the temperature difference) for a component temperature of $T_0 = 400$ K and ambient temperature $T_\mathrm{e} = 300$ K.

8.2 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 (infinite limits) integrals. It can also perform numerical integration of systems of ordinary differential equations.

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

8.2.1 Definite integrals of a single variable

The basic numerical integration routine is `scipy.integrate.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 (that is, estimated absolute or relative error). In its simplest form, it takes three arguments: a Python function object corresponding to the function to integrate, `func`, and the limits of integration, `a` and `b`. `func` must take at least one argument; if it takes more than one it is integrated along the coordinate corresponding to the first argument. In simple usage, `lambda` expressions are a convenient way to define `func`. For example, to evaluate $\int_1^4 x^{-2}\,\mathrm{d}x = \frac{3}{4}$ numerically:

```
In [x]: from scipy.integrate import quad
In [x]: f = lambda x: 1/x**2
Out[x]: quad(f, 1, 4)
(0.7500000000000002, 1.913234548258995e-09)
```

`quad` returns two values in a tuple – the value of the integral and an estimate of the absolute error in the result.

Use `np.inf` to evaluate improper integrals:

```
In [x]: quad(lambda x: np.exp(-x**2), 0, np.inf)
Out[x]: (0.8862269254527579, 7.101318390472462e-09)
In [x]: np.sqrt(np.pi)/2    # analytical result
Out[x]: 0.88622692545275794
```

Note that in this call to `quad` we didn't even give the function a name but simply passed it as an anonymous `lambda` object.

More complicated functions require a Python function object defined with `def`:

```
In [x]: def g(x):
   ...:     if abs(x) < 0.5:
   ...:         return -x
   ...:     return x - np.sign(x)
   ...:
In [x]: quad(g, -0.6, 0.8)
Out[x]: (-0.06000000000000002, 6.661338147750941e-17)
```

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:

```
In [x]: sinc = lambda x: np.sin(x)/x
In [x]: quad(sinc, -2, 2)
...: RuntimeWarning: invalid value encountered in double_scalars
Out[37]: (nan, nan)
```

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

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

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

Note that break points cannot be specified with infinite limits.

The arguments `epsrel` and `epsabs` allow the specification of a desired accuracy of the quadrature as a relative or absolute tolerance. The default values are both `1.49e-8`, but the integration can be done faster if a less-accurate answer is required. As an example, consider integrating the rapidly varying function, $f(x) = e^{-|x|} \sin^2 x^2$:

```
In [x]: f = lambda x: np.sin(x**2)**2 * np.exp(-np.abs(x))
In [x]: quad(f, -1, 2, epsabs=0.1)
Out[x]: (0.29551455828969975, 0.001529571827911671)
In [x]: quad(f, -1, 2, epsabs=1.49e-8)   # (the default absolute tolerance)
Out[x]: (0.29551455505239044, 4.449763315720537e-10)
```

Note that `epsabs` is only a requested upper bound: the actual estimated accuracy in the result may be much better, and in fact 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 `args`. For example, the integral

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

can be evaluated numerically with

```
In [x]: def f(x, n, m):
  ....:     return np.sin(x)**n * np.cos(x)**m
  ....:
In [x]: n, m = 2, 1
In [x]: quad(f, -np.pi/2, np.pi/2, args=(n, m))
(0.6666666666666666, 1.625746841018571e-13)
```

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

Example E8.13 Consider a torus of average radius R and cross-sectional radius r. The volume of this shape may be evaluated analytically in Cartesian coordinates as a volume of revolution:

$$V = 2 \int_{R-r}^{R+r} 2\pi x z \, \mathrm{d}x, \quad \text{where } z = \sqrt{r^2 - (x-R)^2}.$$

The center of the torus is at the origin and the z axis is taken to be its symmetry axis.

The integral is tedious but yields to standard methods: $V = 2\pi^2 R r^2$. Here we take a numerical approach with the values $R = 4$, $r = 1$:

```
In [x]: R, r = 4, 1
In [x]: f = lambda x, R, r: x * np.sqrt(r**2 - (x-R)**2)
In [x]: V, _ = quad(f, R-r, R+r, args=(R, r))
In [x]: V *= 4 * np.pi
In [x]: Vexact = 2 * np.pi**2 * R * r**2
In [x]: print('V = {} (exact: {})'.format(V, Vexact))
Out[x]: V = 78.95683520871499 (exact: 78.95683520871486)
```

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

8.2.2 Integrals of two and more variables

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

`dblquad` evaluates the double integral:

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

It is passed $f(x,y)$ as a function of at least two variables, `func(y, x, ...)`. The function must take `y` as its first argument and `x` as its second argument. The integral limits are passed to `dblquad` in four further arguments. First, the 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, the value of `x` at which the limit applies (i.e., they must themselves be functions of `x`). If either of the y-integral limits does not depend on x, `gfun` or `hfun` can return a constant value.

As a simple example, the integral

$$\int_1^4 \int_0^2 x^2 y\ \mathrm{d}y\mathrm{d}x$$

can be evaluated with

```
In [x]: f = lambda y, x: x**2 * y
In [x]: a, b = 1, 4
In [x]: gfun = lambda x: 0
In [x]: hfun = lambda x: 2
In [x]: dblquad(f, a, b, gfun, hfun)
Out[x]: (42.00000000000001, 4.662936703425658e-13)
```

Here, `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.

Of course, it is possible to wrap all of this into a single line:

```
In [x]: dblquad(lambda y, x: x**2 * y, 1, 4, lambda x: 0, lambda x: 2)
Out[x]: (42.00000000000001, 4.662936703425658e-13)
```

A double integral can be used to find the area of some two-dimensional shape bounded by one or more functions. For an example in polar coordinates, consider the area inside the curve $r = 2 + 2\sin\theta$ but outside the circle defined by $r = 2$ for θ in $[0, 2\pi]$ (see Figure 8.8). These curves intersect at $\theta = 0, \pi$ so the required integral is

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

where $r\,\mathrm{d}r\,\mathrm{d}\theta$ is the infinitesimal area element in polar coordinates. This particular integral is fairly straightforward to evaluate analytically ($A = 8 + \pi$), so the numerical result is easy to check:

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

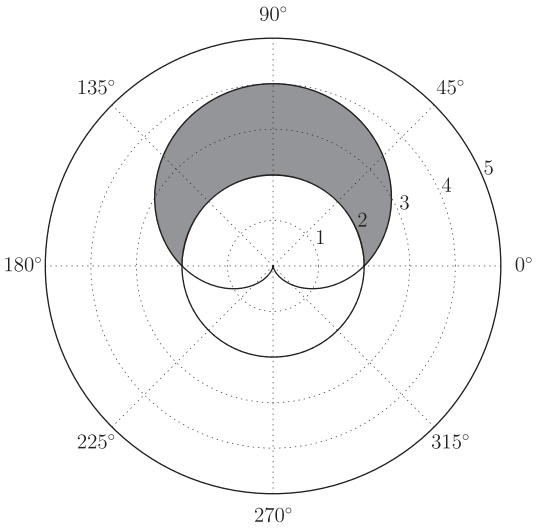


Figure 8.8 The region defined as the area inside $r = 2 + 2\sin\theta$ but outside the circle $r = 2$.

```
In [x]: r1, r2 = lambda theta: 2, lambda theta: 2 + 2*np.sin(theta)
In [x]: A, _ = dblquad(lambda r, theta: r, 0, np.pi, r1, r2)
Out[x]: 11.141592653589791
In [x]: 8 + np.pi       # exact answer
Out[x]: 11.141592653589793
```

The function to evaluate is simply `r`, defined by `lambda r, theta: r`; in the inner integral the limits on r are 2 and $2 + 2\sin\theta$; for the outer integral θ ranges from 0 to π.

The method `tplquad` evaluates triple integrals and takes a function of three variables, `func(z, y, x)` and *six* further arguments: constant x-limits, `a` and `b`, y-limits `gfun(x)` and `hfun(x)` (which are functions, as for `dblquad`, and z-limits `qfun(x, y)` and `rfun(x,y)` (functions of `x` and `y` in that order).

Higher dimensional integrations are handled by the `scipy.integrate.nquad` method which will not be discussed here (documentation and examples are available online).[7]

Example E8.14 The volume of the unit sphere, $4\pi/3$, can be expressed as a triple integral in spherical polar coordinates with constant limits:

$$\int_0^{2\pi}\int_0^{\pi}\int_0^1 r^2\sin\theta\ \mathrm{d}r\mathrm{d}\theta\mathrm{d}\phi.$$

```
In [x]: from scipy.integrate import tplquad
In [x]: tplquad(lambda phi, theta, r: r**2 * np.sin(theta),
                0, 1,
                lambda theta: 0, lambda theta: np.pi,
```

[7] http://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.nquad.html/scipy.integrate.nquad.

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use , available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

```
                    lambda theta, phi: 0, lambda theta, phi: 2*np.pi)
Out[x]: (4.18879020478639, 4.650491330678174e-14)
```

Or in Cartesian coordinates with limits as functions:

$$8\int_0^1 \int_0^{\sqrt{1-x^2}} \int_0^{\sqrt{1-x^2-y^2}} \mathrm{d}z\,\mathrm{d}y\,\mathrm{d}x,$$

where the integral is in the positive octant of the three-dimensonal Cartesian axes.

```
In [x]: A, _ = tplquad(lambda z, y, x: 1,
                       0, 1,
                       lambda x: 0, lambda x: np.sqrt(1 - x**2),
                       lambda x,y: 0, lambda x,y: np.sqrt(1 - x**2 - y**2))
In [x]: 8*A
Out[x]: 4.188790204786391
```

Example E8.15 This example finds the mass and center of mass of the tetrahedron bounded by the coordinate axes and the plane $x + y + z = 1$ with density $\rho = \rho(x, y, z)$ where $\rho(x, y, z)$ is provided as a `lambda` function. We test it with the functions $\rho = 1$, $\rho = x$ and $\rho = x^2 + y^2 + z^2$.

The mass may be written as a triple integral of the density over the volume of the tetrahedron:

$$m = \int_V \rho(x, y, z)\ \mathrm{d}V = \int_0^1 \int_0^{1-x} \int_0^{1-x-y} \rho(x, y, z)\ \mathrm{d}z\,\mathrm{d}y\,\mathrm{d}x,$$

and the coordinates of the center of mass are given by

$$m\bar{x} = \int_V x\rho(x, y, z)\ \mathrm{d}V, \quad m\bar{y} = \int_V y\rho(x, y, z)\ \mathrm{d}V, \quad m\bar{z} = \int_V z\rho(x, y, z)\ \mathrm{d}V.$$

The following program uses `scipy.integrate.tplquad` to perform the necessary integrations (which can also be solved analytically).

Listing 8.8 Calculating the mass and center of mass of a tetrahedron given three different densities

```
# eg8-tetrahedron-cofm.py

import numpy as np
from scipy.integrate import tplquad

# The integration limits on x, y, z:
a, b = 0, 1
gfun, hfun = lambda x: 0, lambda x: 1 - x
qfun, rfun = lambda x, y: 0, lambda x, y: 1 - x - y
❶ lims = (a, b, gfun, hfun, qfun, rfun)

# The three different density functions
rhos = [lambda x, y, z: 1,
        lambda x, y, z: x,
        lambda x, y, z: x**2 + y**2 + z**2]
```

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

```
for rho in rhos:
    # The mass as a triple integral of rho over the volume
    m, _ = tplquad(rho, *lims)
    # The center of mass (xbar, ybar, zbar)
    mxbar, _ = tplquad(lambda x, y, z: x * rho(x,y,z), *lims)
    mybar, _ = tplquad(lambda x, y, z: y * rho(x,y,z), *lims)
    mzbar, _ = tplquad(lambda x, y, z: z * rho(x,y,z), *lims)
    xbar, ybar, zbar = mxbar / m, mybar / m, mzbar / m

    print('mass = {:g}, CofM = ({:g}, {:g}, {:g})'.format(m, xbar, ybar, zbar))
```

❶ Note that the six arguments representing the limits on the triple integral (two constants and two pairs of `lambda` functions) have been packed into a tuple, `lims` (the parentheses are optional here).

The output is:

```
mass = 0.166667, CofM = (0.25, 0.25, 0.25)
mass = 0.0416667, CofM = (0.4, 0.2, 0.2)
mass = 0.05, CofM = (0.277778, 0.277778, 0.277778)
```

8.2.3 Ordinary differential equations

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.[8] `odeint` solves first-order differential equations – *to solve a higher-order equation, it must be decomposed into a system of first-order equations first*, as explained later.

A single first-order ordinary differential equation

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

$$\frac{\mathrm{d}y}{\mathrm{d}t} = f(y, t),$$

`odeint` takes three arguments: a function object returning $\mathrm{d}y/\mathrm{d}t$, an initial condition, y_0, and a sequence of t values at which to calculate the solution, $y(t)$.

For example, consider the first-order differential equation describing the rate of the reaction $\mathrm{A} \rightarrow \mathrm{P}$ in terms of the concentration of the reactant, A:

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

This example has an easily obtainable analytical solution:

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

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

[8] 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.

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

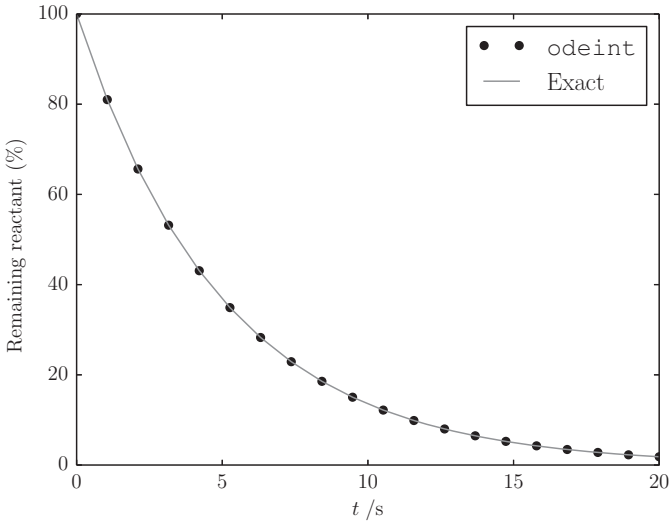


Figure 8.9 Exponential decay of a reactant in a first-order reaction: numerical and exact solutions.

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

$$\frac{\mathrm{d}y}{\mathrm{d}t} = -ky$$

We need to provide a function returning $\mathrm{d}y/\mathrm{d}t$ as $f(y, t)$ (in general a function of both y and 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
```

(the order of the arguments is important). A program comparing the numerical and analytical results for a reaction with $k = 0.2\ \mathrm{s}^{-1}$ and $y(0) \equiv [\mathrm{A}]_0 = 100$ is given later; the resulting plot is Figure 8.9.

Listing 8.9 First-order reaction kinetics

```
import numpy as np
from scipy.integrate import odeint
import pylab

# First-order reaction rate constant, s-1
k = 0.2
# Initial condition on y: 100% of reactant is present at t=0
y0 = 100

# A suitable grid of time points for the reaction
t = np.linspace(0, 20, 20)
```

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use , available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

```
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
pylab.plot(t, y, 'o', color='k', label=r'\texttt{odeint}')
pylab.plot(t, y0 * np.exp(-k*t), color='gray', label='Exact')
pylab.xlabel(r'$t\;/\mathrm{s}$')
pylab.ylabel('Remaining reactant (\%)')
pylab.legend()
pylab.show()
```

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

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

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

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

Coupled first-order ordinary differential equations

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

$$\begin{aligned}
\frac{\mathrm{d}y_1}{\mathrm{d}t} &= f_1(y_1, y_2, \cdots, y_n; t) \\
\frac{\mathrm{d}y_2}{\mathrm{d}t} &= f_2(y_1, y_2, \cdots, y_n; t) \\
&\cdots \\
\frac{\mathrm{d}y_n}{\mathrm{d}t} &= f_n(y_1, y_2, \cdots, y_n; t)
\end{aligned}$$

In this case, the function passed to `odeint()` must return a sequence of derivatives, $\mathrm{d}y_1/\mathrm{d}t, \mathrm{d}y_2/\mathrm{d}t, \cdots, \mathrm{d}y_n/\mathrm{d}t$ for each of the dependent variables; that is, it evaluates the earlier mentioned functions $f_i(y_1, y_2, \cdots, y_n; t)$ for each of the y_i passed to it in a sequence, `y`. The form of this function is:

```
def deriv(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)
    # ... etc
    # Return the derivatives in a sequence such as a tuple:
    return dy1dt, dy2dt, ..., dyndt
```

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use , available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

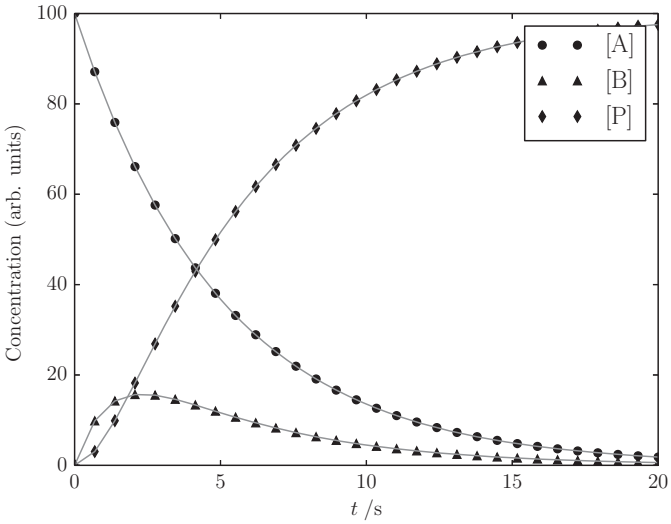


Figure 8.10 Two coupled first-order reactions: numerical and exact solutions.

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

$$\frac{\mathrm{d[A]}}{\mathrm{d}t} = -k_1[\mathrm{A}]$$
$$\frac{\mathrm{d[B]}}{\mathrm{d}t} = k_1[\mathrm{A}] - k_2[\mathrm{B}]$$

Again, we can solve this pair of coupled equations analytically, but in our numerical solution, let $y_1 \equiv [\mathrm{A}]$ and $y_2 \equiv [\mathrm{B}]$:

$$\frac{\mathrm{d}y_1}{\mathrm{d}t} = -k_1 y_1$$
$$\frac{\mathrm{d}y_2}{\mathrm{d}t} = k_1 y_1 - k_2 y_2$$

The code mentioned here integrates these equations 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 (Figure 8.10).

Listing 8.10 Two coupled first-order reactions

```
import numpy as np
from scipy.integrate import odeint
import pylab

# First-order reaction rate constants, 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
```

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use , available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

```
# A suitable grid of time points for the reaction
t = np.linspace(0, 20, 100)

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 differential equation
y0 = A0, B0
y1, y2 = odeint(dydt, y0, t, args=(k1, k2)).T

A, B = y1, y2
# [P] is determined by conservation
P = A0 - A - B

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

pylab.plot(t, A, 'o', label='[A]')
pylab.plot(t, B, '^', label='[B]')
pylab.plot(t, P, 'd', label='[P]')
pylab.plot(t, Aexact)
pylab.plot(t, Bexact)
pylab.plot(t, Pexact)
pylab.xlabel(r'$t\;/\mathrm{s}$')
pylab.ylabel('Concentration (arb. units)')
pylab.legend()
pylab.show()
```

❶ Note that `odeint` returns a two-dimensional 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, we need the transpose of this returned array.

A single second-order ordinary differential equation

To solve an ordinary differential equation of higher than first order, it must first be 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 example, the equation of motion for a harmonic oscillator is a second-order differential equation:

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

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

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use , available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

$$\frac{dx_1}{dt} = x_2,$$
$$\frac{dx_2}{dt_2} = -\omega^2 x_1,$$

where x_1 is identified with x and x_2 with dx/dt.

This pair of coupled first-order equations may be solved as before:

Listing 8.11 Solution of the harmonic oscillator equation of motion

```
import numpy as np
from scipy.integrate import odeint
import pylab

# Harmonic oscillator frequency (s-1)
omega = 0.9
# initial conditions on x1=x and x2=dx/dt at t=0
A, v0 = 3, 0        # cm, cm.s-1
x0 = A, v0

# A suitable grid of time points
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 differential equation
x1, x2 = odeint(dxdt, x0, t, args=(omega,)).T

# Plot and compare the numerical and exact solutions
pylab.plot(t, x1, 'o', color='k', label=r'\texttt{odeint()}')
pylab.plot(t, A * np.cos(omega * t), color='gray', label='Exact')
pylab.xlabel(r'$t\;/\mathrm{s}$')
pylab.ylabel(r'$x\;/\mathrm{cm}$')
pylab.legend()
pylab.show()
```

The plot produced by this code is given in Figure 8.10.

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

Example E8.16 An object falling slowly in a viscous fluid under the influence of gravity is subject to a drag force (*Stokes drag*), which varies linearly with its velocity. Its equation of motion may be written as the second-order differential equation:

$$m\frac{d^2z}{dt^2} = -c\frac{dz}{dt} + mg',$$

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

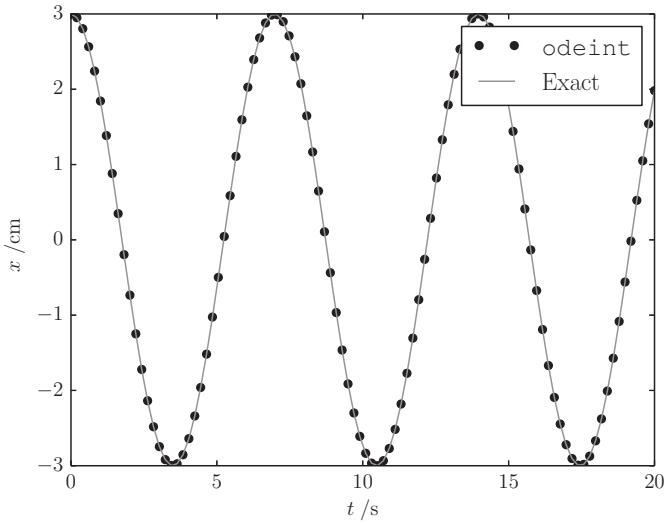


Figure 8.11 The harmonic oscillator: numerical and exact solutions.

where z is the object's position as a function of time, t, c is a drag constant which depends on the shape of the object and the fluid viscosity and

$$g' = g\left(1 - \frac{\rho_{\text{fluid}}}{\rho_{\text{obj}}}\right)$$

is the effective gravitational acceleration, which accounts for the buoyant force due to the fluid (density ρ_{fluid}) displaced by the object (density ρ_{obj}). For a small sphere of radius r in a fluid of viscosity η, Stokes' law predicts $c = 6\pi\eta r$.

Consider a sphere of platinum ($\rho = 21.45\ \text{g cm}^{-3}$) with radius 1 mm, initially at rest, falling in mercury ($\rho = 13.53\ \text{g cm}^{-3}$, $\eta = 1.53 \times 10^{-3}$ Pa s). The earlier mentioned second-order differential equation can be solved analytically, but to integrate it numerically using `odeint`, it must be treated as two first-order ordinary differential equations:

$$\frac{\mathrm{d}z}{\mathrm{d}t} = \dot{z}$$
$$\frac{\mathrm{d}^2 z}{\mathrm{d}t^2} = \frac{\mathrm{d}\dot{z}}{\mathrm{d}t} = g' - \frac{c}{m}\dot{z}$$

In the code mentioned here, the function `deriv` calculates these derivatives and is passed to `odeint` with the intial conditions ($z = 0, \dot{z} = 0$) and a grid of time points.

Listing 8.12 Calculating the motion of a sphere falling under the influence of gravity and Stokes drag

```
# eg8-stokes-drag.py
import numpy as np
from scipy.integrate import odeint
import pylab

# Pt sphere falling from rest in mercury
```

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use , available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

```
# Acceleration due to gravity (m.s-2)
g = 9.81
# Densities (kg.m-3)
rho_Pt, rho_Hg = 21450, 13530
# Viscosity  of Hg (Pa.s)
eta = 1.53e-3

# Radius and mass of the sphere
r = 1.e-3   # radius (m)
m = 4*np.pi/3 * r**3 * rho_Pt
# Drag consant from Stokes' Law:
c = 6 * np.pi * eta * r
# Effective gravitational acceleration
gp = g * (1 - rho_Hg/rho_Pt)

def deriv(z, t, m, c, gp):
    """ Return the dz/dt and d2z/dt2. """
    dz0 = z[1]
    dz1 = gp - c/m * z[1]
    return dz0, dz1

t = np.linspace(0, 20, 50)
# Initial conditions: z = 0, dz/dt = 0 at t=0
z0 = (0, 0)

# Integrate the pair of differential equations
z, zdot = odeint(deriv, z0, t, args=(m, c, gp)).T
pylab.plot(t, zdot)

print('Estimate of terminal velocity = {:.3f} m.s-1'.format(zdot[-1]))

# Exact solution: terminal velocity vt (m.s-1) and characteristic time tau (s)
v0, vt, tau = 0, m*gp/c, m/c
print('Exact terminal velocity = {:.3f} m.s-1'.format(vt))
z = vt*t + v0*tau*(1-np.exp(-t/tau)) + vt*tau*(np.exp(-t/tau)-1)
zdot_exact = vt + (v0-vt)*np.exp(-t/tau)
pylab.plot(t, zdot_exact)
pylab.xlabel('$t$ /s')
pylab.ylabel('$\dot{z}\;/\mathrm{m\, s^{-1}}$')

pylab.show()
```

The plot produced by this program is shown in Figure 8.12: the numerical and analytical results are indistinguishable at this scale but are reported to three decimal places in the output:

```
Estimate of terminal velocity = 11.266 m.s-1
Exact terminal velocity = 11.285 m.s-1
```

8.2.4 Exercises

Questions

Q8.2.1 Use `scipy.integrate.quad` to evaluate the following integral:

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

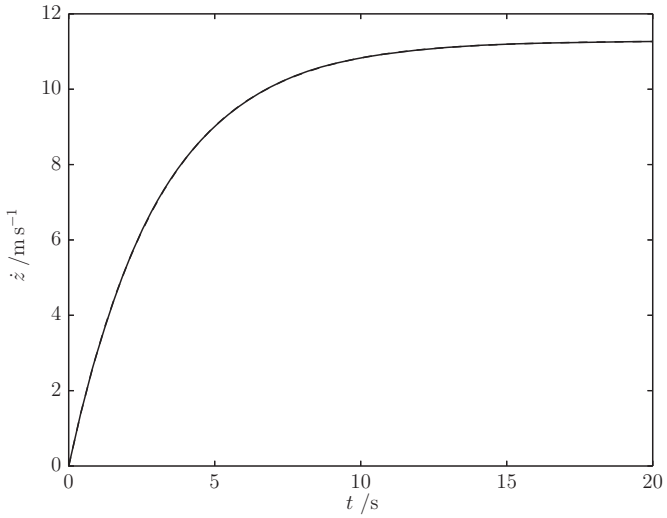


Figure 8.12 The velocity of a platinum sphere falling in mercury as a function of time, modeled with Stokes' law.

$$\int_0^6 \lfloor x \rfloor - 2 \left\lfloor \frac{x}{2} \right\rfloor \, \mathrm{d}x.$$

Q8.2.2 Use `scipy.integrate.quad` to evaluate the following definite integrals (most of which can also be expressed in closed form over the range given but are awkward).

a.
$$\int_0^1 \frac{x^4(1-x)^4}{1+x^2} \, \mathrm{d}x.$$
(Compare with $22/7 - \pi$.)

b. The following integral appears in the Debye theory of the heat capacity of crystals at low temperature
$$\int_0^\infty \frac{x^3}{e^x - 1} \, \mathrm{d}x.$$
(Compare with $\pi^4/15$.)

c. The integral sometimes known as the *Sophomore's dream*:
$$\int_0^1 x^{-x} \, \mathrm{d}x$$
(Compare the value you obtain from the summation $\sum_{n=1}^{\infty} n^{-n}$.)

d.
$$\int_0^1 [\ln(1/x)]^p \, \mathrm{d}x$$

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use , available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

(Compare with $p!$ for integer $0 \leq p \leq 10$.)

e.

$$\int_0^{2\pi} e^{z\cos\theta}\,\mathrm{d}\theta$$

(Compare with $I_0(z)/2\pi$, where $I_0(z)$ is a modified Bessel function of the first kind, for $0 \leq z \leq 2$.)

Q8.2.3 Use `scipy.integrate.dblquad` to evaluate π by integration of the constant function $f(x, y) = 4$ over the quarter circle with unit radius in the quadrant $x > 0, y > 0$.

Q8.2.4 What is wrong with the following attempt to calculate the area of the unit circle (π) as a double integral in polar coordinates?

```
In [x]: dblquad(lambda r, theta: r, 0, 1, lambda r: 0, lambda r: 2*np.pi)
Out[x]: (19.739208802178712, 2.1914924100062363e-13)
```

Problems

P8.2.1 The area of the surface of revolution about the x-axis between a and b of the function $y = f(x)$ is given by the integral

$$S = 2\pi \int_a^b y\,\mathrm{d}s, \quad \text{where } \mathrm{d}s = \sqrt{1 + \left(\frac{\mathrm{d}y}{\mathrm{d}x}\right)^2}\,\mathrm{d}x.$$

Use this equation to write a function to determine the surface area of revolution of a function $y = f(x)$ about the x-axis, given Python function objects that return y and $\mathrm{d}y/\mathrm{d}x$, and test it for the paraboloid obtained by rotation of the function $f(x) = \sqrt{x}$ about the x-axis between $a = 0$ and $b = 1$. Compare with the exact result, $\pi(5^{3/2} - 1)/6$.

P8.2.2 The integral of the secant function,

$$\int_0^{\theta} \sec\phi\,\mathrm{d}\phi$$

for $-\pi/2 < \theta < \pi/2$ is important in navigation and the theory of map projections. It can be expressed in closed form as the inverse Gudermannian function,

$$\mathrm{gd}^{-1}(\theta) = \ln|\sec\theta + \tan\theta|.$$

Use `scipy.integrate.quad` to calculate values for the integral across the relevant range for θ given earlier and compare graphically with the exact answer.

P8.2.3 Consider a torus of uniform density, unit mass, average radius R and cross-sectional radius r. The volume and moments of inertia of such a torus may be evaluated analytically and give the results:

$$V = 2\pi^2 R r^2,$$
$$I_z = R^2 + \tfrac{3}{4}r^2,$$
$$I_x = I_y = \tfrac{1}{2}R^2 + \tfrac{5}{8}r^2,$$

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use , available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

where the center of mass of the torus is at the origin and the z axis is taken to be its symmetry axis.

Here we take a numerical approach. In cylindrical coordinates (ρ, θ, z), it may be shown that:

$$V = 2\int_0^{2\pi}\int_{R-r}^{R+r}\int_0^{\sqrt{r^2-(\rho-R)^2}} \rho \; \mathrm{d}z\,\mathrm{d}\rho\,\mathrm{d}\theta,$$

$$I_z = \frac{2}{V}\int_0^{2\pi}\int_{R-r}^{R+r}\int_0^{\sqrt{r^2-(\rho-R)^2}} \rho^3 \; \mathrm{d}z\,\mathrm{d}\rho\,\mathrm{d}\theta,$$

$$I_x = I_y = \frac{2}{V}\int_0^{2\pi}\int_{R-r}^{R+r}\int_0^{\sqrt{r^2-(\rho-R)^2}} (\rho^2\sin^2\theta + z^2)\rho \; \mathrm{d}z\,\mathrm{d}\rho\,\mathrm{d}\theta.$$

Evaluate these integrals for the torus with dimensions $R = 4$, $r = 1$ and compare with the exact values.

P8.2.4 The *Brusselator* is a theoretical model for an autocatalytic reaction. It assumes the following reaction sequence, in which species A and B are taken to be in excess with constant concentration and species D and E are removed as they are produced. The concentrations of species X and Y can show oscillatory behavior under certain conditions.

$$\begin{aligned} \mathrm{A} &\rightarrow \mathrm{X} && k_1 \\ 2\mathrm{X} + \mathrm{Y} &\rightarrow 3\mathrm{X} && k_2 \\ \mathrm{B} + \mathrm{X} &\rightarrow \mathrm{Y} + \mathrm{D} && k_3 \\ \mathrm{X} &\rightarrow \mathrm{E} && k_4 \end{aligned}$$

It is convenient to introduce the scaled quantities

$$x = [\mathrm{X}]\sqrt{\frac{k_2}{k_4}}, \quad y = [\mathrm{Y}]\sqrt{\frac{k_2}{k_4}},$$

$$a = [\mathrm{A}]\frac{k_1}{k_4}\sqrt{\frac{k_2}{k_4}}, \quad b = [\mathrm{B}]\frac{k_3}{k_4},$$

and to scale the time by the factor k_4, which gives rise to the dimensionless equations

$$\frac{\mathrm{d}x}{\mathrm{d}t} = a - (1+b)x + x^2 y,$$

$$\frac{\mathrm{d}y}{\mathrm{d}t} = bx + x^2 y.$$

Show how these equations predict x and y to vary for (a) $a = 1, b = 1.8$ and (b) $a = 1, b = 2.02$ by plotting in each case (i) x, y as functions of (dimensionless) time and (ii) y as a function of x.

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use , available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

P8.2.5 The equation governing the motion of a pendulum consisting of a mass at the end of a light, rigid rod of length l may be written

$$\frac{\mathrm{d}^2\theta}{\mathrm{d}t^2} = -\frac{g}{l}\sin\theta,$$

where θ is the angle the pendulum makes with the vertical.

Taking $l = 1$ m and $g = 9.81\ \mathrm{m\,s^{-2}}$, determine the subsequent motion of the pendulum if it is started at rest with an initial angle $\theta_0 = 30°$. Compare the motion with the harmonic approximation reached by assuming θ is small, which has the analytical solution $\theta = \theta_0\cos(\omega t)$ with $\omega = \sqrt{g/l}$.

P8.2.6 A simple mechanism for the formation of ozone in the stratosphere consists of the following four reactions (known as the *Chapman cycle*):

$$\begin{aligned}
\mathrm{O_2} + h\nu &\rightarrow 2\mathrm{O} & k_1 &= 3\times10^{-12}\ \mathrm{s^{-1}}\\
\mathrm{O_2} + \mathrm{O} + \mathrm{M} &\rightarrow \mathrm{O_3} + \mathrm{M} & k_2 &= 1.2\times10^{-33}\ \mathrm{cm^6\,molec^{-2}\,s^{-1}}\\
\mathrm{O_3} + h\nu' &\rightarrow \mathrm{O} + \mathrm{O_2} & k_3 &= 5.5\times10^{-4}\ \mathrm{s^{-1}}\\
\mathrm{O} + \mathrm{O_3} &\rightarrow 2\mathrm{O_2} & k_4 &= 6.9\times10^{-16}\ \mathrm{cm^3\,molec^{-1}\,s^{-1}}
\end{aligned}$$

where M is a nonreacting third body taken to be at the total air molecule concentration for the altitude being considered. The earlier mentioned reactions lead to the following rate equations for [O], [O_3] and [O_2]:

$$\begin{aligned}
\frac{\mathrm{d[O_2]}}{\mathrm{d}t} &= -k_1[\mathrm{O_2}] - k_2[\mathrm{O_2}][\mathrm{O}][\mathrm{M}] + k_3[\mathrm{O_3}] + 2k_4[\mathrm{O}][\mathrm{O_3}]\\
\frac{\mathrm{d[O]}}{\mathrm{d}t} &= 2k_1[\mathrm{O_2}] - k_2[\mathrm{O_2}][\mathrm{O}][\mathrm{M}] + k_3[\mathrm{O_3}] - k_4[\mathrm{O}][\mathrm{O_3}]\\
\frac{\mathrm{d[O_3]}}{\mathrm{d}t} &= k_2[\mathrm{O_2}][\mathrm{O}][\mathrm{M}] - k_3[\mathrm{O_3}] - k_4[\mathrm{O}][\mathrm{O_3}]
\end{aligned}$$

The rate constants apply at an altitude of 25 km, where $[\mathrm{M}] = 9\times10^{17}\ \mathrm{molec\,cm^{-3}}$. Write a program to determine the concentrations of O_3 and O as a function of time at this altitude (you should find the [O_2] remains pretty much constant). Start with initial conditions $[\mathrm{O_2}]_0 = 0.21[\mathrm{M}]$, $[\mathrm{O}]_0 = [\mathrm{O_3}]_0 = 0$ and integrate for 10^8 s (starting from scratch it takes about three years to build an ozone layer with this mechanism). Compare the equilibrium concentrations with the approximate analytical result obtained using the *steady-state approximation*:

$$[\mathrm{O_3}] = \sqrt{\frac{k_1k_2}{k_3k_4}}[\mathrm{O_2}][\mathrm{M}]^{\frac{1}{2}}, \quad \frac{[\mathrm{O}]}{[\mathrm{O_3}]} = \frac{k_3}{k_2[\mathrm{O_2}][\mathrm{M}]}.$$

P8.2.7 Hyperion is an irregularly shaped moon of Saturn notable for its chaotic rotation. Its motion may be modeled as follows.

The orbit of Hyperion (H) about Saturn (S) is an ellipse with semi-major axis, a, and eccentricity, e. Let its point of closest approach (*periapsis*) be P. Its distance from the

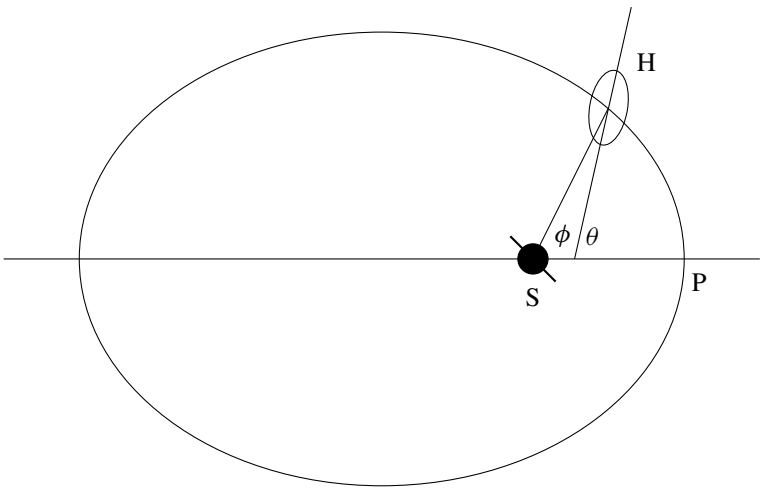
Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

planet, SH, as a function of its *true anomaly* (orbital angle, ϕ, measured from the line SP) is therefore

$$r = \frac{a(1-e^2)}{1+e\cos\phi}.$$

Define the angle θ to be that between the axis of the smallest principal moment of inertia (loosely, the longest axis of the moon) and SP, and the quantity Ω to be a scaled rate of change of θ with ϕ (i.e., the rate at which Hyperion spins as it orbits Saturn) as follows:

$$\Omega = \frac{a^2}{r^2}\frac{\mathrm{d}\theta}{\mathrm{d}\phi}.$$



Now, it can be shown that

$$\frac{\mathrm{d}\Omega}{\mathrm{d}\phi} = -\frac{B-A}{C}\frac{3}{2(1-e^2)}\frac{a}{r}\sin[2(\theta-\phi)],$$

where A, B and C are the principal moments of inertia.

Use `scipy.integrate.odeint` to find and plot the spin rate, Ω, as a function of ϕ for the initial conditions (a) $\theta = \Omega = 0$ at $\phi = 0$, and (b) $\theta = 0$, $\Omega = 2$ at $\phi = 0$. Take $e = 0.1$ and $(B-A)/C = 0.265$.

P8.2.8 The radioactive decay chain of ^{212}Pb to the stable isotope ^{208}Pb may be considered as the following sequence of steps with the given rate constants, k_i:

$$\begin{aligned}
^{212}\mathrm{Pb} &\rightarrow {}^{212}\mathrm{Bi} + \beta^- & k_1 &= 1.816\times10^{-5}\ \mathrm{s}^{-1},\\
^{212}\mathrm{Bi} &\rightarrow {}^{208}\mathrm{Tl} + \alpha & k_2 &= 6.931\times10^{-5}\ \mathrm{s}^{-1},\\
^{212}\mathrm{Bi} &\rightarrow {}^{212}\mathrm{Po} + \beta^- & k_3 &= 1.232\times10^{-4}\ \mathrm{s}^{-1},\\
^{208}\mathrm{Tl} &\rightarrow {}^{208}\mathrm{Pb} + \beta^- & k_4 &= 3.851\times10^{-3}\ \mathrm{s}^{-1},\\
^{212}\mathrm{Po} &\rightarrow {}^{208}\mathrm{Pb} + \alpha & k_5 &= 2.310\ \mathrm{s}^{-1}.
\end{aligned}$$

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

By considering the following first-order differential equations giving the rates of change for each species, plot their concentrations as a function of time.

$$\frac{\mathrm{d}[^{212}\mathrm{Pb}]}{\mathrm{d}t} = -k_1[^{212}\mathrm{Pb}]$$
$$\frac{\mathrm{d}[^{212}\mathrm{Bi}]}{\mathrm{d}t} = k_1[^{212}\mathrm{Pb}] - k_2[^{212}\mathrm{Bi}] - k_3[^{212}\mathrm{Bi}]$$
$$\frac{\mathrm{d}[^{208}\mathrm{Tl}]}{\mathrm{d}t} = k_2[^{212}\mathrm{Bi}] - k_4[^{208}\mathrm{Tl}]$$
$$\frac{\mathrm{d}[^{212}\mathrm{Po}]}{\mathrm{d}t} = k_3[^{212}\mathrm{Bi}] - k_5[^{212}\mathrm{Po}]$$
$$\frac{\mathrm{d}[^{208}\mathrm{Pb}]}{\mathrm{d}t} = k_4[^{208}\mathrm{Tl}] + k_5[^{212}\mathrm{Po}]$$

If all the intermediate species, J, are treated in "steady state" (i.e., $\mathrm{d}[\mathrm{J}]/\mathrm{d}t = 0$, the approximate expression for the ^{208}Pb concentration as a function of time is

$$[^{208}\mathrm{Pb}] = [^{212}\mathrm{Pb}]_0 \left(1 - e^{-k_1 t}\right).$$

Compare the "exact" result obtained by numerical integration of the differential equations with this approximate answer.

8.3 Interpolation

The package `scipy.interpolate` contains a large variety of functions and classes for interpolation and splines in one and more dimensions. Some of the more important are described in this section.

8.3.1 Univariate interpolation

The most straightforward one-dimensional interpolation functionality is provided by `scipy.interpolate.interp1d`. Given arrays of points `x` and `y`, a function is returned, which can be called to generate interpolated values at intermediate values of `x`. The default interpolation scheme is linear, but other options (see Table 8.3) allow for different schemes, as shown in the following example.

Example E8.17 This example demonstrates some of the different interpolation methods available in `scipy.interpolation.interp1d` (see Figure 8.13).

Listing 8.13 A comparison of one-dimensional interpolation types using `scipy.interpolate.interp1d`

```
# eg8-interp1d.py
import numpy as np
from scipy.interpolate import interp1d
import pylab
```

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

Table 8.3 Interpolation methods specified by the `kind` argument to `scipy.interpolate.interp1d`

`kind`	Description
`'linear'`	The default, linear interpolation using 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 (in practice, the same as `'linear'`)
`'quadratic'`	Second-order spline interpolation
`'cubic'`	Cubic spline interpolation

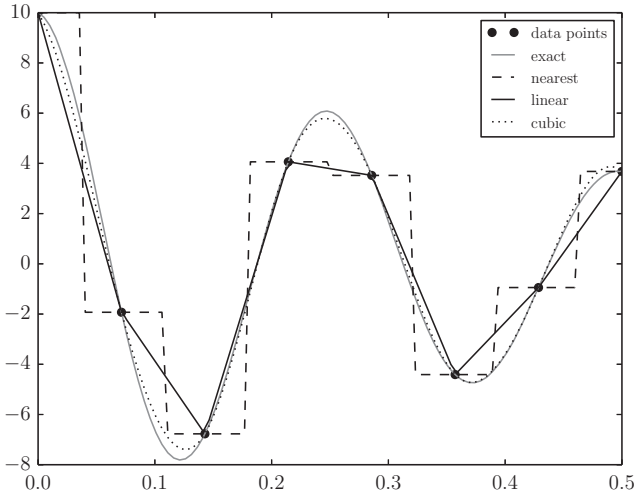


Figure 8.13 An illustration of different one-dimensional interpolation methods with `scipy.interpolation.interp1d`.

```
A, nu, k = 10, 4, 2

def f(x, A, nu, k):
    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)

f_nearest = interp1d(x, y, kind='nearest')
f_linear  = interp1d(x, y)
f_cubic   = interp1d(x, y, kind='cubic')

x2 = np.linspace(0, xmax, 100)
pylab.plot(x, y, 'o', label='data points')
pylab.plot(x2, f(x2, A, nu, k), label='exact')
pylab.plot(x2, f_nearest(x2), label='nearest')
```

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use , available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

```
pylab.plot(x2, f_linear(x2), label='linear')
pylab.plot(x2, f_cubic(x2), label='cubic')
pylab.legend()
pylab.show()
```

8.3.2 Multivariate interpolation

We shall consider two kinds of multivariate interpolation corresponding to whether or not the source data are structured (arranged on some kind of grid) or not.

Interpolation from a rectangular grid

The simplest two-dimensional interpolation routine is `scipy.interpolate.interp2d`. It requires a two-dimensional array of values, `z`, and the two (one-dimensional) coordinate arrays `x` and `y` to which they correspond. These arrays need not have constant spacing. Three kinds of interpolation spline are supported through the `kind` argument: `'linear'` (the default), `'cubic'` and `'quintic'`.

Example E8.18 In the following example, we calculate the function

$$z(x, y) = \sin\left(\frac{\pi x}{2}\right) e^{y/2}$$

on a grid of points (x, y) which is not evenly spaced in the y-direction. We then use `scipy.interpolate.interp2d` to interpolate these values onto a finer, evenly spaced (x, y) grid: see Figure 8.14.

Listing 8.14 Two-dimensional interpolation with `scipy.interpolate.interp2d`

```
# eg8-interp2d.py
import numpy as np
from scipy.interpolate import interp2d
import matplotlib.pyplot as plt

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)

x2 = np.linspace(0, 4, 65)
y2 = np.linspace(0, 4, 65)
f = interp2d(x, y, Z, kind='cubic')
Z2 = f(x2, y2)

fig, ax = plt.subplots(nrows=1, ncols=2)
ax[0].pcolormesh(X, Y, Z)

X2, Y2 = np.meshgrid(x2, y2)
ax[1].pcolormesh(X2, Y2, Z2)

plt.show()
```

❶ Note that `interp2d` requires the *one-dimensional* arrays, `x` and `y`.

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

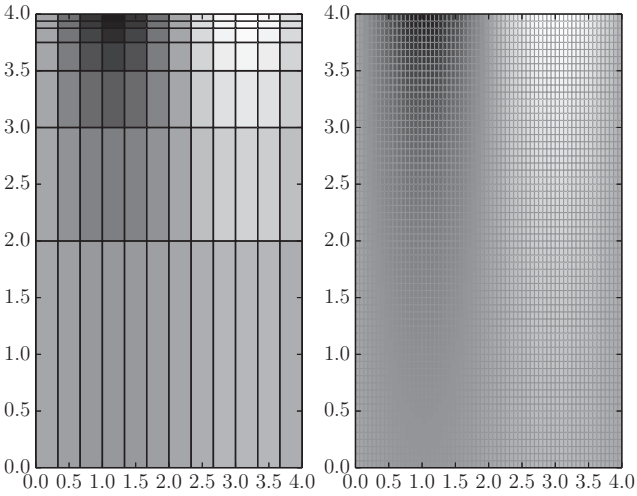


Figure 8.14 Two-dimensional interpolation with `scipy.interpolate.interp2d`.

If the mesh of (x, y) coordinates form a *regularly spaced* grid, the fastest way to interpolate values from values of `z` is to use a `scipy.interpolate.RectBivariateSpline` object as in the following example.

Example E8.19 In the following code, the function

$$z(x, y) = e^{-4x^2} e^{-y^2/4}$$

is calculated on a regular, coarse grid and then interpolated onto a finer one (Figure 8.15).

Listing 8.15 Interpolation onto a regular two-dimensional grid with `scipy.interpolate.RectBivariateSpline`

```
# eg8-RectBivariateSpline.py
import numpy as np
from scipy.interpolate import RectBivariateSpline
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Regularly spaced, coarse grid
dx, dy = 0.4, 0.4
xmax, ymax = 2, 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)

❶ interp_spline = RectBivariateSpline(y, x, Z)

# Regularly spaced, fine grid
dx2, dy2 = 0.16, 0.16
x2 = np.arange(-xmax, xmax, dx2)
```

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use , available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

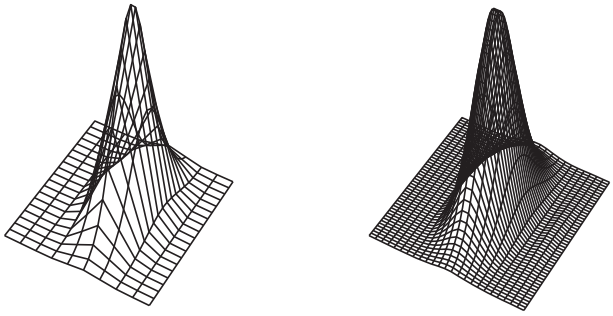


Figure 8.15 Two-dimensional interpolation from a coarse rectangular grid (left-hand plot) to a finer one (right-hand plot) with `scipy.interpolate.RectBivariateSpline`.

```
y2 = np.arange(-ymax, ymax, dy2)
X2, Y2 = np.meshgrid(x2,y2)
Z2 = interp_spline(y2, x2)

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()
```

❶ Note that for our function, `Z`, defined using the `meshgrid` set up here, the `RectBivariateSpline` method expects the corresponding one-dimensional arrays `y` and `x` to be passed in this order (opposite to that of `interp2d`).[9]

Interpolation of unstructured data

To interpolate unstructured data (that is, 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')
```

where 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. `xi` is an array of the coordinate grid to by interpolated onto (of shape `(m,2)`.) The `methods` available are `'linear'` (the default), `'nearest'` and `'cubic'`.

[9] This issue is related to the way that `meshgrid` is indexed, which is based on the conventions of MATLAB.

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use , available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

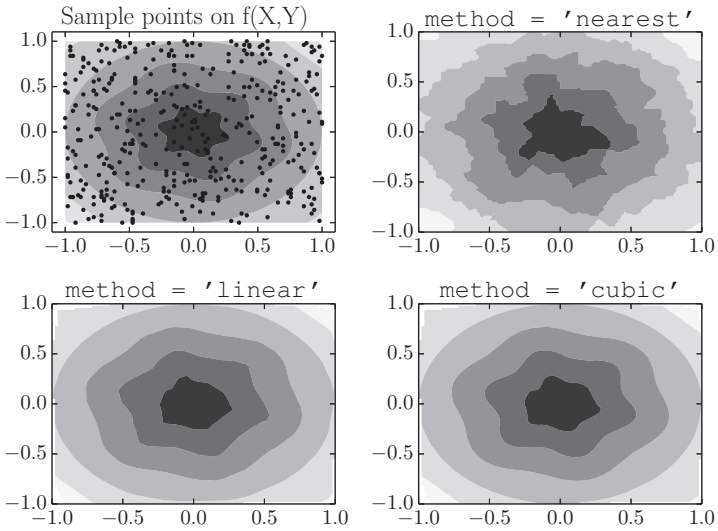


Figure 8.16 Some different interpolation schemes for `scipy.interpolate.griddata`.

Example E8.20 The code mentioned here illustrates the different kinds of interpolation method available for `scipy.interpolate.griddata` using 400 points chosen randomly from an interesting function. The results can be compared in Figure 8.16.

Listing 8.16 Interpolation from an unstructured array of two-dimensional points with `scipy.interpolate.griddata`

```
# eg8-gridinterp.py
import numpy as np
from scipy.interpolate import griddata
import matplotlib.pyplot as plt

x = np.linspace(-1,1,100)
y =  np.linspace(-1,1,100)
X, Y = np.meshgrid(x,y)

def f(x, y):
    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 random point from the discrete domain of our model function
npts = 400
px, py = np.random.choice(x, npts), np.random.choice(y, npts)

fig, ax = plt.subplots(nrows=2, ncols=2)
# Plot the model function and the randomly selected sample points
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)')
```

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

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

plt.show()
```

8.4 Optimization, data-fitting and root-finding

The `scipy.optimize` package provides a range of popular algorithms for minimization of multidimensional functions (with or without additional constraints), least-squares data-fitting and multidimensional equation solving (root-finding). This section will give an overview of the more important options available, but it should be borne 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 minimum (or root, etc.), 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 – consider a two-dimensional function as a landscape with several valleys separated by steep ridges: an initial guess placed within one valley is likely to lead most algorithms to wander downhill and find the minimum in that valley (even if it isn't the *global* minimum) without climbing the ridges. Similarly, you might expect (but cannot *guarantee*) that most numerical root-finders return the "nearest" root to the initial guess.

8.4.1 Minimization

SciPy's optimization routines *minimize* a function of one or more variables, $f(x_1, x_2, \cdots, x_n)$. To find the *maximum*, one determines the minimum of $-f(x_1, x_2, \cdots, x_n)$.

Some of the minimization algorithms only require the function itself to be evaluated; others require its first derivative with respect to each of the variables in an array known as the *Jacobian*:

$$J(f) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \cdots, \frac{\partial f}{\partial x_n} \right)$$

Some algorithms will attempt to estimate the Jacobian numerically if it cannot be provided as a separate function.

Furthermore, some sophisticated optimization algorithms require information about the second derivatives of the function, a symmetric matrix of values called the *Hessian*:

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

$$H(f) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_2 \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_1} \\ \frac{\partial^2 f}{\partial x_1 \partial x_2} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_1 \partial x_n} & \frac{\partial^2 f}{\partial x_2 \partial x_n} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{pmatrix}$$

Just as the Jacobian represents local *gradient* of a function of several variables, the Hessian represents the local *curvature*.

Unconstrained minimization

The general algorithm for the minimization of multivariate scalar functions is `scipy.optimize.minimize`, which takes two mandatory arguments:

```
minimize(fun, x0, ...)
```

The first is a function object, `fun`, for evaluating the function to be minimized: this function should take an array of values, `x`, defining the point at which it is to be evaluated $(x_1, x_2, \cdots, x_n)$ followed by any further arguments it requires. The second required argument, `x0`, is an array of values representing the initial guess for the minimization algorithm to start at.

In this section we will demonstrate the use of `minimize` with *Himmelblau's function*, a simple two-dimensional function with some awkward features that make it a good test-function for optimization algorithms. Himmelblau's function is

$$f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2.$$

The region $-5 \leq x \leq 5$, $-5 \leq y \leq 5$ contains one local maximum,

$$f(-0.270845, -0.923039) = 181.617$$

(though the function climbs steeply outside of this region). There are four minima:

$$\begin{aligned} f(3, 2) &= 0, \\ f(-2.805118, 3.131312) &= 0, \\ f(-3.779310, -3.283186) &= 0, \\ f(3.584428, -1.848126) &= 0. \end{aligned}$$

and four saddle points. Figure 8.17 shows a contour plot of the function.

The function may be defined in Python in the usual way:

```
In [x]: def f(X):
   ...:     x, y = X
   ...:     return (x**2 + y - 11)**2 + (x + y**2 - 7)**2
```

where for clarity we have unpacked the array, `X`, holding (x_1, x_2) into the named values $x_1 \equiv x$ and $x_2 \equiv y$.

To find a minimum, call `minimize` with some initial guess, say $(x, y) = (0, 0)$:

```
In [x]: from scipy.optimize import minimize
In [x]: minimize(f, (0,0))
```

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use , available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

Table 8.4 Minimization information dictonary returned by `scipy.optimize.minimize`

Key	Description
`success`	A boolean value indicating whether or not the minimization was successful
`x`	If successful, the solution: the values of $(x_1, x_2, \cdots, x_n)$ at which the function is a minimum. If the algorithm was not successful, `x` indicates the point at which it gave up
`fun`	If successful, the value of the function at the minimum identified as `x`
`message`	A string describing of the outcome of the minimization
`jac`	The value of the Jacobian: if the minimization is successful the values in this array should be close to zero
`hess, hess_inv`	The Hessian and its inverse (if used)
`nfev, njev, nhev`	The number of evaluations of the function, its Jacobian and its Hessian

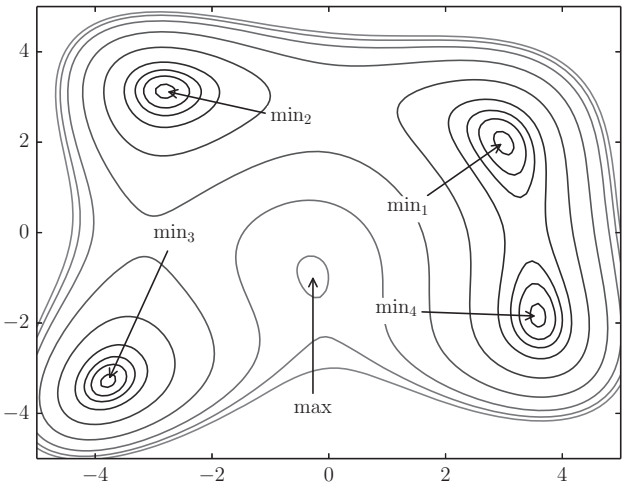


Figure 8.17 Contour plot of Himmelblau's function.

```
     jac: array([ -8.77780211e-06,  -3.52519449e-06])
 message: 'Optimization terminated successfully.'
     fun: 6.15694370233122e-13
    njev: 16
hess_inv: array([[ 0.01575433, -0.00956965],
       [-0.00956965,  0.03491686]])
  status: 0
    nfev: 64
 success: True
       x: array([ 2.99999989,  1.99999996])
```

`minimize` returns a dictionary-like object with information about the minimization. The important fields are described in Table 8.4: if the minimization is successful, the minimum appears as `x` in this object – here we have converged close to the minimum $f(3, 2) = 0$.

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use , available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

Table 8.5 Some of the minimization methods used by `scipy.optimize.minimize`

`method`	Description
`BFGS`	Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm, the default for minimization without constraints or bounds
`Nelder-Mead`	Nelder-Mead algorithm, also known as the downhill simplex or amoeba method. No derivatives are needed
`CG`	Conjugate gradient method
`Powell`	Powell's method (no derivatives are needed with this algorithm)
`dogleg`	Dog-leg trust-region algorithm (unconstrained minimization). Requires the Jacobian and the Hessian (which must be positive-definite)
`TNC`	Truncated Newton algorithm for minimization within bounds
`l-bfgs-b`	Bound-constrained minimization with the L-BFGS-B algorithm
`slsqp`	"Sequential least squares programming" method for minimization with bounds and equality and inequality constraints
`cobyla`	"Constrained optimization by linear approximation" method for constrained minimization

The algorithm to be used by `minimize` is specified by setting its `method` argument to one of the strings given in Table 8.5. The default algorithm, BFGS, is a good general-purpose quasi-Newton method that can approximate the Jacobian if it is not provided and does not use the Hessian. However, it struggles to find the maximum of Himmelblau's function:

```
In [x]: mf = lambda X: -f(X)    # to find the maximum, minimize -f(x,y)
In [x]: minimize(mf, (0,0))
Out[x]:
      jac: array([  1.17853903e+13,   4.57328118e+13])
  message: 'Desired error not necessarily achieved due to precision loss.'
      fun: -2.9978221235736595e+17
     njev: 16
 hess_inv: array([[ 1.03696455, -0.26722678],
       [-0.26722678,  0.0688646 ]])
   status: 2
     nfev: 76
  success: False
        x: array([-14336., -22528.])
```

Starting at $(0, 0)$, the BFGS algorithm has wandered up one of the steep sides of the Himmelblau function (note the size of the Jacobian) and failed to converge. In fact, we need to start quite close to the maximum to succeed:

```
In [x]: minimize(mf, (-0.2,-1))
Out[x]:
      jac: array([  3.81469727e-06,   1.90734863e-06])
  message: 'Optimization terminated successfully.'
      fun: -181.61652152258262
     njev: 8
 hess_inv: array([[ 0.0232834 , -0.00626945],
       [-0.00626945,  0.06137267]])
   status: 0
     nfev: 32
```

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

```
  success: True
        x: array([-0.27084453, -0.92303852])
```

This is, of course, not much help if we don't know in advance where the maximum is! Let's try a different minimization algorithm, starting at our arbitrary guess, $(0, 0)$:

```
In [x]: minimize(mf, (0,0), method='nelder-mead')
Out[x]:
  status: 0
    nfev: 115
 success: True
 message: 'Optimization terminated successfully.'
     fun: -181.61652150549165
     nit: 59
       x: array([-0.27086815, -0.92300745])
```

The Nelder-Mead algorithm is a simplex method that does not need or estimate the derivatives of the function, so it isn't tempted up the steep sides of the function. However, it has taken 115 function evaluations to converge on the local maximum.

As a final example, consider the `dogleg` method, which requires `minimize` to be passed functions evaluating the Jacobian and the Hessian. The necessary derivatives have simple analytical forms for Himmelblau's function:

$$\frac{\partial f}{\partial x} = 4x(x^2 + y - 11) + 2(x + y^2 - 7)$$
$$\frac{\partial f}{\partial y} = 2(x^2 + y - 11) + 4y(x + y^2 - 7)$$
$$\frac{\partial^2 f}{\partial x^2} = 12x^2 + 4y - 42$$
$$\frac{\partial^2 f}{\partial y^2} = 12y^2 + 4x - 26$$
$$\frac{\partial^2 f}{\partial y \partial x} = \frac{\partial^2 f}{\partial x \partial y} = 4x + 4y$$

The Jacobian and Hessian can be coded up as follows:

```
In [x]: def df:
   ...:     x, y = X
   ...:     f1, f2  = x**2 + y - 11, x + y**2 - 7
   ...:     dfdx = 4*x*f1 + 2*f2
   ...:     dfdy = 2*f1 + 4*y*f2
   ...:     return np.array([dfdx, dfdy])
   ...:
In [x]: def ddf:
   ...:     x, y = X
   ...:     d2fdx2 = 12*x**2 + 4*y - 42
   ...:     d2fdy2 = 12*y**2 + 4*x - 26
   ...:     d2fdxdy = 4*(x + y)
   ...:     return np.array([[d2fdx2, d2fdxdy], [d2fdxdy, d2fdy2]])
   ...:
❶ In [x]: mdf = lambda X: -df(X)
   In [x]: mddf = lambda X: -ddf(X)
```

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

❶ Note that as with the function itself, we need to use the negative of the Jacobian and Hessian if we seek the maximum: these are defined as `lambda` functions `mdf` and `mddf`.

```
In [x]: minimize(mf, (0,0), jac=mdf, hess=mddf, method='dogleg')
Out[x]:
     jac: array([ -1.26922473e-10,   1.23685240e-09])
 message: 'Optimization terminated successfully.'
     fun: -181.6165215225827
    hess: array([[ 44.81187272,   4.77553259],
       [  4.77553259,  16.85937624]])
     nit: 4
    njev: 5
       x: array([-0.27084459, -0.92303856])
  status: 0
    nfev: 5
 success: True
    nhev: 4
```

The algorithm has converged successfully on the local maximum in five function evaluations, five Jacobian evaluations and four Hessian evaluations.

◊ Constrained optimization

Sometimes it is necessary to find the maximum or minimum of a function subject to one or more constraints. To use the earlier mentioned function as an example, you may wish for the single minimum of $f(x, y)$ that satisfies $x > 0$, $y > 0$; or the minimum value of the function along the line $x = y$.

The algorithms `l-bfgs-b`, `tnc` and `slsqp` support the `bounds` argument to `minimize`. `bounds` is a sequence of tuples, each giving the `(min, max)` pairs for each variable of the function defining the bounds on that variable to the minimization. If there is no bound in either direction, use `None`.

For example, if we try to find a minimum in $f(x, y)$ starting at $(-\frac{1}{2}, -\frac{1}{2})$ without specifying any bounds, the `slsqp` method converges (just about) on the one at $(-2.805118, 3.131312)$:

```
In [x]: minimize(f, (-0.5,-0.5), method='slsqp')
Out[x]:
     jac: array([-0.00721077,  0.00037714,  0.        ])
 message: 'Optimization terminated successfully.'
     fun: 4.0198760213901536e-07
     nit: 10
    njev: 10
       x: array([-2.80522924,  3.131319  ])
  status: 0
    nfev: 46
 success: True
```

To stay in the quadrant $x < 0$, $y < 0$, set `bounds` with no minimum on x or y and a maximum bound at $x = 0$ and $y = 0$:

```
In [x]: xbounds = (None, 0)
In [x]: ybounds = (None, 0)
In [x]: bounds = (xbounds, ybounds)
In [x]: minimize(f, (-0.5,-0.5), bounds=bounds, method='slsqp')
```

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

```
Out[x]:
     jac: array([-0.00283595, -0.00034243,  0.        ])
 message: 'Optimization terminated successfully.'
     fun: 4.115667606325133e-08
     nit: 11
    njev: 11
       x: array([-3.77933774, -3.28319868])
  status: 0
    nfev: 50
 success: True
```

Suppose we wish to find the extrema of Himmelblau's function that also satisfy the condition $x = y$ (that is, they lie along the diagonal of Figure 8.17). Two of the minimization methods listed in Table 8.5 allow for constraints, `cobyla` and `slsqp`, so we must use one of these.

Constraints are specified as the argument `constraints` to the `minimize` function as a sequence of dictionaries defining string keys `'type'`: the *type* of constraint and `'fun'`: a callable object implementing the constraint. `'type'` may be `'eq'` or `'ineq'` for a constraint based on an equality (such as $x = y$) or an inequality (e.g., $x > 2y - 1$). *Note that* `cobyla` *does not support equality constraints*.

An equality constraint function should return zero if the constraint function is met; an inequality constraint function should return a non-negative value if the inequality is met.

To find the minima in $f(x, y)$ subject to the constraint $x = y$, we can use the `slsqp` method with an equality constraint function returning $x - y$:

```
In [x]: con = {'type': 'eq', 'fun': lambda X: X[0] - X[1]}
In [x]: minimize(f, (0,0), constraints=con, method='slsqp')
     jac: array([-16.33084416,  16.33130538,   0.        ])
 message: 'Optimization terminated successfully.'
     fun: 8.0000000007160867
     nit: 7
    njev: 7
       x: array([ 2.54138438,  2.54138438])
  status: 0
    nfev: 32
 success: True
```

The method converged on one of the minima (there is another: start at, for e.g., $(-2, -2)$ to find it). What about the maximum?

```
In [x]: minimize(mf, (0,0), constraints=con, method='slsqp')
Out[x]:
     jac: array([ 0.,  0.,  0.])
 message: 'Singular matrix C in LSQ subproblem'
     fun: -3.1826053300603689e+68
     nit: 4
    njev: 4
       x: array([ -1.12315113e+17,  -1.12315113e+17])
  status: 6
    nfev: 16
 success: False
```

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

That didn't go so well – the algorithm wandered up the side of a valley. A better choice of algorithm here is `cobyla`, but this method doesn't support equality constraints, so we will build one from a pair of inequalities: $x = y$ if both of $x > y$ and $x < y$ are not satisified:

```
In [x]: con1 = {'type': 'ineq', 'fun': lambda X: X[0] - X[1]}
In [x]: con2 = {'type': 'ineq', 'fun': lambda X: X[1] - X[0]}
In [x]: minimize(mf, (0,0), constraints=(con1, con2), method='cobyla')
Out[x]:
  status: 1
    nfev: 34
 success: True
 message: 'Optimization terminated successfully.'
     fun: -179.12499987327624
   maxcv: 0.0
       x: array([-0.49994148, -0.49994148])
```

Here, the constraint function defined in `con1` returns a non-negative value if $x > y$ and that defined in `con2` returns a non-negative value if $x < y$. The only way both can be satisfied is if $x = y$.

Minimizing a function of one variable

If the function to be minimized is *univariate* (i.e., takes only one variable, a scalar), a faster algorithm is provided by `scipy.optimize.minimize_scalar`. To simply return a minimum, this function can be called with `method='brent'`, which implements Brent's method for locating a minimum.

Ideally, one should "bracket" the minimum first by providing values for x, (a, b, c) such that $f(a) > f(b)$ and $f(c) > f(b)$. This can be done with the `bracket` argument which takes the tuple `(a,b,c)`. If this isn't possible or feasible, provide an interval of two values of x on which to start a search for such a bracket (in the downhill direction). If no `bracket` argument is specified, this search is initiated from the interval $(0, 1)$.

Figure 8.18 gives an example polynomial with two minima and a maximum.

With no `bracket`, `minimize_scalar` converges on the minimum at -2.841 for this function:

```
In [x]: Polynomial = np.polynomial.Polynomial
In [x]: from scipy.optimize import minimize_scalar
In [x]: f = Polynomial( (48., 28., -24., -3., 1.))
In [x]: minimize_scalar(f)
Out[x]:
  fun: -91.32163915433344
 nfev: 11
    x: -2.8410443265958261
  nit: 10
```

If we bracket the other minimum by providing values `(a,b,c)=(3, 4, 6)` which can be seen from Figure 8.18 to satisfy $f(a) > f(b) < f(c)$, the algorithm converges on 4.549:

```
In [x]: minimize_scalar(f, bracket=(3,4,6))
Out[x]:
  fun: -175.45563549487974
```

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use , available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

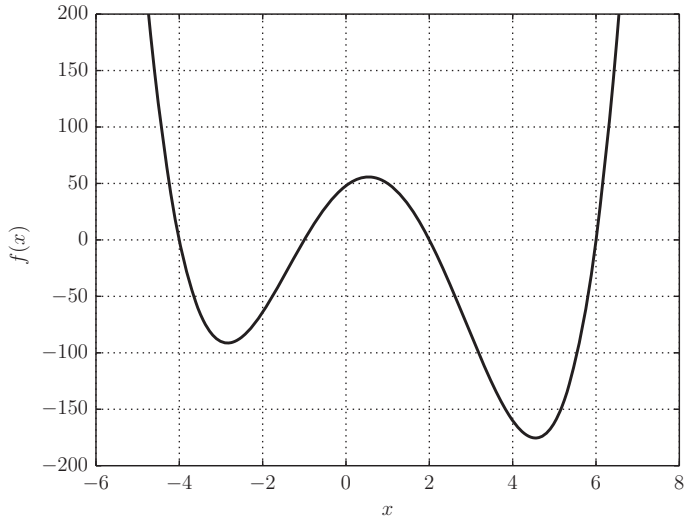


Figure 8.18 The polynomial $f(x) = x^4 - 3x^3 - 24x^2 + 28x + 48$.

```
 nfev: 11
    x: 4.5494683642571934
  nit: 10
```

Finally, to find the maximum, call `minimize_scalar` with $-f(x)$. This time we will initialize a search for a bracket to the minimum of $-f(x)$ with the pair of values $(-1, 0)$:

```
In [x]: minimize_scalar(-f, bracket=(-1, 0))
Out[x]:
  fun: -55.734305899213226
 nfev: 9
    x: 0.54157595897344157
  nit: 8
```

Example E8.21 A simple model for the envelope of an airship treats it as the volume of revolution obtained from a pair of quarter-ellipses joined at their (equal) semi-minor axes. The semi-major axis of the aft ellipse is taken to be longer than that representing the bow by a factor $\alpha = 6$. Equations describing the cross section (in the vertical plane) of the airship envelope may be written

$$y = \begin{cases} \frac{b}{a}\sqrt{x(2a - x)} & (x \le a), \\ \frac{b}{a}\sqrt{a^2 - \frac{(x-a)^2}{\alpha^2}} & (a < x \le \alpha(a + 1)). \end{cases}$$

The drag on the envelope is given by the formula

$$D = \tfrac{1}{2}\rho_{\text{air}} v^2 V^{2/3} C_{\text{DV}},$$

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use , available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

where ρ_{air} is the air density, v the speed of the airship, V the envelope volume and the drag coefficient, C_{DV} is estimated using the following empirical formula:[10]

$$C_{\text{DV}} = \text{Re}^{-1/6}[0.172(l/d)^{1/3} + 0.252(d/l)^{1.2} + 1.032(d/l)^{2.7}].$$

Here, $\text{Re} = \rho_{\text{air}} vl/\mu$ is the Reynold's number and μ the dynamic viscosity of the air. l and d are the airship length and maximum diameter ($= 2b$) respectively.

Suppose we want to minimize the drag with respect to the parameters a and b but fix the total volume of the airship envelope, $V = \frac{2}{3}\pi ab^2(1+\alpha)$. The following program does this using the `slsqp` algorithm, for a volume of 200000 m^3, that of the Hindenburg.

Listing 8.17 Minimizing the drag on an airship envelope

```
# eg8-airship.py
import numpy as np
from scipy.optimize import minimize

# air density (kg.m-3) and dynamic viscosity (Pa.s) at cruise altitude
rho, mu = 1.1, 1.5e-5
# air speed (m.s-1) at cruise altitude
v = 30

def CDV(L, d):
    """ Calculate the drag coefficient. """
    Re = rho * v * L / mu       # Reynold's number
    r = L / d                   # "Fineness" ratio
    return (0.172 * r**(1/3) + 0.252 / r**1.2 + 1.032 / r**2.7) / Re**(1/6)

def D(X):
    """ Return the total drag on the airship envelope. """
    a, b = X
    L = a * (1+alpha)
    return 0.5 * rho * v**2 * V(X)**(2/3) * CDV(L, 2*b)

# Fixed total volume of the airship envelope (m3)
V0 = 2.e5
# Parameter describing the tapering of the stern of the envelope
alpha = 6

def V(X):
    """ Return the volume of the envelope. """
    a, b = X
    return 2 * np.pi * a * b**2 * (1+alpha) / 3

# Minimize the drag, constraining the volume to be equal to V0
a0, b0 = 70, 45     # initial guesses for a, b
con = {'type': 'eq', 'fun': lambda X: V(X)-V0}
res = minimize(D, (a0, b0), method='slsqp', constraints=con)
if res['success']:
    a, b = res['x']
    L, d = a * (1+alpha), 2*b       # length, greatest diameter
```

[10] S. F. Hoerner, *Fluid Dynamic Drag*, Hoerner Fluid Dynamics (1965).

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use , available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

```
    print('Optimum parameters: a = {:g} m, b = {:g} m'.format(a, b))
    print('V = {:g} m3'.format(V(res['x'])))
    print('Drag, D = {:g} N'.format(res['fun']))
    print('Total length, L = {:g} m'.format(L))
    print('Greatest diameter, d = {:g} m'.format(d))
    print('Fineness ratio, L/d = {:g}'.format(L/d))
else:
    # We failed to converge: output the results dictionary
    print('Failed to minimize D!', res, sep='\n')
```

This example is a little contrived, since for fixed α the requirement that V be constant means that a and b are not independent, but a solution is found readily enough:

```
Optimum parameters: a = 32.9301 m, b = 20.3536 m
V = 200000 m3
Drag, D = 20837.6 N
Total length, L = 230.51 m
Greatest diameter, d = 40.7071 m
Fineness ratio, L/d = 5.66266
```

The actual dimensions of the Hindenburg were $l = 245$ m, $d = 41$ m giving the ratio $l/d = 5.98$; so we didn't do too badly.

8.4.2 Nonlinear least squares fitting

SciPy's general *nonlinear* least squares fitting routine is `scipy.optimize.leastsq`, which has the most basic call signature:

`scipy.optimize.leastsq(func, x0, args=()).`

This will attempt to fit a sequence of data points, `y`, to a model function, `f`, which depends on one or more fit parameters. `leastsq` is passed a related function object, `func`, which returns the *difference* between `y` and `f` (the *residuals*). `leastsq` also requires an initial guess for the fitted parameters, `x0`. If `func` requires any other arguments (typically, arrays of the data, `y`, and one or more independent variables), pass them in the sequence `args`. For example, consider fitting the artificial noisy decaying cosine function, $f(t) = Ae^{t/\tau}\cos 2\pi\nu t$ (Figure 8.19).

```
In [x]: import numpy as np
In [x]: import pylab

In [x]: A, freq, tau = 10, 4, 0.5
In [x]: def f(t, A, freq, tau):
   ...:     return A * np.exp(-t/tau) * np.cos(2*np.pi * freq * t)
   ...:
In [x]: tmax, dt = 1, 0.01
In [x]: t = np.arange(0, tmax, dt)
In [x]: yexact = f(t, A, freq, tau)
In [x]: y = yexact + np.random.randn(len(yexact))*2
In [x]: pylab.plot(t, yexact)
In [x]: pylab.plot(t, y)
In [x]: pylab.show()
```

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

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

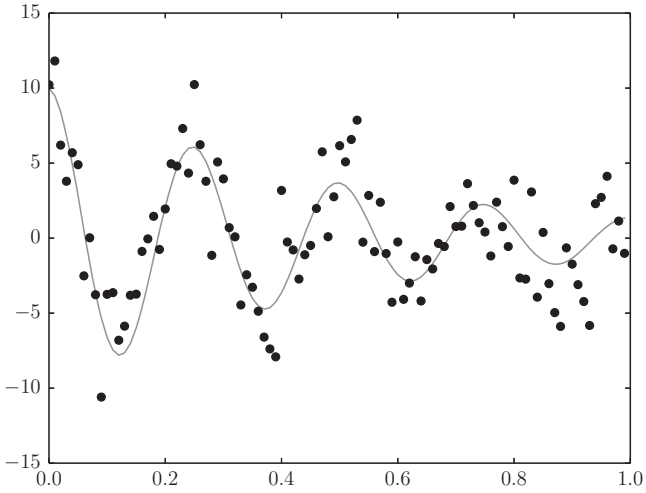


Figure 8.19 A synthetic noisy decaying cosine function.

```
In [x]: 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`:

```
In [x]: from scipy.optimize import leastsq
In [x]: p0 = 5, 5, 1
In [x]: plsq = leastsq(residuals, p0, args=(y, t))
In [x]: plsq[0]
Out[x]: [ 9.33962672  4.04958427  0.48637434]
```

As with SciPy's other optimization routines, `leastsq` can be configured to return more information about its working, but here we report only the solution (best fit parameters), which is always the first item in the `plsq` tuple.

The true values are `A, freq, tau = 10, 4, 0.5`, so given the noise we haven't done badly. Graphically,

```
In [x]: pylab.plot(t, y, 'o', c='k', label='Data')
In [x]: pylab.plot(t,yexact,c='gray', label='Exact')
In [x]: pylab.plot(t,f(t, *pfit),c='k', label='Fit')
In [x]: pylab.legend()
In [x]: pylab.show()
```

The fit is illustrated in Figure 8.20.

If it is known, it is also possible to pass the Jacobian to `leastsq`, as the following example demonstrates.

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use , available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

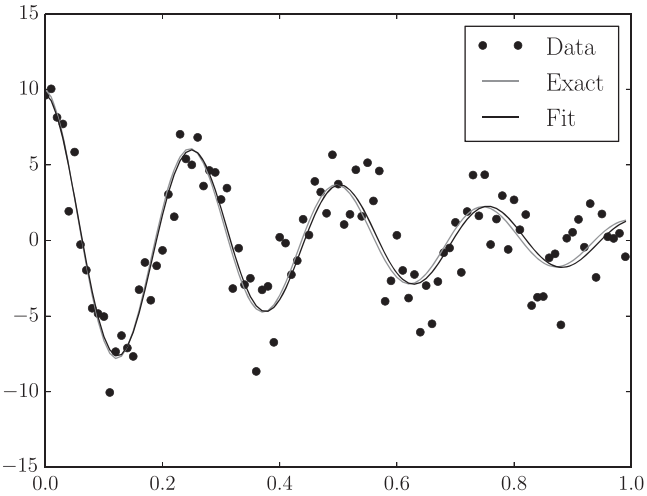


Figure 8.20.

Example E8.22 In this example, we are given a noisy series of data points that we want to fit to an ellipse. The equation for an ellipse may be written as a nonlinear function of angle, θ ($0 \leq \theta < 2\pi$), which depends on the parameters a (the semi-major axis) and e (the eccentricity):

$$r(\theta; a, e) = \frac{a(1-e^2)}{1 - e\cos\theta}.$$

To fit a sequence of data points (θ, r) to this function, we first code it as a Python function taking two arguments: the independent variable, `theta`, and a tuple of the parameters, `p = (a, e)`. The function we wish to minimize is the difference between this model function and the data, `r`, defined as the method `residuals`:

```
def f(theta, p):
    a, e = p
    return a * (1 - e**2)/(1 - e*np.cos(theta))

def residuals(p, r, theta):
    return r - f(theta, p)
```

We also need to give `leastsq` an initial guess for the fit parameters, say `p0 = (1,0.5)`. The simplest call to fit the function would then pass to `leastsq` the objects `residuals`, `p0` and `args=(r, theta)` (the additional arguments needed by the `residuals` function:

```
plsq = leastsq(residuals, p0, args=(r, theta))
```

If at all possible, however, it is better to also provide the Jacobian (the first derivative of the fit function with respect to the parameters to be fitted). Expressions for these are straightforward to calculate and implement:

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

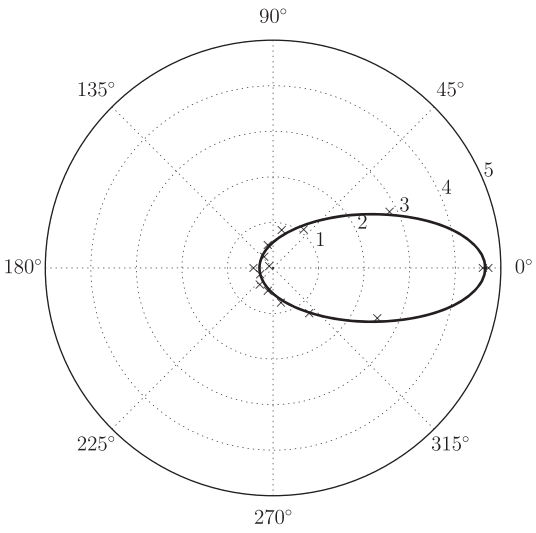


Figure 8.21 Nonlinear least squares fitting of data to the equation of an ellipse in polar coordinates.

$$\frac{\partial f}{\partial a} = \frac{(1-e^2)}{1-e\cos\theta},$$
$$\frac{\partial f}{\partial e} = \frac{a(1-e^2)\cos\theta - 2ae(1-e\cos\theta)}{(1-e\cos\theta)^2}.$$

However, the function we wish to minimize is the residuals function, $r-f$, so we need the negatives of these derivatives. Here is the working code and the fit result (Figure 8.21).

Listing 8.18 Nonlinear least squares fit to an ellipse

```
# eg8-leastsq.py

import numpy as np
from scipy import optimize
import pylab

def f(theta, p):
    a, e = p
    return a * (1 - e**2)/(1 - e*np.cos(theta))

# The data to fit
theta = np.array([0.0000,0.4488,0.8976,1.3464,1.7952,2.2440,2.6928,
                  3.1416,3.5904,4.0392,4.4880,4.9368,5.3856,5.8344,6.2832])
r = np.array([4.6073, 2.8383, 1.0795, 0.8545, 0.5177, 0.3130, 0.0945, 0.4303,
              0.3165, 0.4654, 0.5159, 0.7807, 1.2683, 2.5384, 4.7271])

def residuals(p, r, theta):
    """ Return the observed - calculated residuals using f(theta, p). """
    return r - f(theta, p)
```

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use , available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

```
def jac(p, r, theta):
    """ Calculate and return the Jacobian of residuals. """
    a, e = p
    da = (1 - e**2)/(1 - e*np.cos(theta))
    de = (-2*a*e*(1-e*np.cos(theta)) + a*(1-e**2)*np.cos(theta))/(1 -
                                                        e*np.cos(theta))**2
    return -da,  -de
    return np.array((-da, -de)).T

# Initial guesses for a, e
p0 = (1, 0.5)
plsq = optimize.leastsq(residuals, p0, Dfun=jac, args=(r, theta), col_deriv=True)
print(plsq)

pylab.polar(theta, r, 'x')
theta_grid = np.linspace(0, 2*np.pi, 200)
pylab.polar(theta_grid, f(theta_grid, plsq[0]), lw=2)
pylab.show()
```

SciPy also includes a curve-fitting function, `scipy.optimize.curve_fit`, that can fit data to a function directly (without the need for an additional function to calculate the residuals) and supports weighted least squares fitting. The call signature is

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

where `f` is the function to fit to the data `(xdata, ydata)`. `p0` is the initial guess for the parameters, and `sigma`, if provided, give the weights of the `ydata` values. If `absolute_sigma` is `True`, these are treated as one standard deviation error (that is, *absolute* weights); the default, `absolute_sigma=False`, treats them as *relative* weights.

The `curve_fit` function returns `popt`, the best-fit values of the parameters and `pcov`, the covariance matrix of the parameters.

Example E8.23 To illustrate the use of `curve_fit` in weighted and unweighted least squares fitting, the following program fits the Lorentzian line shape function centered at x_0 with half width at half-maximum (HWHM), γ, amplitude, A:

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

to some artificial noisy data. The fit parameters are A, γ and x_0. The noise is such that a region of the data close to the line center is much noisier than the rest.

Listing 8.19 Weighted and unweighted least squares fitting with `curve_fit`

```
# eg8-curve-fit.py
import numpy as np
from scipy.optimize import curve_fit
import pylab

x0, A, gamma = 12, 3, 5
```

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

```
n = 200
x = np.linspace(1, 20, n)
yexact = A * gamma**2 / (gamma**2 + (x-x0)**2)

# 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)<1] = 3
noise = np.random.randn(n) * sigma
y = yexact + noise

def f(x, x0, A, gamma):
    """ The Lorentzian entered at x0 with amplitude A and HWHM gamma. """
    return A *gamma**2 / (gamma**2 + (x-x0)**2)

def rms(y, yfit):
    return np.sqrt(np.sum((y-yfit)**2))

# 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:'); print(pcov)
print('rms error in fit:', rms(yexact, yfit))
print()

# 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:'); print(pcov2)
print('rms error in fit:', rms(yexact, yfit2))

pylab.plot(x, yexact, label='Exact')
pylab.plot(x, y, 'o', label='Noisy data')
pylab.plot(x, yfit, label='Unweighted fit')
pylab.plot(x, yfit2, label='Weighted fit')
pylab.ylim(-1,4)
pylab.legend(loc='lower center')
pylab.show()
```

As Figure 8.22 shows, the unweighted fit is thrown off by the noisy region. Data in this region are given a lower weight in the weighted fit and so the parameters are closer to their true values and the fit better. The output is

```
Unweighted fit parameters: [ 11.61282984   3.64158981   3.93175714]
Covariance matrix:
[[ 0.0686249  -0.00063262  0.00231442]
 [-0.00063262  0.06031262 -0.07116127]
 [ 0.00231442 -0.07116127  0.16527925]]
rms error in fit: 4.10434012348

Weighted fit parameters: [ 11.90782988   3.0154818    4.7861561 ]
Covariance matrix:
```

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use , available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

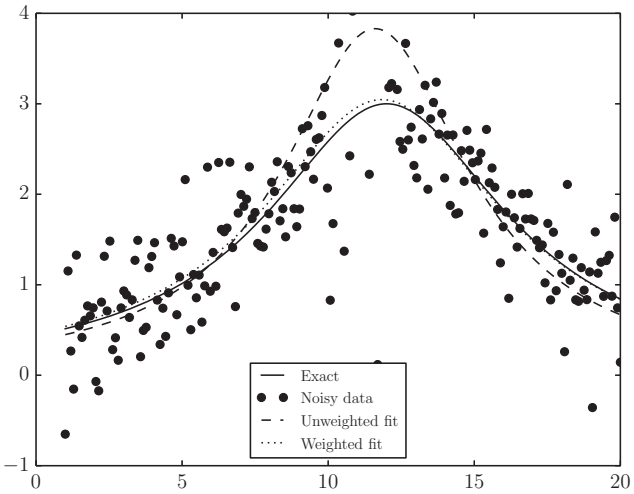


Figure 8.22 Example of least squares fit with `scipy.optimize.curve_fit`.

```
[[ 0.01893474 -0.00333361  0.00639714]
 [-0.00333361  0.01233797 -0.02183039]
 [ 0.00639714 -0.02183039  0.06062533]]
rms error in fit: 0.694013741786
```

8.4.3 Root-finding

`scipy.optimize` provides several methods for obtaining the roots of both univariate and multivariate functions. We describe here only the algorithms relating to functions of a single variable: `brentq`, `brenth`, `ridder` and `bisect`. Each of these methods requires a continuous function, $f(x)$, and a pair of numbers defining a *bracketing interval* for the root to find; that is, values a and b such that the root lies in the interval $[a, b]$ and $f(a) = -f(b)$. Details of the algorithms behind these root-finding methods can be found in standard textbooks on numerical analysis.[11]

In general, the method of choice for finding the root of a well-behaved function is `scipy.optimize.brentq`, which 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 $-1 \leq x \leq 1$:

$$f(x) = \frac{1}{5} + x\cos\left(\frac{3}{x}\right).$$

[11] For example, Press et al., (2007). *Numerical Recipes*, 3rd ed., Cambridge University Press.

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use , available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

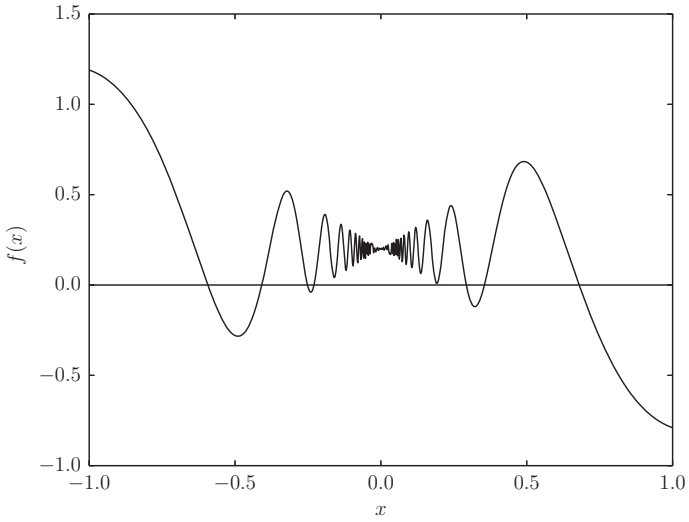


Figure 8.23 The function $f(x) = \frac{1}{5} + x\sin(3/x)$ and its roots.

A plot of this function (Figure 8.23) suggests there is a root between -0.7 and -0.5:

```
In [x]: f = lambda x: 0.2 + x*np.cos(3/x)
In [x]: x = np.linspace(-1, 1, 1000)
In [x]: pylab.plot(x,f(x))
In [x]: pylab.axhline(0, color='k')
In [x]: pylab.show()

In [x]: from scipy.optimize import brentq
In [x]: brentq(f, -0.7, -0.5)
Out[x]: -0.5933306271014237
```

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

Finally, root-finding by the Newton-Raphson algorithm can be very fast (quadratic) 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 method `scipy.optimize.newton` as the argument `fprime` along with a starting point, `x0`, which should (in general) be as near to the root as possible. It is not necessary to bracket the root. If the $f'(x)$ cannot be provided, the secant method is used by `newton`. If you are in the happy position of being able to provide the second derivative, $f''(x)$, as `fprime2` as well as the first, Halley's method (which converges even faster than the basic Newton-Raphson algorithm) is used instead.

Note 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: the result should be verified by evaluating the function at the returned value to check that it is (close to) zero.

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use , available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

Table 8.6 Population data for voles measured by Leslie and Ranson

x /weeks	$m(x)$	$P(x)$
8	0.6504	0.83349
16	2.3939	0.73132
24	2.9727	0.58809
32	2.4662	0.43343
40	1.7043	0.29277
48	1.0815	0.18126
56	0.6683	0.10285
64	0.4286	0.05348
72	0.3000	0.02549

Example E8.24 In ecology, the *Euler-Lotka equation* describes the growth of a population in terms of $P(x)$, the fraction of individuals alive at age x and $m(x)$, the mean number of live females born per time period per female alive during that time period:

$$\sum_{x=\alpha}^{\beta} P(x)m(x)e^{-rx} = 1,$$

where α and β are the boundary ages for reproduction defining the discrete growth rate, $\lambda = e^r$. $r = \ln\lambda$ is known as *Lotka's intrinsic rate of natural increase*.

In a paper by Leslie and Ranson,[12] $P(x)$ and $m(x)$ were measured for a population of voles (*Microtus agrestis*) using a time period of eight weeks. The data are given in Table 8.6.

The sum $R_0 = \sum_{x=\alpha}^{\beta} P(x)m(x)$ gives the ratio between the total number of female births in successive generations; a population grows if $R_0 > 1$ and r determines how fast this growth is. In order to find r, Leslie and Ranson used an approximate numerical method; the code mentioned here determines r by finding the real root of the Lotka-Euler equation directly (it can be shown that there is only one).

Listing 8.20 Solution of the Euler-Lotka equation

```
# eg8-euler-lotka.py
import numpy as np
from scipy.optimize import brentq

# The data, from Table 6 of:
# P. H. Leslie and R. M. Ranson, J. Anim. Ecol. 9, 27 (1940)
x = np.linspace(8, 72, 9)
m = np.array( [0.6504, 2.3939, 2.9727, 2.4662, 1.7043,
               1.0815, 0.6683, 0.4286, 0.3000] )
P = np.array( [0.83349, 0.73132, 0.58809, 0.43343, 0.29277,
               0.18126, 0.10285, 0.05348, 0.02549] )
```

[12] P. H. Leslie and R. M. Ranson, (1940). *J. Anim. Ecol.* **9**, 27.

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use , available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

```
# Calculate the product sequence f and R0, the ratio between the number of
# female births in successive generations.
f = P * m
R0 = np.sum(f)
if R0 > 1:
    msg = 'R0 > 1: population grows'
else:
    msg = 'Population does not grow'

# The Euler-Lotka equation: we seek the one real root in r
def func(r):
    return np.sum(f * np.exp(-r * x)) - 1

# Bracket the root and solve with scipy.optimize.brentq
a, b = 0, 10
r = brentq(func, a, b)
print('R0 = {:.3f} ({})'.format(R0, msg))
print('r = {:.5f} (lambda = {:.5f})'.format(r, np.exp(r)))
```

The output of this program is as follows:

```
R0 = 5.904 (R0 > 1: population grows)
r = 0.08742 (lambda = 1.09135)
```

This value of r may be compared with the approximate value obtained by Leslie and Ranson, who comment:

The required root is 0.087703 which slightly overestimates the value of r, to which the series is approaching. This lies between 0.0861 (the third degree approximation) and 0.0877, but nearer the latter than the former, the error being probably in the last decimal place.

8.4.4 Exercises

Questions

Q8.4.1 Use `scipy.optimize.brentq` to find the solutions to the equation

$$x + 1 = -\frac{1}{(x-3)^3}$$

Q8.4.2 Using `scipy.optimize.newton` to find a root of the following functions (with the given starting point, x_0) fails. Explain why and find the roots either by modifying the call to `newton` or by using a different method.

a.

$$f(x) = x^3 - 5x, \quad x_0 = 1$$

b.

$$f(x) = x^3 - 3x + 1, \quad x_0 = 1$$

c.

$$f(x) = 2 - x^5, \quad x_0 = 0.01$$

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

d.

$$f(x) = x^4 - (4.29)x^2 - 5.29, \quad x_0 = 0.8$$

Q8.4.3 The trajectory of a projectile in the xz-plane launched from the origin at an angle θ_0 with speed $v_0 = 25\ \mathrm{m\,s^{-1}}$ is

$$z = x\tan\theta_0 - \frac{g}{2v_0^2\cos\theta_0}x^2.$$

If the projectile passes through the point $(5, 15)$, use Brent's method to determine the possible values of θ_0.

Problems

P8.4.1 A rectangular field with area $A = 10,000\ \mathrm{m}^3$ is to be fenced-off beside a straight river (the boundary with the river does not need to be fenced). What dimensions a, b minimize the amount of fencing required? Verify that a constrained minimization algorithm gives the same answer as the algebraic analysis.

P8.4.2 Find all of the roots of

$$f(x) = \frac{1}{5} + x\cos\left(\frac{3}{x}\right)$$

using (a) `scipy.optimize.brentq` and (b) `scipy.optimize.newton`.

P8.4.3 The *Wien displacement law* predicts that the wavelength of maximum emission from a black body described by Planck's law is proportional to $1/T$:

$$\lambda_{\max}T = b,$$

where b is a constant known as *Wien's displacement constant*. Given the Planck distribution of emitted energy density as a function of wavelength,

$$u(\lambda, T) = \frac{8\pi^2 hc}{\lambda^5}\frac{1}{e^{hc/\lambda k_\mathrm{B}T} - 1},$$

determine the constant b by using `scipy.optimize.minimize_scalar` to find the maximum in $u(\lambda, T)$ for temperatures in the range $500\ \mathrm{K} \le T \le 6000\ \mathrm{K}$ and fitting $\lambda_{\max}$ to a straight line against $1/T$. Compare with the "exact" value of b, which is available within `scipy.constants` (see Section 8.1.1).

P8.4.4 Consider a one-dimensional quantum mechanical particle in a box ($-1 \le x \le 1$) described by the Schrödinger equation:

$$-\frac{\mathrm{d}^2\psi}{\mathrm{d}x^2} = E\psi,$$

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008

in energy units for which $\hbar^2/(2m) = 1$ with m the mass of the particle. The exact solution for the ground state of this system is given by

$$\psi = \cos\left(\frac{\pi x}{2}\right), \quad E = \frac{\pi^2}{4}.$$

An approximate solution may be arrived at using the *variational principle* by minimizing the expectation value of the energy of a trial wavefunction,

$$\psi_{\text{trial}} = \sum_{n=0}^{N} a_n \phi_n(x)$$

with respect to the coefficients a_n. Taking the basis functions to have the following symmetrized polynomial form,

$$\phi_n = (1-x)^{N-n+1}(x+1)^{n+1},$$

use `scipy.optimize.minimize` and `scipy.integrate.quad` to find the optimum value of the expectation value (Rayleigh-Ritz ratio):

$$\mathcal{E} = \frac{\langle\psi_{\text{trial}}|\hat{H}|\psi_{\text{trial}}\rangle}{\langle\psi_{\text{trial}}|\psi_{\text{trial}}\rangle} = -\frac{\int_{-1}^{1} \psi_{\text{trial}} \frac{\mathrm{d}^2}{\mathrm{d}x^2}\psi_{\text{trial}}\,\mathrm{d}x}{\int_{-1}^{1} \psi_{\text{trial}}\psi_{\text{trial}}\,\mathrm{d}x}.$$

Compare the estimated energy, $\mathcal{E}$, with the exact answer for $N = 1, 2, 3, 4$. (*Hint:* use `np.polynomial.Polynomial` objects to represent the basis and trial wavefunctions.)

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:04:40, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781139871754.008