6 NumPy

NumPy has become the de facto standard package for general scientific programming in Python. Its core object is the `ndarray`, a multidimensional array of a single data type which can be sorted, reshaped, subject to mathematical operations and statistical analysis, written to and read from files, and much more. The NumPy implementations of these mathematical operations and algorithms have two main advantages over the "core" Python objects we have used until now. First, they are implemented as precompiled C code and so approach the speed of execution of a program written in C itself; second, NumPy supports *vectorization*: a single operation can be carried out on an entire array, rather than requiring an explicit loop over the array's elements. For example, compare the multiplication of two one-dimensional lists of `n` numbers, `a` and `b`, in the core python language:

```
c = []
for i in range(n):
    c.append(a[i] * b[i])
```

and using NumPy arrays:[1]

```
c = a * b
```

The elementwise multiplication is handled by optimized, precompiled C and so is very fast (much faster for large `n` than the core Python alternative). The absence of explicit looping and indexing makes the code cleaner, less error-prone and closer to the standard mathematical notation it reflects.

All of NumPy's functionality is provided by the `numpy` package. To use it, it is strongly advised to import with

```
import numpy as np
```

and then to refer to its attributes with the prefix `np.` (e.g., `np.array`). This is the way we use NumPy in this book.

6.1 Basic array methods

The NumPy array class is `ndarray`, which consists of a multidimensional table of elements indexed by a tuple of integers. Unlike Python lists and tuples, the elements

[1] We will use the terms NumPy array and `ndarray` interchangeably

cannot be of different types: each element in a NumPy array has the same type, which is specified by an associated *data type* object (`dtype`). The `dtype` of an array specifies not only the broad class of element (integer, floating point number, etc.) but also how it is represented in memory (e.g., how many bits it occupies) – see Section 6.1.2.

The dimensions of a NumPy array are called *axes*; the number of axes an array has is called its *rank*.

6.1.1 Creating an array

Basic array creation

The simplest way to create a small NumPy array is to call the `np.array` constructor with a list or tuple of values:

```
In [x]: import numpy as np
In [x]: a = np.array( (100, 101, 102, 103) )
In [x]: a
Out[x]: array([100, 101, 102, 103])
In [x]: b = np.array( [[1.,2.], [3.,4.]] )
Out[x]:
array([[ 1.,  2.],
       [ 3.,  4.]])
```

Note that passing a list of lists creates a two-dimensional array (and similarly for higher dimensions).

Indexing a multidimensional NumPy array is a little different from indexing a conventional Python list of lists: instead of `b[i][j]`, refer to the index of the required element as a tuple of integers, `b[i,j]`:

```
In [x]: b[0,1]          # same as b[(0,1)]
Out[x]: 2.0
In [x]: b[1,1] = 0.     # also for assignment
Out[x]:
array([[ 1.,  2.],
       [ 3.,  0.]])
```

The data type is deduced from the type of the elements in the sequence and "upcast" to the most general type if they are of mixed but compatible types:

```
In [x]: np.array( [-1, 0, 2.])  # mixture of int and float: upcast to float
Out[x]: array([-1.,  0.,  2.])
```

You can also explicitly set the data type using the optional `dtype` argument (see Section 6.1.2):

```
In [x]: np.array( [0, 4, -4], dtype=complex)
In [x]: array([ 0.+0.j,  4.+0.j, -4.+0.j])
```

If your array is large or you do not know the element values at the time of creation, there are several methods to declare an array of a particular shape filled with default or arbitrary values. The simplest and fastest, `np.empty`, takes a tuple of the array's shape and creates the array without initializing its elements: the initial element values

are undefined (typically random junk defined from whatever were the contents of the memory that Python allocated for the array).

```
In [x]: np.empty((2,2))
Out[x]:
array([[ -2.31584178e+077,  -1.72723381e-077],
       [  2.15686807e-314,   2.78134366e-309]])
```

There are also helper methods `np.zeros` and `np.ones`, which create an array of the specified shape with elements prefilled with 0 and 1 respectively. `np.empty`, `np.zeros` and `np.ones` also take the optional `dtype` argument.

```
In [x]: np.zeros((3,2))     # default dtype is 'float'
Out[x]:
array([[ 0.,  0.],
       [ 0.,  0.],
       [ 0.,  0.]])
In [x]: np.ones((3,3), dtype=int)
Out[x]:
array([[1, 1, 1],
       [1, 1, 1],
       [1, 1, 1]])
```

If you already have an array and would like to create another with the same shape, `np.empty_like`, `np.zeros_like` and `np.ones_like` will do that for you:

```
In [x]: a
Out[x]: array([100, 101, 102, 103])
In [x]: np.ones_like(a)
Out[x]: array([1, 1, 1, 1])
In [x]: np.zeros_like(a, dtype=float)
Out[x]: array([ 0.,  0.,  0.,  0.])
```

Note that the array created inherits its `dtype` from the original array; to set its data type to something else, use the `dtype` argument.

Initializing an array from a sequence

To create an array containing a sequence of numbers there are two methods: `np.arange` and `np.linspace`. `np.arange` is the NumPy equivalent of `range`, except that it can generate floating point sequences. It also actually allocates the memory for the elements in an `ndarray` instead of returning a generator-like object – compare Section 2.4.3.

```
In [x]: np.arange(7)
Out[x]: array([0, 1, 2, 3, 4, 5, 6])
In [x]: np.arange(1.5, 3., 0.5)
Out[x]: array([ 1.5,  2. ,  2.5]))
```

As with `range` the array generated in these examples does not include the last elements, 7 and 3. However, `arange` has a problem: because of the finite precision of floating point arithmetic it is not always possible to know how many elements will be created. For this reason, and because one often often wants the last element of a specifed sequence, the `np.linspace` function can be a more useful way of creating

an sequence.[2] For example, to generate an evenly spaced array of the five numbers between 1 and 20 *inclusive*:

```
In [x]: np.linspace(1, 20, 5)
Out[x]: array([  1.  ,   5.75,  10.5 ,  15.25,  20.  ])
```

`np.linspace` has a couple of optional boolean arguments. First, setting `retstep` to `True` returns the number spacing (step size):

```
In [x]: x, dx = np.linspace(0., 2*np.pi, 100, retstep=True)
In [x]: dx
Out[x]: 0.06346651825433926
```

This saves you from calculating `dx = (end-start)/(num-1)` separately; in this example, the 100 points between 0 and 2π inclusive are spaced by $2\pi/99 = 0.0634665\cdots$. Finally, setting `endpoint` to `False` omits the final point in the sequence, as for `np.arange`:

```
In [x]: x = np.linspace(0, 5, 5, endpoint=False)
Out[x]: array([ 0.,  1.,  2.,  3.,  4.])
```

Note that the array generated by `np.linspace` has the `dtype` of floating point numbers, even if the sequence generates integers.

Initializing an array from a function

To create an array initialized with values calculated using a function, use NumPy's `np.fromfunction` method, which takes as its arguments a function and a tuple representing the shape of the desired array. The function should itself take the same number of arguments as dimensions in the array: these arguments index each element at which the function returns a value. An example will make this clearer:

```
In [x]: def f(i, j):
   ...:     return 2 * i * j
   ...:
In [x]: np.fromfunction(f,(4,3))
array([[  0.,   0.,   0.],
       [  0.,   2.,   4.],
       [  0.,   4.,   8.],
       [  0.,   6.,  12.]])
```

The function `f` is called for every index in the specified shape and the values it returns are used to initialize the corresponding elements.[3] A simple expression like this one can be replaced by an anonymous `lambda` function (see Section 4.3.3) if desired:

```
In [x]: np.fromfunction(lambda i,j: 2*i*j, (4,3))
```

Example E6.1 To create a "comb" of values in an array of length N for which every nth element is one but with zeros everywhere else:

```
In [x]: N, n = 101, 5
In [x]: def f(i):
```

[2] We came across `linspace` in Example E3.1.

[3] Note that the indexes are passed as `ndarrays` and expect the function, `f`, to use vectorized operations.

```
   ...:     return (i % n == 0) * 1
   ...:
In [x]: comb = np.fromfunction(f, (N,), dtype=int)
In [x]: print(comb)
[1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1
 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0
 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 1]
```

`ndarray` attributes for introspection

A NumPy array knows its rank, shape, size, `dtype` and one or two other properties: these can be determined directly from the attributes described in Table 6.1. For example,

```
In [x]: a = np.array(((1, 0, 1), (0, 1, 0)))
In [x]: a.shape
Out[x]: (2, 3)       # 2 rows, 3 columns
In [x]: a.ndim       # rank (number of dimensions)
Out[x]: 2
In [x]: a.size       # total number of elements
Out[x]: 6
In [x]: a.dtype
Out[x]: dtype('int64')
In [x]: a.data
Out[x]: <memory at 0x102387308>
```

The `shape` attribute returns the axis dimensions in the same order as the axes are indexed: a two-dimensional array with `n` rows and `m` columns has a `shape` of `(n,m)`.

6.1.2 NumPy's basic data types (`dtype`s)

So far, the NumPy arrays we have created have contained either integers or floating point numbers, and we have let Python take care of the details of how these are represented. However, NumPy provides a powerful way of determining these details explicitly using *data type* objects. This is necessary, because in order to interface with the underlying compiled C code the elements of a NumPy array must be stored in a compatible

Table 6.1 `ndarray` Attributes

Attribute	Description
`shape`	The array dimensions: the size of the array along each of its axes, returned as a tuple of integers
`ndim`	Number of axes (dimensions). Note that `ndim == len(shape)`
`size`	The total number of elements in the array, equal to the product of the elements of `shape`
`dtype`	The array's data type (see Section 6.1.2)
`data`	The "buffer" in memory containing the actual elements of the array
`itemsize`	The size in bytes of each element

Table 6.2 Commmon NumPy data types

Data Type	Description
`int_`	The default integer type, corresponding to C's `long`: *platform-dependent*
`int8`	Integer in a single byte: -128 to 127
`int16`	Integer in 2 bytes: -32768 to 32767
`int32`	Integer in 4 bytes: -2147483648 to 2147483647
`int64`	Integer in 8 bytes: -2^{63} to $2^{63}-1$
`uint8`	Unsigned integer in a single byte: 0 to 255
`uint16`	Unsigned integer in 2 bytes: 0 to 65535
`uint32`	Unsigned integer in 4 bytes: 0 to 4294967295
`uint64`	Unsigned integer in 8 bytes: 0 to $2^{64}-1$
`float_`	The default floatng point number type, another name for `float64`
`float32`	Single-precision, signed float: $\sim 10^{-38}$ to $\sim 10^{38}$ with ~ 7 decimal digits of precision
`float64`	Double-precision, signed float: $\sim 10^{-308}$ to $\sim 10^{308}$ with ~ 15 decimal digits of precision
`complex_`	The default complex number type, another name for `complex128`
`complex64`	Single-precision complex number (represented by 32-bit floating point real and imaginary components)
`complex128`	Double-precision complex number (represented by 64-bit floating point real and imaginary components)
`bool_`	The default boolean type represented by a single byte

format: that is, each element is represented in a fixed number of bytes that are interpreted in a particular way.

For example, consider an *unsigned* integer stored in 2 bytes (16 bits) of memory (the C-type `uint16_t`). Such a number can take a value between 0 and $2^{16}-1=65535$. No equivalent native Python type exists for this exact representation: Python integers are signed quantities and memory is dynamically assigned for them as required by their size. So NumPy defines a data type object, `np.uint16` to describe data stored in this way.

Furthermore, different systems can order the two bytes of this number differently, a distinction known as *endianness*. The *big-endian* convention places the most-significant byte in the smallest memory address; the *little-endian* convention places the least-significant byte in the smallest memory address. In creating your own arrays, NumPy will use the default convention for the hardware your program is running on, but it is essential to set the endianness correctly if reading in a binary file generated by a different computer.

Table 6.3 Common NumPy data type strings

String	Description
`i`	Signed integer
`u`	Unsigned integer
`f`	Floating point number[a]
`c`	Complex floating point number
`b`	Boolean value
`S`, `a`	String (fixed-length sequence of characters)
`U`	Unicode

[a] Note that without specifying the byte size, setting `dtype='f'` creates a *single-precision* floating point data type, equivalent to `np.float32`.

A full list of the numerical data types[4] is given in the NumPy documentation,[5] but the more common ones are listed in Table 6.2. They all exist within the `numpy` package and so can be referred to as, for example, `np.uint16`. The data types that get created by default when using the native Python numerical types are those with a trailing underscore: `np.float_`, `np.complex_` and `np.bool_`.

Apparently higher-precision floating point number data types such as `float96`, `float128` and `longdouble` are available but are not to be trusted: their implementation is platform dependent, and on many systems they do not actually offer any extra precision but simply align array elements on the appropriate byte-boundaries in memory.

To create a NumPy array of values using a particular data type, use the `dtype` argument of any array constructor function (such as `np.array`, `np.zeros`, etc.). This argument takes either a data type object (such as `np.uint8`) or something that can be converted into one. It is common to specify the `dtype` using a string consisting of a letter indicating the broad category of data type (integer, unsigned integer, complex number, etc.) optionally followed by a number giving the byte size of the type. For example,

```
In [x]: b = np.zeros((3,3), dtype='u4')
```

creates a 3×3 array of unsigned, 32-bit (4-byte) integers (equivalent to `np.uint32`). A list of supported data type letters and their meanings is given in Table 6.3.

To specify the endianness, use the prefixes `>` (big-endian), `<` (little-endian) or `|` (endianness not relevant). For example,

```
In [x]: a = np.zeros((3,3), dtype='>f8')
In [x]: b = np.zeros((3,3), dtype='<f')
In [x]: c = np.empty((3,3), dtype='|S4')
```

create arrays of big-endian double-precision numbers, little-endian single-precision numbers and four-character strings respecively.

[4] Strictly speaking, these types are *array scalar types* and not `dtypes`, but for our use here the distinction is not important.

[5] http://docs.scipy.org/doc/numpy/user/basics.types.html.

In these examples we have passed a *typecode* string to an array constructor's `dtype` argument, but it is also possible to create a `dtype` object first and pass that instead:

```
In [x]: dt = np.dtype('f8')
In [x]: dt
dtype('float64')    # i.e. 8 bytes, double-precision floating point
In [x]: a = np.array([0., 1., -2.], dtype=dt)
```

`dtype` objects have a handful of useful introspection methods:

```
In [x]: dt.str      # a string identifying the data type
'<f8'
In [x]: dt.name     # data type name and bit-width
'float64'
In [x]: dt.itemsize # data type size in bytes
8
```

To copy an array to a new array with a different data type, pass the desired `dtype` or typecode to the `astype` method:

```
In [x]: a = np.array([1.2345678, 2.5, 3.9])
In [x]: a.astype('float32')     # cast to single-precision float
Out[x]: array([ 1.23456776,  2.5       ,  3.9000001 ], dtype=float32)
In [x]: a.astype(np.uint8)      # cast to unsigned, 1-byte integer
Out[x]: array([1, 2, 3], dtype=uint8)
```

Strings in NumPy arrays are *byte strings* of a fixed size: each "character" is represented by a single byte, in contrast to the variable size UTF-8 encoding commonly used to represent Unicode strings. This is necessary because NumPy arrays have a predefined, fixed size in which all the elements occupy the same amount of memory so that they can be indexed efficiently with a constant stride. Unicode strings encoded with UTF-8, however, represent characters as code points with a variable width (see Section 2.3.3). Of course, any string is ultimately stored as a sequence of bytes and Python provides methods for translating between encodings. For example, on a system encoding strings with UTF-8 by default:

```
In [x]: s = 'piñata'        # UTF-8 encoded Unicode string
In [x]: b = s.encode()
In [x]: b
b'pi\xc3\xb1ata'            # byte string: ñ is stored in two bytes: hex C3B1
In [x]: len(s), len(b)
(6,7)                       # 6 UTF-8 encoded characters stored in 7 bytes
In [x]: arr = np.empty((2,2), 'S7')
In [x]: arr[:] = b          # Store the byte string b in array arr
In [x]:
array([[b'pi\xc3\xb1ata', b'pi\xc3\xb1ata'],
       [b'pi\xc3\xb1ata', b'pi\xc3\xb1ata']],
      dtype='|S7')
In [x]: arr[0,0]            # returns the byte string
b'pi\xc3\xb1ata'
In [x]: arr[0,0].decode()   # decode the byte string back assuming UTF-8
'piñata'
```

6.1.3 Universal functions (ufuncs)

In addition to the basic arithmetic operations of addition, division and more, NumPy provides many of the familiar mathematical functions that the `math` module (Section 2.2.2) does, implemented as so-called *universal functions* that act on each element of an array, producing an array in return without the need for an explicit loop. Universal functions are the way NumPy allows for *vectorization*, which promotes clean, efficient and easy-to-maintain code. For example,

```
In [x]: x = np.linspace(1,5,5)
In [x]: x**2
Out[x]: array([  1.,   4.,   9.,  16.,  25.])
In [x]: x - 1
Out[x]: array([ 0.,  1.,  2.,  3.,  4.])
In [x]: np.sqrt(x - 1)
Out[x]: array([ 0.,  1.,  1.41421356,  1.73205081,  2.])
In [x]: y = np.exp(-np.linspace(0., 2., 5))
In [x]: np.sin(x - y)
Out[x]: array([ 0.,  0.98431873,  0.48771645, -0.59340065, -0.98842844])
```

Array multiplication occurs *elementwise*: matrix multiplication is implemented by NumPy's `dot` function (or using `matrix` objects, see Section 6.6):

```
In [x]: a = np.array( ((1,2), (3,4)) )
In [x]: b = a
In [x]: a * b       # elementwise multiplication
Out[x]:
array([[ 1,  4],
       [ 9, 16]])
In [x]: a.dot(b)    # or np.dot(a, b)
Out[x]:
array([[ 7, 10],
       [15, 22]])
```

Comparison and logic operators (`~`, `&` and `|` for *not*, *and* and *or* respectively) are also vectorized and result in arrays of boolean values:

```
In [x]: a = np.linspace(1,6,6)**3
In [x]: print(a)
[   1.    8.   27.   64.  125.  216.]
In [x]: print(a > 100)
[False False False False  True  True]
In [x]: print((a < 10) | (a > 100))
[ True  True False False  True  True]
```

6.1.4 NumPy's special values, `nan` and `inf`

NumPy defines two special values to represent the outcome of calculations, which are not mathematically defined or not finite. The value `np.nan` ("not a number," NaN) represents the outcome of a calculation that is not a well-defined mathematical operation (e.g., 0/0); `np.inf` represents infinity.[6] For example,

[6] These quantities are defined in accordance with the IEEE 754 standard for floating point numbers.

```
In [x]: a = np.arange(4)
In [x]: a /= 0       # [0/0 1/0 2/0 3/0]
In [x]: a
Out[x]: array([ nan,  inf,  inf,  inf])
```

Do not test nans for equality (`np.nan == np.nan` is `False`). Instead, NumPy provides methods `np.isnan`, `np.isinf` and `np.isfinite`:

```
In [x]: np.isnan(a)
Out[x]: array([ True, False, False, False], dtype=bool)
In [x]: np.isinf(a)
Out[x]: array([False,  True,  True,  True], dtype=bool)
In [x]: np.isfinite(a)
Out[x]: array([False, False, False, False], dtype=bool)
```

Note that `nan` is neither finite nor infinite! (See also Section 9.1.4.)

Example E6.2 A *magic square* is an $N \times N$ grid of numbers in which the entries in each row, column and main diagonal sum to the same number (equal to $N(N^2 + 1)/2$). A method for constructing a magic square for odd N is as follows:

Step 1. Start in the middle of the top row, and let $n = 1$;

Step 2. Insert n into the current grid position;

Step 3. If $n = N^2$ the grid is complete so stop. Otherwise, increment n;

Step 4. Move diagonally up and right, wrapping to the first column or last row if the move leads outside the grid. If this cell is already filled, move vertically down one space instead;

Step 5. Return to step 2.

The following program creates and displays a magic square.

Listing 6.1 Creating a magic square

```
# Create an N x N magic square. N must be odd.
import numpy as np

N  = 5
magic_square = np.zeros((N,N), dtype=int)

n = 1
i, j = 0, N//2

while n <= N**2:
    magic_square[i, j] = n
    n += 1
    newi, newj = (i-1) % N, (j+1)% N
    if magic_square[newi, newj]:
        i += 1
    else:
        i, j = newi, newj

print(magic_square)
```

The 5×5 magic square output by the earlier example is

```
[[17 24  1  8 15]
 [23  5  7 14 16]
 [ 4  6 13 20 22]
 [10 12 19 21  3]
 [11 18 25  2  9]]
```

6.1.5 Changing the shape of an array

Whatever the rank of an array, its elements are stored in sequential memory locations that are addressed by a single index (internally, the array is one-dimensional, but knowing the shape of the array, Python is able to resolve a tuple of indexes into a single memory address). NumPy's arrays are stored in memory in C-style, *row-major* order, that is, with the elements of the last (rightmost) index stored contiguously. In a two-dimensional array, for example, the element `a[0,0]` is followed by `a[0,1]`. The array that follows

```
In [x]: a = np.array( ((1,2),(3,4)) )
In [x}: print(a)
[[1 2]
 [3 4]]
```

is stored in memory as the sequential elements `[1,2,3,4]`.[7]

`flatten` and `ravel`

Suppose you wish to "flatten" a multidimensional array onto a single axis. NumPy provides two methods to do this: `flatten` and `ravel`. Both flatten the array into its internal (row-major) ordering, as described earlier. `flatten` returns an independent *copy* of the elements and is generally slower than `ravel` which, tries to return a *view* to the flattened array. An array view is a new NumPy array with, in this case, a different shape from the original, but it does not "own" its data elements: it references the elements of another array. Thus, just as with mutable lists (Section 2.4.1), a reassignment of an element of one array affects the other. An example should make this clear:

```
In [x]: a = np.array( [[1,2,3], [4,5,6], [7,8,9]] )
In [x]: b = a.flatten()     # create and independent, flattened copy of a
In [x]: b
Out[x]: array([1, 2, 3, 4, 5, 6, 7, 8, 9])
In [x]: b[3] = 0
In [x]: b
Out[x]: array([1, 2, 3, 0, 5, 6, 7, 8, 9])
In [x]: a       # a is unchanged
Out[x]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

[7] This contrasts with Fortran's *column-major* ordering, which would store the elements as `[1,3,2,4]`.

Assignment to `b` didn't change `a` because they are completely independent objects that do not share their data. In contrast, the flattened array created by taking a view on `a` with `ravel` refers to the same underlying data:

```
In [x]: c = a.ravel()
In [x]: c
Out[x]:array([1, 2, 3, 4, 5, 6, 7, 8, 9])
In [x]: c[3] = 0
In [x]: c
Out[x]: array([1, 2, 3, 0, 5, 6, 7, 8, 9])
In [x]: a
Out[x]:
array([[1, 2, 3],
       [0, 5, 6],
       [7, 8, 9]])
```

You should be aware that although the `ravel` method "does its best" to return a view to the underlying data, various array operations (including *slicing*; see Section 6.1.6) can leave the elements stored in noncontiguous memory locations in which case `ravel` has no choice but to make a copy.

`resize` and `reshape`

An array may be resized (in place) to a compatible shape[8] with the `resize` method, which takes the new dimensions as its arguments. If the array doesn't reference another array's data and doesn't have references to it, resizing to a smaller shape is allowed and truncates the array; resizing to a larger shape pads with zeros. Array references are created when, for example, one array is a view on another (they share data) or simply by assignment: (`b=a`).

```
In [x]: a = np.linspace(1, 4, 4)     # the array [1. 2. 3. 4.]
Out[x]: print(a)
[1. 2. 3. 4.]
In [x]: a.resize(2,2)   # reshapes a in place, doesn't return anything
In [x]: print(a)
[[ 1.  2.]
 [ 3.  4.]]
In [x]: a.resize(3,2)   # OK: nothing else references a
In [x]: print(a)
[[ 1.  2.]
 [ 3.  4.]
 [ 0.  0.]]
```

The `reshape` method returns a view on the array with its elements reshaped as required. The original array is not modified.

```
In [x]: a = np.linspace(1, 4, 4)
In [x]: a.resize(3,2)
In [x]: a
[[ 1.  2.]
 [ 3.  4.]
 [ 0.  0.]]
```

[8] That is, a shape with the same total number of elements.

```
In [x]: b = a.reshape(6)
In [x]: print(b)
[ 1.  2.  3.  4.  0.  0.]
In [x]: b.resize(3,2)   # OK: same number of elements
In [x]: b.resize(2,2)   # not OK: b is a view on (shares) the same data as a
...

ValueError: cannot resize this array: it does not own its data
In [x]: a.resize(2,2)   # also not OK: a shares its data with b
ValueError: cannot resize this array: it does not own its data
```

Transposing an array

The method `transpose` returns a view of an array with the axes transposed. For a two-dimensional array, this is the usual matrix transpose:

```
In [x]: a = np.linspace(1,6,6).reshape(3,2)
In [x]: a
Out[x]:
array([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.]])
In [x]: a.transpose()           # or simply a.T
Out[x]:
array([[ 1.,  3.,  5.],
       [ 2.,  4.,  6.]]
```

Note that transposing a one-dimensional array returns the array unchanged:

```
In [x]: b = np.array([100, 101, 102, 103])
In [x]: b.transpose()
Out[x]: array([100, 101, 102, 103])
```

The `np.matrix` object has methods for converting between column and row vectors if this is what you want; see also Section 6.1.6.

Merging and splitting arrays

A clutch of NumPy methods merge and split arrays in different ways. `np.vstack`, `np.hstack` and `np.dstack` stack arrays vertically (in sequential rows), horizontally (in sequential columns) and depthwise (along a third axis). For example,

```
In [x]: a = np.array([0, 0, 0, 0])
In [x]: b = np.array([1, 1, 1, 1])
In [x]: c = np.array([2, 2, 2, 2])
In [x]: np.vstack((a,b,c))
Out[x]:
array([[0, 0, 0, 0],
       [1, 1, 1, 1],
       [2, 2, 2, 2]])
In [x]: np.hstack((a,b,c))
Out[x]:
array([0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2])
In [x]: np.dstack((a,b,c))
Out[x]:
array([[[0, 1, 2],
```

```
        [0, 1, 2],
        [0, 1, 2],
        [0, 1, 2]]])
```

Note that the array created contains an independent *copy* of the data from the original arrays.[9]

The inverse operations, `np.vsplit`, `np.hsplit` and `np.dsplit` split a single array into multiple arrays by rows, columns or depth. In addition to the array to be split, these methods require an argument indicating how to split the array. If this argument is a *single integer*, the array is split into that number of equal-sized arrays along the appropriate axis. For example,

```
In [x]: a = np.arange(6)
In [x]: a
Out[x]: array([ 0,  1,  2,  3,  4,  5])
In [x]: np.hsplit(a, 3)
Out[x]: [array([ 0,  1]), array([ 2,  3]), array([ 4,  5])]
```

– a list of array objects is returned. If the second argument is a sequence of integer indexes, the array is split on those indexes:

```
In [x]: a
Out[x]: array([ 0,  1,  2,  3,  4,  5])
In [x]: np.hsplit(a, (2, 3, 5))
[array([0, 1]), array([2]), array([3, 4]), array([5])]
```

– this is the same as the list `[a[:2], a[2:3], a[3:5], a[5:]]`. Unlike with `np.hstack`, etc., the arrays returned are *views* on the original data.[10]

Example E6.3 Suppose you have a 3×3 array to which you wish to add a row or column. Adding a row is easy with `np.vstack`:

```
In [x]: a = np.ones((3, 3))
In [x]: np.vstack( (a, np.array((2,2,2))) )
Out[x]:
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 2.,  2.,  2.]])
```

Adding a column requires a bit more work, however. You can't use `np.hstack` directly:

```
In [x]: a = np.ones((3, 3))
In [x]: np.hstack( (a, np.array((2,2,2))) )

... [Traceback information] ...
ValueError: all the input arrays must have same number of dimensions
```

---

[9] NumPy has to copy the data because it has to store its data in one contiguous block of memory and the original arrays may be dispersed in different noncontiguous locations.

[10] NumPy does this for efficiency reasons – copying large amounts of data is expensive and not necessary to fulfill the function of these splitting methods.

This is because `np.hstack` cannot concatenate two arrays with different numbers of rows. Schematically:

```
[[ 1.,  1.,  1.],       [2., 2., 2.]
 [ 1.,  1.,  1.],   +                  = ?
 [ 1.,  1.,  1.]]
```

We can't simply transpose our new row, either, because it's a one-dimensional array and its transpose is the same shape as the original. So we need to *reshape* it first:

```
In [x]: a = np.ones((3, 3))
In [x]: b = np.array((2,2,2)).reshape(3,1)
In [x]: b
array([[2],
       [2],
       [2]])
In [x]: np.hstack((a, b))
Out[x]:
array([[ 1.,  1.,  1.,  2.],
       [ 1.,  1.,  1.,  2.],
       [ 1.,  1.,  1.,  2.]])
```

6.1.6 Indexing and slicing an array

An array is indexed by a tuple of integers, and as for Python sequences negative indexes count from the end of the axis. Slicing and striding is supported in the same way as well. For one-dimensional arrays there is only one index:

```
In [x]: a = np.linspace(1,6,6)
In [x]: print(a)
[ 1.  2.  3.  4.  5.  6.]
In [x]: a[1:4:2]    # elements a[1] and a[3] (a stride of 2)
Out[x]: array([ 2.,  4.])
In [x]: a[3::-2]    # elements a[3] and a[1] (a stride of -2)
Out[x]: array([ 4.,  2.]
```

Multidimensional arrays have an index for each axis. If you want to select every item along a particular axis, replace its index with a single colon:

```
In [x]: a = np.linspace(1,12,12).reshape(4,3)
In [x]: a
Out[x]:
array([[  1.,   2.,   3.],
       [  4.,   5.,   6.],
       [  7.,   8.,   9.],
       [ 10.,  11.,  12.]])
In [x]: a[3, 1]
Out[x]: 11.0
In [x]: a[2, :]         # everything in the third row
Out[x]:
array([ 7.,  8.,  9.])
In [x]: a[:, 1]         # everything in the second column
Out[x]: array([  2.,   5.,   8.,  11.])
In [x]: a[1:-1, 1:]     # middle rows, second column onwards
Out[x]:
```

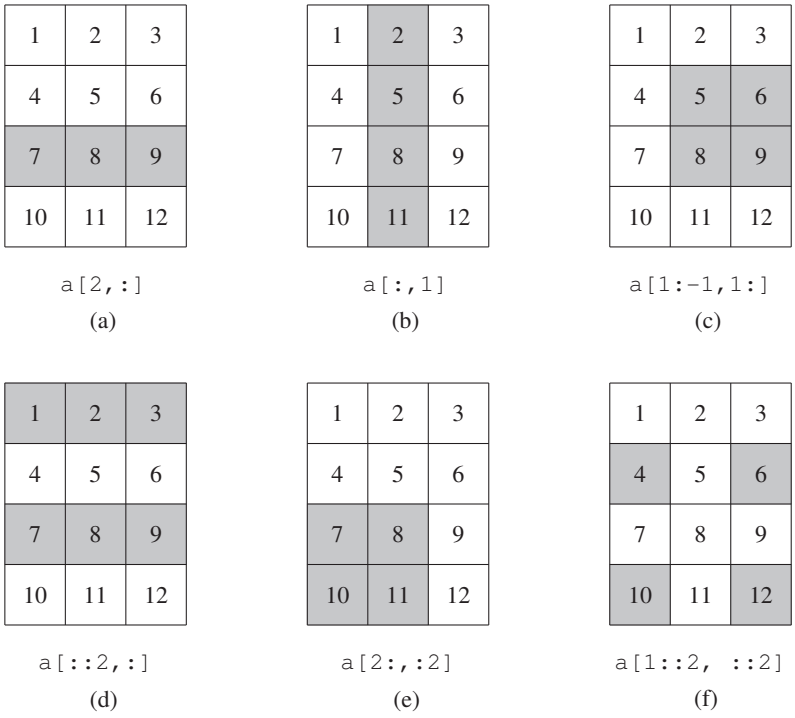


Figure 6.1 Various ways to slice a NumPy array.

```
array([[ 5.,  6.],
       [ 8.,  9.]])
```

These and further examples of NumPy array slicing are illustrated in Figure 6.1.

The special *ellipsis* notation (`...`) is useful for high-rank arrays: in an index, it represents as many colons as are necessary to represent the remaining axes. For example, for a four-dimensional array, `a[3, 1, ...]` is equivalent to `a[3, 1, :, :]` and `a[3,...,1]` is equivalent to `a[3, :, :, 1]`.

The colon and ellipsis syntax also works for assignment:

```
In [x]: a[:,1] = 0      # set all elements in the second column to zero
In [x]: print(a)
[[  1.   0.   3.]
 [  4.   0.   6.]
 [  7.   0.   9.]
 [ 10.   0.  12.]]
```

Advanced indexing

NumPy arrays can also be indexed by sequences that aren't simple tuples of integers, including other lists, arrays of integers and tuples of tuples. Such "advanced indexing" creates a new array with its own copy of the data, rather than a view:

```
In [x]: a = np.linspace(0.,0.5,6)
In [x]: print(a)
```

```
[ 0.   0.1  0.2  0.3  0.4  0.5]
In [x]: ia = [1, 4, 5]       # a list of indexes
In [x]: print(a[ia])
[ 0.1  0.4  0.5]
In [x]: ia = np.array( ((1,2), (3,4)) )
In [x]: print(a[ia])     # an array to be formed from the specified indexes
[[ 0.1  0.2]
 [ 0.3  0.4]]
```

One can even index a multidimensional array with multidimensional arrays of indexes, picking off individual elements at will to build an array of a specified shape. This can lead to some rather baroque code:

```
In [x]: a = np.linspace(1,12,12).reshape(4,3)
In [x]: print(a)
[[  1.   2.   3.]
 [  4.   5.   6.]
 [  7.   8.   9.]
 [ 10.  11.  12.]]
In [x]: ia = np.array( ((1,0),(2,1)) )
In [x]: ja = np.array( ((0,1),(1,2)) )
In [x]: print(a[ia,ja])
[[ 4.  2.]
 [ 8.  6.]]
```

Here we build a 2×2 array (the shape of the index arrays) whose elements are `a[1,0]`, `a[0,1]` on the top row and `a[2,1]`, `a[1,2]` on the bottom row.

Instead of indexing an array with a sequence of integers, it is also possible to use an array of boolean values. The `True` elements of this indexing array identify elements in the target array to be returned:

```
In [x]: a = np.array([-2,-1,0,1,2])
In [x]: ia = np.array([False, True, False, True, True])
In [x]: print(a[ia])
[-1  1  2]
```

Because comparisons are vectorized across arrays just like mathematical operations, this leads to some useful shortcuts:

```
In [x]: print(a)
[-2 -1  0  1  2]
In [x]: ib = a < 0
In [x]: print(ib)
[ True  True False False False]
In [x]: a[ib] = 0   # set all negative elements to zero
In [x]: print(a)
[0 0 0 1 2]
```

It is not actually necessary to store the intermediate boolean array, `ib`, and `a[a<0]=0` does the same job:

```
In [x]: a = np.array([-2,-1,0,1,2])
In [x]: a[a<0]=0
In [x]: print(a)
[0 0 0 1 2]
```

The boolean operations *not*, *and* and *or* are implemented on boolean arrays with the operators ~, & and | respectively. For example,

```
In [x]: years = array([1900, 1904, 1990, 1993, 2000, 2014, 2016, 2100])
In [x]: leap_year = (years % 400 == 0) | (years % 4 == 0) & ~(years % 100 == 0)
In [x]: print(list(zip(years, leap_year)))
Out[x]: [(1900, False), (1904, True), (1990, False), (1993, False),
         (2000, True), (2014, False), (2016, True), (2100, False)]
```

Adding an axis

To add an axis (i.e., dimension) to an array, insert `np.newaxis` in the desired position:

```
In [x]: a = np.linspace(1, 4, 4).reshape(2, 2)
In [x]: print(a)     # a 2x2 array (rank=2)
[[ 1.  2.]
 [ 3.  4.]]
In [x]: a.shape()
(2, 2)
In [x]: b = a[:, np.newaxis, :]
In [x]: print(b)     # a 2x1x2 array (rank=3)
[[[ 1.  2.]]

 [[ 3.  4.]]]
In [x]: b.shape
(2, 1, 2)
```

In fact, `np.newaxis` is the `None` object, so `None` can be used directly in its place if desired.

Example E6.4 A *Sudoku* square consists of a 9 × 9 grid with entries such that each row, column and each of the 9 nonoverlapping 3 × 3 tiles contains the numbers 1–9 once only. The following program verifies that a provided grid is a valid Sudoku square.

Listing 6.2 Verifying the validity of a Sudoku square

```
import numpy as np

def check_sudoku(grid):
    """ Return True if grid is a valid Sudoku square, otherwise False. """
    for i in range(9):
        # j, k index the top left-hand corner of each 3x3 tile
        j, k = (i // 3) * 3, (i % 3) * 3
❶       if len(set(grid[i,:])) != 9 or len(set(grid[:,i])) != 9\
                    or len(set(grid[j:j+3, k:k+3].ravel())) != 9:
            return False
    return True

sudoku = """145327698
            839654127
            672918543
            496185372
            218473956
```

```
            753296481
            367542819
            984761235
            521839764"""
# Turn the provided string, sudoku, into an integer array
grid = np.array([[int(i) for i in line] for line in sudoku.split()])
print(grid)

if check_sudoku(grid):
    print('grid valid')
else:
    print('grid invalid')
```

❶ Here we use the fact that an array of length 9 contains nine unique elements if the *set* formed from these elements has cardinality 9. No check is made that the elements themselves are actually the numbers 1–9.

Meshes

To evaluate a multidimensional function on a grid of points, a *mesh* is useful. The function `np.meshgrid` is passed a series of N one-dimensional arrays representing coordinates along each dimension and returns a set of N-dimensional arrays comprising a mesh of coordinates at which the function can be evaluated. For example, in the two-dimensional case:

```
In [x]: x = np.linspace(0, 5, 6)
In [x]: y = np.linspace(0, 3, 4)
In [x]: X, Y = np.meshgrid(x, y)
In [x]: X
Out[x]:
array([[ 0.,  1.,  2.,  3.,  4.,  5.],
       [ 0.,  1.,  2.,  3.,  4.,  5.],
       [ 0.,  1.,  2.,  3.,  4.,  5.],
       [ 0.,  1.,  2.,  3.,  4.,  5.]])

In [x]: Y
Out[x]:
array([[ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  1.,  1.,  1.],
       [ 2.,  2.,  2.,  2.,  2.,  2.],
       [ 3.,  3.,  3.,  3.,  3.,  3.]])
```

The arrays `X` and `Y` can *each* be indexed with indexes `i,j`: the `x` array is repeated as rows down `X` and the `y` array as columns across `Y`. A function of two coordinates can therefore be evaluated on the grid as simply `f(X,Y)`.

Setting the optional argument `sparse` to `True` will return sparse grid to conserve memory. In the previous example, instead of two arrays, both with shapes `(6, 4)`, arrays with shapes `(1,6)` and `(4, 1)` that can be broadcast against each other (see Section 6.1.7) will be returned:

```
In [X]: X, Y = np.meshgrid(x, y, sparse=True)
In [X]: X
Out[X]: array([[ 0.,  1.,  2.,  3.,  4.,  5.]])
```

```
In [X]: Y
Out[X]:
array([[ 0.],
       [ 1.],
       [ 2.],
       [ 3.]])
```

6.1.7 ♢ Broadcasting

We have already seen that simple operations such as addition and multiplication can be carried out elementwise on two arrays of the same shape (*vectorization*):

```
In [x]: a = np.array([1, 2, 3])
In [x]: b = np.array([0, 10, 100])
In [x]: a * b
Out[x]: array([  0,  20, 300])
```

Broadcasting describes the rules that NumPy uses to carry out such operations when the arrays have *different* shapes. This allows the operation to be carried out using precompiled C loops instead of slower, Python loops, but there are constraints as to which array shapes can be broadcast against each other. The rules are applied on each dimension of the arrays, starting with the last and working backward. Two dimensions compared in this way are said to be *compatible* if they are *equal* or *one of them is 1*.

The simplest example of broadcasting involves the operation between an array and a scalar (which may be considered for this purpose to be a one-dimensional array of length 1). Consider

```
In [x]: a = np.array([[1, 2, 3], [4, 5, 6]])
In [x]: b = 2
In [x]: c = a * b
In [x]: c
Out[x]:
array([[ 2,  4,  6],
       [ 8, 10, 12]])
```

The dimensions of `a` and `b` are compatible:

```
a:      2 x 3
b:          1
c:      2 x 3
```

Here, `b` can be broadcast across the two dimensions of array `a` by repetition of its value for every element in that array. Similarly, an array of shape `(3,)` can be broadcast across both rows of `a`:

```
In [x]: b = np.array([1, 2, 3])
In [x]: a*b
Out[x]:
array([[ 1,  4,  9],
       [ 4, 10, 18]])


a:      2 x 3
b:          3
c:      2 x 3
```

That is, for each row of `a`, its entries are multiplied by the corresponding entries of the one-dimensional array `b`. However, attempting to multiply `a` by an array whose last dimension is not 1 or 3 is a `ValueError` here:

```
In [x]: b = np.array([1,2])
In [x]: a * b


---------------------------------------------------------------------------
...
----> 1 a * b

ValueError: operands could not be broadcast together with shapes (2,3) (2,)
```

In the example of the sparse mesh created in the previous section, the arrays with shapes `(1,6)` and `(4,1)` are compatible. For example,

```
In [x]: f = X*Y
Out[x]: f
array([[  0.,   0.,   0.,   0.,   0.,   0.],
       [  0.,   1.,   2.,   3.,   4.,   5.],
       [  0.,   2.,   4.,   6.,   8.,  10.],
       [  0.,   3.,   6.,   9.,  12.,  15.]])
```

The broadcasting process "stretches out" the second axis of `Y` from 1 to 6 to match that of `X` and the first axis of `X` from 1 to 4 to match that of `Y`:

```
X:      1 x 6
Y:      4 x 1
f:      4 x 6
```

To force a broadcast on an array with insufficient dimensions to meet your requirements, you can always add an axis with `np.newaxis`. For example, one way to take the *outer product* of two arrays is by adding a dimension to one of them and broadcasting the multiplication:

```
In [x]: a = np.array([1, 2, 3])
In [x]: b = np.array([0, 10, 100])
In [x]: c = a[:, np.newaxis] * b
In [x]: c
Out[x]:
array([[  0,  20, 300],
       [  0,  40, 600],
       [  0,  60, 900]])
```

Thus, instead of matching elements in the two arrays with shapes `(3,)`, the extra axis on `a` creates an array with shape `(3,1)` and this dimension is stretched across the array `b`:

```
a[:,np.newaxis]:   3 x 1
              b:       3
              c:   3 x 3
```

6.1.8 Maximum and minimum values

NumPy arrays have the methods `min` and `max`, which return the minimum and maximum values in the array. By default, a single value for the flattened array is returned; to find maximum and minimum values along a given axis, use the `axis` argument:

```
In [x]: a = np.array([[3, 0, -1, 1], [2, -1, -2, 4], [1, 7, 0, 4]])
In [x]: print(a)
[[ 3  0 -1  1]
 [ 2 -1 -2  4]
 [ 1  7  0  4]]
In [x]: a.min()      # "global" minimum
Out[x]: -2
In [x]: a.max()      # "global" maximum
Out[x]: 7
In [x]: print( a.min(axis=0) )
[ 1 -1 -2  1]        # minima in each column
In [x]: print( a.max(axis=1) )
[3 4 7]              # maxima in each row
```

Often one wants not the maximum (or minimum) value itself but its index in the array. This is what the methods `argmin` and `argmax` do. By default, the index returned is into the *flattened* array, so the actual value can be retrieved using a view on the array created by `ravel`:

```
In [x]: a.argmin()
6
In [x]: a.ravel()[a.argmin()]
-2
In [x]: print(a.argmax(axis=0))
[0 2 2 1]        # row indexes of maxima in each column
In [x]: print(a.argmax(axis=1))
[0 3 1]          # column indexes of maxima in each row
```

Figure 6.2 illustrates the process for `axis=0` and for `axis=1`. Notice that if more than one equal maximum exists in a column, the index of the first is returned.

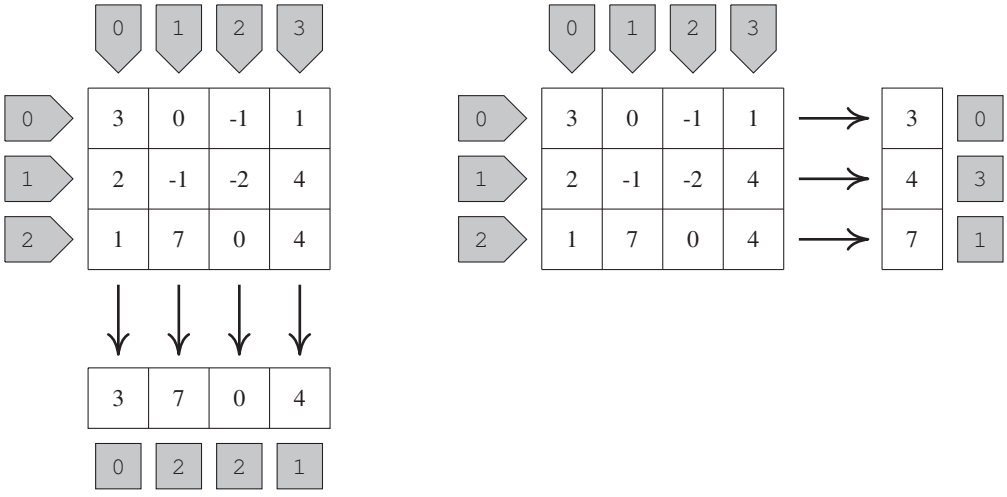
Example E6.5 Consider the following oscillating functions on the interval $[0, L]$:

$$f_n(x) = x(L-x)\sin\frac{2\pi x}{\lambda_n}; \quad \lambda_n = \frac{2L}{n}, \quad n = 1, 2, 3, \cdots .$$

The following code defines a two-dimensional array holding values of these functions for $L = 1$ on a grid of $N = 100$ points (rows) for $n = 1, 2, \cdots, 5$ (columns). The position of the maximum and minimum in each column is calculated with `argmax(axis=0)` and `argmin(axis=0)`. (See Figure 6.3.)

Listing 6.3 `argmax` and `argmin`

```
# eg6-array_maxmin.py
import numpy as np
import pylab

N = 100
L = 1
```



(a) `axis=0` (b) `axis=1`

Figure 6.2 (a) `a.max(axis=0)` giving the maximum values and `a.argmax(axis=0)` giving the indexes of the maximum values of each column in array `a` (that is, maintaining the *row* dimension) and (b) The same for `axis=1`: maximum values along each row.

```
def f(i, n):
    x = i * L / N
    lam = 2*L/(n+1)
    return x * (L-x) * np.sin(2*np.pi*x/lam)

a = np.fromfunction(f, (N+1, 5))
min_i = a.argmin(axis=0)
max_i = a.argmax(axis=0)
pylab.plot(a, c='k')
pylab.plot(min_i, a[min_i, np.arange(5)], 'v', c='k', markersize=10)
pylab.plot(max_i, a[max_i, np.arange(5)], '^', c='k', markersize=10)
pylab.xlabel(r'$x$')
pylab.ylabel(r'$f_n(x)$')
pylab.show()
```

6.1.9 Sorting an array

NumPy arrays can be sorted in several different ways with the `sort` method, which orders the numbers in an array *in place*. By default, this method sorts multidimensional arrays along their *last* axis. To sort along some other axis, set the `axis` argument. For example,

```
In [x]: a = np.array([5, -1, 2, 4, 0, 4])
In [x]: a.sort()
```

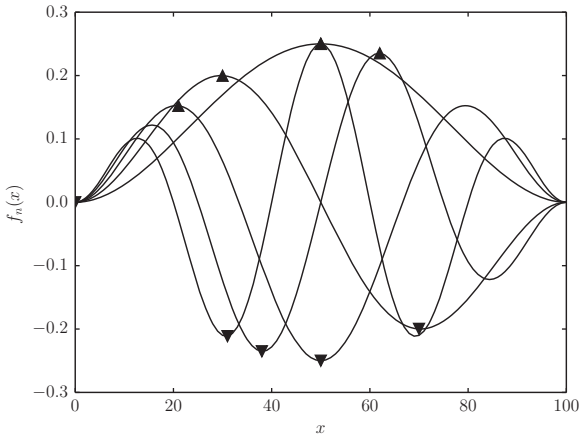


Figure 6.3 Maxima and minima of the functions $f_n(x)$ described in Example E6.5. Note that only the "global" maximum and minimum are returned for each function, and that where more than one point has the same maximum or minimum value, only the first is returned.

```
In [x]: print(a)
[-1  0  2  4  4  5]
In [x]: b = np.array([[0, 3, -2], [7, 1, 3], [4, 0, -1]])
In [x]: print(b)
[[ 0  3 -2]
 [ 7  1  3]
 [ 4  0 -1]]
In [x]: b.sort()          # sort the numbers along each row
In [x]: print(b)
[[-2  0  3]
 [ 1  3  7]
 [-1  0  4]]
```

This is the same as `b.sort(axis=1)` – "for each row, order the numbers by column." To sort the numbers in each column – "for each column, order the numbers by row," set `axis=0`:

```
In [x]: b.np.array([[0, 3, -2], [7, 1, 3], [4, 0, -1]])
In [x]: b.sort(axis=0)    # sort the numbers along each column
In [x]: print(b)
[[0  0  -2]
 [4  1  -1]
 [7  3  3]]
```

The sorting algorithm used is the "quicksort" algorithm, which is a good general-purpose choice.[11]

[11] Some arrays can be sorted faster with the alternative *mergesort* or *heapsort* algorithms; these can be selected by setting the optional `kind` argument to the string literal values `'mergesort'` and `'heapsort'`, for example: `b.sort(axis=1, kind='heapsort')`.

Two other sorting functions are worth mentioning. `np.argsort` returns the *indexes* that would sort an array rather than the sorted elements themselves:

```
In [x]: a = np.array([3, 0, -1, 1])
In [x]: np.argsort(a)
Out[x]: array([2, 1, 3, 0])
```

Therefore,

```
In [x]: a[np.argsort(a)]
Out[x]: array([-1,  0,  1,  3])
```

`np.argsort` also takes the `axis` and `kind` arguments previously described.

The method `np.searchsorted` takes a, *sorted* array, `a` and one or more values, `v`, and returns the indexes in `a` at which the values should be entered to maintain its order:

```
In [x]: a = np.array([1, 2, 3, 4])
In [x]: np.searchsorted(a, 3.5)
Out[x]: 3
In [x]: np.searchsorted(a, (3.5, 0, 1.1))
Out[x]: array([3, 0, 1])
```

6.1.10 Structured arrays

Also known as *record arrays*, structured arrays are arrays consisting of rows of values where each value may have its own data type and name. These rows are the "records." This is very much like a table of data with rows (records) consisting of values that fall into columns (fields) and provides a very convenient and natural way to manipulate scientific data that is often obtained or presented in tabular form.

Creating a structured array

The structure of a record array is defined by its `dtype` using a more complex syntax than we have used previously. For example,

```
In [x]: a = np.zeros(5, dtype='int8, float32, complex_')
In [x]: print(a)
[(0, 0.0, 0j) (0, 0.0, 0j) (0, 0.0, 0j) (0, 0.0, 0j) (0, 0.0, 0j)]
In [x]: a.dtype
dtype([('f0', '|i1'), ('f1', '<f4'), ('f2', '<c16')])
```

Here we have created an array of five records, each of which has three fields, defined by constructing a `dtype` specified by the string `'int8, float32, complex_'`.

- The first field is a single-byte, signed integer (`int8` which is described by the string `'|i1'` – clearly the endianness (byte order) is not relevant in a one-byte quantity);
- The second is a single-precision floating point number (which on my system) is stored in memory as a little-endian 4-byte sequence, indicated by `'<f4'`;
- The final field is defined to be a complex number to default precision, which on my system is stored in 16-bytes, little-endian (`complex_` is equivalent to `complex128` which corresponds to a data type `'<c16'`).

Because we did not explicitly name the fields, they are given the default names `'f0'`, `'f1'` and `'f2'`. To name the fields of our structured array explicitly, pass the `dtype` constructor a list of (*name, dtype descriptor*) tuples: for example,

```
In [x]: dt = np.dtype( [('time', 'f8'), ('signal', 'i4')] )
In [x]: a = np.zeros(10, dtype=dt)
In [x]: a
Out[x]:
array([(0.0, 0), (0.0, 0), ..., (0.0, 0)],
      dtype=[('time', '<f8'), ('signal', '<i4')])
```

A structured array can therefore be visualized as a table of data values with column headings for each field.

Assigning records in a structured array is as expected:

```
In [x]: a[0] = (0., 4)
In [x]: a[1:3] = [(0.5, -3), (1., -5)]
In [x]: a
Out[x]:
array([(0.0, 4), (0.5, -3), (1.0, -5), ..., (0.0, 0)],
      dtype=[('time', '<f8'), ('signal', '<i4')])
```

but the real power of this approach is in the ability to reference a field by its name. For example, to set the `'time'` column in our array to a linear sequence:

```
In [x]: a['time'] = np.linspace(0., 4.5, 10)
In [x]: print(a)
[(0.0, 4) (0.5, -3) (1.0, -5) (1.5, 0) (2.0, 0) (2.5, 0) (3.0, 0) (3.5, 0)
 (4.0, 0) (4.5, 0)]
In [x]: print(a['time'][-1])
4.5
```

Likewise, to obtain a view on a column, refer to it by name:

```
In [x]: print(a['time'])
[ 0.   0.5  1.   1.5  2.   2.5  3.   3.5  4.   4.5]
In [x]: print( a['signal'].min() )
-5
```

More ways to create a structured array

There are several (arguably, too many) ways to define the `dtype` describing a structured array. So far we have used a string of comma-separated identifiers and a list of tuples. A third way is to use a *dictionary*. The basic usage assigns a list of values to the two keys `'names'` and `'formats'` naming the fields and specifying their formats respectively:

```
In [x]: dt = np.dtype({ 'names': ['time', 'signal'],
                        'formats':['f8', 'i4']
                      })
In [x]: a = np.zeros(10, dtype=dt)
```

defines the same structured array of `(time, signal)` records as before. A third key, `'titles'`, can be used to give each field a more detailed description; each title can then be used as an alias to its name in referring to that field in the array.[12]

[12] In fact, `title` can be any Python object and can be used to provide detailed "metadata" concerning the corresponding field.

```
In [x]: dt = np.dtype({'names': ['candidate', 'mark', 'grade'],
                       'formats': ['|S50', 'u1', '|S2'],
                       'titles': ['Candidate Name', 'Percentage Mark', 'Grade: A-F']})
In [x]: a = np.zeros(10, dtype=dt)
In [x]: a[0] = ('John Brown', 64, 'B-')
In [x]: a[1] = ('Jane Smith', 78, 'A')
In [x]: print(a['Candidate Name'])
[b'John Brown' b'Jane Smith' b'' b'' b'' b'' b'' b'' b'' b'']
In [x]: print(a['Percentage Mark'])
[64 78  0  0  0  0  0  0  0  0]
```

Sorting structured arrays

Structured arrays can be sorted by giving a specific order to the fields used with the `order` argument. For example, with the following structured array:

```
In [x]: data = [ ('NiCd', 1.2, 0.14, 2000),
                 ('Lead acid', 2.1, 0.14, 700),
                 ('Lithium ion', 3.6, 0.46, 800) ]
In [x]: dtype = [ ('name', '|S20'),
                  ('voltage', 'f8'),
                  ('specific energy', 'f8'),
                  ('cycle durability', 'i4') ]
In [x]: a = np.array(data, dtype=dtype)
In [x]: a.sort(order='specific energy')
In [x]: print(a)
[(b'Lead acid', 2.1, 0.14, 700) (b'NiCd', 1.2, 0.14, 2000)
 (b'Lithium ion', 3.6, 0.46, 800)]

In [x]: a.sort(order=['specific energy', 'voltage'])
In [x]: print(a)
[(b'NiCd', 1.2, 0.14, 2000) (b'Lead acid', 2.1, 0.14, 700)
 (b'Lithium ion', 3.6, 0.46, 800)]
```

The second sort operation here sorts the records by specific energy, and if this is the same for two or more records, then it sorts by voltage.

6.1.11 Arrays as vectors

Although NumPy provides a `matrix` class that specializes `ndarray` to make linear algebra calculations easier and can be used to represent vectors, for many purposes it is just as convenient to define a vector with n components as a regular one-dimensional array with `n` elements.

In addition to elementwise operations such as vector addition, subtraction and so on, NumPy array objects implement scalar (dot) product and vector (cross) product methods:

```
In [x]: a = np.array([1, 0, -3])      # vector as a one-dimensional array
In [x]: b = np.array([2, -2, 5])
In [x]: a.dot(b)                      # or b.dot(a) or np.dot(a,b)
Out[x]: -13
In [x]: np.cross(a, b)
array([ -6, -11,  -2])
```

You can only take the cross product of an array with two or three elements; the third component is assumed to be zero in the former case. To use `dot` and `cross` on two individual vectors, ensure that they are row vectors as described previously and not column vectors represented as an `(n,1)` array:

```
In [x]: a = np.array([[1], [0],[-3]])   # 3x1 two-dimensional array
In [x]: b = np.array([[2],[-2], [5]])
In [x]: print(a)
[[ 1]
 [ 0]
 [-3]]
In [x]: np.dot(a,b)     # tries matrix multiplication: won't work
...

ValueError: objects are not aligned
```

If you do want to take the dot product of two column vectors using `np.dot`, they need to be turned into row vectors:

```
In [x]: np.dot(a.T[0], b.T[0])    # transpose to row vectors
Out[x]: -13
```

This is a bit tortuous: the index is needed because the transpose of our `(n,1)` (two-dimensional) array is a `(1,n)` array from which we want the first and only row for our vector. Alternatively, we can operate using a flattened view of the column vectors obtained with `ravel`:

```
In [x]: a.ravel().dot(b.ravel())
Out[x]: -13
```

See also Section 6.6.

6.1.12 Logic and comparisons

NumPy provides a set of methods for comparing and performing logical operations on arrays elementwise. The more useful of these are summarized in Table 6.4.

Table 6.4 `ndarray` Attributes

Function	Description
`np.all(a)`	Determine whether *all* array elements of `a` evaluate to `True`.
`np.any(a)`	Determine whether *any* array element of `a` evaluates to `True`.
`np.isreal(a)`	Determine whether each element of array `a` is real.
`np.iscomplex(a)`	Determine whether each element of array `a` is a complex number.
`np.isclose(a, b)`	Return a boolean array of the comparison between arrays `a` and `b` for equality within some tolerance.
`np.allclose(a, b)`	Return a `True` if *all* the elements in the arrays `a` and `b` are equal to within some tolerance.

`np.all` and `np.any` work the same as Python's built-in functions of the same name[13] (see Section 2.4.3):

```
In [x]: a = np.array([[1, 2, 0, 3], [4, 0, 1, 1]])
In [x]: np.any(a), np.all(a)
Out[x]: (True, False)     # Some (but not all) elements are equivalent to True
```

`np.isreal` and `np.iscomplex` return boolean arrays:

```
In [x]: b = np.array([1, -1j, 0.5j, 0, 1-2.5j])
In [x]: np.isreal(b)
Out[x]: array([ True, False, False,  True, False], dtype=bool)
In [x]: np.iscomplex(b)
Out[x]: array([False,  True,  True, False,  True], dtype=bool)
```

Because the representation of floating point numbers is not exact, comparing two `float` or `complex` arrays with the `==` operator is not always reliable and is not recommended. Instead, the best we can do is see if two values are "close" to one another within some (typically small) absolute or relative tolerance – NumPy provides the function `np.isclose(a, b)` for elementwise comparisons of two arrays: it returns `True` for elements satisfying

```
abs(a-b) <= (atol + rtol * abs(b))
```

with absolute tolerance, `atol` and relative tolerance, `rtol` which are 10^{-8} and 10^{-5} respecively by default but can be changed by setting the corresponding arguments.[14] An additional argument, `equal_nan`, defaults to `False`, meaning that `nan` values in corresponding positions in the two arrays are treated as different; to treat such elements as equal, set `equal_nan=True`.

```
In [x]: a = np.array([1.66e-27, 1.38e-23, 6.63e-34, 6.02e23, np.nan])
In [x]: b = np.array([1.66e-27, 1.66e-27, 1.66e-27, 6.00e23, np.nan])
In [x]: np.isclose(a, b)
Out[x]: array([ True,  True,  True, False, False], dtype=bool)
In [x]: np.isclose(a, b, equal_nan=True)
Out[x]: array([ True,  True,  True, False,  True], dtype=bool)
```

Note that small numbers compare as equal even though they may differ by many orders of magnitude – to correct this, set `atol=0` to compare within relative tolerance only:

```
In [x]: np.isclose(a, b, atol=0)
Out[x]: array([ True, False, False, False, False], dtype=bool)
```

Finally, `allclose(a, b)` returns a single value: `True` only if every element in `a` is equal to the corresponding element in `b` (within the tolerance defined by `atol` and `rtol`), and otherwise `False`.

```
In [x]: x = np.linspace(0, np.pi, 100)
In [x]: np.allclose(np.sin(x)**2, 1 - np.cos(x)**2)
Out[x]: True
```

[13] Except that they don't work on generator or iterator objects.

[14] Note that this relation is not symmetric in `a` and `b`, so it is possible that `isclose(a, b)` may not equal `isclose(b, a)`.

6.1.13 Exercises

Questions

Q6.1.1 What is the difference between the objects `np.ndarray` and `np.array`?

Q6.1.2 Why doesn't this create a two-dimensional array?

```
>>> np.array((1,0,0), (0,1,0), (0,0,1), dtype=float)
```

What is the correct way?

Q6.1.3 What is the difference, if any, between the following statements:

```
>>> a = np.array([0,0,0])
>>> a = np.array([[0,0,0]])
```

Q6.1.4 Explain the following behavior:

```
In [x]: a, b = np.zeros((3,)), np.ones((3,))
In [x]: a.dtype = 'int'
In [x]: a
Out[x]: array([0, 0, 0])
In [x]: b.dtype = 'int'
In [x]: b
Out[x]: array([4607182418800017408, 4607182418800017408, 4607182418800017408])
```

What is the correct way to convert an array of one data type to an array of another?

Q6.1.5 A $3 \times 4 \times 4$ array is created with

```
In [x]: a = np.linspace(1,48,48).reshape(3,4,4)
```

Index or slice this array to obtain the following:

a.
```
20.0
```

b.
```
[ 9. 10. 11. 12.]
```

c. The 4×4 array:

```
[[ 33.  34.  35.  36.]
 [ 37.  38.  39.  40.]
 [ 41.  42.  43.  44.]
 [ 45.  46.  47.  48.]]
```

d. The 3×2 array:

```
[[  5.,   6.],
 [ 21.,  22.],
 [ 37.,  38.]]
```

e. The 4×2 array:

```
[[ 36.  35.]
 [ 40.  39.]
 [ 44.  43.]
 [ 48.  47.]]
```

f. The 3×4 array:

```
[[ 13.   9.   5.   1.]
 [ 29.  25.  21.  17.]
 [ 45.  41.  37.  33.]]
```

g. (Harder) Using an array of indexes, the 2×2 array:

```
[[  1.   4.]
 [ 45.  48.]]
```

Q6.1.6 Write an expression, using boolean indexing, which returns only the values from an array that have magnitudes between 0 and 1.

Q6.1.7 Why does the following statement evaluate to `True` even though the two numbers passed to `np.isclose()` differ by more than `atol`?

```
In [x]: np.isclose(-2.00231930436153, -2.0023193043615, atol=1.e-14)
Out[x]: True
```

Q6.1.8 Explain why the following evaluates to `True` even though the two approximations to π differ by more than 10^{-16}:

```
In [x]: np.isclose(3.1415926535897932, 3.141592653589793, atol=1.e-16, rtol=0)
Out[x]: True
```

whereas this statement works as expected:

```
In [x]: np.isclose(3.14159265358979, 3.1415926535897, atol=1.e-14, rtol=0)
Out[x]: False
```

Q6.1.9 Verify that the magic square created in Example E6.2 satisfies the conditions that it contains the numbers 1 to N^2 and that its rows, columns and main diagonals sum to $N(N^2 + 1)/2$.

Q6.1.10 Write a one-line statement that returns `True` if an array is a monotonically increasing sequence or `False` otherwise.

Hint: `np.diff` returns the *difference* between consecutive elements of a sequence. For example,

```
In [x]: np.diff([1,2,3,3,2])
Out[x]: array([ 1,  1,  0, -1])
```

♢ **Q6.1.11** (Harder) The `dtype` `np.uint8` represents an unsigned integer in 8 bits. Its value may therefore be in the range 0−255. Explain the following behavior:

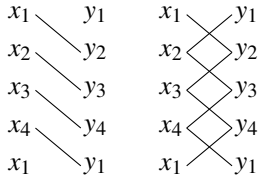
```
In [x]: x = np.uint8(250)
In [x]: x*2
Out[x]: 500

In [x]: x = np.array([250,], dtype=np.uint8)
In [x]: x*2
Out[x]: array([244], dtype=uint8)
```

Problems

P6.1.1 Turn the following data concerning various species of cetacean into a NumPy structured array and order it by (a) mass and (b) population. Determine in each case the index at which *Bryde's whale* (population: 100000, mass: 25 tonnes) should be inserted to keep the array ordered.

Name	Population	Mass/tonnes
Bowhead whale	9000	60
Blue whale	20000	120
Fin whale	100000	70
Humpback whale	80000	30
Gray whale	26000	35
Atlantic white-sided dolphin	250000	0.235
Pacific white-sided dolphin	1000000	0.15
Killer whale	100000	4.5
Narwhal	25000	1.5
Beluga	100000	1.5
Sperm whale	2000000	50
Baiji	13	0.13
North Atlantic right whale	300	75
North Pacific right whale	200	80
Southern right whale	7000	70

P6.1.2 The *shoelace algorithm* for calculating the area of a simple polygon (that is, one without holes or self-intersections) proceeds as follows: Write down the (x, y) coordinates of the N vertices in an $N \times 2$ array and then repeat the coordinates of the first vertex as the last row to make an $(N + 1) \times 2$ array. Now (a) multiply each x-coordinate value in the first N rows by the y-coordinate value in the next row down and take the sum, $S_1 = x_1y_2 + x_2y_3 + \cdots + x_Ny_1$. Then (b) multiply each y-coordinate value in the first N rows by the x-coordinate in the next row down and take the sum, $S_2 = y_1x_2 + y_2x_3 + \cdots + y_Nx_1$. The area of the polygon is then $\frac{1}{2}|S_1 - S_2|$.



Implement this algorithm as a function that takes a NumPy array of vertices as its argument and returns the area of the polygon. Do not use Python loops!

P6.1.3 Using NumPy, it is possible to do this exercise without using a single (Python) loop.

The normalized Gaussian function with mean μ and standard deviation σ is

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

Write a program to calculate and plot the Gaussian functions with $\mu = 0$ and the three values $\sigma = 0.5, 1, 1.5$. Use a grid of 1,000 points in the interval $-10 \leq x \leq 10$.

Verify (by direct summation) that the functions are normalized with area 1.

Finally, calculate the first derivative of these functions on the same grid using the first-order central difference approximation:

$$g'(x) \approx \frac{g(x+h) - g(x-h)}{2h}$$

for some suitably chosen, small h.

6.2 Reading and writing an array to a file

Scientific data are frequently read in from a text file, which may contain comments, missing values and blank lines. Columns of values may be either aligned in a fixed-width format or separated by one or more delimiting characters (such as spaces, tabs or commas). Furthermore, there may be a descriptive header and even footnotes to the file, which make it hard to parse directly using Python's string methods.

NumPy provides several functions for reading data from a text file. The simpler `np.loadtxt` handles many common cases; the more sophisticated `np.genfromtxt` allows for better handling of missing values and footers. These are described in the following sections.

6.2.1 `np.save` and `np.load`

There is a platform-independent *binary* format for saving a NumPy array:

```
In [x]: np.save('my-array.npy', a)
```

will save the array `a` to the binary file `my-array.npy` (the `.npy` extension is appended if it is not provided). The array can then be reloaded using NumPy on any other operating system with

```
In [x]: a = np.load('my-array.npy')
```

(the `.npy` extension must be provided).

6.2.2 `np.loadtxt`

The method prototype for `np.loadtxt` is

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

The arguments are as follows:

- `fname`: The only required argument, `fname`, which can be a filename, an open file, or a generator returning the lines of data to be parsed.
- `dtype`: The data type of the array defaults to `float` but can be set explicitly by the `dtype` argument. In particular this is the place to set up names and types for a structured array (see Section 6.1.10).
- `comments`: Comments in a file are usually started by some character such as `#` (as with Python) or `%`. To tell NumPy to ignore the contents of any line following this character, use the `comments` argument – by default it is set to `#`.
- `delimiter`: The string used to separate columns of data in the file; by default it is `None`, meaning that any amount of whitespace (spaces, tabs) delimits the data. To read a comma-separated (csv) file, set `delimiter=','`.
- `converters`: An optional dictionary mapping the column index to a function converting string values in that column to data (e.g., `float`).
- `skiprows`: An integer giving the number of lines at the start of the file to skip over before reading the data (e.g., to pass over header lines). Its default is 0 (no header).
- `usecols`: A sequence of column indexes determining which columns of the file to return as data; by default it is `None`, meaning all columns will be parsed and returned.
- `unpack`: By default, the data table is returned in a single array of rows and columns reflecting the structure of the file read in. Set `unpack=True` will transpose this array so that individual columns can be picked off an assigned to different variables.
- `ndmin`: The minimum number of dimensions the returned array should have. By default, 0 (so a file containing a single number is read in as a scalar), it can be set to 1 or 2.

For example, to read the first, third and fourth columns from the file `data.txt` into three separate one-dimensional arrays:

```
col1, col3, col4 = np.loadtxt('data.txt', usecols=(0,2,3), unpack=True)
```

Example E6.6 The use of `np.loadtxt` is best illustrated using an example. Consider the following text file of data relating to a (fictional) population of students. This file can be downloaded as `eg6-a-student-data.txt` from scipython.com/eg/aac.

```
# Student data collected on 17 July 2014
# Researcher: Dr Wicks, University College Newbury

# The following data relate to N = 20 students. It
# has been totally made up and so therefore is 100%
# anonymous.

Subject Sex    DOB       Height  Weight       BP      VO2max
(ID)    M/F  dd/mm/yy     m       kg         mmHg   mL.kg-1.min-1
JW-1     M    19/12/95    1.82     92.4    119/76    39.3
```

```
JW-2     M    11/1/96     1.77     80.9    114/73   35.5
JW-3     F    2/10/95     1.68     69.7    124/79   29.1
JW-6     M    6/7/95      1.72     75.5    110/60   45.5
# JW-7   F    28/3/96     1.66     72.4    101/68   -
JW-9     F    11/12/95    1.78     82.1    115/75   32.3
JW-10    F    7/4/96      1.60     -       -        30.1
JW-11    M    22/8/95     1.72     77.2    97/63    48.8
JW-12    M    23/5/96     1.83     88.9    105/70   37.7
JW-14    F    12/1/96     1.56     56.3    108/72   26.0
JW-15    F    1/6/96      1.64     65.0    99/67    35.7
JW-16    M    10/9/95     1.63     73.0    131/84   29.9
JW-17    M    17/2/96     1.67     89.8    101/76   40.2
JW-18    M    31/7/96     1.66     75.1    -        -
JW-19    F    30/10/95    1.59     67.3    103/69   33.5
JW-22    F    9/3/96      1.70     -       119/80   30.9
JW-23    M    15/5/95     1.97     89.2    124/82   -
JW-24    F    1/12/95     1.66     63.8    100/78   -
JW-25    F    25/10/95    1.63     64.4    -        28.0
JW-26    M    17/4/96     1.69     -       121/82   39.
```

Let's find the average heights of the male and female students. The columns we need are the second and fourth, and there's no missing data in these columns so we can use `np.loadtxt`. First construct a record `dtype` for the two fields, then read the relevant columns after skipping the first nine header lines:

```
In [x]: fname = 'eg6-a-student-data.txt'
In [x]: dtype1 = np.dtype([('gender', '|S1'), ('height', 'f8')])
In [x]: a = np.loadtxt(fname, dtype=dtype1, skiprows=9, usecols=(1,3))
In [x]: a
Out[x]:
array([(b'M', 1.8200000524520874), (b'M', 1.7699999809265137),
       (b'F', 1.6799999475479126), (b'M', 1.7200000286102295),
       ...
       (b'M', 1.690000057220459)],
      dtype=[('gender', 'S1'), ('height', '<f8')])
```

To find the average heights of the male students, we only want to index the records with the gender field as `M`, for which we can create a boolean array:

```
In [x]: m = a['gender'] == b'M'
In [x]: m
Out[x]: array([ True,  True, False,  True, ...,  True], dtype=bool)
```

`m` has entries that are `True` or `False` for each of the 19 valid records (one is commented out) according to whether the student is male or female. So the heights of the male students can be seen to be:

```
In [x]: print(a['height'][m])
[ 1.82000005  1.76999998  1.72000003  1.72000003  1.83000004  1.63
  1.66999996  1.65999997  1.97000003  1.69000006]
```

Therefore, the averages we need are

```
In [x]: m_av = a['height'][m].mean()
❶ In [x]: f_av = a['height'][~m].mean()
In [x]: print('Male average: {:.2f} m, Female average: {:.2f} m'.format(m_av,f_av))
Male average: 1.75 m, Female average: 1.65 m
```

❶ Note that `~m` ("not `m`") is the inverse boolean array of `m`.

To perform the same analysis on the student weights we have a bit more work to do because there are some missing values (denoted by '-'). We could use `np.genfromtxt` (see Section 6.2.3), but let's write a converter method instead. We'll replace the missing values with the nicely unphysical value of −99. The function `parse_weight` expects a string argument and returns a `float`:

```
def parse_weight(s):
    try:
        return float(s)
    except ValueError:
        return -99.
```

This is the function we want to pass as a converter for column 4:

```
In [x]: dtype2 = np.dtype([('gender', '|S1'), ('weight', 'f8')])
In [x]: b = np.loadtxt(fname, dtype=dtype2, skiprows=9, usecols=(1,4),
                       converters={4: parse_weight})
```

Now mask off the invalid data and index the array with a boolean array as before:

```
In [x]: mv = b['weight'] > 0    # elements only True for valid data
In [x]: m_wav = b['weight'][mv & m].mean()      # valid and male
In [x]: f_wav = b['weight'][mv & ~m].mean()     # valid and female
In [x]: print('Male average: {:.2f} kg,
               Female average: {:.2f} kg'.format(m_wav,f_wav))
Male average: 82.44 kg, Female average: 66.94 kg
```

Finally, let's read in the blood pressure data. Here we have a problem, because the systolic and diastolic pressures are not separated by whitespace but by a forward slash (/). One solution is to reformat each line to replace the slash with a space before it is fed to `np.loadtxt`. Recall that `fname` can be a generator instead of a filename or open file: we write a suitable generator function, `reformat_lines`, which takes an open file object and yields its lines to `np.loadtxt`, one by one, after the replacement. This is going to mess with the column numbering because it has the side effect of splitting up the birth dates into three columns, so in our reformatted lines the blood pressure values are now in the columns indexed at 7 and 8.

Listing 6.4 Reading the blood pressure column

```
# eg6-a-read-bp.py
import numpy as np

fname = 'eg6-a-student-data.txt'
dtype3 = np.dtype([('gender', '|S1'), ('bps', 'f8'), ('bpd', 'f8')])

def parse_bp(s):
    try:
        return float(s)
    except ValueError:
        return -99.

def reformat_lines(fi):
    for line in fi:
        line = line.replace('/',' ')
        yield line
```

```
with open(fname) as fi:
    gender, bps, bpd = np.loadtxt(reformat_lines(fi), dtype3, skiprows=9,
                usecols=(1,7,8),converters={7: parse_bp, 8: parse_bp},
                unpack=True)

# now do something with the data...
```

6.2.3 np.genfromtxt

NumPy's `genfromtxt` function is similar to `np.loadtxt` but has a few more options and is able to cope with missing data.

The following arguments to this function are the same as for `np.loadtxt`: `fname` (the only required argument), `dtype`, `comments`, `converters`, `usecols` and `unpack`.

Headers and footers

Instead of `np.loadtxt`'s `skiprows`, the `np.genfromtxt` function has two optional arguments, `skip_header` and `skip_footer`, giving the number of lines to skip at the beginning and the end of the file, respectively.

Fixed-width fields

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

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

so that the first row is to be split: `' 1', '2', ' 100.231', '.03'`. There is no delimiter character, so this isn't possible with `np.loadtxt`, but with `np.genfromtxt`:

```
In [x]: np.genfromtxt(fname='data.txt',delimiter=[2,1,9,3],
                      dtype='i4, i4, f8, f8')
array([(1, 2, 100.231, 0.03), (1, 1, 1201.842, 0.04), (1, 1, 99.324, 0.02)],
      dtype=[('f0', '<i4'), ('f1', '<i4'), ('f2', '<f8'), ('f3', '<f8')])
```

as required.

Missing data

If a data set is incomplete, `np.loadtxt` will be unable to parse the fields with missing data into valid values for the array and will raise an exception. `np.genfromtxt`, however, sets missing or invalid entries equal to the default values given in Table 6.5.

For example, the comma-separated file here has two ways of indicating missing data: empty fields and entries with '`???`':

Table 6.5 Default filling values for missing data used by `genfromtxt`

Data type	Default value
`int`	`-1`
`float`	`np.nan`
`bool`	`False`
`complex`	`np.nan + 0.j`

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

Accordingly, `np.genfromtxt` sets the missing fields to its defaults:

```
In [x]: data = np.genfromtxt(fname='data.txt', dtype='f8, i4, f8, i4',
   ...:                      delimiter=',')
In [x]: print(data)
[(10.1, 4, -0.1, 2) (10.2, 4, nan, 0) (10.3, -1, nan, 4) (10.4, 2, 0.0, -1)
 (10.5, -1, nan, 3)]
```

The `missing_values` and `filling_values` arguments allow closer control over which default values to use for which columns. If `missing_values` is given as a sequence of strings, each string is associated with a column in the data file, in order; if given as a dictionary of string values, the keys denote either column indexes (if they are integers) or column names (if they are strings). The corresponding argument, `filling_values`, maps these column indexes or names to default values. If `filling_values` is provided as a single value, this value is used for missing data in all columns.

For example, to replace the invalid values in column 1 (indicated by `'???'`) with 999, the missing or invalid values in column 2 (also indicated by `'???'`) with −99 and the missing values in column 3 with 0:

```
In [x]: data =np.genfromtxt(fname='data.txt', dtype='f8, i4, f8, i4',
   ...:                     delimiter=',', missing_values={1: '???', 2: '???'},
   ...:                     filling_values={1: 999, 2: -99., 3: 0})
   ...:
In [x]: print(data)
[(10.1, 4, -0.1, 2) (10.2, 4, -99.0, 0) (10.3, 999, -99.0, 4)
 (10.4, 2, 0.0, 0) (10.5, -1, -99.0, 3)]
```

Note in particular how the missing entry in the second column has been replaced by `999` instead of the default `-1` – this would be particularly important if `-1` is a valid value for this column (however, it is now up to the rest of your code to recognize and know what to do with values such as `999`.[15]

[15] For more advanced handling of missing values, see the `genfromtxt` documentation for details on the `usemask` argument and *masked arrays* in general.

Column names

The argument `names` provides a way of setting names for the columns of data read in. If it is the boolean value `True`, the names are read from the first valid line after the number of lines skipped over specified by the `skip_header` argument; if `names` is a comma-separated string of names or a sequence of strings, those strings will be used as names. By default, `names` is `None` and the field names are taken from the `dtype`, if given.

Example E6.7 In an experiment to investigate the *Stroop effect*, a group of students were timed reading out 25 randomly ordered color names, first in black ink and then in a color other than the one they name (e.g., the word "red" in blue ink). The results are presented in the text file. Missing data are indicated by the character `X`.

```
Subject Number, Gender, Time (words in black), Time (words in color)
1,F,18.72,31.11
2,F,21.14,52.47
3,F,19.38,33.92
4,M,22.03,50.57
5,M,21.41,29.63
6,M,15.18,24.86
7,F,14.13,33.63
8,F,19.91,42.39
9,F,X,43.60
10,F,26.56,42.31
11,F,19.73,49.36
12,M,18.47,31.67
13,M,21.38,47.28
14,M,26.05,45.07
15,F,X,X
16,F,15.77,38.36
17,F,15.38,33.07
18,M,17.06,37.94
19,M,19.53,X
20,M,23.29,49.60
21,M,21.30,45.56
22,M,17.12,42.99
23,F,21.85,51.40
24,M,18.15,36.95
25,M,33.21,61.59
```

We can read in this data with `np.genfromtxt` and summarize the results with the code here.

Listing 6.5 Analyzing data from a Stroop effect experiment

```
# eg6-stroop.py
import numpy as np

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

```
nwords = 25

# Remove invalid rows from data set
❷ filtered_data = data[np.isfinite(data['black']) & np.isfinite(data['color'])]

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

# Produce statistics: mean and standard deviation by gender and word color
mu_fb, sig_fb = np.mean(fb), np.std(fb)
mu_fc, sig_fc = np.mean(fc), np.std(fc)
mu_mb, sig_mb = np.mean(mb), np.std(mb)
mu_mc, sig_mc = np.mean(mc), np.std(mc)

print('Mean and (standard deviation) times per word (sec)')
print('gender |    black      |    color     | difference')
print('   F   | {:4.3f} ({:4.3f}) | {:4.3f} ({:4.3f}) |   {:4.3f}'
                    .format(mu_fb, sig_fb, mu_fc, sig_fc, mu_fc - mu_fb))
print('   M   | {:4.3f} ({:4.3f}) | {:4.3f} ({:4.3f}) |   {:4.3f}'
                    .format(mu_mb, sig_mb, mu_mc, sig_mc, mu_mc - mu_mb))
```

❶ In the absence of any provided `filling_values`, `np.genfromtxt` will replace the invalid fields with `np.nan`.

❷ We only want to consider students with times for both parts of the experiment, so create a filtered data set here.

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

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

6.2.4 Exercises

Problems

P6.2.1 The following text file gives some data concerning the 8,000 m peaks, in alphabetical order.

```
ex6-2-b-mountain-data.txt This file contains a list of the 14
highest mountains in the world with their names, height, year
of first ascent, year of first winter ascent, and location as
longitude and latitude in degrees (d), minutes (m) and seconds
(s). Note: as of 2013, no winter ascent has been made of K2 or
Nanga Parbat.
```

```
----------------------------------------------------------------------
Name          Height  First ascent First winter Location
                m         date     ascent date   (WGS84)
----------------------------------------------------------------------
Annapurna I   8091    3/6/1950    3/2/1987     28d35m46sN 83d49m13sE
Broad Peak    8051    9/6/1957    5/3/2013     35d48m39sN 76d34m06sE
Cho Oyu       8201    19/10/1954  12/2/1985    28d05m39sN 86d39m39sE
Dhaulagiri I  8167    13/5/1960   21/1/1985    27d59m17sN 86d55m31sE
Everest       8848    29/5/1953   17/2/1980    27d59m17sN 86d55m31sE
Gasherbrum I  8080    5/7/1958    9/3/2012     35d43m28sN 76d41m47sE
Gasherbrum II 8034    7/7/1956    2/2/2011     35d45m30sN 76d39m12sE
K2            8611    31/7/1954   -            35d52m57sN 76d30m48sE
Kangchenjunga 8568    25/5/1955   11/1/1986    27d42m09sN 88d08m54sE
Lhotse        8516    18/5/1956   31/12/1988   27d57m42sN 86d56m00sE
Makalu        8485    15/5/1955   9/2/2009     27d53m21sN 87d05m19sE
Manaslu       8163    9/5/1956    12/1/1984    28d33m0sN  84d33m35sE
Nanga Parbat  8126    3/7/1953    -            35d14m15sN 74d35m21sE
Shishapangma  8027    2/5/1964    14/1/2005    28d21m8sN  85d46m47sE
----------------------------------------------------------------------
```

Use NumPy's `loadtxt` method to read these data into a suitable structured array to determine the following:

1. The lowest 8,000 m peak
2. The most northely, easterly, southerly and westerly peaks
3. The most recent first ascent of the peaks
4. The first of the peaks to be climbed in winter

Also produce another structured array containing a list of mountains with their height in *feet* and first ascent date, ordered by increasing height.[16]

P6.2.2 The file `busiest_airports.txt`, available to download from scipython.com/ex/afa, provides details of the 30 busiest airports in the world in 2014. The tab-delimited fields are: three-letter IATA code, airport name, airport location, latitude and longitude (both in degrees).

Write a program to determine the distance between two airports identified by their three-letter IATA code, using the Haversine formula (see, for example, Exercise 4.4.2) and assuming a spherical Earth of radius 6378.1 km.

P6.2.3 The World Bank provides an extensive collection of data sets on a wide range of "indicators," which is searchable at http://data.worldbank.org/. Data sets concerning child immunization rates for BCG (against tuberculosis), Pol3 (Polio) and measles in three South-East Asian countries between 1960 and 2013 are available at scipython.com/ex/afb. Fields are delimited by semicolons and missing values are indicated by `'..'`.

Use NumPy methods to read in this data and create three plots (one for each vaccine) comparing immunization rates in the three countries.

[16] 1 metre = 3.2808399 feet.

6.3 Statistical methods

NumPy provides several methods for performing statistical analysis, either on an entire array or an axis of it.

6.3.1 Ordering statistics

Maxima and minima

We have already used `np.min` and `np.max` to find the minimum and maximum values of an array (these methods are also available using the names `np.amin` and `np.amax`). If the array contains one or more NaN values, the corresponding minimum or maximum value will be `np.nan`. To ignore NaN values instead, use `np.nanmin` and `np.nanmax`:

```
In [x]: a = np.sqrt(np.linspace(-2, 2, 4))
In [x]: print(a)
[        nan         nan  0.          1.          1.41421356]
In [x]: np.min(a), np.max(a)
Out[x]: (nan, nan)
In [x]: np.nanmin(a), np.nanmax(a)
(0.0, 1.4142135623730951)
```

We have also met the functions `np.argmin` and `np.argmax`, which return the *index* of the minimum and maximum values in an array; they too have `np.nanargmin` and `np.nanargmax` variants:

```
In [x]: np.argmin(a), np.argmax(a)
Out[x]: (0, 0)          # The first nan in the array
In [x]: np.nanargmin(a), np.nanargmax(a)
Out[x]: (2, 4)          # The indexes of 0, 1.41421356
```

The related methods, `np.fmin` / `np.fmax` and `np.minimum` / `np.maximum`, compare two arrays, *element by element* and return another array of the same shape. The first pair of methods ignores NaN values and the second pair propagates them into the output array. For example,

```
In [x]: np.fmin([1, -5, 6, 2], [0, np.nan, -1, -1])
array([ 0., -5., -1., -1.])         # NaNs are ignored
In [x]: np.maximum([1, -5, 6, 2], [0, np.nan, -1, -1])
array([  1.,  nan,   6.,   2.])     # NaNs are propagated
```

Percentiles

The `np.percentile` method returns a specified percentile, `q`, of the data along an axis (or along a flattened version of the array if no axis is given). The minimum of an array is the value at `q=0` (0th percentile), the maximum is the value at `q=100` (100th percentile) and the median is the value at `q=50` (50th percentile). Where no single value in the array corresponds to the requested value of `q` exactly, a weighted average of the two nearest values is used. For example,

```
In [x]:  a = np.array([[0., 0.6, 1.2], [1.8, 2.4, 3.0]])
In [x]: np.percentile(a, 50)
1.5
```

```
In [x]: np.percentile(a, 75)
2.25
In [x]: np.percentile(a, 50, axis=1)
array([ 0.6,  2.4])
In [x]: np.percentile(a, 75, axis=1)
array([ 0.9,  2.7])
```

6.3.2 Averages, variances and correlations

Averages

In addition to `np.mean`, which calculates the arithmetic mean of the values along a specified axis of an array, NumPy provides methods for calculating the weighted average, median, standard deviation and variance. The weighted average is calculated as

$$\bar{x}_w = \frac{\sum_i^N w_i x_i}{\sum_i^N w_i}$$

where the weights, w_i, are supplied as a sequence the same length as the array. For example,

```
In [x]: x = np.array([1., 4., 9., 16.])
In [x]: np.mean(x)
7.5
In [x]: np.median(x)
6.5
In [x]: np.average(x, weights=[0., 3., 1., 0.])
5.25        # ie (3.*4. + 1.*9.) / (3. + 1.)
```

If you want the sum of the weights as well as the weighted average, set the `returned` argument to `True`. In the following example, we do this and find the weighted averages in each row (`axis=1` averages values across *columns* of a two-dimensional array):

```
In [x]: x = np.array( [[1., 8., 27], [-0.5, 1., 0.]] )
In [x]: av, sw = np.average(x, weights=[0., 1., 0.1], axis=1, returned=True)
In [x]: print(av)
[ 9.72727273  0.90909091]
In [x]: print(sw)
[ 1.1  1.1]
```

The averages are therefore $(1 \times 8 + 0.1 \times 27)/1.1 = 9.72727273$ and $(1 \times 1.)/1.1 = 0.90909091$ where 1.1 is the sum of the weights.

Standard deviations and variances

The function `np.std` calculates, by default, the *uncorrected sample standard deviation*:

$$\sigma_N = \sqrt{\frac{1}{N}\sum_i^N (x_i - \bar{x})^2}.$$

where x_i are the N observed values in the array and $\bar{x}$ is their mean. To calculate the *corrected sample standard deviation*,

$$\sigma = \sqrt{\frac{1}{N-\delta}\sum_{i}^{N}(x_i - \bar{x})^2},$$

pass to the argument `ddof` the value of δ such that $N - \delta$ is the number of degrees of freedom in the sample. For example, if the sample values are drawn from the population independently with replacement and used to calculate $\bar{x}$ there are $N - 1$ degrees of freedom in the vector of residuals used to calculate σ: $(x_1 - \bar{x}, x_2 - \bar{x}, \cdots, x_N - \bar{x})$ and so $\delta = 1$. For example,

```
In [x]: x = np.array([1., 2., 3., 4.])
In [x]: np.std(x)           # or x.std(), uncorrected standard deviation
1.1180339887498949
In [x]: np.std(x, ddof=1)   # corrected standard deviation
1.2909944487358056
```

The function `np.nanstd` calculates the standard deviation ignoring `np.nan` values (so that N is the number of non-NaN values in the array). NumPy also has methods for calculating the *variance* of the values in an array: `np.var` and `np.nanvar`.

The covariance is returned by the `np.cov` method. In its simplest invocation, it can be passed a single two-dimensional array, `X`, in which the rows represent variables, x_i, and the columns observations of the value of each variable. `np.cov(X)` then returns the covariance matrix, C_{ij}, indicating how variable x_i varies with x_j: the element C_{ij} is said to be an estimate of the covariance of variables x_i and x_j:

$$C_{ij} \equiv \mathrm{cov}(x_i, x_j) = \mathrm{E}[(x_i - \mu_i)(x_j - \mu_j)]$$

where μ_i is the mean of the variable x_i and E[ ] denotes the expected value. If there are N observed values for each of the variables, $\mu_i = \frac{1}{N}\sum_k x_{ik}$. The *unbiased* estimate of the covariance is then

$$C_{ij} = \frac{1}{N-1}\sum_{k}[(x_{ik} - \mu_i)(x_{jk} - \mu_j)]$$

This is the default behavior of `np.cov`, but if the `bias` argument is set to 1, then N is used in the denominator here to give the *biased* estimate of the covariance. Finally, the denominator can be set explicitly to $N - \delta$ by passing δ as the argument to the `ddof` argument of `cov`.

Example E6.8 As an example, consider the matrix of five observations each of three variables, x_0, x_1 and x_2 whose observed values are held in the three rows of the array `X`:

```
X = np.array([ [0.1, 0.3, 0.4, 0.8, 0.9],
               [3.2, 2.4, 2.4, 0.1, 5.5],
               [10., 8.2, 4.3, 2.6, 0.9]
             ])
```

The covariance matrix is a 3×3 array of values,

```
In [x]: print( np.cov(X) )
[[  0.115 ,   0.0575,  -1.2325],
 [  0.0575,   3.757 ,  -0.8775],
 [ -1.2325,  -0.8775,  14.525 ]]
```

The diagonal elements, C_{ii}, are the variances in the variables x_i assuming $N - 1$ degrees of freedom:

```
In [x]: print(np.var(X, axis=1, ddof=1))
[  0.115   3.757  14.525]
```

Although the magnitude of the covariance matrix elements is not always easy to interpret (because it depends on the magnitude of the individual observations which may be very different for different variables), it is clear that there is a strong anticorrelation between x_0 and x_2 ($C_{02} = -1.2325$: as one increases the other decreases) and no strong correlation between x_0 and x_1 ($C_{01} = 0.0575$: x_0 and x_1 do not trend strongly together).

The *correlation coefficient matrix* is often used in preference to the covariance matrix as it is normalized by dividing C_{ij} by the product of the variables' standard deviations:

$$P_{ij} = \mathrm{corr}(x_i, x_j) = \frac{C_{ij}}{\sigma_i \sigma_j} = \frac{C_{ij}}{\sqrt{C_{ii} C_{jj}}}.$$

This means that the elements P_{ij} have values between -1 and 1 inclusive, and the diagonal elements, $P_{ii} = 1$. In our example, using `np.corrcoef` gives:

```
In [x]: print( np.corrcoef(X) )
[[ 1.          0.0874779  -0.95363007]
 [ 0.0874779   1.         -0.11878687]
 [-0.95363007 -0.11878687  1.        ]]
```

It is easy to see from this correlation coefficient matrix the strong anticorrelation between x_0 and x_2 ($C_{0,2} = -0.954$) and the lack of correlation between x_1 and the other variables (e.g., $C_{1,0} = 0.087$).

Both the `np.cov` and `np.corrcoef` methods can take a second array-like object containing a further set of variables and observations, so they can be called on a pair of one-dimensional arrays without stacking them into a single matrix:

```
In [x]: x = np.array([1., 2., 3., 4., 5.])
In [x]: y = np.array([0.08, 0.31, 0.41, 0.48, 0.62])
In [x]: print( np.corrcoef(x,y) )
[[ 1.          0.97787645]
 [ 0.97787645  1.        ]]
```

That is

```
np.corrcoef(x, y)
```

is a convenient alternative to

```
np.corrcoef(np.vstack((x,y)))
```

Finally, if your observations happen to be in the rows of your matrix, with the variables corresponding to the columns (instead of the other way round) there is no need

to transpose the matrix, just pass `rowvar=0` to either `np.cov` or `np.corrcoef` and NumPy will take care of it for you.

Example E6.9 The Cambridge University Digital Technology Group have been recording the weather from the roof of their department building since 1995 and make the data available to download in a single CSV file at www.cl.cam.ac.uk/research/dtg/weather/.

The following program determines the correlation coefficient between pressure and temperature at this site.

Listing 6.6 Calculating the correlation coefficient between air temperature and pressure

```
# eg6-pT.py
import numpy as np
import pylab

data = np.genfromtxt('weather-raw.csv', delimiter=',', usecols=(1,4))
# Remove any rows with either missing T or missing p
data = data[~np.any(np.isnan(data), axis=1)]
# Temperatures are reported after multiplication by a factor of 10 so remove
# this factor
data[:,0] /= 10

# Get the correlation coefficient
corr = np.corrcoef(data, rowvar=0)[0,1]
print('p-T correlation coefficient: {:.4f}'.format(corr))

# Plot the data on a scatterplot: T on x-axis, p on y-axis.
pylab.scatter(*data.T, marker='.')
pylab.xlabel('$T$ /$\mathrm{^\circ C}$')
pylab.ylabel('$p$ /mbar')
pylab.show()
```

The output (Figure 6.4) gives a correlation coefficient of 0.0260: as expected, there is little correlation between air temperature and pressure (since the air density also varies).

6.3.3 Histograms

The NumPy function, `np.histogram`, creates a histogram from the values in an array. That is, a set of *bins* is defined with lower and upper limits and each is filled with the number of elements from the array whose value falls within its limits. For example, suppose the following array holds the percentage marks of 10 students in a test:

```
In [x]: marks = np.array([45, 68, 56, 23, 60, 87, 75, 59, 63, 72])
```

There are several ways to define the histogram bins. If the `bins` argument is a sequence, it defines the boundaries of the sequential bins:

```
In [x]: bins = [20, 40, 60, 80, 100]
```

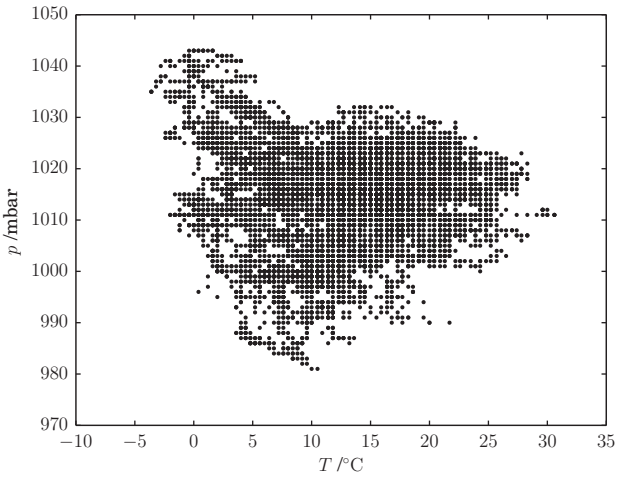


Figure 6.4 There is virtually no correlation between air temperature and air pressure in this data set.

defines four bins with ranges [20 – 40%), [40 – 60%), [60 – 80%) and [80 – 100%]. All but the last bin is half open; that is, the first bin includes marks from and including 20% up to but not including 40%. Note that a sequence of $N + 1$ numbers is required to create N bins. The `np.histogram` method returns a tuple consisting of the values of the histogram and the bin edges we defined (both as NumPy arrays).

```
In [x]: hist, bins = np.histogram(marks, bins)
In [x]: hist
Out[x]: array([1, 3, 5, 1])

In [x]: bins
Out[x]: array([ 20,  40,  60,  80, 100])
```

This shows that there is one mark in the 20 – 40% bin, three in the 40 – 60% bin and so on.

If you just want a certain number of evenly spaced bins, an integer can be passed as `bins` instead of a sequence:

```
In [x]: np.histogram(marks, bins=5)
Out[x]: (array([1, 1, 2, 4, 2]),
         array([ 23. ,  35.8,  48.6,  61.4,  74.2,  87. ]))
```

By default, the requested number of bins range between the minimum and maximum values of the array (here, `23` and `87`); to specify a different minimum and maximum, set the `range` argument tuple:

```
In [x]: np.histogram(marks, bins=5, range=(0,100))
Out[x]: (array([0, 1, 3, 5, 1]),
         array([   0.,   20.,   40.,   60.,   80.,  100.]))
```

The `np.histogram` method also has an optional boolean argument `density`: by default it is `False`, meaning that the histogram array returned contains the *number* of

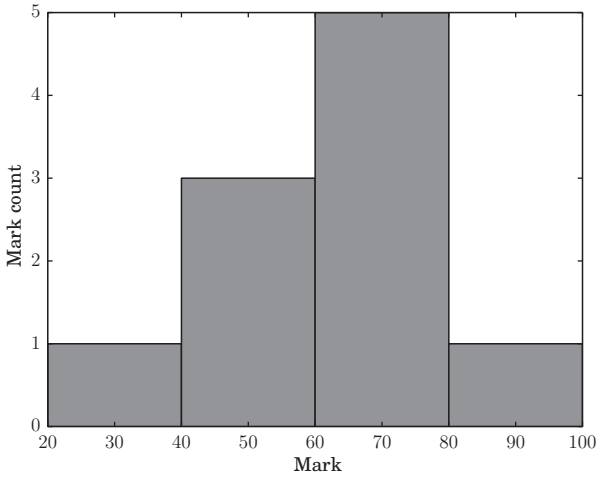


Figure 6.5 An example histogram.

values from the original array in each bin. If `density` is set to `True`, the histogram array will contain the *probability density function*, normalized so that the integral over the entire range of the bins is equal to unity:

```
In [x]: hist, bins = np.histogram(marks, bins=5, range=(0,100),
                                  density=True)
In [x]: print(hist)
[ 0.      0.005   0.015   0.025   0.005]
In [x]: bin_width = 100/5
In [x]: print(np.sum(hist) * bin_width)
1.0
```

(By integral here we mean the area under the histogram, which is the sum of each histogram bar height times its corresponding bin width.)

To plot a histogram with `pylab`, use `pylab.hist`, passing it the same arguments you would to `np.histogram`:[17]

❶
```
In [x]: hist, bins, patches = pylab.hist(marks, bins=5, range=(0,100))
In [x]: hist, bins
Out[x]:
(array([ 0.,  1.,  3.,  5.,  1.]),
 array([   0.,   20.,   40.,   60.,   80.,  100.]))
In [x]: pylab.show()
```

❶ In addition to the bin counts (`hist`) and boundaries (`bins`), `pylab` returns a list of references to the "patches" which appear in the plotted figure (see Section 7.1.5 for more information about this advanced feature).

The resulting histogram is plotted in Figure 6.5. See also Sections 3.3.2 and 7.1.2.

[17] Note that the `density` argument is not supported as of Matplotlib 1.3.1: instead, set `normed=True` for a probability density plot.

6.3.4 Exercises

Problems

P6.3.1 A certain lottery involves players selecting six numbers without replacement from the range [1,49]. The jackpot is shared among the players who match all six numbers ("balls") selected in the same way at random in a twice-weekly draw (in any order). If no player matches every drawn number, the jackpot "rolls over" and is added to the following draw's jackpot.

Although the lottery is *fair* in the sense that every combination of drawn numbers is equally likely, it has been observed that many players show a preference in their selection for certain numbers, such as those that represent dates (i.e., more of their numbers are chosen from [1,31] than would be expected if they chose randomly). Hence, to avoid sharing the jackpot and hence to maximize one's expected winnings, it would be reasonable to avoid these numbers.

Test this hypothesis by establishing if there is any correlation between the number of balls with values less than 13 (representing a month) and the jackpot winnings per person. Ignore draws immediately following a rollover. The necessary data can be downloaded from scipython.com/ex/afe.

P6.3.2 We have seen how to create a histogram plot from an array with `pylab.hist`, but suppose you have already created arrays `hist` and `bins` using `np.hist` and want to plot the resulting histogram from these arrays. You can't use `pylab.hist` because this function expects to act on the original array of data. Use `pylab.bar`[18] to plot a `hist` array as a bar chart.

P6.3.3 The heights, in cm, of a sample of 1,000 adult men and 1,000 adult women from a certain population are collected in the data files `ex6-3-f-male-heights.txt` and `ex6-3-f-female-heights.txt` available at scipython.com/ex/afd. Read in the data and establish the mean and standard deviation for each sex. Create histograms for the two data sets using a suitable binning interval and plot them on the same figure.

Repeat the exercise in imperial units (feet and inches).

6.4 Polynomials

NumPy provides a powerful set of classes for representing polynomials, including methods for evaluation, algebra, root-finding and fitting of several kinds of polynomial basis functions. In this section, the simplest and most familiar basis, the power series, will be described first, before a discussion of a few other classical orthogonal polynomial basis functions.

[18] Documentation for this method is at http://matplotlib.org/api/pyplot_api.html/matplotlib.pyplot.bar; see also Section 7.1.2.

6.4.1 Defining and evaluating a polynomial

A (finite) polynomial power series has as its basis the powers of x: $1(=x^0), x, x^2, x^3, \cdots, x^N$, with coefficients c_i:

$$P(x) = \sum_{i=0}^{N} c_0 + c_1 x + c_2 x^2 + c_3 x^3 + \cdots + c_N x^N$$

This section describes the use of the `Polynomial` *convenience class* which provides a natural interface to the underlying functionality of NumPy's `polynomial` package.

The polynomial convenience class is `numpy.polynomial.Polynomial`. To import it directly, use

```
In [x]: from numpy.polynomial import Polynomial
```

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

```
In [x]: import numpy as np
In [x]: Polynomial = np.polynomial.Polynomial
```

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

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

$$P(x) = 6 - 5x + x^2$$

define a the object

```
In [x]: p = Polynomial([6, -5, 1])
```

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

```
In [x]: print(p)
poly([ 6. -5.  1.])
In [x]; P.coef
Out[x]: array([ 6., -5.,  1.])
```

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

To evaluate a polynomial for a given value of x, "call" it as follows:

```
In [x]: p(4)    # calculate p at a single value of x
2.0
In [x]: x = np.linspace(-5, 5, 11)
In [x]: print(p(x))     # calculate p on a sequence of x values
Out[x]: [ 56.  42.  30.  20.  12.   6.   2.   0.   0.   2.   6.]
```

6.4.2 Polynomial algebra

The `Polynomial` convenience class implements the familiar Python operators: `+`, `-`, `*`, `//`, `**`, `%` and `divmod`[19] on `Polynomial` objects. These are illustrated in the following examples using the polynomials

$$P(x) = 6 - 5x + x^2$$

$$Q(x) = 2 - 3x$$

```
In [x]: p = Polynomial([6, -5, 1])
In [x]: q = Polynomial([2, -3])
In [x]: print(p + q)
poly([ 8. -8.  1.])

In [x]: print(p - q)
poly([ 4. -2.  1.])

In [x]: print(p * q)
poly([ 12. -28.  17.  -3.])

In [x]: print(p // q)
poly([ 1.44444444 -0.33333333])

In [x]: print(p % q)
poly([ 3.11111111])     # i.e. 28/9
```

Division of a polynomial by another polynomial is analogous to integer division (and uses the same `//` operator): that is, the result is another polynomial (with no reciprocal powers of x), possibly leaving a remainder.

Hence $p = q(-\frac{1}{3}x + \frac{13}{9}) + \frac{28}{9}$ and the `//` operator returns the quotient polynomial, $-\frac{1}{3}x + \frac{13}{9}$. The remainder (which, in general, will be another polynomial) is returned, as might be expected, by the modulus operator, `%`. The `divmod()` built-in returns both quotient and remainder in a tuple:

```
In [x]: quotient, remainder = divmod(p, q)
In [x]: print(quotient)

poly([ 1.44444444 -0.33333333])     # i.e. p(x) // q(x) is 13/9 - x/3

In [x]: print(remainder)
poly([ 3.11111111])
```

Exponentiation is supported through the `**` operator; polynomials can only be raised to a non-negative integer power:

```
In [x]: print(q ** 2)
poly([  4. -12.   9.])
```

It isn't always convenient to create a new polynomial object in order to use these operators on one another, so many of the operators described here also work with scalars:

[19] The `divmod` function returns the quotient and remainder of a division operation as a tuple.

```
In [x]: print(p * 2)    # multiplication by a scalar
poly([ 12. -10.   2.])
In [x]: print(p / 2)    # division by a scalar
poly([ 3.  -2.5  0.5])
```

and even tuples, lists and arrays of polynomial coefficients. For example, to multiply $P(x)$ by $x^2 - 2x^3$:

```
In [x]: print(p * [0, 0, 1, -2])
poly([  0.   0.   6. -17.  11.  -2.])
```

Finally, one polynomial can be substituted into another. To evaluate $P(Q(x))$, simply use `p(q)`:

```
In [x]: print(p(q))
poly([ 0.  3.  9.])
```

That is, $P(Q(x)) = 3x + 9x^2$.

6.4.3 Root-finding

The roots of a polynomial are returned by the `roots` method. Repeated roots are simply repeated in the returned array:

```
In [x]: p.roots()
array([ 2.,  3.])
In [x]: (q*q).roots()
array([ 0.66666667,  0.66666667])
In [x]: Polynomial([5, 4, 1]).roots()
array([-2.-1.j, -2.+1.j])
```

Polynomials can also be created from their roots with `Polynomial.fromroots`:

```
In [x]: print( Polynomial.fromroots([-4, 2, 1]) )
poly([  8. -10.   1.   1.])
```

That is, $(x + 4)(x - 2)(x - 1) = 8 - 10x + x^2 + x^3$. Note that the way the polynomial is constructed means that the coefficient of the highest power of x will be 1.

Example E6.10 The tanks used in the storage of cryogenic liquids and rocket fuel are often spherical (why?). Suppose a particular spherical tank has a radius R and is filled with a liquid to a height h. It is (fairly) easy to find a formula for the volume of liquid from the height:

$$V = \pi R h^2 - \frac{1}{3}\pi h^3.$$

Suppose that there is a *constant* flow of liquid from the tank at a rate $F = -\mathrm{d}V/\mathrm{d}t$. How does the height of liquid, h, vary with time? Differentiating the earlier mentioned equation with respect to t leads to

$$(2\pi R h - \pi h^2)\frac{\mathrm{d}h}{\mathrm{d}t} = -F.$$

If we start with a full tank ($h = 2R$) at time $t = 0$, this ordinary differential equation may be integrated to yield the equation

$$-\frac{1}{3}\pi h^3 + \pi R h^2 + \left(Ft - \frac{4}{3}\pi R^3\right) = 0,$$

a cubic polynomial in h. Because this equation cannot be inverted analytically for h, let's use NumPy's `Polynomial` class to find $h(t)$, given a tank of radius $R = 1.5$ m from which liquid is being drawn at 200 $\text{cm}^3\,\text{s}^{-1}$.

The total volume of liquid in the full tank is $V_0 = \frac{4}{3}\pi R^3$. Clearly, the tank is empty when $h = 0$, which occurs at time $T = V_0/F$, since the flow rate is constant. At any particular time, t, we can find h by finding the roots of this equation.

Listing 6.7 Liquid height in a spherical tank

```
# eg6-c-spherical-tank-a.py
import numpy as np
import pylab
Polynomial  = np.polynomial.Polynomial

# Radius of the spherical tank in m
R = 1.5
# Flow rate out of the tank, m^3.s-1
F = 2.e-4
# Total volume of the tank
V0 = 4/3 * np.pi * R**3
# Total time taken for the tank to empty
T = V0 / F

# coefficients of the quadratic and cubic terms
# of p(h), the polynomial to be solved for h
c2, c3 = np.pi * R, -np.pi / 3

N = 100
# array of N time points between 0 and T inclusive
time = np.linspace(0, T, N)
# create the corresponding array of heights h(t)
h = np.zeros(N)
for i, t in enumerate(time):
    c0 = F*t - V0
    p = Polynomial([c0, 0, c2, c3])
    # find the three roots to this polynomial
    roots = p.roots()
    # we want the one root for which 0 <= h <= 2R
    h[i] = roots[(0 <= roots) & (roots <= 2*R)][0]

pylab.plot(time, h, 'o')
pylab.xlabel('Time /s')
pylab.ylabel('Height in tank /m')
pylab.show()
```

❶ We construct an array of time points between $t = 0$ and $t = T$.

❷ For each time point find the roots of the above cubic polynomial. Only one of the roots is physically meaningful, in that $0 \leq h \leq 2R$ (the height of the level of liquid

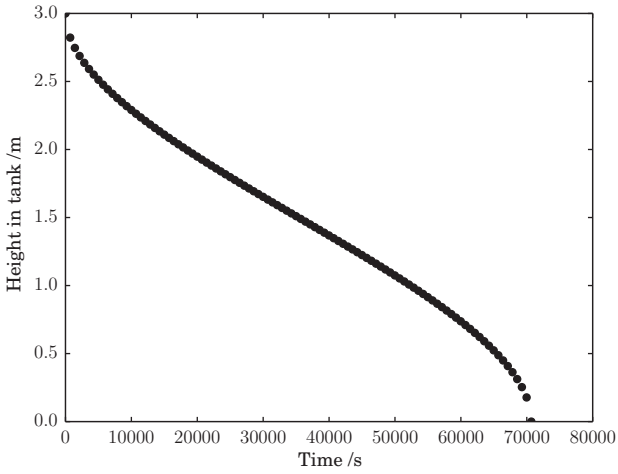


Figure 6.6 The height of liquid as a function of time, $h(t)$, for the spherical tank problem.

cannot be negative or greater than the diameter of the tank), so we extract that root (by boolean indexing) and store it in the array h.

Finally, we plot h as a function of time (Figure 6.6).

6.4.4 Calculus

Polynomials can be differentiated with the `Polynomial.deriv` method. By default, this function returns the first derivative, but the optional argument `m` can be set to return the mth derivative:

```
In [x]: print(p)
poly([ 6. -5.  1.]) # 6 - 5x + x^2
In [x]: print(p.deriv())
poly([-5.  2.])
In [x]: print(p.deriv(2))
poly([ 2.])
```

A `Polynomial` object can also be integrated with an optional lower bound, L, and constant of integration, k, treated as shown in the following example:

$$\int_L^x 2 - 3x\,\mathrm{d}x = \left[2x - \tfrac{3}{2}x^2\right]_L^x = 2x - \tfrac{3}{2}x^2 - 2L + \tfrac{3}{2}L^2$$

$$\int 2 - 3x\,\mathrm{d}x = 2x - \tfrac{3}{2}x^2 + k$$

By default, L and k are zero, but can be specified by passing the arguments `lbnd` and `k` to the `Polynomial.integ` method:

```
In [x]: print(q)
poly([ 2. -3.])
In [x]: print(q.integ())
poly([ 0.   2.  -1.5])
```

```
In [x]: print(q.integ(lbnd=1))
poly([-0.5  2.  -1.5])
In [x]: print(q.integ(k=2))
poly([ 2.   2.  -1.5])
```

Polynomials can be integrated repeatedly by passing a value to `m`, giving the number of integrations to perform.[20]

6.4.5 ◊ Classical orthogonal polynomials

In addition to the `Polynomial` class representing simple power series such as $a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$, NumPy provides classes to represent a series composed of any of a number of classical orthogonal polynomials. These polynomials and linear combinations of them are widely used in physics, statistics and mathematics. As of NumPy version 1.8, the polynomial convenience classes provided are `Chebyshev`, `Legendre`, `Laguerre`, `Hermite` ("physicists' version") and `HertmiteE` ("probabilists' version"). Many good textbooks exist describing the properties of these polynomial classes; to illustrate their use we will focus here on the Legendre polynomials,[21] denoted $P_n(x)$. These are the solutions to Legendre's differential equation,

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

The first few Legendre polynomials are

$$\begin{aligned} P_0(x) &= 1 \\ P_1(x) &= x \\ P_2(x) &= \tfrac{1}{2}(3x^2 - 1) \\ P_3(x) &= \tfrac{1}{2}(5x^3 - 3x) \\ P_4(x) &= \tfrac{1}{8}(35x^4 - 30x^2 + 3) \end{aligned}$$

and are plotted in Figure 6.7.

A useful property of the Legendre polynomials is their orthogonality on the interval [-1, 1]:

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

which is important in their use as a basis for representing suitable functions.[22]

To create a linear combination of Legendre polynomials, pass the coefficients to the `Legendre` constructor, just as for `Polynomial`. For example, to construct the polynomial expansion $5P_1(x) + 2P_2(x)$:

[20] Different constants of integration for each can be specified by setting `k` to an array of values.

[21] The Legendre Polynomials are named after the French mathematician Adrien-Marie Legendre (1752–1833); for 200 years until 2005 many publications mistakenly used a portrait of the unrelated French politician Louis Legendre as that of the mathematician.

[22] In particular, in physics, the multipole expansion of electrostatic potentials.

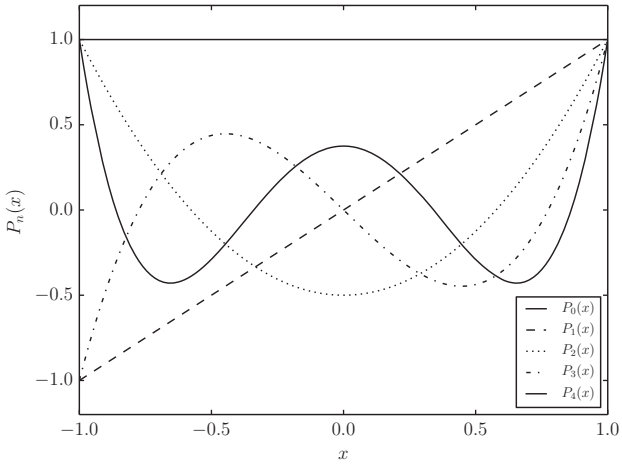


Figure 6.7 The first five Legendre Polynomials, $P_n(x)$ for $x = 0, 1, 2, 3, 4$.

```
In [x]: Legendre = np.polynomial.Legendre
In [x]: A = Legendre([0, 5, 2])
```

An existing polynomial object can be converted into a Legendre series with the `cast` method:

```
In [x]: P = Polynomial([0,1,1])
In [x]: Q = Legendre.cast(P)
In [x]: print(Q)
leg([ 0.33333333  1.          0.66666667])
```

That is, $x + x^2 = \frac{1}{3}P_0 + P_1 + \frac{2}{3}P_2$.

An instance of a single Legendre polynomial basis function can be created with the `basis` method:

```
In [x]: L3 = Legendre.basis(3)
```

creates an object representing $P_3(x)$, and is equivalent to calling `Legendre([0,0,0,1])`. To obtain a regular power series, we can cast it back to a `Polynomial`:

```
In [x]: print(Polynomial.cast(L3))
poly([ 0.  -1.5  0.   2.5])
```

In addition to the functions just described for `Polynomial`, including differentiation and integration of polynomial series, the convenience classes for the classical orthogonal polynomials expose several useful methods.

`convert` converts between different kinds of polynomials. For example, the linear combination $A(x) = 5P_1(x) + 2P_2(x) = 5x + 2\frac{1}{2}(3x^2 - 1) = -1 + 5x + 3x^2$, as a power series of monomials (a Maclaurin series), is represented by an instance of the `Polynomial` class as:

```
In [x]: A = Legendre([0, 5, 2])
In [x]: B = A.convert(kind=Polynomial)
In [x]: print(B)
In [x]: poly([-1.  5.  3.])
```

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

```
In [x]: A(-2) == B(-2)
Out[x]: True
In [x]: print(A.roots(), B.roots(), sep='\n')
[-1.84712709  0.18046042]
[-1.84712709  0.18046042]
```

6.4.6 Fitting polynomials

A common use of polynomial expansions is in fitting and approximating data series. NumPy's polynomial modules provide methods for the least squares fitting of functions. The `fit` function of the polynomial convenience classes is described in this section.[23]

The `domain` and `window` attributes

A typical one-dimensional fitting problem requires the best-fit polynomial to a finite, continuous function over some finite region of the x-axis (the *domain*). However, polynomials themselves can differ from each other wildly and diverge as $x \to \pm\infty$. This makes any attempt to blindly find the least squares fit on the domain of the function itself potentially risky: the fitted polynomial is frequently subject to numerical instability, overflow, underflow and other types of ill-conditioning (see Section 9.2). As an example, consider the function

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

in the interval (100, 100.1). There is nothing particularly tricky about this function: it is well-behaved everywhere and $f(x)$ takes very moderate values between e^{-1} and e^{1}. Yet a straightforward least squares fit to a fourth-order polynomial on this domain gives:

$$-11.881851 + 2379.22228x - 119.741202x^2 - 23828009.7x^3 + 1192894610x^4$$

and clearly the potential for numerical instability and loss of accuracy with even moderate values of x: our approximation to $f(x)$ is built up from difference between very large monomial terms.

Each class of polynomial has a default *window* over which it is optimal to take a linear combination in fitting a function. For example, the Legendre polynomials window is the region $[-1,1]$ plotted above, on which $P_n(x)$ are orthogonal and everywhere $|P_n(x)| < 1$. The problem is that it is rather unlikely that the function to be fitted falls

[23] *Note:* The older `np.poly1d` class representing one-dimensional polynomials is still available (as of NumPy 1.9) for backward-compatibility reasons. It is documented at http://docs.scipy.org/doc/numpy/reference/routines.polynomials.poly1d.html and provides a simpler but less reliable least squares fitting method, `polyfit`. It is recommended, however, to use the new `Polynomial` class in new code.

within the chosen polynomials' window. It is therefore necessary to relate the domain of the function to the window. This is done by shifting and scaling the x-axis: that is, by mapping points in the function's domain to points in the fitting polynomials' window. The polynomial `fit` function does this automatically, so the fourth-order least squares fit to the earlier mentioned function yields

```
In [x]: x = np.linspace(100, 100.1, 1001)
In [x]: f = lambda x: np.exp(-np.sin(40*x))
In [x]: p = Polynomial.fit(x, f(x), 4)
In [x]: print(p)
poly([ 1.49422551 -2.54641449  0.63284641  1.84246463 -1.02821956])
```

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

```
In [x]: p.domain
array([ 100. ,  100.1])
In [x]: p.window
array([-1.,  1.])
```

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

```
In [x]: q = Polynomial([1.49422551, -2.54641449,  0.63284641,
                        1.84246463, -1.02821956])
```

it is created with the default domain and window, which are *both* `(-1,1)`:

```
In [x]: print(q.domain, q.window)
[-1.  1.] [-1.  1.]
```

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

```
In [x]: q(100.05), p(100.05)
(-101176442.96772559, 1.4942255113760108)
```

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

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

These are the parameters returned by the polynomial's `mapparms` function:

```
In [x]: chi, mu = p.mapparms()
In [x]: print(chi, mu)
-2001.0, 20.0
```

Therefore,

```
In [x]: print(q(chi + mu*100.05))
1.49422551
```

It is possible to change `domain` and `window` by direct assignment:

```
In [x]: q.domain = np.array((100., 100.1))
In [x]: print(q(100.05))
1.49422551
```

To evaluate a polynomial on a set number of evenly distributed points in its domain, for example, to plot it, use the `Polynomial`'s `linspace` method:

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

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

`Polynomial.fit`

The `Polynomial` method `fit` returns a least squares fitted polynomial to data, `y`, sampled at values `x`. In its simplest use, `fit` needs only to be passed array-like objects `x` and `y`, and a value for `deg`, the degree of polynomial to fit. It returns the polynomial which minimizes the sum of the squared errors,

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

For example,

```
In [x]: x = np.linspace(400, 700, 1000)
In [x]: y = 1 / x**4
In [x]: p = Polynomial.fit(x, y, 3)
```

produces the best-fit cubic polynomial to the function x^{-4} on the interval $(400, 700)$.

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

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

The domain and window of the fitted polynomial may be specified with the arguments `domain` and `window`; by default a minimal domain covering the points `x` is used.

It is wise to check the *quality* of the fit before using the returned polynomial. Setting the argument `full=True` causes `fit` to return two objects: the fitted polynomial and a `list` of various statistics about the fit itself:

```
In [x]: deg = 3
In [x]: p, [resid, rank, sing_val, rcond] = Polynomial.fit(x, y, deg, full=True)
In [x]: p
Out[x]:
Polynomial([  1.07041864e-11,  -1.16488662e-11,   1.02545751e-11,
        -5.64068914e-12], [ 400.,  700.], [-1.,  1.])
```

```
In [x]: resid
Out[x]: array([  4.57180972e-23])

In [x]: rank
Out[x]: 4

In [x]: sing_val
Out[x]: array([ 1.3843828 ,  1.32111941,  0.50462215,  0.28893641])

In [x]: rcond
Out[x]: 2.2204460492503131e-13
```

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

$$\texttt{resid} = \sum_i |y_i - p(x_i)|^2$$

– a smaller value indicates a better fit. `rank` and `sing_val` are the rank and singular values of the matrix inverted in the least squares algorithm to find the polynomial coefficients: ill-conditioning of this matrix can lead to poor fits (particularly if the fitted polynomial degree is too high). `rcond` is the cutoff ratio for small singular values within this matrix: values smaller than this value are set to zero in the fit (to protect the fit from spurious artifacts introduced by round-off error) and a `RankWarning` exception is raised. If this happens, the data may be too noisy or not well described by the polynomial of the specified degree. Note that least squares fitting should always be carried out at double precision and be aware of "over-fitting" the data (attempting to fit a function with too many coefficients, i.e., a polynomial of too high order).

Example E6.11 A straight-line best fit is just a special case of a polynomial least squares fit (with `deg=1`). Consider the following data giving the absorbance over a path length of 5 mm of UV light at 280 nm, A, by a protein as a function of the concentration, [P]:

[P] / μg/mL	A
0	2.287
20	3.528
40	4.336
80	6.909
120	8.274
180	12.855
260	16.085
400	24.797
800	49.058
1500	89.400

We expect the absorbance to be linearly related to the protein concentration: $A = m[\mathrm{P}] + A_0$ where A_0 is the absorbance in the absence of protein (e.g., due to the solvent and experimental components).

Listing 6.8 Straight line fit to absorbance data

```
# eg6-polyfit.py
import numpy as np
import pylab
Polynomial = np.polynomial.Polynomial

# The data: conc = [P] and absorbance, A
conc = np.array([0, 20, 40, 80, 120, 180, 260, 400, 800, 1500])
A = np.array([2.287, 3.528, 4.336, 6.909, 8.274, 12.855, 16.085, 24.797,
              49.058, 89.400])

cmin, cmax = min(conc), max(conc)
pfit, stats = Polynomial.fit(conc, A, 1, full=True, window=(cmin, cmax),
                                                    domain=(cmin, cmax))

print('Raw fit results:', pfit, stats, sep='\n')

A0, m = pfit
resid, rank, sing_val, rcond = stats
rms = np.sqrt(resid[0]/len(A))

print('Fit: A = {:.3f}[P] + {:.3f}'.format(m, A0),
      '(rms residual = {:.4f})'.format(rms))

pylab.plot(conc, A, 'o', color='k')
pylab.plot(conc, pfit(conc), color='k')
pylab.xlabel('[P] /$\mathrm{\mu g\cdot mL^{-1}}$')
pylab.ylabel('Absorbance')
pylab.show()
```

The output shows a good straight-line fit to the data (Figure 6.8):

```
Raw fit results:
poly([ 1.92896129  0.0583057 ])
[array([ 2.47932733]), 2, array([ 1.26633786,  0.62959385]), 2.2204460492503131e-15]
Fit: A = 0.058[P] + 1.929 (rms residual = 0.4979)
```

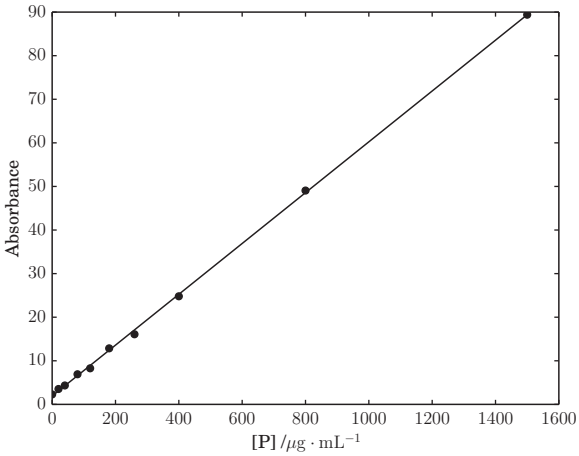


Figure 6.8 Line of least squares best fit to absorbance data as a function of concentration.

Table 6.6 Radius of the ball of fire produced by the "Trinity" nuclear test as a function of time

t /ms	R /m	t /ms	R /m	t /ms	R /m
0.1	11.1	1.36	42.8	4.34	65.6
0.24	19.9	1.50	44.4	4.61	67.3
0.38	25.4	1.65	46.0	15.0	106.5
0.52	28.8	1.79	46.9	25.0	130.0
0.66	31.9	1.93	48.7	34.0	145.0
0.80	34.2	3.26	59.0	53.0	175.0
0.94	36.3	3.53	61.1	62.0	185.0
1.08	38.9	3.80	62.9		
1.22	41.0	4.07	64.3		

Note: This data can be downloaded from scipython.com/ex/afg.

6.4.7 Exercises

Questions

Q6.4.1 The third derivative of the polynomial function $P(x) = 3x^3 + 2x - 7$ is 18, so why does the following evaluate as `False`?

```
In [x]: Polynomial((-7, 2, 0, 3)).deriv(3) == 18
Out[x]: False
```

Q6.4.2 Find and classify the stationary points of the polynomial

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

Problems

P6.4.1 The expansion of the spherical ball of fire generated in an explosion may be analyzed to deduce the initial energy, E, released by a nuclear weapon. The British physicist Geoffrey Taylor used dimensional analysis to demonstrate that the radius of this sphere, $R(t)$, should be related to E, the air density, ρ_{air}, and time, t, through

$$R(t) = CE^{\frac{1}{5}}\rho_{\text{air}}^{-\frac{1}{5}}t^{-\frac{2}{5}},$$

where, using model-shock wave problems, Taylor estimated the dimensionless constant $C \approx 1$. Using the data obtained from declassified timed images of the first New Mexico atomic explosion, Taylor confirmed this law and produced an estimate of the (then unknown) value of E. Use a log-log plot to fit the data in Table 6.6[24] to the model and confirm the time-dependence of R. Taking $\rho_{\text{air}} = 1.25\ \text{kg m}^{-3}$ deduce E and express its value in Joules and in "kilotons of TNT" where the explosive energy released by 1 ton of TNT is arbitrarily defined to be 4.184×10^9 J.

P6.4.2 Find the mean and variance of both x and y, the correlation coefficient and the equation of the linear regression line for each of the four data sets given in Table 6.7. Comment on these values in the light of a plot of the data.

[24] G. I. Taylor, (1950) *Proc. Roy. Soc. London* **A201**, 159.

Table 6.7 Four sample data sets for analysis of mean, variance and correlation

x_1	y_1	x_2	y_2	x_3	y_3	x_4	y_4
10.0	8.04	10.0	9.14	10.0	7.46	8.0	6.58
8.0	6.95	8.0	8.14	8.0	6.77	8.0	5.76
13.0	7.58	13.0	8.74	13.0	12.74	8.0	7.71
9.0	8.81	9.0	8.77	9.0	7.11	8.0	8.84
11.0	8.33	11.0	9.26	11.0	7.81	8.0	8.47
14.0	9.96	14.0	8.10	14.0	8.84	8.0	7.04
6.0	7.24	6.0	6.13	6.0	6.08	8.0	5.25
4.0	4.26	4.0	3.10	4.0	5.39	19.0	12.50
12.0	10.84	12.0	9.13	12.0	8.15	8.0	5.56
7.0	4.82	7.0	7.26	7.0	6.42	8.0	7.91
5.0	5.68	5.0	4.74	5.0	5.73	8.0	6.89

These data can be downloaded as the file `ex6-4-a-anscombe.tex` from scipython.com/ex/aff.

P6.4.3 The van der Waals equation of state may be written as follows to give the pressure, p, of a gas from its molar volume, V, and temperature, T:

$$p = \frac{RT}{V-b} - \frac{a}{V^2},$$

where a and b are molecule-specific constants and $R = 8.314\ \mathrm{J\,K^{-1}\,mol^{-1}}$ is the gas constant. It can readily be rearranged to yield the temperature for a given pressure and volume, but its form giving the molar volume in terms of pressure and temperature is a cubic equation:

$$pV^3 - (pb + RT)V^2 + aV - ab = 0$$

Of the three roots to this equation, below the *critical point*, (T_c, p_c) all are real: the largest and smallest give the molar volume of the gas phase and liquid phase respectively; above the critical point, where no liquid phase exists, only one root is real and gives the molar volume of the gas (also known in this region as a *supercritical fluid*). The critical point is given by the condition $(\partial p/\partial V)_T = (\partial^2 p/\partial V^2)_T = 0$ and for a van der Waals gas is given by the formulas

$$T_c = \frac{8a}{27Rb}, \qquad p_c = \frac{a}{27b^2}$$

For ammonia the van der Waals constants are $a = 4.225\ \mathrm{L^2\,bar\,mol^{-2}}$ and $b = 0.03707\ \mathrm{L\,mol^{-1}}$.

a. Find the critical point of ammonia, and then determine the molar volume at room temperature and pressure, (298 K, 1 atm) and at (500 K, 12 MPa).
b. An *isotherm* is the set of points (p, V) at a constant temperature satisfying an equation of state. Plot the isotherm (p against V) for ammonia at 350 K using the

van der Waals equation of state and compare it with the 350 K isotherm for an ideal gas, which has the equation of state $p = RT/V$.

P6.4.4 The first-stage rockets of the Saturn V rocket that launched the Apollo 11 mission generated an acceleration which increases with time throughout their operation (mostly because of the decrease in mass as it burns its fuel). This acceleration may be modeled (in units of $\mathrm{m\,s^{-2}}$) as a function of time after launch, t in seconds, by the quadratic function:

$$a(t) = 2.198 + (2.842 \times 10^{-2})t + (1.061 \times 10^{-3})t^2$$

Determine the distance traveled by the rocket at the end of the stage-one center-engine burn, 2 minutes, 15.2 seconds, after launch.

(Harder) Assuming a constant lapse rate of $\Gamma = -\mathrm{d}T/\mathrm{d}z = 6\ \mathrm{K\,km^{-1}}$ and a ground temperature of 302 K, at what time and altitude, z, did the rocket achieve Mach 1? During the relevant phase of the launch, take the average pitch angle to be 12°, and assume the speed of sound can be calculated as a function of absolute temperature to be

$$c = \sqrt{\frac{\gamma RT}{M}},$$

where the constant $\gamma = 1.4$ and the mean molar mass of the atmosphere is $M = 0.0288\ \mathrm{kg\,mol^{-1}}$.

6.5 Linear algebra

6.5.1 Basic matrix operations

Although NumPy does have a `matrix` object (see Section 6.6), all the same matrix operations can be carried out on a regular two-dimensional NumPy array. These include scalar multiplication, matrix (dot) product, elementwise multiplication and transpose:

```
In [x]: A = np.array([[0, 0.5], [-1, 2]])
In [x]: A
Out[x]:
array([[ 0. ,  0.5],
       [-1. ,  2. ]])
In [x]: A * 5               # multiplication by a scalar
Out[x]:
array([[  0. ,   2.5],
       [ -5. ,  10. ]])
In [x]: B = np.array([[2, -0.5], [3, 1.5]])

In [x]: B
Out[x]:
array([[ 2. , -0.5],
       [ 3. ,  1.5]])

In [x]: A.dot(B)            # or np.dot(A,B): matrix product
```

```
Out[x]:
array([[ 1.5 ,  0.75],
       [ 4.  ,  3.5 ]])

In [x]: A * B                # elementwise multiplication
Out[x]:
array([[ 0.  , -0.25],
       [-3.  ,  3.  ]])
In [x]: A.transpose()        # or simply A.T
Out[x]:
array([[ 0. , -1. ],
       [ 0.5,  2. ]])
```

Note that the transpose returns a *view* on the original matrix.

The identity matrix is returned by passing the two dimensions of the matrix to the method `np.eye`:

```
In [x]: np.eye(3,3)
Out[x]:
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

Matrix products

NumPy contains further methods for vector and matrix products. For example,

```
In [x]: a = np.array([1,2,3])
In [x]: b = np.array([0,1,2])
In [x]: np.inner(a,b)           # inner product; here, the same as a.dot(b)
Out[x]: 8

In [x]: np.outer(a,b)           # outer product
Out[x]:
array([[0, 1, 2],
       [0, 2, 4],
       [0, 3, 6]])
```

To raise a matrix to an (integer) power, however, requires a method from the `np.linalg` module:

```
In [x]: A = np.array([[0, 0.5], [-1, 2]])
In [x]: np.linalg.matrix_power(A, 3)        # the same as A.dot(A.dot(A))
Out[x]:
array([[-1.  ,  1.75],
       [-3.5 ,  6.  ]])
```

Note that the `**` operator performs *elementwise* exponentiation:

```
In [x]: A**3                    # the same as A * A * A
Out[x]:
array([[ 0.   ,  0.125],
       [-1.   ,  8.   ]])
```

Other matrix properties

The norm of a matrix or vector is returned by the function `np.linalg.norm`. It is possible to calculate several different norms (see the documentation), but the ones used

by default, are the Frobenius norm for two-dimensional arrays:

$$||A|| = \left(\sum_{i,j} |a_{ij}|^2 \right)^{1/2}$$

and the Euclidean norm for one-dimensional arrays:

$$||a|| = \left(\sum_{i} |z_i|^2 \right)^{1/2} = \sqrt{|z_0|^2 + |z_1|^2 + \cdots + |z_{n-1}|^2}.$$

Thus,

```
In [x]: np.linalg.norm(A)
Out[x]: 2.2912878474779199

In [x]: c = np.array([1, 2j, 1 - 1j])
In [x]: np.linalg.norm(c)
Out[x]: 2.6457513110645907      # sqrt(1 + 4 + 2)
```

The function `np.linalg.det` returns the determinant of a matrix, and the regular NumPy function `np.trace` returns its trace (the sum of its diagonal elements):

```
In [x]: np.linalg.det(A)
Out[x]: 0.5

In [x]: np.trace(A)
Out[x]: 2.0
```

The *rank* of a matrix is obtained using `np.linalg.matrix_rank`:

```
In [x]: np.linalg.matrix_rank(A)        # matrix A has full rank
Out[x]: 2
In [x]: D = np.array([[1,1],[2,2]])     # a rank deficient matrix

In [x]: np.linalg.matrix_rank(D)
Out[x]: 1
```

To find the inverse of a square matrix, use `np.linalg.inv`. A `LinAlgError` exception is raised if the matrix inversion fails:

```
In [x]: np.linalg.inv(A)
Out[x]:
array([[ 4., -1.],
       [ 2.,  0.]])

In [x]: np.linalg.inv(D)
...
LinAlgError: Singular matrix
```

6.5.2 Eigenvalues and eigenvectors

To calculate the eigenvalues and (right) eigenvectors of a general square array with shape `(n,n)`, use `np.linalg.eig`, which returns the eigenvalues, `w`, as an array of shape `(n,)` and the normalized eigenvectors, `v`, as a complex array of shape `(n,n)`.

The eigenvalues are not returned in any particular order, but the eigenvalue `w[i]` corresponds to the eigenvector `v[:,i]`. Note that the eigenvectors are arranged in columns. If the eigenvalue calculation does not converge for some reason a `LinAlgError` is raised.

```
In [x]: vals, vecs = np.linalg.eig(A)
In [x]: vals
Out[x]: array([ 0.29289322,  1.70710678])

❶ In [x]: np.isclose(np.sum(vals), A.trace())
Out[x]: True

In [x]: vecs
Out[x]:
array([[-0.86285621, -0.28108464],
       [-0.50544947, -0.95968298]])
```

❶ Verify that the sum of the eigenvalues is equal to the matrix trace.

If the matrix is Hermitian or real-symmetric, the function `np.linalg.eigh` may be used instead. This method takes an additional argument, `UPLO`, which can be `'L'` or `'U'` according to whether the lower or upper triangular part of the matrix is used. The default is `'L'`.

Two additional methods, `np.linalg.eigvals` and `np.linalg.eigvalsh`, return *only the eigenvalues* (and not the eigenvectors) of a general and Hermitian matrix respectively.

Since NumPy version 1.8, these and most other `linalg` methods follow the usual broadcasting rules so that several matrices can be operated on at once: each matrix is assumed to be stored in the last two dimensions. For example, we may work with an array with shape `(3,2,2)` representing the three 2×2 Pauli matrices:

$$\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad \sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \quad \sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

```
In [x]: pauli_matrices = np.array((
                ((0, 1), (1, 0)),       # sigma_x
                ((0, -1j), (1j, 0)),    # sigma_y
                ((1, 0), (0, -1))       # sigma_z
                ))
In [x]: np.linalg.eigh(pauli_matrices)
Out[x]:
(array([[-1.,  1.],
        [-1.,  1.],
        [-1.,  1.]]),
 array([[[-0.70710678+0.j        ,  0.70710678+0.j        ],
         [ 0.70710678+0.j        ,  0.70710678+0.j        ]],

        [[-0.70710678-0.j        , -0.70710678+0.j        ],
         [ 0.00000000+0.70710678j,  0.00000000-0.70710678j]],

        [[ 0.00000000+0.j        ,  1.00000000+0.j        ],
         [ 1.00000000+0.j        ,  0.00000000+0.j        ]]]))
```

6.5.3 Solving equations

Linear scalar equations

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

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

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

$$\begin{pmatrix} m_{11} & m_{12} & \cdots & m_{1n} \\ m_{21} & m_{22} & \cdots & m_{2n} \\ \vdots & & \ddots & \vdots \\ m_{n1} & m_{n2} & \cdots & m_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}.$$

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

$$\begin{aligned} 3x - 2y &= 8 \\ -2x + y - 3z &= -20 \\ 4x + 6y + z &= 7 \end{aligned}$$

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

$$\begin{pmatrix} 3 & -2 & 0 \\ -2 & 1 & -3 \\ 4 & 6 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 8 \\ -20 \\ 7 \end{pmatrix}$$

and solved by passing arrays corresponding to matrix **M** and vector **b** to `np.linalg.solve`:

```
In [x]: M = np.array([[3,-2,0],[-2,1,-3],[4,6,1]])
In [x]: b = np.array([8,-20,7])
In [x]: np.linalg.solve(M, b)
Out[x]: array([ 2., -1.,  5.])
```

That is, $x = 2, y = -1, z = 5$.

If no unique solution exists (for nonsquare or singular matrix, **M**), a `LinAlgError` is raised.

Linear least squares solutions ("best fit")

Where a set of equations, $\mathbf{Mx} = \mathbf{b}$, does not have a unique solution, a least squares solution that minimizes the L^2 norm, $||b - Mx||^2$ (sum of squared residuals) may be sought using the `np.linalg.lstsq` method. This is the type of problem described as *over-determined* (more data points than the two unknown quantities, m and c). Passed

`M` and `b`, `np.linalg.lstsq` returns the solution array `x`, the sum of squared residuals, the rank of `M` and the singular values of `M`.

A typical use of this method is to find the "line of best-fit", $y = mx + c$, through some data thought to be linearly related as in the following example.

Example E6.12 The Beer-Lambert Law relates the concentration, c, of a substance in a solution sample to the intensity of light transmitted through the sample, I_t across a given path length, l, at a given wavelength, λ:

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

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

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

$$y = \ln \frac{I_t}{I_0} = -\alpha c l.$$

Although this line passes through the origin ($y = 0$ for $c = 0$), we will fit the more general linear relationship:

$$y = mc + k$$

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

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

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

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

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

where $T = \ln(I_t/I_0)$. The code here determines m and hence α using `np.linalg.lstsq`:

Listing 6.9 Linear least squares fitting of the Beer-Lambert Law

```
# eg6-beer-lambert-lstsq.py
import numpy as np
import pylab
```

```
# Path length, cm
path = 0.8
# The data: concentrations (M) and It/I0
c = np.array([0.4, 0.6, 0.8, 1.0, 1.2])
It_over_I0  = np.array([ 0.891 ,  0.841,  0.783,  0.744,  0.692])

n = len(c)
A = np.vstack((c, np.ones(n))).T
T = np.log(It_over_I0)

❶ x, resid, _, _ = np.linalg.lstsq(A, T)
m, k = x
alpha = - m / path
print('alpha = {:.3f} M-1.cm-1'.format(alpha))
print('k =', k)
print('rms residual = ', np.sqrt(resid[0]))

pylab.plot(c, T, 'o')
pylab.plot(c, m*c + k)
pylab.xlabel('$c\;/\mathrm{M}$')
pylab.ylabel('$\ln(I_\mathrm{t}/I_0)$')
pylab.show()
```

❶ Here, _ is the dummy variable name conventionally given to an object we do not need to store or use.

The output produces a best fit value of $\alpha = 0.393\ \mathrm{M}^{-1}\,\mathrm{cm}^{-1}$ and a value of k compatible with experimental error:

```
alpha = 0.393 M-1.cm-1
k = 0.0118109033334
rms residual =  0.0096843591966
```

Figure 6.9 shows the data and fitted line.

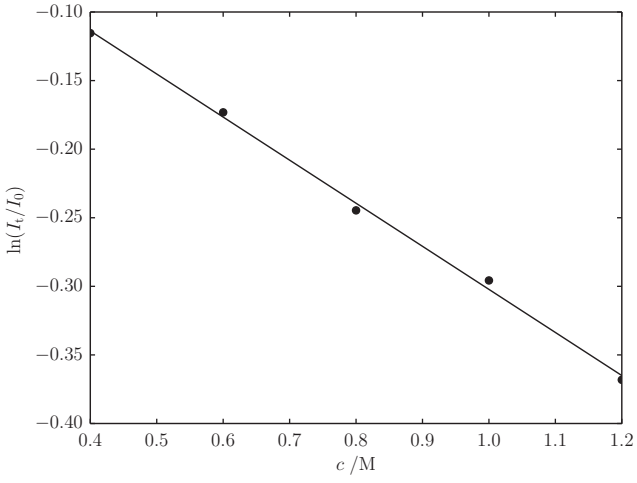


Figure 6.9 Line of least squares best fit to absorbance data as a function of concentration.

6.5.4 Exercises

Questions

Q6.5.1 Demonstrate that the three Pauli matrices given in Section 6.5.2 are *unitary*. That is, that $\sigma_p^\dagger \sigma_p = I_2$ for $p = x, y, z$, where I_2 is the 2×2 identity matrix and $\dagger$ denotes the Hermitian conjugate (conjugate transpose).

Q6.5.2 The ticker timer, much used in school physics experiments, is a device that marks dots on a strip of paper tape at evenly spaced intervals of time as the tape moves through it at some (possibly variable) speed. The following data relate to the positions (in cm) of marks on a tape pulled through a ticker timer by a falling weight. The marks are made every 1/10 sec.

```
x = [1.3, 6.0, 20.2, 43.9, 77.0, 119.6, 171.7, 233.2, 304.2, 384.7,
     474.7, 574.1, 683.0, 801.3, 929.2, 1066.4, 1213.2, 1369.4, 1535.1,
     1710.3, 1894.9]
```

Fit these data to the function $x = x_0 + v_0 t + \frac{1}{2} g t^2$ and determine an approximate value for the acceleration due to gravity, g.

Problems

P6.5.1 In physics, the *Planck units* of measurement are those defined such that the five universal physical constants, c (the speed of light), G (the gravitational constant), $\hbar$ (the reduced Planck constant), $(4\pi\epsilon_0)^{-1}$ (the Coulomb constant) and k_B (the Boltzmann constant) are set to unity. The dimensions of these quantities in terms of length (L), mass (M), time (T), charge (Q) and thermodynamic temperature (Θ) are given in Table 6.8, along with their values in SI units.

This suggests the following matrix relationship between the constants and their dimensions:

$$\begin{array}{c} \\ c \\ G \\ \hbar \\ (4\pi\epsilon_0)^{-1} \\ k_\mathrm{B} \end{array} \begin{array}{c} \begin{array}{ccccc} \mathrm{L} & \mathrm{M} & \mathrm{T} & \mathrm{Q} & \Theta \end{array} \\ \begin{pmatrix} 1 & 0 & -1 & 0 & 0 \\ 3 & -1 & -2 & 0 & 0 \\ 2 & 1 & -1 & 0 & 0 \\ 3 & 1 & -2 & -2 & 0 \\ 2 & 1 & -2 & 0 & -1 \end{pmatrix} \end{array}$$

Table 6.8 Some physical constants and their dimensions

c	Speed of light	$2.99792458 \times 10^8\ \mathrm{m\,s^{-1}}$	$\mathrm{L\,T^{-1}}$
G	Gravitational constant	$6.67384 \times 10^{-11}\ \mathrm{m^3\,kg^{-1}\,s^{-2}}$	$\mathrm{L^3\,M^{-1}\,T^{-2}}$
$\hbar$	Reduced Planck constant	$1.054571726 \times 10^{-34}\ \mathrm{J\,s}$	$\mathrm{L^2\,M\,T^{-1}}$
$(4\pi\epsilon_0)^{-1}$	Coulomb constant	$8.9875517873681764 \times 10^9\ \mathrm{N\,m^2\,C^{-2}}$	$\mathrm{L^3\,M\,T^{-2}\,Q^{-2}}$
k_B	Boltzmann constant	$1.3806488 \times 10^{-23}\ \mathrm{J\,K^{-1}}$	$\mathrm{L^2\,M\,T^{-2}\,\Theta^{-1}}$

Using the inverse of this matrix, determine the SI values of length, mass, time, charge and temperature in the base Planck units; that is, the combination of these physical constants yielding the dimensions L, M, T, Q and Θ. For example, the *Planck length* is found to be $l_{\mathrm{P}} = \sqrt{\hbar G/c^3} = 1.616199 \times 10^{-35}$ m.

P6.5.2 The (symmetric) matrix representing the inertia tensor of a collection of masses, m_i, with positions (x_i, y_i, z_i) relative to their center of mass is

$$\mathbf{I} = \begin{pmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{xy} & I_{yy} & I_{yz} \\ I_{xz} & I_{yz} & I_{zz} \end{pmatrix},$$

where

$$I_{xx} = \sum_i m_i(y_i^2 + z_i^2), \qquad I_{yy} = \sum_i m_i(x_i^2 + z_i^2), \qquad I_{zz} = \sum_i m_i(x_i^2 + y_i^2),$$
$$I_{xy} = -\sum_i m_i x_i y_i, \qquad I_{yz} = -\sum_i m_i y_i z_i, \qquad I_{xz} = -\sum_i m_i x_i z_i.$$

There exists a transformation of the coordinate frame such that this matrix is diagonal: the axes of this transformed frame are called the *principal* axes and the diagonal inertia matrix elements, $I_a \leq I_b \leq I_c$, are the *principal moments of inertia*.

Write a program to calculate the principal moments of inertia of a molecule, given the position and masses of its atoms *relative to some arbitrary origin*. Your program should first relocate the atom coordinates relative to its center of mass and then determine the principal moments of inertia as the eigenvalues of the matrix **I**.

A molecule may be classified as follows according to the relative values of I_a, I_b and I_c:

- $I_a = I_b = I_c$: spherical top;
- $I_a = I_b < I_c$: oblate symmetric top;
- $I_a < I_b = I_c$: prolate symmetric top;
- $I_a < I_b < I_c$: asymmetric top.

Determine the principal moments of inertia and classify the molecules NH_3, CH_4, CH_3Cl and O_3 given the data available at scipython.com/ex/afh. Also determine the *rotational constants*, A, B and C, related to the moments of inertia through $Q = h/(8\pi^2 c I_q)$ $(Q = A, B, C; q = a, b, c)$ and usually expressed in cm^{-1}.

◊ **P6.5.3** The NumPy method `numpy.linalg.svd` returns the *singular value decomposition* (SVD) of a matrix, **M**, as the arrays **U**, Σ and **V** satisfying the factorization: $\mathbf{M} = \mathbf{U}\Sigma\mathbf{V}^\dagger$ where † denotes the Hermitian conjugate (the conjugate transpose).

The SVD and the eigendecomposition are related in that the left-singular row vectors, **U** are the eigenvectors of $\mathbf{M}\mathbf{M}^*$ and the right-singular column vectors, **V**, are the eigenvectors of $\mathbf{M}^*\mathbf{M}$. Furthermore, the diagonal entries of Σ are the square roots of the nonzero eigenvalues of both $\mathbf{M}\mathbf{M}^*$ and $\mathbf{M}\mathbf{M}^*$.

Show that this is the case for the special case of $\mathbf{M}$ a 3×3 matrix with random real entries by comparing the output of `numpy.linalg.svd` with that of `numpy.linalg.eig`.

Hint: the singular values of $\mathbf{M}$ are sorted in descending order, but the eigenvalues returned by `numpy.linalg.eig` are in no particular order. Both methods produce normalized eigenvectors, but may differ by sign (ignore the possibility that any of the eigenvalues could have an eigenspace with dimension greater than 1).

6.6 Matrices

The NumPy `matrix` class is a subclass of the regular `ndarray` which provides some convenient functionality for dealing with matrices. There are some important differences from conventional arrays, and care should be taken in using some of the familiar array operations as they have been overridden in the matrix subclass and behave differently.

A matrix is *always a two-dimensional array*. Even a row or column matrix has two dimensions (with shape `(1,n)` or `(n,1)` respectively), and flattening a matrix with `flatten` or `ravel` (see Section 6.1.5) returns a `(1,n)` array rather than a one-dimensional array.

6.6.1 Creating a matrix

As an alternative to the regular `ndarray` construction methods, a `matrix` object can be created using the MATLAB-like syntax using a string of values in which columns are separated by spaces and rows by semicolons:

```
In [x]: A = np.matrix([[0, -1], [1, -2]])     # as for np.array()
In [x]: B = np.matrix('0 -1; 1 -2')           # MATLAB-like
In [x]: print(B)
[[ 0 -1]
 [ 1 -2]]
```

The data type of the matrix can be set with the `dtype` attribute as for regular arrays. If a matrix is created from an existing `ndarray` object, the default behavior is to *copy* the data into the new matrix object; to construct a matrix consisting of a *view* on an existing `ndarray`'s data, set the attribute `copy=False`:

```
In [x]: a = np.array([[1, 2], [3, 4]])
In [x]: A = np.matrix(a, copy=False)
In [x]: B = np.matrix(a)
In [x]: a[0,0] = -1
In [x]: print(A[0,0], B[0,0])
-1 1
```

That is, `A` is updated by the assignment `a[0,0]= -1`, but `B` owns its own data and is not updated. Special matrices such as the identity matrix are best created using the corresponding `ndarray` constructor and passing the resulting array object to `matrix` with `copy=False`:

```
In [x]: I = np.matrix(np.eye(2,2), copy=False)
In [x]: N = np.matrix(np.zeros((2,2)), copy=False)
In [x]: W = np.matrix(np.ones((2,2)), copy=False)
```

Example E6.13 One way to create the two-dimensional rotation matrix,

$$\mathbf{R} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

which rotates points in the *xy* plane counterclockwise through $\theta = 30°$ about the origin:

```
In [x]: theta = np.radians(30)
In [x]: c, s = np.cos(theta), np.sin(theta)
In [x]: R = np.matrix('{} {}; {} {}'.format(c, -s, s, c))
In [x]: print(R)
[[ 0.8660254 -0.5      ]
 [ 0.5        0.8660254]]
```

6.6.2 Matrix operations

The most important difference between `matrix` objects and arrays is in the behavior of the `*` and `**` operators. As we have seen, these act *elementwise* on `ndarrays`:

```
In [x]: a = np.array([[0, -1], [1, -2]])
In [x]: a * a       # arrays: elementwise (Hadamard) product
Out[x]:
array([[0, 1],
       [1, 4]])
In [x]: a ** 3      # arrays: elementwise exponentiation
Out[x]:
array([[ 0, -1],
       [ 1, -8]])
```

That is,

$$\begin{pmatrix} 0 & -1 \\ 1 & -2 \end{pmatrix} \circ \begin{pmatrix} 0 & -1 \\ 1 & -2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 4 \end{pmatrix},$$

$$\begin{pmatrix} 0^3 & -1^3 \\ 1^3 & -2^3 \end{pmatrix} = \begin{pmatrix} 0 & -1 \\ 1 & -8 \end{pmatrix}.$$

For `matrix` objects these operators are *matrix* multiplication and exponentiation:

```
In [x]: A = np.matrix([[0, -1], [1, -2]])
In [x]: A * A       # matrix multiplication
matrix([[-1,  2],
        [-2,  3]])
In [x]: A ** 3      # ie A.A.A
matrix([[ 2, -3],
        [ 3, -4]])
```

That is,

$$\begin{pmatrix} 0 & -1 \\ 1 & -2 \end{pmatrix}\begin{pmatrix} 0 & -1 \\ 1 & -2 \end{pmatrix} = \begin{pmatrix} -1 & 2 \\ -2 & 3 \end{pmatrix},$$

and

$$\begin{pmatrix} 0 & -1 \\ 1 & -2 \end{pmatrix}^3 = \begin{pmatrix} 0 & -1 \\ 1 & -2 \end{pmatrix}\begin{pmatrix} 0 & -1 \\ 1 & -2 \end{pmatrix}\begin{pmatrix} 0 & -1 \\ 1 & -2 \end{pmatrix} = \begin{pmatrix} 2 & -3 \\ 3 & -4 \end{pmatrix},$$

This simplifies the otherwise slightly cumbersome equivalents: `a.dot(a)` and `a.dot(a).dot(a)`. Note that for both `ndarray` and `matrix` objects, multiplication by a *scalar* acts elementwise:

```
In [x]: A * 4
matrix([[ 0, -4],
        [ 4, -8]])
```

As might be expected, matrix operations that already have methods implemented by the `ndarray` class are retained, including `transpose` (also available as the `T` attribute) and `diagonal`. Additionally, there are attributes for the Hermitian transpose (`H`) and matrix inverse (`I`). If the matrix is singular, a `LinAlgError` exception is raised if an attempt is made to take its inverse.

For eigenfunctions and eigenvalues, see the description of NumPy's `linalg` module (Section 6.5.2).

Example E6.14 The matrix $\mathbf{B}$, defined here, may be manipulated as follows:

$$\mathbf{B} = \begin{pmatrix} 1 & 3-j \\ 3j & -1+j \end{pmatrix}, \quad \mathbf{B}^{\mathrm{T}} = \begin{pmatrix} 1 & 3j \\ 3-j & -1+j \end{pmatrix}$$

$$\mathbf{B}^{\dagger} = \begin{pmatrix} 1 & -3j \\ 3+j & -1-j \end{pmatrix}, \quad \mathbf{B}^{-1} = \begin{pmatrix} -\frac{1}{20} - \frac{3}{20}j & \frac{1}{20} - \frac{7}{20}j \\ \frac{3}{10} + \frac{3}{20}j & -\frac{1}{20} + \frac{1}{10}j \end{pmatrix}.$$

```
In [x]: B = np.matrix([[1, 3-1j], [3j, -1+1j]])
In [x]: print(B)
[[ 1.+0.j  3.-1.j]
 [ 0.+3.j -1.+1.j]]

In [x]: print(B.T)
[[ 1.+0.j  0.+3.j]
 [ 3.-1.j -1.+1.j]]

In [x]: print(B.H)
[[ 1.-0.j  0.-3.j]
 [ 3.+1.j -1.-1.j]]

In [x]: print(B.I)
[[-0.05-0.15j  0.05-0.35j]
 [ 0.30+0.15j -0.05+0.1j ]]
```

Note that although these derived matrices look like attributes, they are not calculated until requested,[25] and so the use of the `matrix` class is not significantly slower than using regular `ndarrays`.

A few other common matrix operations are found elsewhere in the NumPy package, including the trace, determinant, eigenvalues and (right) eigenvectors:

```
In [x]: print(np.trace(B))
1j
In [x]: print(np.linalg.det(B))
(-4-8j)
In [x]: eigenvalues, eigenvectors = np.linalg.eig(B)
In [x]: print(eigenvalues, eigenvectors, sep='\n\n')
[ 2.50851535+2.09456868j -2.50851535-1.09456868j]

[[ 0.77468569+0.j         -0.52924821+0.38116633j]
 [ 0.18832434+0.60365224j  0.75802940+0.j        ]]
```

6.6.3 Should you use NumPy matrices?

The NumPy Matrix class is convenient if you have a lot of operations to perform with matrices and like the MATLAB-style syntax for manipulating them, but it does not provide any functionality that isn't already available to `ndarray` objects. The multiplication operator, `*`, acting to produce matrix products can make code clearer but other common matrix operations still require the use of the main NumPy library's modules and functions. Indeed, the `matrix` class's insistence in turning everything into a two-dimensional array can be rather trying. For example, a $1 \times n$ row matrix must be indexed `M[0,j]` where $j = 0, 1, ..., n - 1$, and, bizarrely, even the `trace` method called on a matrix object returns a two-dimensional matrix object:

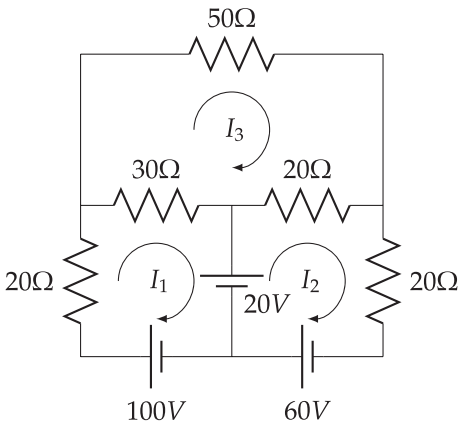
```
In [x]: A.trace()
matrix([[-2]])          # ?!
In [x]: np.trace(A)     # recommended alternative
-2
```

In short, while `matrix` objects may have the edge for simple calculations in an interactive session, they do not have much to commend them over regular `ndarrays` to any but the most die-hard MATLAB fans.[26]

Example E6.15 The currents flowing in the closed regions labeled I_1, I_2 and I_3 of the circuit given here may be analyzed by *mesh analysis*.

[25] They are *properties* of the `matrix` class which, in this case, are really class methods masquerading as attributes.

[26] Moreover, at the time of writing it seems that Python 3.5 is likely to include a specific infix operator for matrix multiplication, `@`.



For each closed loop, we can apply Kirchoff's Voltage Law ($\sum_k V_k = 0$) in conjunction with Ohm's Law ($V = IR$), to give three simultaneous equations:

$$50I_1 - 30I_3 = 80,$$

$$40I_2 - 20I_3 = 80,$$

$$-30I_1 - 20I_2 + 100I_3 = 0.$$

These can be expressed in matrix form as $\mathbf{RI} = \mathbf{V}$:

$$\begin{pmatrix} 50 & 0 & -30 \\ 0 & 40 & -20 \\ -30 & -20 & 100 \end{pmatrix} \begin{pmatrix} I_1 \\ I_2 \\ I_3 \end{pmatrix} = \begin{pmatrix} 80 \\ 80 \\ 0 \end{pmatrix},$$

We could use the numerically stable `np.linalg.solve` method (Section 6.5.3) to find the loop currents, $\mathbf{I}$ here, but in this well-behaved system[27], let's find them by left multiplication by the matrix inverse, $\mathbf{R^{-1}}$:

$$\mathbf{R^{-1}RI} = \mathbf{I} = \mathbf{R^{-1}V}.$$

Using NumPy's `matrix` module:

```
In [x]: R = np.matrix('50 0 -30; 0 40 -20; -30 -20 100')
In [x]: V = np.matrix('80; 80; 0')
In [x]: I = np.linalg.inv(R) * V
In [x]: print(I)
[[ 2.33333333]
 [ 2.61111111]
 [ 1.22222222]]
```

Thus, $I_1 = 2.33$ A, $I_2 = 2.61$ A, $I_3 = 1.22$ A.

[27] In general, matrix inversion may be an ill-conditioned problem, but this particular matrix is easy to invert accurately. See Section 9.2.2 for more on conditioning.

6.6.4 Exercises

Problems

P6.6.1 Let the column matrix

$$\mathbf{F_n} = \begin{pmatrix} p_n \\ q_n \end{pmatrix}$$

describe the number of non-negative integers less than 10^n ($n \geq 0$) that do (p_n) and do not (q_n) contain the digit 5. Hence, for $n = 1, p_1 = 1$ and $q_1 = 9$. Devise a matrix-based recursion relation for finding $\mathbf{F_{n+1}}$ from $\mathbf{F_n}$.

How many numbers less than 10^{10} contain the digit 5?

For each $n \leq 10$, find p_n and verify that $p_n = 10^n - 9^n$.

P6.6.2 The matrix

$$\mathbf{F} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

can be used to produce the Fibonacci sequence by repeated multiplication: the element F_{11}^n of the matrix $\mathbf{F}^n$ is the $(n+1)$th Fibonacci number (for $n = 0, 1, 2 \cdots$). Use NumPy's `matrix` objects to calculate the first 10 Fibonacci numbers.

One can show that

$$\mathbf{F}^n = \mathbf{C}\mathbf{D}^n\mathbf{C}^{-1}, \quad \text{where} \quad \mathbf{D} = \mathbf{C}^{-1}\mathbf{F}\mathbf{C}$$

is the diagonal matrix related to $\mathbf{F}$ through the similarity transformation associated with matrix $\mathbf{C}$. Use this relationship to find the 1100th Fibonacci number.

P6.6.3 The *implicit* formula for a conic section may be written as the second-degree polynomial,

$$Q = Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0,$$

or in matrix form using the homogeneous coordinate vector,

$$\mathbf{x} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix},$$

as $\mathbf{x}^T\mathbf{Q}\mathbf{x} = 0$, where

$$\mathbf{Q} = \begin{pmatrix} A & B/2 & D/2 \\ B/2 & C & E/2 \\ D/2 & E/2 & F \end{pmatrix}.$$

Conic sections may be classified according to the following properties of $\mathbf{Q}$, where the submatrix $\mathbf{Q}_{33}$ is

$$\mathbf{Q}_{33} = \begin{pmatrix} A & B/2 \\ B/2 & C \end{pmatrix}.$$

- If $\det\mathbf{Q} = 0$, the conic is *degenerate* in one of the following forms:
  - if $\det\mathbf{Q}_{33} < 0$, the equation represents two intersecting lines,
  - if $\det\mathbf{Q}_{33} = 0$, the equation represents two parallel lines,
  - if $\det\mathbf{Q}_{33} > 0$, the equation represents a single point.
- If $\det\mathbf{Q} < 0$, the conic is a *hyperbola*.
- If $\det\mathbf{Q} > 0$, the conic is an *ellipse*:
  - If $A = C$ and $B = 0$, the ellipse is a *circle*.

Write a program to classify the conic section represented by the six coefficients A, B, C, D, E and F.

Some test-cases (coefficients not given are zero):

- Hyperbola: $B = 1, F = -9$.
- Parabola: $A = \frac{1}{2}, D = 2, E = -\frac{1}{2}$.
- Circle: $A = \frac{1}{2}, C = \frac{1}{2}, D = -2, E = -3, F = 2$.
- Ellipse: $A = 9, C = 4, F = -36$.
- Two parallel lines: $A = 1, F = -1$.
- A single point: $A = 1, C = 1$.

6.7 Random sampling

NumPy's `random` module provides methods for obtaining random numbers from any of several distributions as well as convenient ways to choose random entries from an array and to randomly shuffle the contents of an array.

As with the core library's `random` module (Section 4.5.1), `np.random` uses a Mersenne Twister *pseudorandom* number generator (PRNG). The way it seeds itself is operating-system dependent, but it can be reseeded with any hashable object (e.g., an immutable object such as an integer) by calling `np.random.seed`. For example, using the `randint` method described here:

```
In [x]: np.random.seed(42)
In [x]: np.random.randint(1, 10, 10)    # 10 random integers in [1,10)
array([7, 4, 8, 5, 7, 3, 7, 8, 5, 4])
In [x]: np.random.randint(1, 10, 10)
array([8, 8, 3, 6, 5, 2, 8, 6, 2, 5])
In [x]: np.random.randint(1, 10, 10)
array([1, 6, 9, 1, 3, 7, 4, 9, 3, 5])
In [x]: np.random.seed(42)              # reseed the PRNG
In [x]: np.random.randint(1,10, 10)
array([7, 4, 8, 5, 7, 3, 7, 8, 5, 4])   # same as before
```

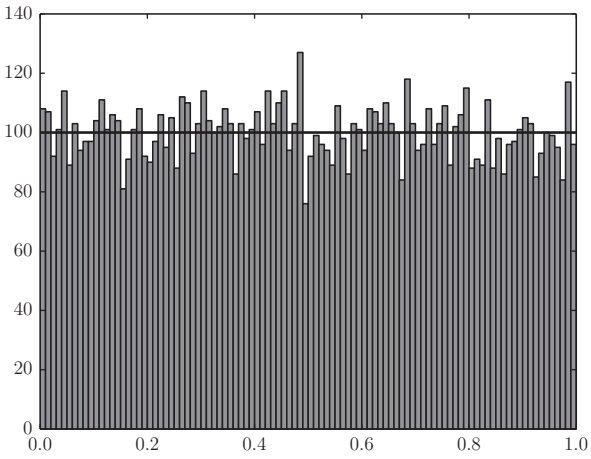


Figure 6.10 Histogram of 10,000 random samples from the uniform distribution on [0, 1) provided by `np.random.random_sample()`.

6.7.1 Uniformly distributed random numbers

Random floating point numbers

The basic random method, `random_sample`[28] takes the shape of an array as its argument and creates an array of the corresponding shape filled with numbers sampled randomly from the uniform distribution over [0, 1); that is, the interval between 0 and 1 inclusive of 0 but exclusive of 1:

```
In [x]: np.random.random_sample((3,2))
array([[ 0.92338355,  0.2978852 ],
       [ 0.75175429,  0.88110707],
       [ 0.16759816,  0.32203783]])
```

(called without an argument, it returns a single random number). If you want numbers sampled from the uniform distribution over $[a, b)$, you need to do a bit of work:

```
In [x]: a, b = 10, 20
In [x]: a + (b-a) * np.random.random_sample((3,2))
array([[ 18.07084068,  12.11591797],
       [ 14.08171741,  19.34857282],
       [ 13.06759203,  11.07003867]])
```

In a uniform distribution, every number has the same probability of being sampled, as can be seen from a histogram of a large number of samples (Figure 6.10):

```
In [x]: pylab.hist(np.random.random_sample(10000), bins=100)
In [x]: pylab.show()
```

The `np.random.rand` method is similar, but is passed the dimensions of the desired array as separate arguments. For example,

[28] `np.random.random_sample` is also available under the aliases `np.random.random`, `np.random.ranf` and `np.random.sample`.

```
In [x]: np.random.rand(2,3)
Out[x]:
array([[ 0.61075227,  0.37459455,  0.95670676],
       [ 0.25276732,  0.1601836 ,  0.3746576 ]])
```

Random integers

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

- If both `low` and `high` are supplied, then the random number(s) are sampled from the discrete half-open interval `[low, high)`.[29]
- If `low` is supplied but `high` is not, then the sampled interval is `[0,low)`.
- `size` is the shape of the array of random integers desired. If it is omitted, as with `np.random.rand` a single random integer is returned.

```
In [x]: np.random.randint(4)                # random integer from [0, 4)
2
In [x]: np.random.randint(4, size=10)       # 10 random integers from [0,4)
array([3, 2, 2, 2, 0, 2, 2, 1, 3, 1])
In [x]: np.random.randint(4, size=(3,5))    # array of random ints from [0,4)
array([[0, 1, 1, 2, 2],
       [2, 0, 3, 3, 0],
       [0, 1, 0, 1, 1]])
In [x]: np.random.randint(1, 4, (3,5))      # array of random ints from [1,4)
array([[1, 1, 1, 3, 2],
       [1, 1, 2, 1, 3],
       [1, 3, 1, 3, 1]])
```

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

```
In [x]: a = np.array([6,6,6,7,7,7,7,7,7])
In [x]: a[np.random.randint(len(a), size=5)]
array([7, 7, 7, 6, 7])
```

The other method for sampling random integers, `np.random.random_integers` has the same syntax but returns integers sampled from the uniform distribution over the *closed* interval `[low, high]` (if `high` is supplied) or `[0, low]` (if it is not).

Example E6.16 These random integer methods can be used for sampling from a set of evenly spaced real numbers, though it requires a bit of extra work: to pick a number from `n` evenly spaced real numbers between `a` and `b` (inclusive), use

```
In [x]: a + (b-a) * (np.random.random_integers(n) - 1) / (n-1.)
```

For example, to sample from $[\frac{1}{2}, \frac{3}{2}, \frac{5}{2}, \frac{7}{2}]$,

```
In [x]: a, b, n = 0.5, 3.5, 4
In [x]: a + (b-a) * (np.random.random_integers(n, size=10) - 1) / (n-1.)
array([ 1.5,  0.5,  1.5,  1.5,  3.5,  2.5,  3.5,  3.5,  3.5,  3.5])
```

---

[29] Note that this is different from the behavior of the standard library's `random.randint(a,b)` method (see Section 4.5.1) which picks numbers uniformly from the *closed* interval, `[a,b]`.

Example E6.17 In a famous experiment, a group of volunteers are asked to toss a fair coin 100 times and note down the results of each toss (heads, `H`, or tails, `T`). It is generally easy to spot the participants who fake the results by writing down what they think is a random sequence of `H`s and `T`s instead of actually tossing the coin because they tend not to include as many "streaks" of repeated results as would be expected by chance.

If they had access to a Python interpreter, here's how they could produce a more plausibly random set of results:

```
In [x]: res = ['H', 'T']
In [37]: tosses = ''.join([res[i] for i in np.random.randint(2, size=100)])
In [38]: tosses
Out[38]: 'TTHHTHHTTHHHTHTTHHHTHHTHTTHHTHHTTTTHHHHHHHHTTTHTTHHHHHHHTHHHTHHHH
THTTTHTTHHHHTHTTTTHTTTHTHHTTHHHHHHH'
```

This virtual experiment features a run of eight heads in a row, and two runs of seven heads in a row:

```
  TAILS    | i |   HEADS
---------------------------
           | 8 | *
           | 7 | **
           | 6 |
           | 5 |
        ** | 4 | **
       *** | 3 | ***
   ******* | 2 | ******
********** | 1 | ********
```

6.7.2 Random numbers from nonuniform distributions

The full range of random distributions supported by NumPy is described in the official documentation.[30] In the next section we describe in detail only the *normal*, *binomial* and *Poisson* distributions.

The normal distribution

The normal probability distribution is described by the Gaussian function,

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

where μ is the mean and σ the standard deviation. The NumPy function, `np.random.normal`, selects random samples from the normal distribution. The mean and standard deviation are specified by `loc` and `scale` respectively, which default to 0 and 1. The shape of the returned array is specified with the `size` attribute.

```
In [x]: np.random.normal()
-0.34599057326978105
In [x]: np.random.normal(scale=5., size=3)
```

[30] http://docs.scipy.org/doc/numpy/reference/routines.random.html.

```
array([  4.38196707,  -5.17358738,  11.93523167])
In [x]: np.random.normal(100., 8., size=(4,2))
array([[ 107.730434  ,  101.06221195],
       [ 100.75627505,   88.79995561],
       [  88.82658615,   94.89630767],
       [ 105.91254312,   98.21190741]])
```

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

```
In [x]: np.random.randn(2, 2)
array([[-1.25092263,  2.6291925 ],
       [ 0.34158642,  0.40339403]])
```

Although `np.random.randn` does not provide a way to set the mean and standard deviation explicitly, the standard distribution can be rescaled easily enough:

```
In [x]: mu, sigma = 100., 8.
In [x]: mu + sigma * np.random.randn(4, 2)
array([[ 104.92454826,   98.84646729],
       [ 109.43568726,   92.9568489 ],
       [  90.21632016,   96.25271625],
       [ 102.65745451,   89.94890264]])
```

Example E6.18 The normal distribution may be plotted from sampled data as a histogram (Figure 6.11):

```
In [x]: mu, sigma = 100., 8.
In [x]: samples = np.random.normal(loc=mu, scale=sigma, size=10000)
In [x]: counts, bins, patches = pylab.hist(samples, bins=100, normed=True)
In [x]: pylab.plot(bins, 1/(sigma * np.sqrt(2 * np.pi)) *
...                np.exp( -(bins - mu)**2 / (2 * sigma**2) ), lw=2)
In [x]: pylab.show()
```

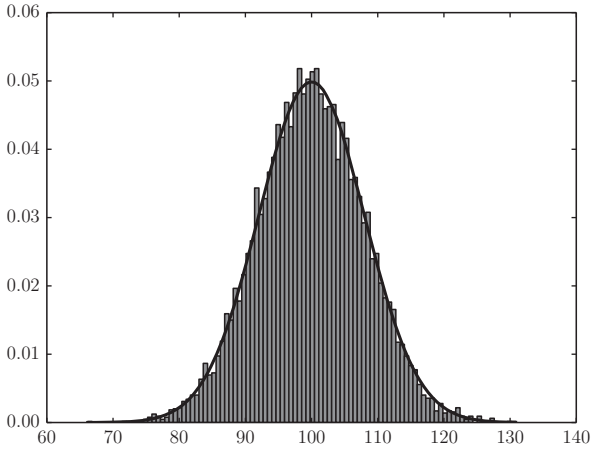


Figure 6.11 Histogram of 10,000 random samples from the normal distribution provided by `np.random.normal`.

6.7.3 The binomial distribution

The binomial probability distribution describes the number of particular outcomes in a sequence of n *Bernoulli trials* – that is, n independent experiments, each of which can yield exactly two possible outcomes (e.g., *yes*/*no*, *success*/*failure*, *heads*/*tails*). If the probability of a single particular outcome (say, *success*) is p, the probability that such a sequence of trials yields exactly k such outcomes is

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

For example, when a fair coin is tossed, the probability of it coming up heads each time is $\frac{1}{2}$. The probability of getting exactly three heads out of four tosses, is therefore $4(\frac{1}{2})(\frac{1}{2})^3 = \frac{1}{4}$, where the factor of $\binom{4}{3} = 4$ accounts for the four possible equivalent outcomes: THHH, HTHH, HHTH, HHHT.

To sample from the binomial distribution described by parameters `n` and `p`, use `np.random.binomial(n, p)`. Again, the shape of an array of samples can be specified with the third argument, `size`:

```
In [x]: np.random.binomial(4, 0.5)
2
In [x]: np.random.binomial(4, 0.5, (4,4))
array([[1, 2, 2, 4],
       [2, 1, 3, 2],
       [2, 3, 1, 1],
       [2, 4, 2, 3]])
```

Example E6.19 There are two stable isotopes of carbon, ^{12}C and ^{13}C (the radioactive ^{14}C nucleus is present in nature in only trace amounts of the order of parts per trillion). Taking the abundance of ^{13}C to be $x = 0.0107$ (i.e., about 1%), we will calculate the relative amounts of buckminsterfullerene, C_{60}, with exactly zero, one, two, three and four ^{13}C atoms. (This is important in nuclear magnetic resonance studies of fullerenes, for example, because only the ^{13}C nucleus is magnetic and so detectable by NMR.)

The number of ^{13}C atoms in a population of carbon atoms sampled at random from a population with natural isotopic abundance follows a binomial distribution: the probability that, out of n atoms, m will be ^{13}C (and therefore $n - m$ will be ^{12}C) is

$$p_m(n) = \binom{n}{m} x^m (1-x)^{n-m}.$$

We can, of course, calculate $p_m(60)$ exactly from this formula for $0 \le m \le 4$, but we can also simulate the sampling with the `np.random.binomial` method:

Listing 6.10 Modeling the distribution of ^{13}C atoms in C_{60}

```
# eg6-e-c13-a.py
import numpy as np

n, x = 60, 0.0107
mmax = 4
m = np.arange(mmax+1)
```

```
# Estimate the abundances by random sampling from the binomial distribution
ntrials = 10000
pbin = np.empty(mmax+1)
for r in m:
❶    pbin[r] = np.sum(np.random.binomial(n, x, ntrials)==r)/ntrials

# Calculate and store the binomial coefficients nCm
nCm = np.empty(mmax + 1)
nCm[0] = 1
for r in m[1:]:
    nCm[r] = nCm[r-1] * (n - r + 1)/ r
# The "exact" answer from binomial distribution
p = nCm * x**m * (1-x)**(n-m)

print('Abundances of C60 as (13C)[m](12C)[60-m]')
print('m   "Exact"    Estimated')
print('-'*24)
for r in m:
    print('{:1d}   {:6.4f}      {:6.4f}'.format(r, p[r], pbin[r]))
```

❶ For each value of `r` in the array `m`, we sample a large number of times (`ntrial`) from the binomial distribution described by $n = 60$ and probability, $x = 0.0107$. The comparison of these sample values with a given value of `r` yields a boolean array which can be summed (remembering that `True` evaluates to 1 and `False` evaluates to 0); division by `ntrials` then gives an estimate of the probability of exactly `r` atoms being of type ^{13}C and the remainder of type ^{12}C.

The explicit loop over `m` could be removed by creating an array of shape `(ntrials, mmax+1)` containing all the samples, and summing over the first axis of this array in the comparison with the `m` array:

```
samples = np.random.binomial(n, x, (ntrials, mmax+1))
pbin = np.sum(samples == m, axis=0) / ntrials
```

The abundances of $^{13}C_m^{12}C_{60-m}$ produced by our program are given as the following output.

```
Abundances of C60 as (13C)[m](12C)[60-m]
m   "Exact"    Estimated
------------------------
0   0.5244      0.5199
1   0.3403      0.3348
2   0.1086      0.1093
3   0.0227      0.0231
4   0.0035      0.0031
```

That is, almost 48% of C_{60} molecules contain at least one magnetic nucleus.

6.7.4 The Poisson distribution

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

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

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

where the parameter λ is the expected (average) number of events occurring within the considered interval.[31] The NumPy implementation `np.random.poisson` takes λ as its first argument (which defaults to 1) and, as before the shape of the desired array of samples can be specified with a second argument, `size`. For example, if I receive an average of 2.5 emails an hour, a sample of the number of emails I receive each hour over the next 8 hours could be obtained as:

```
In [x]: np.random.poisson(2.5, 8)
array([4, 1, 3, 0, 4, 1, 3, 2])
```

Example E6.20 The endonuclease enzyme *Eco*RI is used as a restriction enzyme which cuts DNA at the nucleic acid sequence GAATTC. Suppose a given DNA molecule contains 12000 base pairs and a 50% G+C content. The Poisson distribution can be used to predict the probability that *Eco*RI will fail to cleave this molecule as follows:

The recognition site, GAATTC, consists of six nucleotide base pairs; the probability that any given six-base sequence corresponds to GAATTC is $1/4^6 = 1/4096$ and so the expected number of cleavage sites for *Eco*RI in this DNA molecule is $\lambda = 12000/4096 = 2.93$. From the Poisson distribution, we expect the probability that the endonuclease will fail to cleave this molecule is therefore

$$P(0) = \frac{\lambda^0 e^{-\lambda}}{0!} = 0.053,$$

or about 5.3%. To simulate the possibilities stochastically:

```
In [x]: lam = 12000 / 4**6
In [x]: N = 100000
In [x]: np.sum(np.random.poisson(lam, N)==0)/N
Out[x]: 0.053699999999999998
```

6.7.5 Random selections, shuffling and permutations

It is often the case that given an array of values, you wish to pick one or more at random (with or without replacement). This is the purpose of the `np.random.choice` method. Given a single argument, an one-dimensional sequence, it returns a random element drawn from the sequence:

```
In [x]: np.random.choice([ 1,  5,  2, -5,  5,  2,  0])
2
In [x]: np.random.choice(np.arange(10))
7
```

[31] The Poisson distribution is the limit of the binomial distribution as $n \to \infty$ and $p \to 0$ such that $\lambda = np$ tends to some finite constant value.

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

```
In [x]: a = np.array([1, 2, 0, -1, 1])
In [x]: np.random.choice(a, 6)  # six random selections from a
array([ 1, -1,  2,  1, -1,  1])
In [x]: np.random.choice(a, (2,2), replace=False)
array([[ 2, -1],
       [ 1,  0]])
In [x]: np.random.choice(a, (3,2), replace=False)
... <some traceback information> ...
ValueError: Cannot take a larger sample than population when 'replace=False'
```

This last example shows that, as you might expect, it is not possible to draw a larger number of elements than there are in the original population if you are sampling without replacement.

To specify the probability of each element being selected, pass a sequence of the same length as the population to be sampled as the argument `p`. The probabilities should sum to 1.

```
In [x]: a = np.array([1, 2, 0, -1, 1])
In [x]: np.random.choice(a, 5, p=[0.1, 0.1, 0., 0.7, 0.1])
Out[x]: array([-1, -1, -1, -1,  1])
In [x]: np.random.choice(a, 2, False, p=[0.1, 0.1, 0., 0.8, 0.])
Out[x]: array([-1,  2])          # sample without replacement
```

There are two methods for permuting the contents of an array: `np.random.shuffle` randomly rearranges the order of the elements *in place* whereas `np.random.permutation` makes a copy of the array first, leaving the original unchanged:

```
In [x]: a = np.arange(6)
In [x]: np.random.permutation(a)
array([4, 2, 5, 1, 3, 0])
In [x]: a
array([0, 1, 2, 3, 4, 5])
In [x]: np.random.shuffle(a)
In [x]: a
array([5, 4, 1, 3, 0, 2])
```

These methods only act on the first dimension of the array:

```
In [x]: a = np.arange(6).reshape(3, 2)
In [x]: a
array([[0, 1],
       [2, 3],
       [4, 5]])
In [x]: a.random.permutation(a) # permutes the rows, but not the columns
array([[2, 3],
       [4, 5],
       [0, 1]])
```

6.7.6 Exercises

Questions

Q6.7.1 Explain the difference between

```
In [x]: a = np.array([6,6,6,7,7,7,7,7,7])
In [x]: a[np.random.randint(len(a), size=5)]
array([7, 7, 7, 6, 7])      # (for example)
```

and

```
In [x]: np.random.randint(6, 8, 5)
array([6, 6, 7, 7, 7])      # (for example)
```

Q6.7.2 In Example E6.16 we used `random.random_integers` to sample from the uniform distribution on the floating point numbers $[\frac{1}{2}, \frac{3}{2}, \frac{5}{2}, \frac{7}{2}]$. How can you do the same using the `random.randint` instead?

Q6.7.3 The American lottery, Mega Millions, at the time of writing, involves the selection of five numbers out of 75 *and* one from 15. The jackpot is shared among the players who match all of their numbers in a corresponding random draw. What is the probability of winning the jackpot? Write a single line of Python code using NumPy to pick a set of random numbers for a player.

Q6.7.4 Suppose an n-page book is known to contain m misprints. If the misprints are independent of one another, the probability of a misprint occurring on a particular page is $p = 1/n$ and their distribution may be considered to be binomial. Write a short program to conduct a number of trial virtual "printings" of a book with $n = 500, m = 400$ and determine the probability, Pr, that a single given page will contain two or more misprints.

Compare with the result predicted by the Poisson distribution with rate parameter $\lambda = m/n$, $\Pr = 1 - e^{-\lambda}\left(\frac{\lambda^0}{0!} + \frac{\lambda}{1!}\right)$.

Problems

P6.7.1 Simulate an experiment carried out `ntrials` times in which, for each experiment, `n` coins are tossed and the total number of heads each time is recorded.

Plot the results of the simulation on a suitable histogram and compare with the expected binomial distribution of heads.

P6.7.2 A classic problem, first posed by Georges-Louis Leclerc, Comte de Buffon, can be stated as follows:

Given a plane ruled with parallel lines a distance d apart, what is the probability that a needle of length $l \leq d$ dropped at random onto the plane will cross a line?

The problem can be solved analytically, yielding the answer $2l/\pi d$; show that this solution is given approximately for the case $l = d$ using a random simulation (Monte Carlo) method, that is, by simulating the experiment with a large number of random orientations of the needle.

A related problem involves dropping a circular coin of radius a onto a floor consisting of square tiles, each of side d. Show that the probability of a coin crossing a tile edge is $1 - (d - 2a)^2/d^2$ and confirm it with a Monte Carlo simulation.

P6.7.3 Some bacteria, such as *E. coli*, possess helical flagella which enable them to move toward attractants such as nutrients, process known as chemotaxis. When the flagella rotate counterclockwise the bactrium is propelled forward; when they rotate clockwise, it tumbles randomly, changing its orientation. A combination of such movements enables the bacterium to perform a *biased random walk*: if the bacterium senses it is moving up a concentration gradient toward an attractant it will rotate its flagella counterclockwise more often than clockwise so as to continue moving in that direction; conversely, if it is moving away it is more likely to rotate its flagella clockwise so as to tumble with the aim of randomly changing its orientation to one that points it toward the attractant.

The chemotaxis of *E. coli* may be modeled (very) simplistically by considering a bacterium to move in a two-dimensional "world" populated by an attractant with a constant concentration gradient away from some location. At each of a series of time steps, a model bacterium detects whether it is moving up or down this gradient and either continues moving or tumbles according to some pair of probabilities.

Write a Python program to implement this simple model of chemotaxis for a world consisting of the unit square with an attractant at its center. Plot the locations of 10 model bacteria that start off evenly spaced around the unit circle centered on the attractant location.

P6.7.4 One way to simulate the meanders in a river is as the average of a large number of a random walks.[32] Using a coordinate system (x, y), start at point $A = (0, 0)$ and aim to finish at $B = (b, 0)$. Starting from an initial heading of ϕ_0 from the AB direction, at each step change this angle by a random amount drawn from a normal distribition with mean $\mu = 0$ and standard deviation σ, and proceed by unit distance in this direction. Discard any walks which do not, after n steps, finish within one unit of B (this will be the majority!).

Write a program to find the average path meeting the above constraints for $b = 10$, using $\phi_0 = 110°$, $\sigma = 17°$, $n = 40$ and 10^6 random walk trials. Plot the accepted walks and their average, which should resemble a meander.

6.8 Discrete Fourier transforms

6.8.1 One-dimensional Fast Fourier Transforms

`numpy.fft` is NumPy's Fast Fourier Transform (FFT) library for calculating the discrete Fourier transform (DFT) using the ubiquitous Cooley and Tukey algorithm.[33] The

[32] B. Hayes, (2006) *American Scientist* **94**, 490; H. von Schelling, *General Electric Report* No. 64GL92

[33] J. W. Cooley and J. W. Tukey, (1965) *Math. Comput.* **19**, 297=301.

definition for the DFT of a function defined on n points, $f_m, m = 1, 2, \cdots, n-1$ used by NumPy is

$$F_k = \sum_{m=0}^{n-1} f_m \exp\left(-\frac{2\pi imk}{n}\right), \quad k = 0, 1, 2, \cdots, n-1 \tag{6.1}$$

NumPy's basic DFT method, for real and complex functions, is `np.fft.fft`. If the input signal function, f, is considered to be in the time domain, the output Fourier Transform, F, is in the frequency domain and is returned by the `fft(f)` function call in a standard order: `F[:n/2]` are the positive-frequency terms in increasing order, `F[n/2+1:]` contains the negative-frequency terms in decreasing order, and `F[n/2]` is the (positive and negative) Nyquist frequency.[34] `np.abs(F)`, `np.abs(F)**2` and `np.angle(F)` are the *amplitude spectrum*, *power spectrum* and *phase spectrum* respectively.

The frequency bins corresponding to the values of `F` are given by `np.fft.fftfreq(n, d)` where `d` is the sample spacing. For even n, this is equivalent to

$$0, \frac{1}{dn}, \frac{2}{dn}, \cdots, \frac{n/2-1}{dn}, -\frac{n/2}{dn}, -\frac{n/2-1}{dn}, \cdots, -2, -1$$

To shift the spectrum so that the zero-frequency component is at the center, call `np.fft.fftshift`. To undo that shift, call `np.fft.ifftshift`.

For example, consider the following waveform in the time domain with some synthetic Gaussian noise added:

$$f(t) = 2\sin(20\pi t) + \sin(100\pi t)\,.$$

```
In [x]: A1, A2 = 2, 1
In [x]: freq1,freq2 =  10, 50
In [x]: fsamp = 500
In [x]: t = np.arange(0, 1, 1/fsamp)
In [x]: n = len(t)
In [x]: f = A1*np.sin(2*np.pi*freq1*t) + A2*np.sin(2*np.pi*freq2*t)
In [x]: f += 0.2 * np.random.randn(n)
In [x]: pylab.plot(t, f)
In [x]: pylab.xlabel('Time /s')
In [x]: pylab.show()
```

The plot of this waveform is depicted in Figure 6.12.

The Fourier transform of this function is complex; its real and imaginary components are plotted here (Figure 6.13).

```
In [x]: F = np.fft.fft(f)
In [x]: pylab.plot(F.real, 'k', label='real')
In [x]: pylab.plot(F.imag, 'gray', label='imag')
In [x]: pylab.legend(loc=2)
In [x]: pylab.show()
```

[34] Here, `n` is assumed to be even.

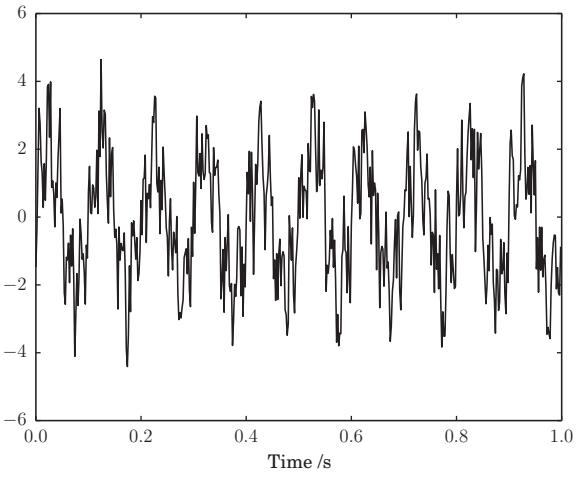


Figure 6.12 The noisy waveform referred to in the text.

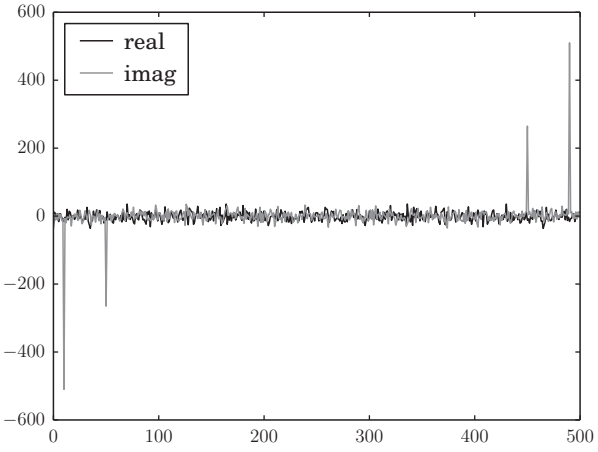


Figure 6.13 The Fourier transform of a noisy waveform with two frequency components, as returned by `np.fft.fft`.

Now look at the shifted amplitude spectrum with the zero-frequency component at the center:[35]

```
In [x]: freq = np.fft.fftfreq(n, 1/fsamp)
In [x]: F_shifted = np.fft.fftshift(F)
In [x]: freq_shifted = np.fft.fftshift(freq)
In [x]: pylab.plot(freq_shifted, np.abs(F_shifted))
In [x]: pylab.xlabel('Frequency /Hz')
In [x]: pylab.show()
```

This plot is given in Figure 6.14.

[35] The shifting here is for illustration: note that it isn't really necessary to shift both `freq` and `F` arrays simply to plot one against the other.

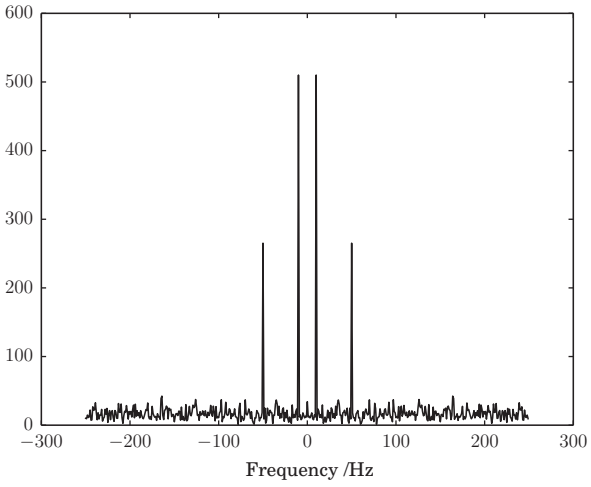


Figure 6.14 The Fourier Transform of a noisy waveform with two frequency components plotted against frequency.

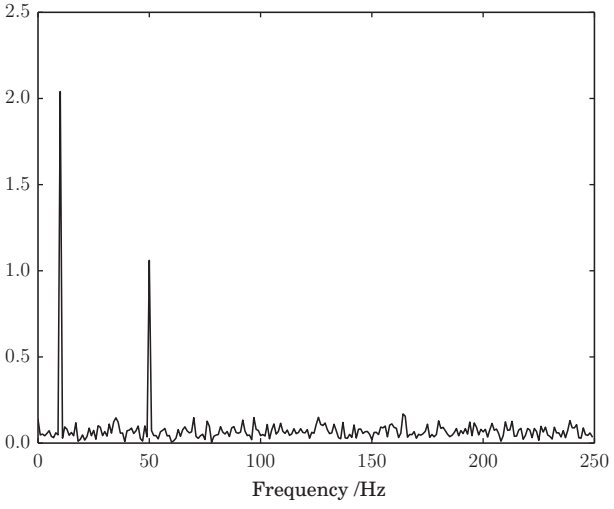


Figure 6.15 The positive-frequency components of the Fourier transform of a noisy waveform, normalized to show their intensities.

Now, because our input function is real, its Fourier transform is Hermitian: the negative frequency components are the complex conjugates of the positive frequency components so they don't contain any further information. Therefore, we only need to deal with the first half of the `F` array. Plotted against its (positive) frequencies as an amplitude spectrum (Figure 6.15):

❶
```
In [x]: spec = 2/n * np.abs(F[:n/2])
In [x]: pylab.plot(freq[:n/2], spec, 'k')
In [x]: pylab.xlabel('Frequency /Hz')
In [x]: pylab.show()
```

❶ Note that because of the way this DFT has been defined, a normalization factor of $\frac{2}{n}$ is required to faithfully regenerate the original amplitudes of each component.

The amplitudes of the 10 Hz and 50 Hz signals are easily resolved in this spectrum.

The inverse Fourier Transform defined through

$$f_m = \frac{1}{n}\sum_{k=0}^{n-1} F_k \exp\left(\frac{2\pi imk}{n}\right) \quad m = 0, 1, 2, \cdots, n-1$$

is returned by the method `np.fft.ifft`.

If, as mentioned earlier, the input function array is real and only the non-negative frequency components are needed, the `np.fft` methods `rfft`, `irfft`, `rfftfreq` can be used.

6.8.2 Two-dimensional Fast Fourier Transforms

Discrete Fourier transforms and their inverses in two and higher dimensions are possible using the `np.fft` methods `fft2`, `ifft2`, `fftn` and `ifftn`. The two-dimensional DFT is defined as

$$F_{jk} = \sum_{p=0}^{m-1}\sum_{q=0}^{n-1} f_{pq} \exp\left[-2\pi i\left(\frac{pj}{m} + \frac{qk}{n}\right)\right],$$
$$j = 0, 1, 2, \cdots, m-1;\ k = 0, 1, 2, \cdots, n-1.$$

and higher dimensions follow similarly.

Example E6.21 The two-dimensional DFT is widely used in image processing.[36] For example, multiplying the DFT of an image by a two-dimensional Gaussian function is a common way to blur an image by decreasing the magnitude of its high-frequency components.

The following code produces an image of randomly arranged squares and then blurs it with a Gaussian filter.

Listing 6.11 Blurring an image with a Gaussian filter

```
# eg6-fft2-blur.py
import numpy as np
import pylab

# image size, square side length, number of squares
ncols, nrows = 120, 120
sq_size, nsq = 10, 20

# The image array (0=background, 1=square) and boolean array of allowed places
# to add a square so that it doesn't touch another or the image sides
```

[36] Note that there is an entire SciPy subpackage, `scipy.ndimage`, not described in this book, devoted to image processing. This example serves simply to illustrate the syntax and format of NumPy's two-dimensional FFT implementation.

```
image = np.zeros((nrows, ncols))
sq_locs = np.zeros((nrows, ncols), dtype=bool)
sq_locs[1:-sq_size-1:,1:-sq_size-1] = True

def place_square():
    """ Place a square at random on the image and update sq_locs. """
    # valid_locs is an array of the indexes of True entries in sq_locs
    valid_locs = np.transpose(np.nonzero(sq_locs))
    # pick one such entry at random, and add the square so its top left
    # corner is there; then update sq_locs
    i, j = valid_locs[np.random.randint(len(valid_locs))]
    image[i:i+sq_size, j:j+sq_size] = 1
    imin, jmin = max(0,i-sq_size-1), max(0, j-sq_size-1)
    sq_locs[imin:i+sq_size+1, jmin:j+sq_size+1] = False

# Add the required number of squares to the image
for i in range(nsq):
    place_square()
pylab.imshow(image)
pylab.show()

# Take the two-dimensional DFT and center the frequencies
ftimage = np.fft.fft2(image)
ftimage = np.fft.fftshift(ftimage)
pylab.imshow(np.abs(ftimage))
pylab.show()


# Build and apply a Gaussian filter.
sigmax, sigmay = 10, 10
cy, cx = nrows/2, ncols/2
x = np.linspace(0, nrows, nrows)
y = np.linspace(0, ncols, ncols)
X, Y = np.meshgrid(x, y)
gmask = np.exp(-(((X-cx)/sigmax)**2 + ((Y-cy)/sigmay)**2))

ftimagep = ftimage * gmask
pylab.imshow(np.abs(ftimagep))
pylab.show()

# Finally, take the inverse transform and show the blurred image
imagep = np.fft.ifft2(ftimagep)
pylab.imshow(np.abs(imagep))
pylab.show()
```

The results are shown in Figure 6.16.

6.8.3 Exercises

Questions

Q6.8.1 Compare the speed of execution of NumPy's `np.fft.fft` algorithm and that of the direct implementation of Equation 6.1.

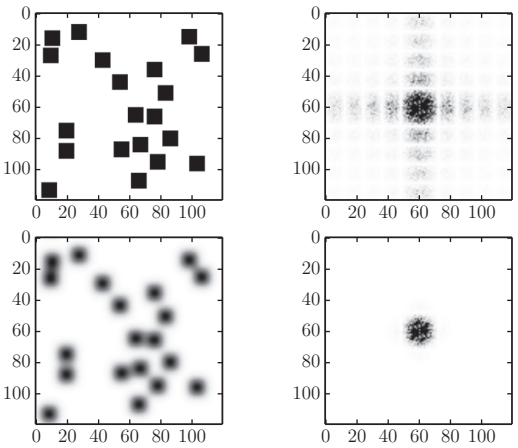


Figure 6.16 Blurring an image with a Gaussian filter applied its two-dimensional Fourier transform.

Hint: treat the direct equation as a matrix multiplication (dot product) of an array of n function values (random ones will do) with the $n \times n$ array with entries $\exp(-2\pi imk/n)$ $(m, k = 0, 1, \cdots, n-1)$. Use IPython's `%timeit` magic.

Problems

P6.8.1 Consider a signal in the time domain defined by the function

$$f(t) = \cos(2\pi \nu t)e^{-t/\tau},$$

with frequency $\nu = 250$ Hz decaying exponentially with a lifetime $\tau = 0.2$ s. Plot the function, sampled at 1,000 Hz, and its discrete Fourier transform against frequency. Examine, by means of a suitable plot, the effect of *apodization* on the DFT by truncating the time sequence after (a) 0.5 s, (b) 0.2 s.

P6.8.2 A square wave of period T may be defined through the following function:

$$f_{\text{sq}}(t) = \begin{cases} 1 & t < T/2 \\ -1 & t \geq T/2 \end{cases}$$

with $f(t) = f(t + nT)$ for $n = \pm 1, \pm 2, \cdots$.

Plot the square wave with $T = 1$ (and hence cycle frequency, $\nu = 1$) for $0 \leq t < 2$ taking a grid of 2,048 time points over this interval. Calculate and plot its discrete Fourier transform.

The Fourier *expansion* of this function is the infinite series

$$f_{\text{sq}}(t) = \frac{4}{\pi} \sum_{k=1}^{\infty} \frac{1}{2k-1} \sin[2\pi(2k-1)\nu t]$$

Compare the square wave function with this Fourier expansion truncated at 3, 9 and 18 terms. Also compare their (suitably normalized) Fourier transforms: the missing

frequencies in each truncated series should appear as zeros in its Fourier transform, whereas the present terms will have intensities $4/[\pi(2k-1)]$.

P6.8.3 The `scipy` library provides a routine for reading in `.wav` files as NumPy arrays:

```
In [x]: from scipy.io import wavfile
In [x]: sample_rate, wav = wavfile.read(\emph{<filename>})
```

For a stereo file, the array `wav` has shape `(n,2)` where `n` is the number of samples.

Use the routines of `np.fft` to identify the chords present in the sound file `chords.wav`, which may be downloaded from scipython.com/ex/afi. Which major chord do they comprise?

The frequencies of musical notes on an equal-tempered scale for which $A_4 = 440$ Hz are provided as a dictionary in the file `notes.py`.