4 Numpy

Numpy is an add-on package which brings enhancements allowing Python to be used constructively for scientific computing, by offering a performance close to that of compiled languages but with the ease of the Python language. The basic object in *Numpy* is the *ndarray*. These are (possibly multi-dimensional) arrays of objects, all of the same data type, and whose size is fixed at creation. (Note that the Python *list* object does not impose homogeneity on its items, and that a *list* object can enlarge or reduce dynamically via the intrinsic `append` and `remove` functions.) The homogeneity requirement ensures that each item occupies the same space in memory. This enables the *numpy* designers to implement many operations involving *ndarray*s as precompiled C-code. Because of this, operations on *ndarrays* can be executed much more efficiently and with much less support code than that required for Python *lists*. Let us illustrate this point with an often-quoted simple example. Suppose that `a` and `b` are two Python *lists* of the same size and we want to multiply them element-wise. Using the Pythonic approach of Chapter 3, we might use

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

However, if the *lists* contain thousands or millions of items, this will be extremely slow. Had we been using a compiled language, e.g., C, and ignoring all variable declarations etc., we might have written

```
for(i=0; i<rows; i++) {
    c[i]=a[i]*b[i];
}
```

and in two dimensions

```
for(i=0; i<rows; i++) {
    for(j=0; j<cols; j++) {
        c[i][j]=a[i][j]*b[i][j];
    }
}
```

Although superficially complicated, this code will execute extremely quickly. However, with *ndarrays* the code would be

```
c=a*b
```

and would use precompiled code to achieve (very nearly) the same speed. To be fair, modern compiled languages, e.g., C++ and Fortran90 can achieve the same simplicity of expression for such a simple example. *Numpy*'s library of "vectorized" functions and operations is however at least as rich as those available in compiled languages, and becomes considerably richer once *scipy*, see Section 4.9.1, is taken into account.

This chapter is an introduction to *numpy*. While the core is pretty stable, extensions near the edges are an ongoing process. The definitive documentation is a recent "user guide", Numpy Community (2013*b*) at 103 pages and the "reference manual", Numpy Community (2013*a*) at 1409 pages. Earlier, more discursive accounts, including a wealth of examples, can be found in Langtangen (2008) and/or Langtangen (2009).

Before we start, we must import the *numpy* module. The preferred approach is to preface the code with

```
import numpy as np
```

Then a *numpy* function `func` needs to be written as `np.func`. It is of course tempting to use the quick-and-dirty-fix

```
from numpy import *
```

and to drop the `np.` prefix. While this is fine for small scale experiments, experience shows that it tends to lead to namespace difficulties for real-life problems. We shall assume that the first **`import`** statement above has always been used.

Once *numpy* has been imported, we need to think about on-line documentation, for this is a very large module. It is very straightforward to see how large *numpy* is if you are following my recommendation to use the *IPython* interpreter, which utilizes *tab completion*, i.e., if we type part of a command and press the TAB key, the range of possible completions is shown, unless there is precisely one, in which case it will be completed. Trying

```
np.<TAB>
```

shows that over 500 possibilities are available, and asks whether you want to see them? Normally we will not want to see all of them. However, it is instructive to answer 'y' once. How do we make sense of this plethora?

In my view, the best place to start is the `np.lookfor` function, try

```
np.lookfor? # or help(np.lookfor)
```

for documentation. As an example, consider looking for functions whose docstrings make use of the word cosine.

```
np.lookfor('cosine')
```

reveals a surprising collection. We explore further with, e.g.

```
np.cos?  # or help(np.cos)
```

which describes the `np.cos` function.

4.1 One-dimensional arrays

Vectors or one-dimensional arrays are the basic building blocks for numerical computation. We look first at how to construct them, and then at how to use them. It makes sense here to distinguish two types of constructors, build a vector from scratch, or construct a vector to "look like" another object.

4.1.1 Ab initio constructors

Perhaps the most useful constructor is `np.linspace`, which builds an equally spaced array of **`float`**s. Its calling sequence is

```
x=np.linspace(start,stop,num=50,endpoint=True,retstep=False)
```

x is an array of length `num`. If `num` is not specified, it defaults to the value 50. The first element is `x[0]=start`. If `endpoint` is `True`, the default, then the final value is `x[-1]=stop`, and the interval spacing is `step=(stop-start)/(num-1)`. However, if we choose `endpoint` to be `False`, then `step= (stop-start)/num`, and so the final value is `x[-1]=stop - step`. Thus the parameter `endpoint` controls whether the interval is closed [start, stop] or half open [start, stop). If `retstep` is `True`, then the function returns a *tuple* consisting of the array and `step`. The following code snippet illustrates the basic use of `linspace`.

```
import numpy as np
xc,dx=np.linspace(0,1,11,retstep=True)
xc,dx
xo=np.linspace(0,1,10,endpoint=False)
xo
```

The function `np.logspace`, is similar, but the numbers are equally spaced on a logarithmic scale. See its docstring for the details and examples of use.

Somewhat closer to the **`range`** function of Python is the function, `np.arange`, which returns an array rather than a *list*. The calling sequence is

```
x=np.arange(start=0,stop,step=1,dtype=None)
```

which generates the open interval [start, stop) with intervals of `step`. Python will try to deduce the type of the array, e.g., **`int`**, **`float`** or **`complex`** from the input arguments, but this choice can be overridden by specifying the type in the last argument. Two examples are

```
import numpy as np
yo=np.arange(1,10)
yo
yoc=np.arange(1,10,dtype=complex)
yoc
```

Three further vector constructors turn out to be surprisingly useful.

```
z=np.zeros(num,dtype=float)
```

constructs an array of length `num` filled with zeros. The function `np.ones` does the same but packs the array with ones, and `np.empty` constructs an array of the same length but leaves the values of the contents unspecified.

Last but not least, we need to introduce the function `np.array`. Its simplest and most used form is

```
ca=np.array(c,dtype=None,copy=True)
```

Here `c` is any container object which can be indexed, e.g., a *list*, a *tuple* or another array. The function will try to guess an appropriate type, but this can be overwritten by using the `dtype` parameter. Possible choices include **`bool`**, **`int`**, **`float`**, **`complex`** and even user-defined objects, see Section 3.9. Here are two examples, one from a *list* and one converting an array of floats to a complex array.

```
la=np.array([1,2,3.0])
la
x = np.linspace(0,1,11)
x
cx=np.array(x, dtype=complex)
cx
```

4.1.2 Look alike constructors

Quite often these will be generated automatically, e.g., with `x` as above the line

```
y=np.sin(x)
```

will generate a new array with identifier `y` whose components are the sines of the corresponding components of `x`.

Otherwise, a very useful constructor is `np.empty_like`

```
xe=np.empty_like(x)
```

sets up an empty array (i.e., the contents have unspecified values) of the same size and type as `x`. Almost as useful are `np.zeros_like` and `np.ones_like`, which behave very much like `np.empty_like`.

Surprisingly often, we need vectors that span an interval but with different spacings on different subintervals. To take a concrete example, suppose we require `xs` to span $[0, 1]$ with spacing of 0.1, except on $[0.5, 0.6]$ where we require a spacing of 0.01. We achieve this by constructing three subintervals, two half-closed and one closed, and then joining them together. It is a useful exercise to consider carefully the following snippet.

```
1 xl=np.linspace(0,0.5,5,endpoint=False)
2 xm=np.linspace(0.5,0.6,10,endpoint=False)
3 xr=np.linspace(0.6,1.0,5)
4 xs=np.hstack((xl,xm,xr))
```

Note that `np.hstack` accepts only one argument, and so we must use a *tuple* in line 4.

We end this section by remarking that a vector is a one-dimensional instance of a *ndarray*. Any *ndarray* is a mutable container object. We studied some of the properties of such objects in the discussion of *lists* in Section 3.5. The remarks we made there, especially on slicing and copies of *lists*, apply equally well to vectors, and indeed to more general *ndarray*s.

4.1.3 Arithmetical operations on vectors

Arithmetical operations between arrays *of the same size* can be performed as the next snippet shows.

```
1 a=np.linspace(0,1,5)
2 c=np.linspace(1,3,5)
3 a+c
4 a*c
5 a/c
```

Let us look in detail at say line 3. The sum is an array of the same size as its operands. The ith component is the sum of the i-components of `a` and `c`. In this sense, the + operator is said to act *component-wise*. All of the arithmetical operations in this code snippet are acting component-wise. Incidentally, there is an efficiency issue, relevant for large arrays, which should be highlighted here. Suppose in line 3 of the snippet above we had set `a=a+c`. Python would have created a temporary array to hold the sum required on the right-hand side, filled it, then attached the identifier `a` to it, and finally deleted the original `a`-array. It is faster to use `a+=c`, which avoids the creation of a temporary array. Similar constructions are available for the other arithmetic operators acting on vectors. There is however a pitfall for the unwary. In Python, if a scalar `a` has type **`int`** and a second scalar `b` has type **`float`**, then the operation `a+=b` widens the type of `a` to float. In fact, the precise opposite holds for numpy arrays! The reader is encouraged to try

```
1 a=np.ones(4,dtype=int)
2 b=np.linspace(0,1,4)
3 a+=b
4 a
```

to see that the type of a is unchanged. The numbers in the new a have been narrowed back to int by truncation towards zero.

In general, arithmetical operations between vectors of different sizes which produce another vector cannot be defined unambiguously, and so cause an error. However, arithmetical operations between an array and a scalar can be given an unambiguous meaning, as shown in the next snippet.

```
import numpy as np
a=np.linspace(0,1,5)
2*a
a*2
a/5
a**3
a+2
```

Here the last line deserves comment. The sum (or difference) of an array, here a, and a scalar, here 2, is not defined. However, *numpy* effectively[1] chooses to *widen* or *broadcast* 2 to 2*ones_like(a) and performs the addition component-wise. Broadcasting is discussed in more detail in Section 4.2.1.

We now have enough information to consider a simple but non-trivial example, the smoothing of data by three-point averaging. Suppose f refers to a Python vector of data. We might smooth the data at interior points as follows

```
f_av=f.copy() # make a copy, just as for lists
for i in range(1,len(f)-1): # loop over interior points
    f_av[i]=(f[i-1]+f[i]+f[i+1])/3.0
```

This works well for small arrays but becomes very slow for larger vectors. Consider instead

```
f_av=f.copy() # as above
f_av[1:-1]=(f[ :-2]+f[1:-1]+f[2: ])/3.0
```

This *vectorized code* will execute much faster for large arrays since the implied loop will be executed using precompiled C code. One way to get the slicings correct is to note that in each slice [a:b], the difference a-b is the same.[2] Finally, we need to point out to beginners that there is an ***extremely important difference*** between arrays and *lists*. If l is a Python *list*, then l[:] is ***always*** a (shallow) copy, but for *numpy* arrays, slicings ***always*** reference the original array. That is why we had to enforce an explicit (deep) copy in the code snippets above.

[1] In fact, it does no such thing! However, the method it uses simulates this effect while being much more memory- and time-efficient.

[2] Recall that the slicing conventions imply [:b] is [0:b] and [a:] is [a:-0].

4.1.4 Ufuncs

The class of *universal functions* or *ufuncs* adds considerably to the usefulness of *numpy*. A *ufunc* is a function which when applied to a scalar generates a scalar, but when applied to an array produces an array of the same size, by operating component-wise. Some of the ufuncs which are most useful for scientists are shown in Table 4.1. Many of these will be familiar, but the docstrings are readily available, e.g.

```
np.fix? # or help(np.fix)
```

Table 4.1 Some commonly used *ufuncs* which can be applied to vectors. Those in the last column require two arguments. For documentation, try, e.g., `np.sign?` or `help(np.sign)`.

sign	cos	cosh	exp	power
abs	sin	sinh	log	dot
angle	tan	tanh	log10	vdot
real	arccos	arccosh	sqrt	
imag	arcsin	arcsinh		
conj	arctan	arctanh		
fix	arctan2			

One point which is not made clear in the documentation is the domain and range of each of these functions. For example, how does *numpy* interpret $\sqrt{-1}$ or $\cos^{-1} 2$? In pure Python, `math.sqrt(-1)` or `math.acos(2)` will produce an error and the program will stop. However, `cmath.sqrt(-1)` returns `1j`, for the argument is widened to a complex value.

The module *numpy* behaves differently, and the type of the argument is critical. Note that `np.sqrt(-1+0j)` takes a complex square root and returns `1j`, but `np.sqrt(-1)` produces a warning and returns `np.nan` or *not a number*, a float of indeterminate value. Further arithmetic may be performed on it, but the result will always be `np.nan`. Similar remarks apply to $\cos^{-1} 2$.

In *numpy*, direct division by zero `1.0/0.0` produces an error and the execution halts. However, if it occurs indirectly, e.g., within a loop, this is not the case. Consider

```
x = np.linspace(-2, 2, 5)
x
1.0/x
```

which produces a warning and `array([-0.5,-1.,inf,1.,0.5])`. Here `inf` behaves pretty much like infinity in further calculations. It available is as `np.inf`. Its negative is given by `np.NINF`. (There is an unfortunate asymmetry in notation here!)

There are of course many more *numpy* functions which can be applied to vectors and some of them are reviewed in Section 4.6.

In almost every non-trivial programme, there will be user-defined functions, and it is highly desirable that, where appropriate, they behave like *ufuncs*, in the sense that when applied to arrays of consistent dimensions, they return arrays of an appropriate dimension without invoking explicit loops over the components. In many cases, e.g., where only arithmetical operations and *ufuncs* are involved, this will be manifestly true.

However, if logical statements are involved, this may not be the case. Consider, e.g., the “top hat” function

$$h(x) = \begin{cases} 0 & \text{if } x < 0, \\ 1 & \text{if } 0 \leqslant x \leqslant 1, \\ 0 & \text{if } x > 1. \end{cases}$$

Using the techniques of Chapter 3, we might try to apply this definition via the code

```
import numpy as np
def h(x):
    """ return 1 if 0<=x<=1 else 0. """
    if x < 0.0:
        return 0.0
    elif x <= 1.0:
        return 1.0
    else:
        return 0.0

v=np.linspace(-2, 2, 401)
hv=h(v)
```

However, this fails, and the reason is very simple. If `x` has more than one element, then `x<0.0` is ambiguous. The next subsection shows how to resolve this problem.

4.1.5 Logical operations on vectors

If `x` is a scalar, then `x<0` is unambiguous. It must evaluate to `True` or `False`. But what if `x` is a *numpy* vector? Clearly, the right-hand side of the inequality is a scalar, and a moment’s thought should suggest that the assertion should be interpreted component-wise, so as to produce a vector of outcomes. Let us test this hypothesis using the interpreter.

```
1  import numpy as np
2  x=np.linspace(-2,2,9)
3  y=x<0
4  x
5  y
6  z=x.copy()
7  z[y]=-z[y]
8  z
```

The primary result, see lines 3 and 5, is that `y` is a vector of length 9 of type *bool*, of which precisely the first four components are `True`. This shows that the surmise above was indeed correct. Lines 6–8 of the code demonstrate an extremely useful feature of *numpy*. We may treat logical arrays such as `y` as slice definitions on one or both sides of an assignment. First, in line 6 we make a copy `z` of `x`. The right-hand side of line 7 first selects those components of z which are negative, i.e., those for which the corresponding component of `y` is `True`, then multiplies them by -1. The assignment then inserts precisely the modified components back into `z`. Thus we have computed `z=|x|` using implicit, C-style, loops. It should be clear that the intermediate logical array `y` is redundant. We could compute `z=|x|` more quickly and clearly by

```
z=x.copy()
z[z<0]=-z[z<0]
```

The reason for the copy is to the leave the original `x` unchanged. See the warning at the end of Section 4.1.3.

Acting on scalars, we can combine logical operators, e.g., `x>0 and x<1` by chaining them, e.g., `0<x<1`. Acting on arrays, we have to carry out the comparisons individually. As an example, we might compute the hat function $h(x)$, defined in the previous subsection, for the current vector `x` via

```
h=np.ones_like(x)
h[x<0]=0.0
h[x>1]=0.0
h
```

Suppose we want to construct a more complicated function, e.g., the example $k(x)$ below. The *numpy* module offers a function `select` of considerable generality to encapsulate a number, say M, of possible choices. Because a formal description is rather complicated, the reader is advised to look ahead to the example and corresponding code snippet while reading it. We assume that `x` is a one-dimensional array of length n and label its elements x_i $(0 \leqslant i < n)$. We assume the choices form a *list* C with members C_J $(0 \leqslant J < M)$. For example, C_0 might be $x \geqslant 2$. Next we might construct a *list* B of answers of length M. Each member B_J is an array of Boolean of length n whose elements are defined by $B_{Ji} = C_J(x_i)$. Finally, we need to supply a *list* of outcomes R of length M. Each member R_J is an array of length n whose elements are defined according to

$$R_{Ji} = \begin{cases} \text{desired outcome} & \text{if } B_{Ji} = True, \\ \text{arbitrary} & \text{if } B_{Ji} = False. \end{cases}$$

Then the `select` function produces a one-dimensional array `k` of length n, and operates as follows. There is an implicit outer loop over i where $0 \leqslant i < n$, and an implicit inner loop over J where $0 \leqslant J < M$. For each fixed i, we search through the array B_{Ji} skipping all the `False` values until we reach the first `True` one. We then set k_i equal to the corresponding R_{Ji}, break out of the J-loop and move to the next i. There is a potential complication. If B_{Ji} returns `False` for fixed i and all J, then the above procedure would produce an arbitrary value for k_i. To guard against this possibility, it might be prudent to supply a default scalar value for k_i to be used in such cases.

The formal syntax is, assuming that `x`, `C` and `R` are already defined, and $m = 3$

```
result=np.select([C0, C1, C2], [R0, R1, R2], default=0)
```

Let us illustrate this by an artificial but non-trivial example

$$k(x) = \begin{cases} -x & \text{if } x < 0, \\ x^3 & \text{if } 0 \leqslant x < 1, \\ x^2 & \text{if } 1 \leqslant x < 2, \\ 4 & \text{otherwise.} \end{cases}$$

The *numpy* code might look like

```
x=np.linspace(-1,3,9)
choices=[ x>=2, x>=1, x>=0, x<0 ]
outcomes=[ 4.0, x**2, x**3, -x ]
y=np.select(choices, outcomes)
```

or more succinctly

```
x=np.linspace(-1,3,9)
y=np.select([ x>=2, x>=1, x>=0, x<0 ],
            [ 4.0, x**2, x**3, -x ])
```

Note that the order of the choices (and their outcomes) needs some care. Can you find a different ordering?

In a certain sense, the `select` function is wasteful, for all possible outcomes need to be evaluated in advance. If this is a performance issue, then *numpy* offers an alternative approach called `piecewise`, involving functions. Consider the example

$$m(x) = \begin{cases} e^{2x} & \text{if } x < 0, \\ 1 & \text{if } 0 \leqslant x < 1, \\ e^{1-x} & \text{if } 1 \leqslant x. \end{cases}$$

We could code this definition as follows

```
def m1(x):
    return np.exp(2*x)
def m2(x):
    return 1.0
def m3(x):
    return np.exp(1.0-x)
conditions=[ x<0, x<1,1<=x ]
functions=[ m1, m2, m3 ]
x=np.linspace(-10,10,2001)
m=np.piecewise(x, conditions, functions)
```

Now the functions will only be called where they are needed. Note that the length of the functions *list* should either be the same as that of the conditions *list*, or one larger.

In this case, the final function is the default choice. There are other options, see the `np.piecewise` docstring for details.

It is perhaps unlikely that the functions `m1`, `m2` and `m3` in the snippet above will be used elsewhere. However, the syntax of `np.piecewise` requires functions to be present, and so this is a situation where the anonymous functions of Section 3.8.7 can be useful. Using them, a more self-contained definition might be

```
m=np.piecewise(x, [x<0, x<1, 1<=x], [lambda x: np.exp(2*x),
                    lambda x: 1.0, lambda x: np.exp(1.0-x)])
```

4.2 Two-dimensional arrays

We turn next to two-dimensional arrays. For this, and the more general case of *n*-dimensional arrays, the official documentation is far from perfect. Fortunately, once we have mastered the basic definitions, it is easy to see that the definitions are consistent with those for vectors, and that much of what we have already learned about one-dimensional arrays or vectors carries through.

A general *numpy* array carries three important attributes: `ndim` the number of dimensions or *axes*, `shape` a *tuple* of dimension `ndim`, which gives the extent or length along each axis, and `dtype`, which gives the type of each element. Let see us them first in a familiar context.

```
v=np.linspace(0,1.0,11)
v.ndim
v.shape
v.dtype
```

Here the shape is `(11,)`, which is a *tuple* with one element and is almost, but not quite, synonymous with the number 11.

A conceptually simple method of generating two-dimensional arrays explicitly is from a *list* of *lists*, e.g.

```
1  x=np.array([[0,1,2,3],[10,11,12,13],[20,21,22,23]])
2  x.ndim
3  x,shape
4  x.dtype
5  x
6  x[2]
7  x[ : ,1]
8  x[2][1]
9  x[2,1]
```

Note first the display produced by line 5 of the code snippet. The trailing axis, corresponding to the last element of the shape is displayed horizontally, then the next axis corresponding to the penultimate element of the shape is displayed vertically. (For large arrays, the default option is that only the corners are printed.) The next two lines 6 and 7 show how we can access individual rows and columns. The good news is that the Python slicing conventions still apply. Line 8 shows one way to access an individual element of `x`. First we create an intermediate temporary vector from the last row, and then accesses an element of that vector. It is however more efficient, especially for large arrays, to access the required element directly as in line 9.

4.2.1 Broadcasting

Suppose that `y` is an array with precisely the same shape as `x`. Then `x+y`, `x-y`, `x*y` and `x/y` are arrays of the same shape, where the operations are carried out componentwise, i.e., component by component. In certain circumstances, these operations are well-defined even when the shapes of `x` and `y` differ, and this is called *broadcasting*. We generalize to a situation where we have a number of arrays, not necessarily of the same shape, to which we are applying arithmetical operations. At first sight broadcasting seems somewhat strange, but it is easier to grasp if we remember two rules:

- The *first rule of broadcasting* is that if the arrays do not have the same number of dimensions, then a “1” will be repeatedly prepended to the shapes of the smaller arrays until all the arrays have the same number of axes.
- The *second rule of broadcasting* ensures that arrays with a size of 1 along a particular dimension or axis act as if they had the size of the array with the largest size along that dimension. The value of the array element is assumed to be the same along that dimension for the “broadcasted” array.

As a simple example, consider the array `v` with shape `(11,)` introduced on the previous page. What does `2*v` mean? Well `2` has shape `(0,)` and so by the first rule we augment the shape of “2” to `(1,)`. Next we use the second rule to increase the shape of “2” to `(11,)`, with identical components. Finally, we perform componentwise multiplication to double each element of `v`. The same holds for other arithmetic operations.[3] However, if `w` is a vector with shape `(5,)`, the broadcasting rules do not allow for the construction of `v*w`.

To see array arithmetic with broadcasting in action, consider

```
x=np.array([[0,1,2,3],[10,11,12,13],[20,21,22,23]])
r=np.array([2,3,4,5])
c=np.array([[5],[6],[7]])
2*x
x*r
r*x
c*x
```

[3] We have already seen this in Section 4.1.3. The discussion there is fully consistent with the broadcasting rules here.

Note that since `x*r=r*x`, `x*r` is NOT the same as matrix multiplication, nor is `x*x` matrix multiplication. For these, try `np.dot(x,r)` or `np.dot(x,x)`. A brief introduction to matrix arithmetic will be given in Section 4.8.

4.2.2 Ab initio constructors

Suppose we are given vectors of x-values x_i, where $0 \leqslant i < m$, and y-values y_k, where $0 \leqslant k < n$, and we want to represent a function $u(x, y)$ by grid values $u_{ik} = u(x_i, y_k)$. Mathematically, we might use a $m \times n$ array ordered in *matrix form*, e.g., for $m = 3$ and $n = 4$

$$\begin{array}{cccc} u_{00} & u_{01} & u_{02} & u_{03} \\ u_{10} & u_{11} & u_{12} & u_{13} \\ u_{20} & u_{21} & u_{22} & u_{23} \end{array}$$

We are thinking of x increasing downwards and y increasing rightwards. However, in the image-processing world many prefer to require x to increase rightwards and y to increase upwards, leading to *image form*

$$\begin{array}{ccc} u_{03} & u_{13} & u_{23} \\ u_{02} & u_{12} & u_{22} \\ u_{01} & u_{11} & u_{21} \\ u_{00} & u_{10} & u_{20} \end{array}$$

Clearly, the two arrays are linear transformations of each other. Because examples in the literature often use arrays corresponding to symmetric matrices, the differences are rarely spelled out explicitly. We therefore need to exhibit care.

For vectors, we found that the most useful ab initio constructors were the *numpy* functions `np.linspace` and `np.arange`, depending on whether we want to model closed or half-open intervals. For two-dimensional arrays, there are four possible intervals. As a concrete example, we try to construct explicitly a grid with $-1 \leqslant x \leqslant 1$ and $0 \leqslant y \leqslant 1$ with a spacing of 0.25 in both directions, and perform a simple arithmetical operation on it.

Perhaps the easiest to understand is the `np.meshgrid` constructor. We first construct vectors `xv` and `yv` which define the two coordinate axes for the intervals, then two arrays `xa` and `ya` on which y and x respectively are held constant. Finally, we compute their product.

```
xv=np.linspace(-1, 1, 9)
yv=np.linspace(0, 1, 5)
[xa,ya]=np.meshgrid(xv,yv)
xa
ya
xa*ya
```

By constructing either or both of the vectors `xv` and `yv` using `np.arange`, we can deal with the half-open possibilities. Notice from the shape of `xa` or `ya` that `np.meshgrid` uses *image form*.

The `np.mgrid` and `np.ogrid` operators use rather different syntax, cobbled together from slicing notation, and the representation of complex numbers. We illustrate this first in one dimension. Try out

```
1 np.mgrid[-1:1:9j]
2 np.mgrid[-1:1:0.25]
```

Note that the first line, with pure imaginary spacing, mimics the effect of the one-dimensional `np.linspace` while the second emulates `np.arange`. Although unfamiliar, this notation is succinct and generalizes to two or more dimensions, where it uses *matrix form*.

```
1 [xm,ym]=np.mgrid[-1:1:9j, 0:1:5j]
2 xm
3 ym
4 xm*ym
```

This is shorter than the first code snippet of this section, but achieves the same result up to a linear transformation. The adaptation to half-closed intervals is handled by line 2 of the snippet before this one.

For large arrays, especially with more dimensions, much of the data in `xm` and `ym` may be redundant. This deficiency is addressed by the `np.ogrid` variant which also uses *matrix form*.

```
1 [xo,yo]=np.ogrid[-1:1:9j, 0:1:5j]
2 xo
3 yo
4 xo.shape
5 yo.shape
6 xo*yo
```

You should verify that `xo` and `yo` have shapes chosen so that the broadcasting rules apply, and that `xm*ym` (from the last snippet) and `xo*yo` are identical.

In the previous section, we introduced the three functions `np.zeros`, `np.ones` and `np.empty` as vector constructors. They work equally well as constructors for more general arrays. All that is necessary is to replace the first argument, which was the length of the vector, by a *tuple* defining the shape of the array, e.g.

```
x=np.zeros((4,3),dtype=float)
```

defines a 4×3 array of floats, in *matrix form*, initialized to zeros. Of course, a *tuple* with one element, e.g., `(9,)` can be replaced by an integer, e.g., 9 in this context.

4.2.3 Look alike constructors

Much of what was said earlier for vector constructors applies here. In particular, the extremely useful `np.zeros_like`, `np.ones_like` and `np.empty_like` constructors can be used with array arguments.

Another very useful look alike constructor is `np.reshape`, which takes an existing array (or even a *list*) and a *tuple*, and, if possible, recasts the array into another whose shape is determined by the *tuple*. A common example is

```
l=range(6)
a=np.reshape(l,(2,3))
a
```

but a potentially more useful idea is

```
v=np.linspace(0, 1.0, 5)
vg=np.reshape(v, (5, 1))
vg.shape
vg
```

4.2.4 Operations on arrays and ufuncs

For arrays of the same shape, the usual arithmetic operations work as expected. There are no surprises. Operations between arrays of different shapes, or even different dimensions, are permitted if the broadcasting rules allow coercion into a common shape. Consider the example

```
u=np.linspace(10,20,3)
vg=np.reshape(np.linspace(0,7,15),(5,3))
u+vg
```

Ufuncs which take a single argument, e.g., `np.sin(x)`, work exactly as expected for arrays. Those which take more arguments, e.g., `np.power(x,y)`, also work exactly as expected if the arguments have the same shape, and are subject to the broadcasting rules otherwise.

Very little needs to be said about slicing. Although various short cuts do exist, it is safest and clearest to exhibit both dimensions, and we can slice one or both dimensions in the obvious way, e.g.

```
vg=np.reshape(np.linspace(0,7,15),(5,3))
vg
vg[1:-1,1: ]=9
vg
```

4.3 Higher-dimensional arrays

The good news here is that almost everything that has been said about two-dimensional arrays carries over into higher dimensions. The single exception is the `np.meshgrid` function, which is restricted to two dimensions. Here is a simple but instructive example of using `np.ogrid` in three dimensions.

```
[xo,yo,zo]=np.ogrid[-1:1:9j, 0:10:5j, 100:200:3j]
xo
yo
zo
xo+yo+zo
```

4.4 Domestic input and output

In this section, we look at how to communicate with humans, other programmes or store intermediate results. We can distinguish at least three scenarios and we consider simple examples of input and output processes for them. The first supposes that we are given a text file which contains both words and numbers, and we wish to read in the latter to *numpy* arrays. Conversely, we might want to output numbers to a text file. The second is similar but simpler, in that we want to process just numbers into and from text files. The third scenario is similar to the second but for reasons of speed and economy of space, we wish to use binary files which can be processed by another *numpy* programme, possibly on a different platform.

There remains the generalization of dealing with data produced by another programme, perhaps a spreadsheet or a "number crunching" non-Pythonic programme. This is discussed in Section 4.5.

4.4.1 Discursive output and input

For the sake of brevity, we assume we are reporting the result, a single float, for each of four quarters. Given *numpy* arrays containing them, we first produce a file called `q4.txt` which can be read both by humans and by other programmes. In fact, this task uses mainly core Python ideas.

```
1  import numpy as np
2  quarter=np.array([1,2,3,4],dtype=int)
3  results=np.array([37.4,47.3,73.4,99])
4  outfile=open("q4.txt","w")
5  outfile.write("The results for the first four quarters\n\n")
6  for q,r in zip(quarter, results):
7      outfile.write("For quarter %d the result is %5.1f\n" %
8                    (q,r))
9  outfile.close()
```

Lines 1–3 generate some artificial data. The Python function **open** in line 4 creates a file called `q4.txt` and opens it for writing.[4] See the docstring for further details of the **open**

[4] The names of files depend strongly on the choice of operating system. The choice here assumes *Unix* and creates or overwrites a file in the current working directory. You should replace the first *string* by the valid address of the desired file in your operating system.

function. Within the programme, the file has the arbitrary identifier `outfile`. In line 5, we write a header *string* terminated by a newline character `n` and a second newline to insert a blank line below the header. Line 6 introduces a new feature, the Python **`zip`** function. This takes a number of iterable objects, here the two *numpy* arrays, and returns the next object from each of them as a *tuple*. The **`for`** loop terminates when the shortest iterable object is exhausted. Within the loop, we write a formatted *string* containing elements from each of the arrays, and terminated with a newline. Finally, in line 9 we close the file. You can check the correctness of this snippet by looking for the file `q4.txt` (it should be in your current directory), and reading it with your favourite text editor.

Now let us consider the reverse process. We have a text file `q4.txt`, about which we know the following. The first two lines form a header, which can be discarded. Next follow an unknown number of lines, which all have the same form: a sequence of "words" separated by white space. Labelling the words in each line from zero, we wish to build two *numpy* arrays: one containing word 2 as an **`int`** and the other word 6 as a **`float`**. (To see why we want words 2 and 6, look at the text file.) The dimension of the arrays will be the same as the number of lines to be read. The following snippet carries out this task and, like the previous one, it is mainly pure Python.

```
1  infile=open("q4.txt","r")
2  lquarter=[]
3  lresult=[]
4  temp=infile.readline()
5  temp=infile.readline()
6  for line in infile:
7      words=line.split()
8      lquarter.append(int(words[2]))
9      lresult.append(float(words[6]))
10 infile.close()
11 import numpy as np
12 aquarter=np.array(lquarter,dtype=int)
13 aresult=np.array(lresult)
```

Lines 1 connects the text file to the programme for reading, and line 10 eventually disconnects it. Lines 2 and 3 create empty *lists* to hold the numbers. The file object `infile` is a *list* of *strings*, one per line. Line 4 then reads the first *string* into a *string* in memory with identifier `temp`, and so effectively deletes the first element of `infile`. Line 5 repeats the process, and so we have discarded the header. (For a more verbose header, a **`for`** loop might be more appropriate.) Now we loop through the remaining lines in `infile`. Acting on the *string* `line`, the string function `split` breaks it into a *list* of substrings, here called `words`. The breaks occur at each white space inside `line` and the white space itself is eliminated. (We could use a different splitting character, e.g., a comma. See the docstring for details. Now `words[2]` contains a *string* which we wish to convert to an integer, and the **`int`** function in line 8 does this. Next we append the integer to the `lquarter` *list*. Line 9 repeats the process for the floatingpoint number. Finally, in lines 12 and 13 we construct *numpy* arrays from the *lists*. Note that at no point have we needed to know how many data lines were present.

4.4.2 *Numpy* text output and input

Text-based output and input of *numpy* data are much more succinct. As an example, we first construct four vectors and an array and show how to save them to text files.

```
1  len=21
2  x=np.linspace(0,2*np.pi,len)
3  c=np.cos(x)
4  s=np.sin(x)
5  t=np.tan(x)
6  arr=np.empty((4,len),dtype=float)
7  arr[0, : ]=x
8  arr[1, : ]=c
9  arr[2, : ]=s
10 arr[3, : ]=t
11 np.savetxt('x.txt',x)
12 np.savetxt('xcst.txt',(x,c,s,t))
13 np.savetxt('xarr.txt',arr)
```

The first ten lines merely set up some data. Line 11 creates a text file, opens it, copies the vector `x` to it using the default format `%.18e`, and closes the file. Line 12 shows how to copy many vectors (or arrays), but of equal shapes. We merely parcel them into a *tuple*. Line 13 shows how to save an array into a file. For more possibilities, see the docstring for `np.savetxt`.

With the current setup, reading the files is easy.

```
xc=np.loadtxt('x.txt')
xc,cc,sc,tc=np.loadtxt('xcst.txt')
arrc=np.loadtxt('xarr.txt')
```

There are of course possibilities that are more elaborate. See the docstring for details.

4.4.3 *Numpy* binary output and input

Writing and reading to binary files should be faster and produce more compact files because the conversions to and from text are not needed. The obvious disadvantage is that the files cannot be read by humans. Since different platforms encode numbers in different ways, there is a danger that binary files may be highly platform-dependent. *Numpy* has its own binary format which should guarantee platform-independence. However, these files cannot be read easily by other non-Python programmes.

A single vector or array is easily written or read. Using the array definitions above, we would write the array to file with

```
np.save('array.npy',arr)
```

and we could recover the array with

```
arrc = np.load('array.npy')
```

File opening and closing is handled silently provided we use the `.npy` postfix in the file name.

A collection of arrays of possibly varying shapes is handled by creating a zipped binary archive. Using the array definitions above, we might write the vectors with a one liner

```
np.savez('test.npz',x=x,c=c,s=s,t=t)
```

Recovering the arrays is a two stage process.

```
1 temp=np.load('test.npz')
2 temp.files
3 xc=temp['x']
4 cc=temp['c']
5 sc=temp['s']
6 tc=temp['t']
```

In line 2, we display the names of the files in the archive, which has the identifier `temp`. Lines 3–6 show how to recover the arrays. Notice that in both processes, file opening, zipping, unzipping and closing is handled silently. There are many other ways of performing binary input/output, in particular the *struct* module. See the documentation for details.

4.5 Foreign input and output

We now turn, albeit very briefly, to the problem of how to read in data from a non-Python source, and the answer depends on the size of the data.

4.5.1 Small amounts of data

Relatively small amounts of data are often available as *comma separated value (CSV)* files, time-series data, observational or statistical data or SQL tables or spreadsheet data. We could try to construct a reader along the lines of Section 4.4.1, but we would need to safeguard against missing or malformed data entries. Fortunately, this problem has already been handled by the *Python data analysis library* which is available as the package *pandas*.[5] This should have been included with the recommended Python distributions (see section A.1), but if not, then it can be downloaded from its website or

[5] Its website, `http://pandas.pydata.org`, is highly informative.

from the Python Package Index (see Section A.3) and that section also gives instructions for installing the package. However, before installation, interested readers should consult the excellent documentation available from the website. A recent more discursive textbook by one of the *pandas* developers, McKinney (2012), contains a wealth of potentially useful examples. It would be a waste of resources to repeat them or similar ones here, and that is why this section is so brief.

4.5.2 Large amounts of data

Large amounts of data, typically terabytes, are usually produced in a standard data format, and we shall consider here the popular *Hierarchical Data Format (HDF)* currently in version 5. The *HDF5* package[6] by itself consists merely of a format definition and associated libraries. To actually use it you need an *application programming interface (API)*. There are officially supported interfaces for C, C++, Fortran90 and Java. Clearly, HDF5 fulfills a need in scientific number crunching! Therefore, the scientific packages Mathematica, Matlab, R and SciLab offer “third party” APIs. Python is perhaps unique in that it offers two and one half, aimed at slightly different audiences.

The half refers to a minimalistic interface within *pandas*, see the relevant documentation. The package *h5py*[7] offers both low- and high-level access to HDF5, aiming for very similar functionality to its compiled language API cousins. The other package *PyTables*[8] offers only a high-level interface, but then provides additional facilities such as sophisticated indexing and query capabilities that we would expect to find only in a database.

The HDF5 documentation is not helpful in deciding which Python API to use, and you really need to study all three of them before making a decision. Note that installation of a sufficiently up-to-date version of HDF5 on Unix/Linux platforms is not for the faint-hearted. This is one instance where Windows users get a helping hand in that both *h5py* and *PyTables* offer binary installations, which include a private copy of HDF5.

4.6 Miscellaneous ufuncs

Numpy contains a number of miscellaneous ufuncs, and we group some of the most useful ones here. For a more complete review, see the *numpy* reference manual (Numpy Community 2013*a*).

4.6.1 Maxima and minima

The basic maximum function has the syntax `np.max(array,axis=None)`. Its usage is illustrated below, where the last two lines build subarrays maximizing over columns and rows respectively.

[6] Its website is `http://www.hdfgroup.org/HDF5/`.
[7] Its website, `http://www.h5py.org` offers documentation and downloads.
[8] There is a very informative website at `http://www.pytables.org`.

```
x=np.array([[5,4,1],[7,3,2]])
np.max(x)
np.max(x,axis=0)
np.max(x,axis=1)
```

Of course, if one of the elements is a `nan`, it is automatically the maximum of the array and any subarrays containing it. The function `np.nanmax` behaves almost identically, but ignores `nan` values. Minima are handled by `np.min` and `np.nanmin`. The range (**p**eak **t**o **p**eak) of the array, or subarray thereof, is given by `np.ptp(x)`. Look at the function docstrings for further information.

Incidentally, `np.isnan(x)` returns an array of Booleans of the same shape as `x` with a `True` entry for each instance of a `nan`. The function `np.isfinite(x)` does the opposite, returning `False` for each `nan` or `inf`.

4.6.2 Sums and products

We can sum the elements of any array `x` with `np.sum(x)`. If `x` has more than one dimension (we use two for illustrative purposes), then `np.sum(x,axis=0)` sums over the individual columns, and `np.sum(x,axis=1)` sums over the rows. With the same syntax, the function `np.cumsum(x)` produces an array of the same shape as `x` but with cumulative sums. The functions `np.prod` and `np.cumprod` do the same but for products. As always, consult the function docstrings for examples of usage.

4.6.3 Simple statistics

The numpy functions `np.mean` and `np.median` have the same syntax as the functions discussed above and produce the mean or median either for the whole array or along the specified axis. The averaging function

```
np.average(x,axis=None,weights=None)
```

is slightly different. If the parameter `weights` is specified, it must be an array with either the same shape as `x` or the shape appropriate to the chosen axis. The result is an appropriately weighted average.

The syntax of the variance function is

```
np.var(x,axis=0,ddof=0)
```

The first two arguments should be familiar. Note that if the elements of `x` are complex, the squaring operation uses a complex conjugate so as to generate a real value. The computation of the variance involves a division by n, where n is the number of elements involved. If `ddof`, the “delta degrees of freedom”, is specified, then the divisor is replaced by `n-ddof`. The function `np.std` is very similar and computes the standard deviation.

Numpy also contains a number of functions for correlating data. The most used ones are `np.corrcoeff`, `np.correlate` and `np.cov`. Their syntax is rather different to the functions discussed above, and so the relevant docstrings need careful perusal before use.

4.7 Polynomials

Polynomials in a single variable occur very frequently in data analysis, and *numpy* offers several approaches for manipulating them. A concise way to describe a polynomial is in terms of its coefficients, e.g.

$$c_0x^4 + c_1x^3 + c_2x^2 + c_3x + c_4 \leftrightarrow \{c_0, c_1, c_2, c_3, c_4\} \leftrightarrow [c[0], c[1], c[2], c[3], c[4]]$$

stored as a Python *list*.

4.7.1 Converting data to coefficients

A rather abstract approach to define a polynomial is to specify a *list* of its roots. This defines the coefficient *list* only up to an overall factor, and the function `np.poly` always chooses `c[0]=1`, i.e., a *monic* polynomial. An example is given below.

More often we have a list or array `x` of x-values and a second one `y` of y-values and we seek a "best fit" *least squares approximation* by an unknown polynomial of given order `n`. This is called *polynomial interpolation* or *polynomial regression*, and the function `np.polyfit(x,y,n)` does precisely that.

4.7.2 Converting coefficients to data

If you are given the coefficient array, then `np.roots` delivers the roots. More usefully, given the coefficient array and a single x-value or an array of x-values, `np.polyval` returns the corresponding y-values. The following snippet illustrates the ideas of this and the preceding subsection.

```python
import numpy as np

roots=[0,1,1,2]
coeffs=np.poly(roots)
coeffs
np.roots(coeffs)
x = np.linspace(0,0.5*np.pi,7)
y=np.sin(x)
c=np.polyfit(x,y,3)
c
y1=np.polyval(c,x)
y1-y
```

4.7.3 Manipulating polynomials in coefficient form

The functions `np.polyadd`, `np.polysub`, `np.polymult` and `np.polydiv` handle the four basic arithmetic functions. The function `np.polyder` obtains the x-derivative of a given polynomial, while `np.polyint` performs x-integration, where the arbitrary constant is set to zero. As ever, more information is given in the docstrings.

4.8 Linear algebra

4.8.1 Basic operations on matrices

For arrays of the same shape, addition and multiplication by a scalar have already been defined and they act componentwise. Mathematicians often think of two-dimensional arrays as matrices.

If `A`, `B`, ..., are *numpy* arrays with two dimensions, they will be called *matrices*. Note that unlike the usual algebra conventions, indices start with zero rather than one. The transpose of `A` is available as `A.transpose()`, or more succinctly as `A.T`. Note that *numpy* does not distinguish between column and row vectors. This means that if `u` is a one-dimensional array or vector, then `u.T = u`.

We already know how to build zero matrices, e.g., `z=np.zeros((4,4))`. The function `np.identity` creates identity matrices, e.g.

```
I=np.identity(3,dtype=float).
```

A more general form of this is the eye function. `np.eye(m,n,k,dtype=float)` returns a $m \times n$ matrix, where the kth diagonal consists of ones and the other elements are zero. Examine carefully

```
C=2*np.eye(3,4,-1)+3*np.eye(3,4,0)+4*np.eye(3,4,1)
C
```

Next consider a different situation where we have a set of m vectors `v1`, `v2`, ..., `vm` all of length n and we want to construct a $m \times n$ matrix with the vectors as rows. This is accomplished easily with the `np.vstack` function as the following example shows.

```
1  v1=np.array([1,2,3])
2  v2=np.array([4,5,6])
3  rows=np.vstack((v1,v2))
4  rows
5  cols=rows.T
6  cols
```

Note that because the actual number of arguments in line 3 is variable, we have to wrap them in a *tuple* first, because `np.vstack` takes precisely one argument. We could of course encode them as columns creating a $n \times m$ matrix, as the last two lines of the code snippet show.

Addition and subtraction of matrices, and multiplication by a scalar are covered by the standard *numpy* componentwise operators. Matrix multiplication is via the `np.dot` function, see below for an example.

4.8.2 More specialized operations on matrices

The module *numpy* contains a submodule `linalg` which handles operations on matrices that are more specialized. Suppose that A is a square $n \times n$ matrix. The determinant of A is given by `np.linalg.det` and, assuming A is non-singular, its inverse is obtained with `np.linalg.inv`. The snippet illustrates the use of these functions.

```python
import numpy as np

a=np.array([[4,2,0],[9,3,7],[1,2,1]])
a
np.linalg.det(a)
b=np.linalg.inv(a)
b
np.dot(b,a)
```

The function `np.linalg.eig` can be used to generate eigenvalues and eigenvectors. This function delivers the eigenvectors as columns of a n-row matrix. Each column has unit Euclidean length, i.e., the eigenvectors are normalized.

```python
import numpy as np

a=np.array([[4,2,0],[9,3,7],[1,2,1]])

evals, evecs = np.linalg.eig(a)
eval1 = evals[0]
evec1 = evecs[:,0]
np.sum(evec1*evec1)
np.dot(a, evec1) - eval1*evec1
```

There are many more functions available in the `np.linalg` module. Try `np.linalg?` for details. They are all Python wrappers around the corresponding LAPACK library subroutines.

4.8.3 Solving linear systems of equations

A very common problem is the need to obtain a "solution" $\mathbf{x}$ to a linear system of equations

$$A\mathbf{x} = \mathbf{b}, \tag{4.1}$$

where A is a matrix and $\mathbf{x}$ and $\mathbf{b}$ are vectors.

The simplest case is where A is $n \times n$ and is non-singular, while $\mathbf{x}$ and $\mathbf{b}$ are n-vectors.

Then the solution vector $\mathbf{x}$ is well-defined and unique. It is straightforward to obtain a numerical approximation to it, as the next snippet shows. We shall treat two cases simultaneously, with

$$A = \begin{pmatrix} 3 & 2 & 1 \\ 5 & 5 & 5 \\ 1 & 4 & 6 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 5 \\ 5 \\ -3 \end{pmatrix} \quad \text{and} \quad \mathbf{b} = \begin{pmatrix} 1 \\ 0 \\ -\frac{7}{2} \end{pmatrix}$$

```python
import numpy as np
a=np.array([[3,2,1],[5,5,5],[1,4,6]])
b=np.array([[5,1],[5,0],[-3,-7.0/2]])
x=np.linalg.solve(a,b)
x
np.dot(a,x)-b
```

Here the last line checks the solution.

This is but the briefest introduction to the linear algebra capabilities of *numpy*. See also Section 4.9.1.

4.9 More *numpy* and beyond

The *numpy* module contains far more resources than those sketched here, in particular several specialized groups of functions including:

- *numpy.fft*, a collection of discrete Fourier transform routines;
- *numpy.random*, which generates random numbers drawn from a variety of distributions. We offer a snapshot of some of its possibilities in Section 7.6 on stochastic differential equations.

Information about these, as well as some specialist functions is available via the interpreter, and also in the reference manual, Numpy Community (2013*a*).

4.9.1 *Scipy*

The *scipy* module contains a wide variety of further specialized groups of functions. A subset of the available groups is:

- *scipy.special* makes many special functions, e.g., Bessel functions, readily available;
- *scipy.integrate* contains various quadrature functions and, most importantly, routines for solving the initial value problems for systems of ordinary differential equations (see Section 7.1);
- *scipy.optimize*, which covers optimization and root finding functions, for which a simple example is given below;
- *scipy.fftpack* which contains a more extensive set of routines involving discrete Fourier transforms;

- *scipy.linalg*, *scipy.sparse*, *scipy.sparse.linalg*, which extend considerably the linear algebra capabilities of *numpy*.

Accessing these within the interpreter is slightly different to the *numpy* case, e.g.

```
import scipy.optimize
scipy.optimize?
scipy.optimize.rootf?
```

As a simple example, we address a problem which arises later in Section 7.4.5, to determine all positive solutions of

$$\coth v = v.$$

A rough sketch of the graphs of each side of the equation as functions of v shows that there is precisely one solution, and that it is greater than 1. Using the information we have gleaned from the previous code snippet, the following code solves the problem.

```
import numpy as np
import scipy.optimize as sco

def fun(x):
    return np.cosh(x)/np.sinh(x)-x

rooots=sco.fsolve(fun,1.0)
root= roots[0]
print "root is %15.12f and value is %e" % (root, fun(root))
```

The required root is approximately 1.1996786402577135, for which the function value is about 3×10^{-14}.

Fuller details, including groups not mentioned here, can be found in the reference manual, Scipy Community (2012). On the whole, the documentation reaches a high standard, if not quite the equal of that for *numpy*.

4.9.2 *Scikits*

In addition, there are a number of scientific packages which are not included in *scipy* for various reasons. They may be too specialized for inclusion, they may involve software licences, e.g., GPL, which are incompatible with *scipy*'s BSD licence or quite simply they reflect work in progress. Many of these can be accessed via the Scikits web page,[9] or via the Python Package Index (see Section A.3). We shall demonstrate their usage with concrete examples. First, in the treatment of boundary value problems for ordinary differential equations, in Section 7.4, the package `scikits.bvp1lg` will be needed. In Section 7.5, which discusses simple delay differential equations, we have chosen not to use the Scikits package `skikits.pydde`, because the package `pydelay`, downloadable

[9] `http://scikits.appspot.com/`

from its web page,[10] appears to be both more versatile and user friendly. Installation of both packages is identical, and is described in Section A.3.

[10] `http://pydelay.sourceforge.net`