Python\_Scripting\_Computational\_Science\_C04

Chapter 4

Numerical Computing in Python

There is a frequent need for processing large amounts of data in computa- tional science applications. Storing data in lists and traversing lists with plain Python for loops leads to slow code, especially when compared with similar code in compiled languages such as Fortran, C, or C++. Fortunately, there is an extension of Python, commonly called Numerical Python, or abbreviated NumPy, which offers efficient array computations. Numerical Python has a fixed-size, homogeneous (fixed-type), multi-dimensional array type and lots of functions for various array operations. The result is a dynamically typed environment for array computing similar to basic Matlab. Usually, the speed of NumPy operations is quite close to what is obtained in pure Fortran, C, or C++.

A glimpse of Numerical Python is presented in Chapter 2.2.5. A more comprehensive, yet compact introduction to basic NumPy computing, is pro- vided in Chapter 4.1. Some non-trivial vectorization techniques are described in Chapter 4.2. More advanced functionality of Numerical Python is listed in Chapter 4.3. Two major scientific computing packages for Python, Scien- tificPython and SciPy, are outlined in Chapter 4.4, along with the Python– Matlab interface and a listing of many useful third-party modules for numer- ical computing in Python.

There are three different implementations of Numerical Python: Numeric, numarray, and numpy. The latter is the newest and contains all features of the former two, plus some new enhancements. It is therefore recommended to apply numpy. This package is documentended in a book which I highly recom- mend to purchase. There are also some resources on the web that exemplify usage of numpy (see doc.html). The free documentation of the old Numeric im- plementation can be used to some extent for numpy programming, but there are some significant changes, especially in coding style.

To use numpy it is common to perform a from numpy import \*

This import statement is require for the examples in this chapter to work.

Mixing Different Numerical Python Implementations. There is much code around using the old Numeric implementation. Numeric arrays work well with numpy arrays, but I will strongly recommend to port Numeric code to numpy, especially since there are fundamental problems with Numeric on 64-bit ma- chines. Usually, the port is a quite simple process as explained well in the

numpy manual and on the webpages. Most of the Numeric functions are mir- rored in numpy. However, numpy encourages the use of array methods instead of functions. For example, in Numeric one can resize an array a to length n with the function call resize(a, n), while the recommended numpy style is a.resize(n). In this book we adapt to the new numpy style.

4.1 A Quick NumPy Primer

In the following sections we cover how to create arrays (Chapter 4.1.1), how to work with indices and slices (Chapter 4.1.2), how to compute with arrays without (slow) loops and explicit indexing (Chapter 4.1.4), how to determine the type of an array and its elements (Chapter 4.1.6), as well as a discussion of how arithmetic expressions generate temporary arrays (Chapter 4.1.4).

All of the code segments to be presented are collected in the script

src/py/intro/NumPy\_basics.py

4.1.1 Creating Arrays

Creating NumPy arrays can be done in a variety of ways. Some common methods are listed below.

Array of Specified Length, Filled with Zeros.

>>> from numpy import \* >>> n = 4

>>> a = zeros(n)

>>> print a

[0. 0. 0. 0.]

>>> a

array([ 0., 0., 0., 0.]) >>> p = q = 2

>>> a = zeros((p,q,3))

>>> print a

[[[ 0. 0. 0.]

[0. 0. 0.]]

[[0. 0. 0.]

[ 0. 0. 0.]]]

# one-dim. array of length n # str(a)

# repr(a)

# p\*q\*3 three-dim. array

By default,zeros generates float elements, which has the same precision as the C type double. Giving a second argument like int, complex, int16 (two- byte integers as frequently used in sound arrays), or bool, other element types can be generated.

There is also corresponding ones function which fills the array with unit values.

Copying an Existing Array. Sometimes we have an array x and want to make a new array r with the same size as x and the same element type. We can either copy x,

r = x.copy()

or we can call zeros with size and element type taken from x: r = zeros(x.shape, x.dtype)

The shape and dtype attributes of arrays are explained later.

Array with a Sequence of Numbers. The call linspace(start, stop, n) pro- duces a set of n uniformly distributed numbers starting with start and ending with stop. For example,

>>> x = linspace(-5, 5, 11)

>>> print x

[-5.-4.-3.-2.-1. 0. 1. 2. 3. 4. 5.]

A special compact syntax is available through the syntax r\_[start,stop,incj]:

>>> a = r\_[-5:5:11j] # same as linspace(-1, 1, 11) >>> print a

[-5.-4.-3.-2.-1. 0. 1. 2. 3. 4. 5.]

Note that in the compact syntax the step is specified as an imaginary number with a j at the end.

Instead of specifying the number of array elements one can specify the increment between two numbers in the sequence, here a unit increment:

>>> x = arange(-5, 5, 1, float)

>>> print x

[-5. -4. -3. -2. -1. 0. 1. 2. 3. 4.]

Note that the upper limit of the interval, here specified as 5, is ruled out be- cause arange works like range, i.e., the largest element is less than the upper limit. Unfortunately, because of round-off errors, the arange function is unre- liable with respect to this behavior, see page 166. We therefore recommend to avoid arange and instead use linspace from numpy or the function seq from scitools.numpytutils:

>>> x = seq(-5, 5, 1)

>>> print x

[-5.-4.-3.-2.-1. 0. 1. 2. 3. 4. 5.]

seq works as arange, but the upper limit (here 5) is ensured to be included in the sequence. Each element also becomes a floating-point number by default. Also for arange there is a quick variant using r\_, as for linspace: Also

here there is a quick variant:

>>> a = r\_[-5:5:1.0]

>>> print a

[-5. -4. -3. -2. -1. 0. 1. 2. 3. 4.]

With 1 as step instead of 1.0 (r\_[-5:5:1]) the elements in a become integers. Array Construction from a Python List. The array function makes an array

out of a Python list, e.g.,

>>> pl = [0, 1.2, 4, -9.1, 5, 8] >>> a = array(pl)

Nested Python lists can be used to construct multi-dimensional NumPy ar- rays:

>>> x = [0, 0.5, 1]; y = [-6.1, -2, 1.2] # Python lists >>> a = array([x, y]) # form array with x and y as rows

If the lists contain integers only, array will produce integer elements in the resulting array unless we add a type argument:

>>> z = array([1, 2, 3])

>>> print z

[1 2 3]

>>> z = array([1, 2, 3], float) >>> print z

[ 1. 2. 3.]

Having a NumPy array, its tolist method creates a Python list. This can be useful since not all functionality for Python lists is available for NumPy arrays. For example, we can locate a specific element in the first row (x values) using list functionality:

>>> i = a.tolist()[0].index(0.5) >>> i

1

Sometimes we have some object a that can be an array, a list, or a tuple, and we want to transform it to a NumPy array. The call

>>> a = asarray(a)

is then handy because it will do nothing if a already is a NumPy array. Otherwise it will take a copy of the data and fill a NumPy array. Especially in functions where you need to work with a NumPy array but would like to offer users to send in anything that can be transformed to a NumPy array, the asarray function is handy.

Changing Array Dimensions. The reshape method or the shape attribute is used both to set and read the array dimensions:

>>> a = array([0, 1.2, 4, >>> a.shape = (2,3)

>>> a.shape = (a.size,) >>> a.shape

(6,)

>>> a = a.reshape(2,3)

>>> a.shape

(2, 3)

4.1. A Quick NumPy Primer 135

-9.1, 5, 8])

# turn a into a 2x3 matrix

# turn a into a vector of length 6 again

# same effect as setting a.shape # get a’s shape

The total number of elements in an array is found by size(a). (A plain len(a) returns 2, i.e., the length of the first dimension, just as len would behave when applied to a nested Python list.)

Array Initialization from a Python Function. We can make a function that maps an array index to an array value and use this function to initialize an array:

>>> def myfunc(i, j):

... ...

* >>> #
* >>> a  
   >>> a  
   array([[ 4., 5., 6., 7.,  
     
  [ 6., 8., 10., 12., [ 6., 9., 12., 15.,

return (i+1)\*(j+4-i)

make 3x6 array where a[i,j] = myfunc(i,j): = fromfunction(myfunc, (3,6))

Fortran vs. C Storage Scheme. Multi-dimensional arrays are stored as a one-dimensional sequence of elements in memory. A two-dimensional C array is stored row by row, while Fortran stores it column by column. In Fortran the first index runs faster than the second index, and so on, whereas in C the first index runs slower than the second, and so forth, with the last index as the fastest one. Figure 4.1 illustrates the differences in storage.

8., 9.],

14., 16.],

18., 21.]])

123456

142536

􏰃􏰄

123 456

Fig. 4.1. Storage of a 2×3 matrix in C/C++/NumPy (upper) and Fortran (lower).

When we send NumPy arrays to C or Fortran code we must be aware of the way the array is stored in memory. By default, NumPy arrays employ the same storage scheme as in C, but we can easily change the ordering of elements used in Fortran. Given any array a, with either C or Fortran

ordering, we can transform the storage to either C or Fortran using NumPy’s asarray function:

>>> af = asarray(a, order=’Fortran’) >>> ac = asarray(a, order=’C’)

If asarray finds that no change in the ordering is necessary, the original array is returned, otherwise a new array is returned with reordered elements. For a two-dimensional array, the reordering corresponds to transposing the array. To check if an array has C or Fortran ordering, we call

>>> isfortran(af)

True

>>> isfortran(ac)

False

When creating arrays using the array, zeros, or ones functions we can also provide an order=’Fortran’ argument to get Fortran ordering. There is more information about arrays and communication with Fortran in Chapter 9.

4.1.2 Array Indexing

Indexing of one-dimensional Numerical Python arrays follows the syntax of Python lists:

a = linspace(-1, a[2:4] = -1 a[-1] = a[0] a[:] =0 a.fill(0)

1, 6)

# set

# set

# set

# set

a[2] and a[3] equal to -1

last element equal to first one all elements of a equal to 0 all elements of a equal to 0

An extended subscripting syntax is offered for multi-dimensional arrays:

a.shape = (2,3) print a[0,1] a[i,j] = 10 a[i][j] = 10 print a[:,k] print a[1,:] a[:,:] = 0

# turn a into a 2x3 matrix

# print element (0,1)

# assignment to element (i,j)

# equivalent syntax (slower)

# print column with index k

# print second row

# set all elements of a equal to 0

A general index has the form start:stop:step, indicating all elements from start up to stop-step in steps of step. Such an index can in general be represented by a slice object (see page 391). We can illustrate slicing further in an interactive session:

>>> a = linspace(0, >>> a.shape = (5,6) >>> a

array([[ 0., 1.,

[ 6., 7.,

29, 30)

2., 3., 4., 5.,]

8., 9., 10., 11.,]

[ 12., 13.,

[ 18., 19.,

[ 24., 25.,

4.1. A Quick NumPy Primer 137

14., 15., 16., 17.,]

20., 21., 22., 23.,]

26., 27., 28., 29.,]])

# a[i,j] for i=1,2 and j=0,2,4

>>> a[1:3,:-1:2]

array([[ 6., 8., 10.],

[ 12., 14., 16.]])

>>> a[::3,2:-1:2] # a[i,j] for i=0,3 and j=2,4 array([[ 2., 4.],

[ 20., 22.]])

>>> i = slice(None, None, 3); j = slice(2, -1, 2) >>> a[i,j]

array([[ 2., 4.],

[ 20., 22.]])

It is important to know that slicing gives a reference to the underlying array, which is different behavior than that of plain Python lists where slices take a copy of the list data, see page 90 and Chapter 3.2.10. For example,

>>> b = a[1,:]

results in a reference to the 2nd row in a. Changing b will also change a (and

vice versa):

>>> print a[1,1] 12.0

>>> b[1] = 2

>>> print a[1,1] 2.0

# change in b is reflected in a

If a true copy of the second row is wanted, we can call the copy method:

>>> b = a[1,:].copy()

>>> print a[1,1]

12.0

>>> b[1] = 2 # b and a are two different arrays now >>> print a[1,1]

12.0 # a is not affected by change in b

Any integer list or array can be in fact be used as index. For example, the slice a[f:t:i] is equivalent to a[range(f:t:i)]. An array b with boolean values can also be used as index. The index set then corresponds to the indices in b for which b’s value is True. This allows for boolean expressions as indices, like a[a<0]. The session below should illustrate som possibilities:

>>> a = linspace(1, 8, 8)

>>> a

array([ 1., 2., 3., 4., 5., 6., 7., 8.])

>>> a[[1,6,7]] = 10

>>> a

array([ 1., 10., 3., 4., 5., 6., 10., 10.])

>>> a[range(2,8,3)] = -2

>>> a

array([ 1., 10., -2., 4., 5., -2., 10., 10.])

>>> a[a < 0] # pick out the negative elements of a

array([-2., -2.])

>>> a[a < 0] = a.max()

>>> a

array([ 1., 10., 10., 4., 5., 10., 10., 10.])

Generalized indexing using integer arrays or lists is important for efficient initialization of array elements.

4.1.3 Loops over Arrays

Iterating over an array can be done with a standard for loop over indices:

for i in xrange(a.shape[0]):

for j in xrange(a.shape[1]):

a[i,j] = (i+1)\*(j+1)\*(j+2)

print ’a[%d,%d]=%g ’ % (i,j,a[i,j]), print # newline after each row

For large arrays, one should use the less memory-consuming and also more efficient1 xrange function instead of range.

There are several ways of iterating over an array a. The standard for e in a construct iterates over the first index:

>>> print a

[[ 2. 6. 12.]

[ 4. 12. 24.]] >>> for e in a: ... print e ...

[ 2. 6. 12.]

[ 4. 12. 24.]

Iterating over all elements can be done by viewing the array as one-dimensional by a.ravel():

>>> for e in a.ravel(): ... print e

...

2.0

6.0

12.0

4.0

12.0

24.0

A more useful iterator iterates over all elements, but extracts both the index tuple and the corresponding array value:

1 src/py/examples/efficiency/pyefficiency.py contains a test showing that xrange is al- most three times as fast range for administering a long empty loop on my laptop.

>>> for index, value in ndenumerate(a): ... print index, value

...

(0, 0) 2.0

(0, 1) 6.0

(0, 2) 12.0

(1, 0) 4.0

(1, 1) 12.0

(1, 2) 24.0

Tests show that this last iteration can be six times more time consuming than the traditional three loops over integer indices using xrange.

4.1.4 Array Computations

Loops over array elements should be avoided as this is computationally in- efficient. Instead, NumPy offers lots of efficient C functions that operate on the whole array at once. Consider, as an example,

b = 3\*a - 1

All elements in a are multiplied by 3 and the result is stored in a temporary array. Then 1 is subtracted from each element in this temporary array, and the result is stored in a new temporary array to which b becomes a reference. All these array operations are performed by looping over the array elements in efficient C code.

We may easily investigate the speed-up of array arithmetics compared to a plain loop:

>>> import time # module for measuring CPU time

>>> a = linspace(0, 1, 1E+07) # create some array

>>> t0 = time.clock()

>>> b = 3\*a -1

>>> t1 = time.clock() # t1-t0 is the CPU time of 3\*a-1 >>> for i in xrange(a.size): b[i] = 3\*a[i] - 1

>>> t2 = time.clock()

>>> print ’3\*a-1: %g sec, loop: %g sec’ % (t1-t0, t2-t1) 3\*a-1: 2.09 sec, loop: 31.27 sec

That is, the array expression 3\*a-1 runs about 15 times faster than the loop- based counterpart.

More memory conserving computation of b=3\*a-1 can be done by in-place modifications in b:

b=a

b \*= 3 # or multiply(b, 3, b) b -= 1 # or subtract(b, 1, b)

These operations require no extra memory as each element in b is modified in-place. The code also runs almost twice as fast (on my laptop). Note that a

is affected by these operations, since b initially shares its data with a, while if we write b=3\*a-1 the a variable remains unaltered. Starting with b=a.copy() instead of b=a prevents changes in a.

The following operators offer in-place arithmetics in arrays:

a \*= 3.0

a -= 1.0

a /= 3.0

a += 1.0

a \*\*= 2.0

# multiply a’s elements by 3

# subtract 1 from each element # divide each element by 3

# add 1 to each element

# square all elements

Another frequently used in-place operation is assignment directly to the ele- ments in an existing array:

a[:] = 3\*c - 1

Note the difference between assignment to a[:] and a. In the former case the elements of the right-hand side array are copied into the elements of the array referred to by a, while in the latter case a refers to a new array object.

NumPy offers trigonometric functions, their inverse counterparts, and hy- perbolic versions as well as the exponential and logarithmic functions. Here are a few examples:

c = sin(b)

c = arcsin(c)

c = sinh(b)

# same functions for the cos and tan families c = b\*\*2.5 # power function

c = log(b)

c = exp(b)

c = sqrt(b)

Many more mathematical functions, such as Bessel functions, are offered by the SciPy package (Chapter 4.4.2).

There are functions for finding maximum and minimum values and cor- responding indices. Let us make a 5 × 4 array of random numbers between 0 and 20:

>>> a = arange(0, 20)

>>> random.seed(10) # fix seed

>>> random.shuffle(a) # in-place modification of a >>> a.shape = 5,4

>>> print a

[[710 5 6]

[31813 2] [14 81716] [191211 1]

Calling a.argmax() returns the index corresponding to the maximum value of a. The index refers to a one-dimensional view of the array. The func- tion a.ravel() makes multi-dimensional arrays one-dimensional (as they are stored in memory). To find the maximum value is then a matter of doing

>>> max\_index = a.argmax()

>>> a1d = a.ravel()

>>> print a1d

[710 5 6 31813 214 81716191211 1 015 4 9] >>> max\_value = a1d[max\_index]

>>> print ’max value = %g for index %d’ % (max\_index, max\_value) max value 19 for index 12

>>> print a1d.max()

19

While a.argmax() returns an index, a.max() returns the largest value in a. Cor- responding a.argmin() and a.min() methods also exist, as expected. Sorting the array can be done as follows:

>>> a1d.sort()

>>> print a1d

[0 1 2 3 4 5 6 7 8 910111213141516171819]

Summing up array elements is a often useful:

>>> print sum(a), sum(a1d)

Large and small values can be clipped away:

>>> a1d = a1d.clip(min=3, max=12)

>>> print a1d

[3 3 3 3 4 5 6 7 8 910111212121212121212]

Simple statistics is available: a.mean() (or mean(a)) for the mean, a.var() (or var(a)) for the variance a.std() (or std(a)) for the standard deviation, median(a) for the median, and cov(x,y) for the covariance of x and y arrays. There are also useful functions piecewise for piecewisely defined functions, trapz for Trapezoidal integration of array values, diff for discrete finite dif- ferences, a polynomial type, etc.

Matlab Compatibility. Most of the basic functions for arrays found in Matlab are mirrored in NumPy. Examples include corrcoef, cov, cumprod, diag, diff, eig, eye, fliplr, flipud, max, min, mean, median, prod, ptp, rot90, squeeze, std, sum, svd, trapz, tri, tril, triu, and var. With the scitools.easyviz package you also get access to plotting functions with names similar to those in Matlab: plot, xlabel, ylabel, legend, title, surf, mesh – to mention some.

Hidden Temporary Arrays. An important feature of NumPy is that most mathematical functions written in plain Python for scalar variables will au- tomatically be applicable to NumPy arrays as well. As an example, consider the mathematical function f(x) = exp􏰅−x2􏰆ln(1 + xsinx) implemented as a plain Python function

def f1(x):

return exp(-x\*x)\*log(1+x\*sin(x))

Sending in a scalar value, say 3.1, f1 evaluates the expression e−3.12 ln(1 + 3.1sin3.1). Sending in a NumPy array as x, returns an array where each element equals f1 applied to the corresponding entry in the input array x. However, “behind the curtain” several temporary arrays are created in order to apply f1 to a vector:

1. temp1 = -x

2. temp2 = temp1\*x

3. temp3 = exp(temp2) 4. temp4 = sin(x)

5. temp5 = x\*temp4

6. temp6 = 1 + temp4

7. temp7 = log(temp5) 8. result = temp3\*temp7

Python quickly removes such temporary arrays.

4.1.5 More Array Functionality

Below we exemplify many useful array methods and attributes.

>>> a = zeros(4) + 3

>>> a

array([ 3., 3., 3., 3.]) # float data

* >>> a.item(2) # more efficient than a[2] 3.0
* >>> a.itemset(3,-4.5) # more efficient than a[3]=-4.5 >>> a  
   array([ 3. , 3. , 3. , -4.5])  
   >>> a.shape = (2,2)  
   >>> a  
   array([[ 3. , 3. ],  
     
  [ 3. , -4.5]])  
   >>> a.ravel() # from multi-dim to one-dim array([ 3. , 3. , 3. , -4.5])

>>> a[0,1]=-88

>>> a

array([[ 3. , -88. ],

[ 3. , -4.5]])

>>> a.transpose()

array([[ 3. , 3. ], [-88. , -4.5]])

>>> a.ndim

2

>>> len(a.shape)

2

>>> rank(a)

2

>>> a.size

# introduce non-symmetry

# no of dimensions

# no of dimensions

# no of dimensions

# total no of elements

4

>>> a.nbytes

32

>>> b = a.astype(int) >>> b

array([3, 3, 3, 3])

4.1. A Quick NumPy Primer 143

# a.size\*a.itemsize

# change data type

Numerical Python supports many data types for the array elements. Besides the standard Python types float, int, complex, and bool, we have float96, float64, float32, int32, int16, complex64, and complex128 to mention some of the most important ones. The trailing number in the names of these data types reflects the number of bits occupied by an array element.

The module numpy.lib.scimath offers enhanced versions of some mathe- matical functions such that both complex and real results can be returned, depending on the input argument. For example, the sqrt function should re- turn a real for a postive argument and a complex for a negative argument. The basic sqrt function from numpy or math do not handle complex results, cmath always returns complex results, while numpy.lib.scimath functions re- turns real if possible, otherwise complex:

>>> from math import sqrt

>>> sqrt(-1)

Traceback (most recent call last):

File "<stdin>", line 1, in <module> ValueError: math domain error

>>> from numpy import sqrt

>>> sqrt(-1)

Warning: invalid value encountered in sqrt nan

>>> from cmath import sqrt

>>> sqrt(-1)

1j

>>> sqrt(4) # cmath functions always return complex... (2+0j)

>>> from numpy.lib.scimath import sqrt

>>> sqrt(4)

2.0

>>> sqrt(-1)

1j

We remark, however, that functions from numpy.lib.scimath may be quite slow compared to those in numpy, as shown below.

Remark on Efficiency. The mathematical functions in NumPy work with both scalar and array arguments. However, they are quite slow for scalar arguments compared with the corresponding functions in the math module. To illustrate this point, we have made a program in

src/py/examples/efficiency/asin\_efficiency.py

which computes sin−1 x using asin from math and arcsin from the various Numerical Python modules numpy, numpy.lib.scimath, numarray, and Numeric.

Calling just sin−1 x and scaling the result of asin from math to one unit of CPU time, arcsin from numpy required 12 units of CPU time, while arcsin from Numeric, numarray, and numpy.lib.scimath led to 14, 18, and 92 units of CPU time, respectively.

Burying the sin−1 x operation inside a function, def f(x, y):

return x\*\*2 + arccos(x)\*arcsin(x)

will naturally not lead to such dramatic differences between the various im- plementations of the inverse sine function since there are more arithmetic operations and function calls involved. Now the numpy-based version of f used 6 units of CPU time, while the enhanced functions from numpy.lib.scimath required almost 40 units of CPU time.

We learn two things from these timings: mathematical NumPy functions are slow for scalar arguments (use math!), and the flexible functions from numpy.lib.scimath are much less efficient than the similar (less flexible) func- tions in numpy.

The efficiency considerations mentioned above are significant only when the mathematical functions are called a (very) large number of times. A profiling (see Chapter 8.10.2) will normally uncover this type of efficiency problems. I therefore recommend to emphasize programming convenience and safety, and when execution speed becomes critical, you may use the comments in this section and the list in Chapter 8.10.3.

4.1.6 Type Testing

The NumPy array class has the name ndarray (“n-dimensional array”):

>>> type(a)

<type ’numpy.ndarray’>

>>> isinstance(a, ndarray) True

The type of the array elements is described by the object a.dtype (“data type”), which contains a name of the data type, a character code (correspond- ing to the codes used in the struct module for binary I/O, see Chapter 8.3.6), and the number of bytes occupied by each array element:

>>> a.dtype.name

’float64’

>>> a.dtype.char

’d’



>>> a.dtype.itemsize

8

* >>> b = zeros(6, float32)
* >>> a.dtype == b.dtype # do a and b have the same data type? False  
   >>> c = zeros(2, float)  
   >>> a.dtype == c.dtype  
   True

Controlling the data type is particularly important when communicating with array processing functions written in Fortran, C, or C++ (Chapters 9 and 10).

Note that if you have an array of integers and assign floating-point num- bers, everyting will be automatically converted to the array’s data type (here integers):

>>> a = zeros(4, int)

>>> a[2] = 2.92

>>> print a

[0 0 2 0] # 2.92 was truncated to 2

4.1.7 Matrix Objects

The arrays created so far have been of type ndarray. NumPy also has a matrix type called matrix or mat, which is similar to the basic matrix data structure in Matlab. That is, one-dimensional arrays are either row or column vectors when converted to the matrix type:

>>> x1 = array([1, 2, 3], float) >>> x2 = matrix(x1)

>>> x2

matrix([[ 1., 2., 3.]])

>>> x3 = mat(x).transpose() >>> x3

matrix([[ 1.],

[ 2.], [ 3.]])

# or mat(x1)

# row vector

# column vector

>>> type(x3)

<class ’numpy.core.defmatrix.matrix’> >>> isinstance(x3, matrix)

True

Arrays of higher dimension than two cannot be represented as matrix in- stances.

A special feature of matrix objects is that the multiplication operator represents the matrix-matrix, vector-matrix, or matrix-vector product as we know from linear algebra:

>>> A = eye(3)

>>> A

array([[ 1., 0., 0.],

[ 0., 1., 0.],

[ 0., 0., 1.]])

>>> A = mat(A)

>>> A

matrix([[ 1., 0., 0.],

[ 0., 1., 0.],

[ 0., 0., 1.]])

>>> y2 = x2\*A

# identity matrix

# vector-matrix product

>>> y2

matrix([[ 1., 2., 3.]]) >>> y3 = A\*x3

>>> y3

matrix([[ 1.],

[ 2.], [ 3.]])

>>> A\*x1

Traceback (most recent call last): ...

ValueError: matrices are not aligned

>>> # try array\*array product:

>>> A = (zeros(9) + 1).reshape(3,3) >>> A

array([[ 1., 1., 1.],

[ 1., 1., 1.],

[ 1., 1., 1.]])

>>> A\*x1

array([[ 1., 2., 3.], [ 1., 2., 3.],

[ 1., 2., 3.]])

>>> B = A + 1

>>> A\*B

array([[ 2., 2., 2.],

[ 2., 2., 2.],

[ 2., 2., 2.]]) >>> A = mat(A); B = mat(B)

>>> A\*B

matrix([[ 6., 6., 6.],

[ 6., 6., 6.],

[ 6., 6., 6.]])

# [A[0,:]\*x1, A[1,:]\*x1, A[2,:]\*x1]

# element-wise product

# matrix-matrix product

# matrix-vector product

# no matrix-array product!

4.1.8 Exercises

Exercise 4.1. Matrix-vector multiply with NumPy arrays.

Define a matrix and a vector, e.g.,

A = array([[1, 2, 3], [4, 5, 6], [7, 8, 10]]) b = array([-3, -2, -1])

Use the NumPy manual to find a function that computes the standard matrix- vector product A times b (i.e., the vector whose i-th component is 􏰇2j=0 A[i,j]\*b[j]). ⋄

Exercise 4.2. Work with slicing and matrix multiplication.

Extract the 2 × 2 matrix in the lower right corner of the matrix A in Exercise 4.1 as a slice. Add this slice to another 2 × 2 matrix, multiply the result by a 2 × 2 matrix, and insert this final result in the upper left corner of the original matrix A. Control the result by hand calculations. ⋄

Exercise 4.3. Assignment and in-place NumPy array modifications.

Consider the following script:

from numpy import linspace x = linspace(0, 1, 3)

# y = 2\*x + 1:

y=x; y\*=2; y+=1

# z = 4\*x - 4:

z=x; z\*=4; z-=4 print x, y, z

Explain why x, y, and z have the same values. How can the script be changed such that y and z get the intended values? ⋄

4.2 Vectorized Algorithms

Below we explain how Python functions with if tests can be vectorized with the aid of the where function. We also describe how difference equations can be vectorized using slices.

4.2.1 From Scalar to Array in Function Arguments

Mathematical Python functions with if tests will not handle NumPy arrays correctly. Consider the sample function

def somefunc(x):

if x < 0:

return 0

else:

return sin(x)

The operation x < 0 results in a boolean array where an element is True if the corresponding element in x is less than zero, and False otherwise. However, this array cannot be evaluated as a boolean value in an if test so a ValueError exception is raised.

How can we extend the somefunc function shown above such that it works with x as a NumPy array? The simplest solution is to use the vectorize class in the numpy package. This class automatically vectorizes any function of scalar arguments such that the function works with array arguments. For example, executing

somefuncv = vectorize(somefunc)

gives a version somefuncv of somefunc where x can also be an array. The array returned from somefuncv has elements of a type that is automatically determined by vectorize. This type may be wrong, which is the case in the present example, and then the output type must be specified explicitly:

somefuncv = vectorize(somefunc, otypes=’d’)

Note that the data type must be specified by a character (and not float or int), here we use ’d’ for float (double precision) elements. The somefuncv object has no function name so we may set one:

somefuncv.\_\_name\_\_ = "vectorize(somefunc)"

Unfortunately, the speed of somefuncv is much lower than the best hand- written versions below (see the end of src/py/intro/NumPy\_basics.py for a timing test that you can run on your own computer).

A possible first try to manually get the scalar code in the somefunc function to work with array arguments is to insert a loop over the array entries:

def somefunc\_NumPy(x):

r = x.copy() # allocate result array for i in xrange(size(x)):

if x[i] < 0:

r[i] = 0.0

else:

r[i] = sin(x[i])

return r

Such loops run very slowly in Python. Moreover, the implementation works only for a one-dimensional array.

To make the code faster, we need to express our mathematical algorithm in terms of vector operations and not elementwise operations based on indexing. Loops will then be executed in fast C code in the Numerical Python library. Such a rewrite is often referred to as vectorization. This technique is in many interactive scientific computing environments, such as Octave and S-PLUS/R (and formerly also in Matlab). Even in C, C++, and Fortran vectorization can speed up the code, because simpler loops may be easier to optimize by the compiler than more complicated loops. (This is particularly the case in the present example because an if-test inside the loop prevents aggressive compiler optimization.)

It is difficult to give general guidelines on how to vectorize a function that does not work with array arguments, because the rewrite depends strongly on the available functionality in the underlying library, here the NumPy package. However, with NumPy, a function like

def f(x):

if condition:

x = <expression1>

else:

x = <expression2>

return x

can be coded like this:

def f\_vectorized(x):

x1 = <expression1>

x2 = <expression2>

return where(condition, x1, x2)

The where function returns an array of the same shape as that of condition, and element no. i equals x1[i] if condition[i] is true, and x2[i] otherwise. In our present example, we can write

def somefunc\_NumPy2(x):

x1 = zeros(x.size, float) x2 = sin(x)

return where(x < 0, x1, x2)

or even simpler

def somefunc\_NumPy2b(x):

return where(x < 0, 0.0, sin(x))

On my laptop, this hand-written function ran over 50 times faster than the function automatically generated by vectorize.

Sometimes the computations cannot be performed for all the values of the incoming array. Consider, as an example,

def logpos(x):

if x <= 0:

return 0.0

else:

return log(x)

Now a simple log(x) when x is an array will not work if x has negative elements. One remedy is to replace all illegal entries in x with legal ones, and then perform log(x). The replaced entries will never enter the final answer anyway:

def logposv(x):

x\_pos = where(x > 0, x, 1) # subst. negative values by 1 r1 = log(x\_pos)

r = where(x < 0, 0.0, r1)

return r

4.2.2 Slicing

Slicing can be an important technique for vectorizing expressions, especially in applications involving finite difference schemes, image processing, or smooth- ing operations. Consider the following numerical recursion scheme:

ul+1 =βul +(1−2β)ul +βul , i=1,...,n−1, i i−1 i i+1

arising from solving a one-dimensional diffusion equation ∂u = ∂2u by an ∂t ∂x2

explicit finite difference scheme. The index l ≥ 0 counts discrete levels in

time, and i is a counter for points in space (i = 0,...,n). The quantity uli is the unknown function u evaluated at grid point i and time level l. In plain Python we would typically code the scheme as

n = size(u)-1

for i in xrange(1,n,1):

u\_new[i] = beta\*u[i-1] + (1-2\*beta)\*u[i] + beta\*u[i+1]

where u\_new holds ul+1 for i = 1,...,n, and u holds ul for the same i values.

ii

The problem is that loops in Python are slow. A vectorized version consists of adding three vectors: u[1:n-1], u[0:n-2], and u[2:n], with suitable scalar coefficients. That is, the loop is replaced by

u[1:n] = beta\*u[0:n-1] + (1-2\*beta)\*u[1:n] + beta\*u[2:n+1]

We now compute slices of the arrays and add these to form the new u. Note that there is no need for a separate array u\_new since u becomes a new ar- ray every time the statement is executed. This leads, of course, to tempo- rary arrays in memory (the additions on the right-hand side of the previous statement also introduce temporary arrays at each time level). It seems that Python is able to deallocate or reuse temporary arrays, because the mem- ory overhead does not increase steadily when the recursion scheme is run for many time levels.

4.2.3 Exercises

Exercise 4.4. Vectorize a constant function.

The function

def initial\_condition(x): return 3.0

does not work properly when x is a NumPy array. In that case the function should return a NumPy array with the same shape as x and with all entries equal to 3.0. Perform the necessary modifications such that the function works for both scalar types and NumPy arrays. ⋄

Exercise 4.5. Vectorize a numerical integration rule.

The integral of a function f(x) from x = a to x = b can be calculated numerically by the Trapezoidal rule:

􏰈

a

n−1 bhh􏰉b−a

f(x)dx≈ 2f(a)+ 2f(b)+h f(a+ih), h= n . i=1

(4.1)

Implement this approximation in a Python function containing a straightfor- ward loop.

The code will run slowly compared to a vectorized version. Make the vectorized version and introduce timings to measure the gain of vectorization. Use the function

f1(x) = 1 + 2x

as test functions for the integration. ⋄

Exercise 4.6. Vectorize a formula containing an if condition.

Consider the following function f(x):

f(x) = n 􏰀0.51+1/n − (0.5 − x)1+1/n, 0 ≤ x ≤ 0.5 (4.2)

1+n 0.51+1/n −(x−0.5)1+1/n,0.5<x≤1

Here, n is a real number, typically 0 < n ≤ 1. (The formula describes the velocity of a pressure-driven power-law fluid in a channel.) Make a vectorized Python function for evaluating f(x) at a set of m equally spaced x values between 0 and 1 (i.e., no loop over the x values should appear). ⋄

Exercise 4.7. Slicing of two-dimensional arrays.

Consider the following recursive relation (arising when generalizing the one-dimensional diffusion equation scheme in Chapter 4.2.2 to two dimen- sions):

ul+1 =β(ul +ul +ul +ul )+(1−4β)ul . i,j i−1,j i+1,j i,j−1 i,j+1 i,j

Write a straight Python loop implementing this recursion. Then replace the loop by a vectorized expression based on slices. ⋄

4.3 More Advanced Array Computing

Numerical Python contains a module random for efficient random number generation, outlined in Chapter 4.3.1. Another Numerical Python module linalg which solves linear systems, computes eigenvalues and eigenvectors, etc., and is presented in Chapter 4.3.2. Tools for curveplotting are described in Chapter 4.3.3. Chapter 4.3.4 deals with a curve fitting example, which ties together linear algebra computations and curve plotting. Chapter 4.3.5 addresses vectorized array computations on structured grids.

Numerical Python comes with its own tools for storing arrays in files and loading them into scripts again. These tools are covered in Chapter 4.3.6. Chapter 4.3.6 also presents a module from the scitools package associated with this book where two-dimensional NumPy arrays can be read from and written to a tabular file format.

4.3.1 Random Numbers

The basic module for generating uniform random numbers in Python is random, which is a part of the standard Python distribution. This module provides the function seed for setting the initial seed. Generating uniformly distributed random numbers in (0, 1) or (a, b) is performed by the random and uniform functions, respectively. Random variates from other distributions are also supported (see the documentation of the random module in the Python Library Reference for details). The next lines illustrate the basic usage of the random module:

import random

random.seed(2198) # control the seed

print ’uniform random number on (0,1):’, random.random() print ’uniform random number on (-1,1):’, random.uniform(-1,1) print ’Normal(0,1) random number:’, random.gauss(0,1)

No call to the seed function implies calculating a seed based on the current time. Giving a manual seed has the advantage that we can work with the same sequence of random numbers each time the program is run. This is important for debugging and code verification.

Calling up the random module in a loop for generating large random sam- ples is a slow process. Much more efficient random number generation is provided by the random module in the NumPy package. This module gets imported by the standard from numpy import \*, but since its name then is identical with Python’s standard random module it is easy to mix the two. The most basic usage of numpy’s random module is illustrated next. The main point is that we can efficiently draw an array of random numbers at once:

from numpy import \* # import random and other stuff

random.seed(12) # set seed

u = random.random(n) # n uniform numbers on (0,1) u = random.uniform(-1, 1, n) # n uniform numbers on (-1,1)

The random module offers more general distributions, e.g., the normal distri- butions:

mean = 0.0; stdev = 1.0

u = random.normal(mean, stdev, n)

m = sum(u)/n # empirical mean

s = sqrt(sum((u - m)\*\*2)/(n-1)) # empirical st.dev. print ’generated %d N(0,1) samples with\nmean %g ’\

’and st.dev. %g using numpy.random.normal’ % (n, m, s)

Logical operators on vectors are often useful when working with large vectors of samples. As an illustrating example, we can find the probability that the samples in u, generated in the previous code snippet, are less than 1.5:

p = sum(where(u < 1.5, 1, 0)) prob = p/float(n)

print ’probability=%.2f’ % prob

The first line deserves a comment. The where(b, c1, c2) call returns an array, say a, where a[i] is c1 if b[i] is True, and c2 if if b[i] is False. The b array is a boolean array arising from a boolean expression involving a NumPy array, such as u < 1.5 in this case. The array resulting from u < 1.5 has element no. i equal to True if u[i] < 1.5, otherwise this element is False. When sum is applied to the array returned from where, having 0 or 1 values, the number of random values less than 1.5 are computed.

Random samples drawn from the uniform, normal, multivariate normal, exponential, beta, chi square, F, binomial, and multinomial distributions are offered by numpy’s random module. We refer to the module’s doc string or the NumPy manual for more details.

4.3.2 Linear Algebra

The linalg module, automatically imported in a from numpy import \* state- ment, contains functions for solving linear systems, finding the inverse and the determinant of a matrix, computing eigenvalues and eigenvectors, solving least-squares problems, finding the singular value decomposition of a matrix, and computing the Cholesky decomposition of a matrix. An illustration of solving a linear system Ax = b is given below.

from numpy import \* A = zeros((n,n))

x = zeros(n)

b = zeros(n)

for i in range(n):

x[i] = i/2.0

# some prescribed solution A[i,j] = 2.0 + float(i+1)/float(j+i+1)

for j in range(n):

b = dot(A, x) # matrix-vector product: adjust rhs to fit x

# solve linear system A\*y=b: y = linalg.solve(A, b)

We can now check if the solution of the linear system, as produced by linalg.solve, coincides with the array x. Testing if x == y does not work, becuase x == y results in an array of length n where element no. i is True if x[i] == y[i]. The problem is that the boolean array arising from x == y can- not be evaluated as a scalar boolean value in an if test. We can use the array method all() to check if all elements are True in this array. Therefore, if (x == y).all() makes sense, but this test involves exact inequalities, which is not a good idea when comparing floating-point numbers. A better test is

if sum(abs(x - y)) < 1.0E-12: print ’correct solution’ else: print ’wrong solution’,x,y

An alternative test is to use the allclose function from numpy, or equivalently float\_eq from scitools.numpyutils (see page 167). This function checks if abs(x-y) is less than an absolute tolerance plus y times a relative tolerance. A typical call is

if allclose(x, y, atol=1.0E-12, rtol=1.0E-12): print ’correct solution’

else:

print ’wrong solution’, x, y

The linalg module has more functionality, for instance functions for matrix determinants and inverses:

d = linalg.det(A)

B = linalg.inv(A)

# check result:

R = dot(A, B) - eye(n) # residual

R\_norm = linalg.norm(R) # Frobenius norm of matrix R print ’Residual R = A\*A-inverse - I:’, R\_norm

Eigenvalues can also be computed:

# eigenvalues only:

A\_eigenvalues = linalg.eigvals(A)

# eigenvalues and eigenvectors: A\_eigenvalues, A\_eigenvectors = linalg.eig(A)

for e, v in zip(A\_eigenvalues, A\_eigenvectors):

print ’eigenvalue %g has corresponding vector\n%s’ % (e, v)

4.3.3 Plotting

There are several Python packages available for plotting curves and visualiz- ing 2D/3D scalar and vector fields. For curve plotting, the Gnuplot package by Michael Haggerty (see doc.html for a link to the software) allows easy access to the popular Gnuplot program from Python scripts. Chapter 5.3.3 has a worked example. A strength of the Gnuplot program is that it is very easy to install on all major platforms. The Gnuplot Python interface comes with a demo.py script which shows the basic usage.

The most promising and comprehensive plotting tool at the time of this writing is Matplotlib. The widely used IDL environment, which has extensive support for plotting, can be interfaced from Python through the pyIDL mod- ule. Another plotting program, Grace, can be interfaced using the pygrace module. With the pymat module (see Chapter 4.4.3) one can easily send NumPy arrays to Matlab and plot them there.

It may be difficult to pick the optimal plotting package for use with a Python script. That is one reason why we have created a unified Python in- terface to several different plotting packages. This interface is called Easyviz. Both curve plots and more advanced 2D/3D visualization of scalar and vector fields are supported by Easyviz. The interface was designed with three ideas in mind: (i) a simple, Matlab-like syntax; (ii) a unified interface to lots of visualization engines (called backends later): Gnuplot, VTK, Matlab, Mat- plotlib, PyX, etc.; and (iii) a minimalistic interface which offers only basic control of plots (fine-tuning is left to programming in the specific backend directly).

The import statements to get access to the interface are either

from numpy import \*

from scitools.easyviz import \*

or

The latter statement performs the former two, plus some more imports of convenient features in scitools. Plotting a curve is very simple:

t = linspace(0, 3, 51) # 51 points between 0 and 3 y = t\*\*2\*exp(-t\*\*2)

plot(t, y)

We can add another curve and some noisy data points, pluss specify legends for the three curves, fix the axis, add a title, and mark the x axis with a t label:

y2 = t\*\*4\*exp(-t\*\*2)

# pick out each 4 points and add random noise:

t3 = t[::4]

random.seed(11)

y3 = y2[::4] + random.normal(loc=0, scale=0.02, size=t3.size)

plot(t, y1, ’r-’)

hold(’on’)

plot(t, y2, ’b-’)

plot(t3, y3, ’bo’)

legend(’t 2\*exp(-t 2)’, ’t 4\*exp(-t 2)’, ’data’) title(’Simple Plot Demo’)

axis([0, 3, -0.05, 0.6]) xlabel(’t’)

ylabel(’y’)

show() hardcopy(’tmp0.eps’) hardcopy(’tmp0.png’)

Matlab users will be familiar with this syntax. However, we also provide a more compact plot command where the individual function calls above are included through keyword arguments:

plot(t, y1, ’r-’, t, y2, ’b-’, t3, y3, ’bo’,

legend=(’t 2\*exp(-t 2)’, ’t 4\*exp(-t 2)’, ’data’), title=’Simple Plot Demo’,

axis=(0, 3, -0.05, 0.6),

xlabel=’t’, ylabel=’y’,

hardcopy=’tmp1.ps’,

show=True)

hardcopy(’tmp0.png’)

A scalar function f(x,y) may be visualized as an elevated surface with colors using these commands:

x = linspace(-2, 2, 41) # 41 point on [-2, 2]

xv, yv = ndgrid(x, x) values = f(xv, yv) surfc(xv, yv, values,

shading=’interp’,

clevels=15,

clabels=’on’,

hidden=’on’,

show=True)

# define a 2D grid with points (xv,yv) # function values

With Easyviz you can quickly write plotting commands in your Python scripts and postpone the decision to employ a specific plotting package. For example, you may start out with Gnuplot and later switch to Matplotlib, if desired. The backend can either be set in a config file or by a command-line option to the Python script,

--SCITOOLS\_easyviz\_backend name

where name is the name of the backend: gnuplot, vtk, matplotlib, blt, etc. The specified backend must of course be installed on your computer system. Easyviz is a light-weight interface and aimed at the functionality you need “95%” of the time. This means that only the most basic plotting operations are found in the interface. If you need more sophisticated operations, you can grab the object that Easyviz applies for communication with the backend and use this object to write plotting package-specific commands. As an example, say you apply the gnuplot backend and want to write a text and display an arrow in your plot. The following commands grab the backend object (a Gnuplot instance), here called g, and then sends Gnuplot-specific commands

for writing the text and drawing the arrow:

g = get\_backend()

if backend == ’gnuplot’:

# g is a Gnuplot object, work with Gnuplot commands directly: g(’set label "global maximum" at 0.1,0.5 font "Times,18"’) g(’set arrow from 0.5,0.48 to 0.98,0.37 linewidth 2’) g.refresh()

g.hardcopy(’tmp.eps’) # make new hardcopy

Easyviz also support making movies through the movie function, which takes a Unix shell-style wildcard specification of a set of hardcopies that are supposed to be the frames in the movie. Here is an example of animating a Gaussian bell where the standard deviation is decreased from 2 to 0.2:

from scitools.all import \*

# Gaussian bell with mean m and standard deviation s: def f(x, m, s):

return (1.0/(sqrt(2\*pi)\*s))\*exp(-0.5\*((x-m)/s)\*\*2)

m=0

s\_start = 2

s\_stop = 0.2

s\_values = linspace(s\_start, s\_stop, 30)

x = linspace(m - 3\*s\_start, m + 3\*s\_start, 1000) max\_f = f(m, m, s\_stop)

# show the movie on the screen

# and make hardcopies of frames simultaneously: counter = 0

for s in s\_values:

y = f(x, 0, s)

plot(x, y, axis=[x[0], x[-1], -0.1, max\_f],

xlabel=’x’, ylabel=’f’, legend=’s=%4.2f’ % s,

hardcopy=’tmp\_%04d.eps’ % counter) counter += 1

movie(’tmp\_\*.eps’) # make movie file the simplest possible way

We refer to the doc string in the Easyviz package for more complete infor- mation on what the package can do:

pydoc scitools.easyviz

Remark. When data are sent from Python to plotting programs, it may happen that the programs need some time to display the data, and if the calling script ends, the plotting program exits and no plot appears on the screen. The remedy is to insert a time.sleep(s) command at the end of the Python script (s is the number of seconds the script should halt at the end to ensure that the plotting program gets enough time to finish the plot).

4.3.4 Example: Curve Fitting

The next example demonstrates how different numerical utilities in Python can be put together to form a flexible and productive working environment in the spirit of environments like Matlab. We shall illustrate how to fit a straight line through a set of data points using the least squares method. The tasks to be performed are

1. generate x as coordinates between 0 and 1,

1. generate eps as random samples from a normal distribution with mean 0 and standard deviation 0.25,
2. compute y as the straight line -2\*x+3 plus the random perturbation eps,
3. form the least squares equations for fitting the parameters a and b in a line a\*x+b to the data points (the coefficient matrix has x in its first column and ones in the second, the right-hand side is the y data),
4. plot the data, the exact line, and the fitted line, with help of Easyviz.

The resulting script, found in src/py/intro/leastsquares.py, is quite short and (hopefully) self-explaining:

import sys

try:

n = int(sys.argv[1]) # no of data points except:

n = 20

from scitools.all import \* # import numpy and much of scitools

# compute data points in x and y arrays:

# x in (0,1) and y=-2\*x+3+eps, where eps is normally # distributed with mean zero and st.dev. 0.25. random.seed(20)

x = linspace(0.0, 1.0, n)

noise = random.normal(0, 0.25, n)

a\_exact = -2.0; b\_exact = 3.0

y\_line = a\_exact\*x + b\_exact

y = y\_line + noise

# create least squares system: A = array([x, zeros(n)+1])

A = A.transpose()

result = linalg.lstsq(A, y)

# result is a 4-tuple, the solution (a,b) is the 1st entry: a, b = result[0]

# plot:

plot(x, y, ’o’,

x, y\_line, ’r’,

x, a\*x + b, ’b’,

legend=(’data points’, ’original line’, ’fitted line’), title=’y = %g\*x + %g: fit to y = %g\*x + %s + normal noise’%\

(a, b, a\_exact, b\_exact), hardcopy=’tmp.ps’)

Figure 4.2 shows the resulting PostScript plot (the Gnuplot program was chosen as the backend for Easyviz).

There is an alternative and easier to use function polyfit in numpy, which fits a polynomial of a given degree d to a set of x-y data points stored in one-dimensional arrays x and y:

coeffs = polyfit(x, y, d)

Fig. 4.2. The result of the script leastsquares.py, demonstrating a least squares fit of a stright line through data points.

The coeffs list starts with the coefficients for the highest degree, i.e., the polynomial is coeffs[0]\*x\*\*d + ... + coeffs[-1]. In the present application of fitting a straight line we can write

a, b = polyfit(x, y, 1)

4.3.5 Arrays on Structured Grids

Suppose we have a two-dimensional grid consisting of points (xi,yj), i = 0,1,...,I, j = 0,1,...,J. The xi and yj coordinates are conveniently made as one-dimensional arrays, e.g.,

x = linspace(0, 1, 5); y = linspace(-1, 1, 5)

A frequently encountered task in this context is to fill a two-dimensional array ai,j with point values of some scalar function f(x,y) of two variables, i.e., ai,j = f(xi,yj) (the a array represents discrete values of the scalar field f(x,y) on a rectangular grid). Filling the array can be accomplished by a double loop:

a = zeros((x.size, y.size)) for i in xrange(x.size):

for j in xrange(y.size): a[i,j] = f(x[i], y[j])

However, these loops run slowly so we may want to vectorize the evalua- tion of a. The plain call a=f(x,y) does not work, as the following example demonstrates:

>>> def f(x,y):

... returnx+y

...

>>> x = linspace(0, 1, 3) >>> y = x.copy()

>>> f(x, y)

array([ 0., 1., 2.])

The expression x+y simply adds the two vectors elementwise, i.e., a = x + y impliesai =xi+yi foralli,whilewhatwewantisai,j =xi+yj.Wemay achieve the latter result if we redimension x as a two-dimensional representa- tion of a column vector, and y as a two-dimensional representation of a row vector.

Extending Coordinate Arrays for 2D Grids. We need to extend the one- dimensional coordinate arrays with one extra dimension of unit length. An obvious method is

xv=x; yv=y xv.shape = (x.size, 1) yv.shape = (1, y.size)

We can equivalently use the reshape method:

xv = x.reshape(x.size, 1); yv = y.reshape(1, y.size)

A third alternative employs the newaxis element to add a dimension to a NumPy array:

xv = x[:, newaxis]; yv = y[newaxis, :]

In all three cases, xv and yv shares the data with x and y.

Now xv+yv evaluates to a two-dimensional array with the i,j element as

x[i] + y[j]:

array([[ 0. , 0.5, 1. ], [ 0.5, 1. , 1.5],

[1., 1.5, 2.]])

The extended xv and yv arrays can be quickly made by calling the ndgrid

function in scitools:

from scitools.numpyutils import \* x = linspace(-2, 2, 101)

xv, yv = ndgrid(x, x)

# evaluate a function

def f(x, y):

return exp(-sqrt(x\*x + y\*y))

values = f(xv, yv)

xv =

yv =

zv =

# or

xv = x[:,newaxis,newaxis]

yv = y[newaxis,:,newaxis]

zv = z[newaxis,newaxis,:]

x.reshape(x.size, 1, 1) y.reshape(1, y.size, 1) z.reshape(1, 1, z.size)

4.3. More Advanced Array Computing 161

# plot values:

from scitools.easyviz import surfc surfc(xv, yv, values)

Extending Coordinate Arrays for 3D Grids. A three-dimensional box-shaped grid has grid-point locations on the form (xi,yj,zk). The coordinates in the three space directions can be represented by three one-dimensional arrays x, y, and z. To evaluate a function f(x,y,z) in a vectorized fashion, we must extend x to a three-dimensional array with unit length in the 2nd and 3rd dimensions, y to a three-dimensional array with unit length in the 1st and 3rd dimensions, and z to a three-dimensional array with unit length in the 1st and 2nd dimensions:

Calling a scalar function of three arguments, f(xv,yv,zv), may now yield a three-dimensional array holding f values at the points in the box grid. We remark that not all functions f(xv,yv,zv) will automatically work in vectorized mode (see Chapter 4.2.1, the example below, and Exercise 4.4).

Sometimes a scalar function is to be evaluated over the grid with one or more of the coordinates constant. For example, f(x,y0) for all x coordinates in the grid is computed straightforwardly by f(x,y\_0). The result is a one- dimensional array since x is a one-dimensional coordinate array and y\_0 is a scalar. In 3D, however, the computations get more involved. Say we want to evaluate f(x,y0,z) for all x and z values, while y0 is the maximum y coordinate. Now we need two-dimensional extensions of the x and z coordinate arrays:

x2 = x[:,newaxis]; z2 = z[newaxis,:] v = f(x2, y[-1], z2)

The result v is a two-dimensional array reflecting the grid in an xz plane. We may assign this array to a slice of a three-dimensional array over all the grid points in a given plane:

u[:,-1,:] = v

Computing f(x0,y0,z) for fixed x0 and y0, while z takes on all legal coordi- nates is simple since this computation only involves a one-dimensional grid. We simply call f(x\_0,y\_0,z).

The ndgrid function mentioned above also handles 3D grids and boundary slices of 3D grids. For example, in a box grid on [0,1]×[0,1]×[0,2] we can extract the extended grid coordinates for a grid in the plane z = 1.5:

4. Numerical Computing in Python

>>> x = linspace(0, 1, 3) >>> y = linspace(0, 1, 2)

>>> # 2D slice

>>> z = 1.5

>>> xv, yv, zv

>>> xv

array([[ 0. ],

[ 0.5],

[ 1. ]]) >>> yv

array([[ 0., 1.]]) >>> zv

1.5

of a 3D grid, with z=const: = ndgrid(x, y, z)

A Class for 2D Grids. To hide the extensions of the coordinate arrays with newaxis or reshape constructions, we can create a more easy-to-use grid class (see Chapter 3.2.9 for a quick intro to Python classes). Limiting the interest to uniform grids with constant spacings in the x and y direction, we could write the class as follows:

class Grid2D:

def \_\_init\_\_(self,

xmin=0, xmax=1, dx=0.5,

ymin=0, ymax=1, dy=0.5):

# coordinates in each space direction: self.xcoor = seq(xmin, xmax, dx) self.ycoor = seq(ymin, ymax, dy)

# store for convenience:

self.dx = dx; self.dy = dy

self.nx = self.xcoor.size; self.ny = self.ycoor.size

# make two-dim. versions of the coordinate arrays: # (needed for vectorized function evaluations) self.xcoorv = self.xcoor[:, newaxis]

self.ycoorv = self.ycoor[newaxis, :]

def vectorized\_eval(self, f):

"""Evaluate a vectorized function f at each grid point.""" return f(self.xcoorv, self.ycoorv)

The class may be used as illustrated below:

g = def

a =

Grid2D(xmax=10, ymax=3, dx=0.5, dy=0.02)

myfunc(x, y):

return x\*sin(y) + y\*sin(x)

g.vectorized\_eval(myfunc)

# check point value:

i = 3; j = g.ny-4; x = g.xcoor[i]; y = g.ycoor[j]

print ’f(%g, %g) = %g = %g’ % (x, y, a[i,j], myfunc(x, y))

# less trivial example:

def myfunc2(x, y): return 2.0

4.3. More Advanced Array Computing 163

a = g.vectorized\_eval(myfunc2)

In the second example, a becomes just the floating-point number 2.0, not an array. We need to vectorize the constant function myfunc2 to get it to work properly in the present context:

def myfunc2v(x, y):

return zeros((x.shape[0], y.shape[1])) + 2.0

a = g.vectorized\_eval(myfunc2v)

Extensions and testing of the class take place in Chapters 8.9.2, 9, and 10.

4.3.6 File I/O with NumPy Arrays

Writing a NumPy array to file and reading it back again can be done with the repr and eval functions2, respectively, as the following code snippet demon- strates:

a = linspace(1, 21, 21) a.shape = (2,10)

# ASCII format:

file = open(’tmp.dat’, ’w’)

file.write(’Here is an array a:\n’)

file.write(repr(a)) # dump string representation of a file.close()

# load the array from file into b:

file = open(’tmp.dat’, ’r’)

file.readline() # load the first line (a comment) b = eval(file.read())

file.close()

Now, b contains the same values as a. Note that repr(a) normally will span multiple lines so storing more than one array in a file requires some delimiter text between the arrays.

When working with large NumPy arrays that are written to or read from files, binary format results in smaller files and significantly faster in- put/output operations. The simplest way of storing and retrieving NumPy arrays in binary format is to use pickling (see Chapter 8.3.2) via the cPickle module:

# a1 and a2 are two arrays

2 See page 363 for examples of how eval and str or repr can be used to read and write Python data structures from/to files.

4. Numerical Computing in Python

import cPickle

file = open(’tmp.dat’, ’wb’) file.write(’This is the array a1:\n’) cPickle.dump(a1, file)

file.write(’Here is another array a2:\n’) cPickle.dump(a2, file)

file.close()

file = open(’tmp.dat’, ’rb’)

file.readline() # swallow the initial comment line b1 = cPickle.load(file)

file.readline() # swallow next comment line

b2 = cPickle.load(file)

file.close()

One can also store NumPy arrays in binary format using the technique of shelving (Chapter 8.3.3).

NumPy has special functions for converting an array to and from binary format. The binary format is just a sequence of bytes stored in a plain Python string. This sequence of bytes only contains the array elements and not infor- mation on the shape and data type. In the code segment below we therefore store the size of the array and its shape as plain text preceding the binary data:

file = open(’tmp.dat’, ’wb’)

a\_binary = a.tostring() # convert to binary format string # store first length (in bytes):

file.write(’%d\n%s\n’ % (a\_binary.size, str(a.shape))) file.write(a\_binary) # dump string

file.close()

file = open(’tmp.dat’, ’rb’)

# load binary data into b:

nbytes = int(file.readline()) # or eval(file.readline()) b\_shape = eval(file.readline())

b = fromstring(file.read(nbytes))

b.shape = b\_shape

file.close()

As always when working with binary files, be careful with potential little- or big-endian problems when the files are moved from one computer platform to another (see page 369). NumPy has functions for checking which endian format the elements have, and array objects have a byteswap() method for swapping between little- and big-endian.

Chapters 8.4.2–8.4.5 demonstrate and evaluate the use of standard Python pickling, C-implemented (cPickle) pickling, formatted ASCII storage, and shelving of NumPy arrays. The technique utilizing the cPickle module has the fastest I/O and the lowest storage costs.

More general information on binary files and related input/output oper- ations is provided in Chapter 8.3.6 and in the documentation of the struct module in the Python Library Reference.

Numerical data are often stored in plain ASCII files with numbers in rows and columns. Such files can be read into two-dimensional NumPy arrays for numerical processing. We have made a module scitools.filetable for reading and writing such tabular data from/to files. A simple example will illustrate how the module can be used. Assume we have a data file tmp.dat like this:

0 1 2 3 4 5

0.0 0.0 1.0 1.0 1.0 2.0 4.0 8.0 17.0 9.0 27.0 82.0

16.0 64.0 257.0 25.0 125.0 626.0

The following interactive session demonstrates how we can load this file into a two-dimensional NumPy array:

>>> import scitools.filetable as ft >>> s = open(’tmp.dat’, ’r’)

>>> table = ft.read(s)

>>> s.close()

>>> print table

[[ 0. 0. 0. 1.]

[ 1. 1. 1. 2.] [ 2. 4. 8. 17.] [ 3. 9. 27. 82.]

[ 4. 16. 64. 257.]

* [ 5. 25. 125. 626.]]  
     
  Instead of reading the tabular data into two-dimensional array, the function read\_columns returns a list of one-dimensional arrays, one for each column of data:  
     
  >>> s = open(’tmp.dat’, ’r’)  
   >>> x, y1, y2, y3 = ft.read\_columns(s) >>> s.close()  
   >>> print x  
   [ 0. 1. 2. 3. 4. 5.]  
   >>> print y1  
   [ 0. 1.  
   >>> print y2

[ 0. 1.

>>> print y3

[ 1. 2.

There are corresponding functions write and write\_columns for writing a two-dimensional array and a set of one-dimensional arrays (columns) to file, respectively. We refer to the documentation of the scitools.filetable mod- ule for more details and examples.

The scripts src/py/intro/datatrans3x.py, with x as a, b, c, and d, imple- ment different strategies for reading tabular data from files. There is a test script datatrans-eff.py in the same directory which can be used to measure the efficiency of the various strategies.

The numpyutils module in the scitools package provides some useful add on functions to what is found in NumPy:

– seq: The seq function is similar to arange and linspace. It does the same as arange, but guarantees to include the upper limit of the array. Contrary to linspace, seq requires the increment between two elements and not the total number of elements as argument.

seq(0, 1, 0.2)

seq(min=0, max=1, inc=0.2) seq(0, 6, 2, int)

seq(3)

The signature of the function reads

# 0., 0.2, 0.4, 0.6, 0.8, 1.0 # same as previous line

# 0, 2, 4, 6 (integers)

# 0., 1., 2., 3.

def seq(min=0.0, max=None, inc=1.0, type=float, return\_type=’NumPyArray’):

The return\_type string argument specifies the returned data structure holding the generated numbers: ’NumPyArray’ or ndarray implies a NumPy array, ’list’ returns a standard Python list, and ’tuple’ returns a tuple. Basically, the function creates a NumPy array using

r = arange(min, max + inc/2.0, inc, type)

and coverts r to list or tuple if necessary.

A warning is demanded regarding the standard use of arange: This func- tion claims to not include the upper limit, but sometimes the upper limit is included due to round-off errors. Try out the following code segment on your computer to see how often the last element in a contains the upper limit 1.0 or not:

N = 1001

for n in range(1, N):

a = arange(0, 1, 1.0/n) last = a[-1]

print a.size-n, n, last

On my computer, the upper limit was included in 58 out 1001 cases, and a then contained an extra element. Therefore, I suggest to avoid arange for floating-point numbers and stick to linspace or seq.

– iseq: The fact that range and xrange do not include the upper limit in integer sequences can be confusing or misleading sometimes when im- plementing mathematical algorithms. The numpyutils module therefore offers a function for generating integers from start up to and including stop in increments of inc:

def iseq(start=0, stop=None, inc=1):

if stop is None: # simulate xrange(start+1) behavior

stop = start; start = 0; inc = 1 return xrange(start, stop+inc, inc)

A relevant example may be coding of a formula like

xk =(ck −Ak,2xk+1)/dk, i=n−2,n−3,...,0,

which translates into

for k in iseq(n-2, 0, -1):

x[k] = (c[k] - A[k,2]\*x[k+1])/d[k]

Many find this more readable and easier to debug than a loop built with range(n-2,-1,-1).

The iseq function is in general recommended when you need to iterate over a part of an array, because it is easy to control that the arguments to iseq correspond exactly to the loop limits used in the mathematical specification of the algorithm. Such details are often important to quickly get a correct implementation of an algorithm.

– float\_eq: float\_eq(a, b, rtol, atol) returns a true value if a and b are equal within a relative tolerance rtol (default 1014) and an absolute tol- erance atol (default 1014). More precisely, the float\_eq function returns a true value if

abs(a-b) < atol + rtol\*abs(b)

The arguments a and b can be float variables or NumPy arrays. In the

latter case, float\_eq just calls allclose in numpy.

– ndgrid: This function extends one-dimensional coordinate arrays with extra dimensions, which is required for vectorized operations for com- puting scalar and vector fields over 2D and 3D grids, as explained in Chapter 4.3.5. For example,

>>> x = linspace(0, 1, 3) >>> y = linspace(0, 1, 2) >>> xv, yv = ndgrid(x, y) >>> xv

array([[ 0. ],

[ 0.5],

[ 1. ]]) >>> yv

array([[ 0., 1.]])

# coordinates along x axis # coordinates along y axis

The ndgrid function also handles boundary grids, i.e., 1D/2D slices of 3D grids with one/two of the coordinates kept constant, see the documenta- tion of the function for further details.

(Remark. There are several ndgrid-like functions in numpy: meshgrid, mgrid, and ogrid, but scitools has its own ndgrid function because meshgrid in numpy is limited to 2D grids only and it always returns a full 2D array and not the “sparse” extensions used in Chapter 4.3.5 (unit length in the added dimensions). The ogrid function can produce “sparse” extensions, but neither ogrid nor mgrid allow for non-uniform grid spacings. The ndgrid in scitools.numpyutils also allow for both “matrix” indexing and “grid” indexing of the coordinate arrays. All of these additional features are important when working with 2D and 3D grids.)

– wrap2callable: This is a function for turning integers, real numbers, func- tions, user-defined objects (with a \_\_call\_\_ method), string formulas, and discrete grid data into some object that can be called as an ordinary func- tion (see Chapters 12.2.1 and 12.2.2). You can write a function

def df(f, x, h):

f = wrap2callable(f) # ensure f is a function: f(x) return (f(x+h) - f(x-h))/(2.0\*h)

and call df with a variety of arguments:

x = 2; h = 0.01

print df(4.2, x, h) print df(’sin(x)’, x, h)

def q(x):

return sin(x)

print df(q, x, h)

# constant 4.2

# string function, sin(x)

# user-defined function q

xc = seq(0, 4, 0.05); yc = sin(xc)

print df((xc,yc), x, h) # discrete data xc, yc

The constant 4.2, user-defined function q, discrete data (xc,yc), and string formula ’sin(x)’ will all be turned, by wrap2callable, into an ob- ject f, which can be used as an ordinary function inside the df function. Chapter 12.2.2 explains how to construct the wrap2callable tool.

– arr: This function provides a unified short-hand notation for creating arrays in many different ways:

a = arr(100) # as zeros(100)

a = arr((M,N)) # as zeros((M,N))

a = arr((M,N), element\_type=complex) # Complex elements

a = arr(N, interval=[1,10])

a = arr(data=mylist)

a = arr(data=myarr, copy=True) a = arr(file\_=’tmp.dat’)

# as linspace(1,10,N)

# as asarray(mylist)

# as array(myarr, copy=1)

# load tabular data from file

The arr function is just a simple, unified interface to the zeros and array function in NumPy, plus some file reading statements. The file format is a table with a fixed number of columns and rows where whitespace is the delimiter between numbers in a row. One- and two-dimensional arrays can be read this way. The arr function makes several consistency and error checks that are handy to have automated and hidden.

4.3.8 Exercises

Exercise 4.8. Implement Exercise 2.9 using NumPy arrays.

Solve the same problem as in Exercise 2.9, but use Numerical Python and a vectorized algorithm. That is, generate a (long) random vector e of 2n uniform integer numbers ranging from 1 to 6, find the entries that are 6 by using where(e == 6, 1, 0), reshape the vector to a two-dimensional

2 × n array, add the two rows of this array to a new array e2, count how many of the elements in e2 that are greater than zero (these are the events where at least one die shows a 6) by sum(where(e2 > 0, 1, 0)). Estimate the probability from this count. Insert CPU-time measurements in the scripts (see Chapter 4.1.4 or 8.10.1) and compare the plain Python loop and the standard random module with the vectorized version utilizing random, where, and sum from numpy. ⋄

Exercise 4.9. Implement Exercise 2.10 using NumPy arrays.

Solve the same problem as in Exercise 2.10, but use Numerical Python and a vectorized algorithm. Generate a random vector of 4n uniform integer numbers ranging from 1 to 6, reshape this vector into an array with four rows and n columns, representing the outcome of n throws with four dice, sum the eyes and estimate the probability. Insert CPU-time measurements in the scripts (see Chapter 4.1.4 or 8.10.1) and compare the plain Python solution in Exercise 2.10 with the version utilizing NumPy functionality.

Hint: You may use the numpy functions random.randint, sum, and < (read about them in the NumPy reference manual, and notice especially that sum can sum the rows or the columns in a two-dimensional array). ⋄

Exercise 4.10. Replace lists by NumPy arrays in convert2.py.

Modify the convert2.py such that the data are read into NumPy arrays and written to files using either the scitools.filetable or TableIO modules (see Chapter 4.3.6). The y variable should be a dictionary where the values are one-dimensional NumPy arrays. ⋄

Exercise 4.11. Use Easyviz in the simviz1.py script.

The simviz1.py script from Chapter 2.3 creates a file with Gnuplot com-

mands and executes Gnuplot via an operating system call. As an alterna- tive to this approach, use Easyviz from Chapter 4.3.3 to make the graphics. Load the data in the sim.dat file into NumPy arrays in the script, using the filetable module from Chapter 4.3.6. Thereafter, use the plot function with appropriate parameters to plot the data, set a title reflecting input parame- ters, and create a hardcopy. ⋄

Exercise 4.12. Extension of Exercise 2.8.

Make a script as described in Exercise 2.8, but now you should modify the src/py/intro/datatrans3.py script instead, i.e., all columns in the input file are stored in NumPy arrays. Construct a new NumPy array with the averages and write all arrays to an output file. ⋄

Exercise 4.13. NumPy arrays and binary files.

Make a version of the src/py/intro/datatrans3a.py script (see Chap- ter 4.3.6) that works with NumPy arrays and binary files (see Chapter 4.3.6). For testing purposes, you will need two additional scripts for generating and viewing binary files (see also Exercise 8.21).

Exercise 4.14. One-dimensional Monte Carlo integration.

One of the earliest applications of random numbers was numerical compu- tation of integrals. Let x1,...,xn be uniformly distributed random numbers

between a and b. Then

b − a 􏰉n

f(xi) (4.3) is an approximation to the integral 􏰊 b f (x)dx. This method is usually referred

n

i=1 a

to as Monte Carlo integration. The uncertainty in the approximation of the integral is estimated by the standard deviation

􏰋􏰌 􏰋􏰌

σ ̄= √n n−1 f(xi) −n−1(f) ≈ √

b − a 􏰌 1 􏰉n n b − a 􏰌 1 􏰉n 􏰍2 ̄2􏰍2 ̄2

f(xi) −(f), (4.4)

where f ̄ = n−1

is needed to compute integrals accurately (standard rules, such as Simpson’s rule, the Trapezoidal rule, or Gauss-Legendre rules are more efficient). How- ever, Monte Carlo integration is efficient for higher-dimensional integrals (see next exercise).

Implement the Monte Carlo integration (4.3) in a Python script with an

explicit loop and calls to the random.random() function for generating random

􏰇

f(xi). Since σ ̄ tends to zero as n−1/2, a quite large n

i=1

i=1

n

n

i=1

numbers. Print the approximation to the integral and the error indicator

(4.4). Test the script on the integral 􏰊 π sin x dx. Add code in the script where 0

you utilize NumPy functionality for random number generation, i.e., a long vector of random samples are generated, f is applied to this vector, followed by a sum operation and division by n. Compare timings of the plain Python code and the NumPy code.

We remark that the straightforward Monte Carlo algorithm presented above can often be significantly improved by introducing more clever sam- pling strategies [30, Ch. 7.8]. ⋄

Exercise 4.15. Higher-dimensional Monte Carlo integration.

This exercise is a continuation of Exercise 4.14. Our aim now is to compute the m-dimensional integral

􏰈

f(x1,...,xm)dx1 ···dxm, (4.5) Ω

where Ω is a domain of general shape in IRm. Monte Carlo integration is well suited for such integrals. The idea is to embed Ω in a box B,

B = [α1,β1]×···[αm,βm], such that Ω ⊂B. Define a new function F on B by

􏰀f(x1,...,xm) if (x1,...,xm) ∈ Ω F(x1,...,xm) = 0, otherwise (4.6)

The integral (4.5) can now be computed as

􏰈 volume(B) 􏰉n

4.3. More Advanced Array Computing 171

F(x(i),...,x(i)), (4.7) where x1 ,...,xm , for i = 1,...,n and j = 1,...,m, are mn independent,

f(x ,...,x )dx ···dx ≈ 1m1mn1m

Ω i=1 (i) (i)

uniformly distributed random numbers. To generate x(i), we just draw a j

number from the one-dimensional uniform distribution on [αj,βj].

Make a Python script for higher-dimensional integration using Monte Carlo simulation. The function f and the domain Ω should be given as Python

functions. Make use of NumPy arrays.

Apply the script to functions where the integral is known, compute the

errors, and estimate the convergence rate empirically. ⋄

Exercise 4.16. Load data file into NumPy array and visualize.

The file src/misc/temperatures.dat contains monthly and annual temper- ature anomalies on the northern hemisphere in the period 1856–2000. The anomalies are relative to the 1961–1990 mean. Visualizing these anomalies may show if the temperatures have increased towards the end of the last century.

Make a script taking the uppercase three-letter name of a month as command-line argument (JAN, FEB, etc.), and visualizes how the tempera- ture anomalies vary with the years. Hint: Load the file data into a NumPy array, as explained in Chapter 4.3.6, and send the relevant columns of this array to Gnuplot for visualization. You can use a dictionary to map from month names to column indices. ⋄

Exercise 4.17. Analyze trends in the data from Exercise 4.16.

This is a continuation of Exercise 4.16. Fit a straight line (by the method of least squares, see Chapter 4.3.4) to the temperature data in the period 1961-1990 and another straight line to the data in the period 1990-2000. Plot the two lines together with the noisy temperature anomalies. If the straight line fit for the period 1990-2000 is significantly steeper than the straight line fit for the period 1961-1990 it indicates a significant temperature rise in the 1990s. Hint: To find the index corresponding to (say) the entry 1961, you can convert the NumPy data to a Python list by the tolist method and then use the index method for lists (i.e., data[:,0].tolist().index(1961)).

On http://cdiac.ornl.gov/trends/temp/jonescru/data.html one can find more temperature data of this kind. ⋄

Exercise 4.18. Evaluate a function over a 3D grid.

Write a class Grid3D for representing a three-dimensional uniform grid on a box with user-defined dimensions and cell resolution. The class should be able to compute a three-dimensional array of function values over the grid points, given a Python function. Here is an exemplifying code segment:

Fig. 4.3. Plot to be made by the script in Exercise 4.17. Temperature deviations in November, relative to the 1961–1990 mean, are shown together with a straight line fit to the 1961–1990 and the 1990-2000 data.

g = Grid3D(xmin=0, xmax=1, dx=0.1, ymin=0, ymax=10, dy=0.5,

zmin=0, zmax=2, dz=0.02)

f = g.vectorized\_eval(lambda x,y,z: sin(x)\*y + 4\*z)

i=2; j=3; k=0

print ’value at (%g,%g,%g) = f[%d,%d,%d] = %g’ % \

(g.xcoor[i], g.ycoor[j], g.zcoor[k], i, j, k, f[i,j,k])

Read Chapter 4.3.5 about a similar class Grid2D and extend the code to

three-dimensional grids.

⋄

Extend Exercise 4.18 such that we can evaluate a function over a 3D grid when one or two coordinates are held constant. Given a Grid3D object g, we can typically write

a = g.vectorized\_eval2(f, x=ALL, y=MIN, z=ALL)

to evaluate f(x,y0,z) for all x and z coordinates, while y0 is the minimum y coordinate. Another example is

a = g.vectorized\_eval2(f, x=MAX, y=MIN, z=ALL)

where we evaluate some f(x0,y0,z) for all z coordinates and with x0 as the maximum x coordinate and y0 as the minimum y coordinate. We can of course use a numerical value for the x, y, and z arguments as well, e.g.,

a = g.vectorized\_eval2(f, x=MAX, y=2.5, z=ALL)

You may use the trick on page 401 and implement the function in a subclass, still with name Grid3D, to avoid touching the original Grid3D.py file.

Implement MIN, MAX, and ALL as global constants in the file. These constants must have values that do not interfer with floating-point numbers so strings might be an appropriate type (say MIN=’min’, etc.).

⋄

Several Python packages offer numerical computing functionality beyond what is found in Numerical Python. Some of the most important ones are de- scribed in the following. This covers ScientificPython, SciPy, and the Python– Matlab interface, presented in Chapters 4.4.1–4.4.3, respectively. Such pack- ages are built on Numerical Python. We also provide, in Chapter 4.4.5, a list of many other packages of relevance for scientific computing with Python.

4.4.1 The ScientificPython Package

The ScientificPython package, developed by Konrad Hinsen, contains nu- merous useful modules for scientific computing. For example, the package offers functionality for automatic differentiation, interpolation, data fitting via nonlinear least-squares, root finding, numerical integration, basic statis- tics, histogram computation, visualization, and parallel computing (via MPI or BSP). The package defines several data types, e.g., physical quantities with dimension, 3D vectors, tensors, and polynomials, with associated oper- ations. I/O functionality includes reading and writing netCDF files and files with Fortran-style format specifications. The ScientificPython web page (see link in doc.html) provides a complete overview of the various modules in the package. Some simple examples are provided below.

The strength of ScientificPython is that the package contains (mostly) pure Python code, which is trivial to install. A subset of ScientificPython, dealing with integration, interpolation, statistics, root finding, etc., is also offered by SciPy (Chapter 4.4.2), usually in a faster compiled implementation. However, SciPy is more difficult to install on Unix, so if ScientificPython has the desired functionality and is fast enough, it represents an interesting alternative.

Both a tutorial and a reference manual are available for ScientificPython. The code itself is very cleanly written and constitutes a good source for doc- umentation as well as a starting point for extensions and customizations to fit special needs. ScientificPython is also a primary example on how to orga- nize a large Python project in terms of classes and modules into a package, and how to embed extensive documentation in doc strings. Before you dive into the source code, you should gain considerable familiarity with Numerical Python.

The next pages show some examples of the capabilities of ScientificPython. Our applications here are mostly motivated by needs later in the book.

Physical Quantities with Dimension. A very useful feature of ScientificPython is the ability to perform calculations with physical units and convert from one unit to another. The basic tool is class PhysicalQuantity, which repre- sents a number and an associated unit (dimension). An interactive session demonstrates some of the capabilities:

>>> from Scientific.Physics.PhysicalQuantities \ import PhysicalQuantity as PQ

>>> m = PQ(12, ’kg’) # number, dimension

>>> a = PQ(’0.88 km/s\*\*2’) # alternative syntax (string) >>> F = m\*a

>>> F

PhysicalQuantity(10.56,’kg\*km/s\*\*2’)

>>> F = F.inBaseUnits()

>>> F

PhysicalQuantity(10560.0,’m\*kg/s\*\*2’)

>>> F.convertToUnit(’MN’) # convert to Mega Newton

>>> F

PhysicalQuantity(0.01056,’MN’)

>>> F = F + PQ(0.1, ’kPa\*m\*\*2’) # kilo Pascal m 2

>>> F

PhysicalQuantity(0.010759999999999999,’MN’)

>>> str(F)

’0.010759999999999999 MN’

>>> value = F.getValue()

>>> value

0.010759999999999999

>>> F.inBaseUnits() PhysicalQuantity(10759.999999999998,’m\*kg/s\*\*2’)

>>> PQ(’0 degC’).inUnitsOf(’degF’) # Celcius to Farenheit PhysicalQuantity(31.999999999999936,’degF’)

I recommend reading the source code of the module to see the available units. Unum by Pierre X. Denis (see link from doc.html) is another and more advanced Python module for computing with units and performing unit con- version. Unum supports unit calculations also with NumPy arrays. One dis- advantage with Unum is that the input and output formats are different. I therefore prefer to use PhysicalQuantity from ScientificPython when this

module provides sufficient functionality.

Automatic Differentiation. The module Derivatives enables differentiation of expressions:

>>> from Scientific.Functions.Derivatives import DerivVar as D

>>> def

somefunc(x, y, z): return 3\*x - y + 10\*z\*\*2

>>> x =

>>> y =

>>> z =

>>> r =

>>> r

(6.0250000000000004, [3.0, -1.0, 1.0])

D(2, index=0) # variable no. 0 with value 2 D(0, index=1) # variable no. 1 with value 0 D(0.05, index=2) # variable no. 2 with value 0.05 somefunc(x, y, z)

The DerivVar (with short form D in this example) defines the value of a variable and, optionally, its number in case of multi-valued functions. The result of computing an expression with DerivVar instances is a new DerivVar instance, here named r, containing the value of the expression and the value of the partial derivatives of the expression. In our example, 6.025 is the value of somefunc, while [3.0, -1.0, 1.0] are the values of somefunc differentiated with respect to x, y, and z (the list index corresponds to the index argument in the construction of DerivVar instances). There is, naturally, no need for numbering the independent variable in the single-variable case:

>>> from numpy import \*

>>> print sin(D(0.0))

(0.0, [1.0]) # (sin(0), [cos(0)])

Note that the sin function must allow NumPy array arguments. Higher-order derivatives can be computed by specifying an order keyword argument to the DerivVar constructor:

>>> x = D(1, order=3)

>>> x\*\*3

(1, [3], [[6]], [[[6]]]) # 0th, 1st, 2nd, 3rd derivative

A derivative of n-th order is represented as an n-dimensional list. For example, 2nd order derivatives of somefunc can be computed by

>>> x

>>> y

>>> z

>>> r

>>> r

(40, [3, -1, 20], [[0, 0, 0], [0, 0, 0], [0, 0, 20]]) >>> r[2][2][0] # d 2(somefunc)/dzdx

= D(10, index=0, order=2) = D(0, index=1, order=2) = D(1, index=2, order=2) = somefunc(x, y, z)

0

>>> r[2][2][2] # d 2(somefunc)/dz 2 20

The module FirstDerivatives is more efficient than Derivatives for comput- ing first-order derivatives. To use it, just do

from Scientific.Functions.FirstDerivatives import DerivVar

An alternative to automatic differentiation with ScientificPython is to use the SymPy package for symbolic differentiation, see Chapter 4.4.4.

Interpolation. Class InterpolatingFunction in the Interpolation module offers interpolation of an m-valued function of n variables, defined on a box- shaped grid. Let us first illustrate the usage by interpolating a scalar function of one variable:

>>> from Scientific.Functions.Interpolation \ import InterpolatingFunction as Ip

>>> from scitools.numpyutils import \* >>> t = linspace(0, 10, 101)

>>> v = sin(t)

>>> vi = Ip((t,), v)

>>> # interpolate and compare with exact result: >>> vi(5.05), sin(5.05)

(-0.94236947849543551, -0.94354866863590658)

>>> # interpolate the derivative of v:

>>> vid = vi.derivative()

>>> vid(5.05), cos(5.05)

(0.33109592335406074, 0.33123392023675369)

>>> # compute the integral of v over all t values: >>> vi.definiteIntegral(), -cos(t[-1]) - (-cos(t[0])) (1.837538713981457, 1.8390715290764525)

As a two-dimensional example, we show how we can easily interpolate func- tions defined via class Grid2D from Chapter 4.3.5:

>>> # make sure we can import Grid2D.py:

>>> sys.path.insert(0, os.path.join(os.environ[’scripting’],

’src’, ’py’, ’examples’)) # location of Grid2D >>> from Grid2D import Grid2D

>>> g = Grid2D(dx=0.1, dy=0.2)

>>> f = g(lambda x, y: sin(pi\*x)\*sin(pi\*y))

>>> fi = Ip((g.xcoor, g.ycoor), f)

>>> # interpolate at (0.51,0.42) and compare with exact result: >>> fi(0.51,0.42), sin(pi\*0.51)\*sin(pi\*0.42) (0.94640171438438569, 0.96810522380784525)

Nonlinear Least Squares. Suppose you have a scalar function of d variables (x1,...,xd) and n parameters (p1,...,pn),

f(x1,...,xd;p1,...,pn),

and that we have m measurements of values of this function:

f(i) = f(x(i),...,x(i);p1,...,pn), i = 1,...,m. 1d

To fit the parameters p1,...,pn in f to the data points

((x(i),...,x(i)),f(i)), i = 1,...,m, 1d

a nonlinear least squares method can be used. This method is available through the leastSquaresFit function in the LeastSquares module in Sci- entificPython. The function makes use of the standard Levenberg-Marquardt algorithm, combined with automatic derivatives of f.

The user needs to provide a function for evaluating f:

def f(p, x):

...

return scalar\_value

Here, p is a list of the n parameters p1,...,pn, and x is a list of the values of the d independent variables x1,...,xd in f. The set of data points is collected in a nested tuple or list:

((x1, f1), ..., (xm, fm))

# or

((x1, f1, s1), ..., (xm, fm, sm))

The x1,. . . ,xm tuples correspond to the (x(i) , . . . , x(i) ) set of independent vari- 1d

ables, and f1,. . . ,fm correspond to f(i). The s1,. . . , sm parameters are optional, default to unity, and reflect the statistical variance of the data point, i.e., the inverse of the point’s statistical weight in the fitting procedure.

The nonlinear least squares fit is obtained by calling

from Scientific.Functions.LeastSquares import leastSquaresFit r = leastSquaresFit(f, p\_guess, data, max\_iterations=None)

where f is the function f in our notation, p\_guess is an initial guess of the solution, i.e., the p1, . . . , pn values, data holds the nested tuple of all data points (((x1,f1),...,(xm,fm))), and the final parameter limits the number of iterations in case of convergence problems. The return value r contains a list of the optimal p1, . . . , pn values and the chi-square value describing the quality of the fit.

A simple example may illustrate the use further. We want to fit the pa- rameters C, a, D, and b in the model

e(Δx,Δt;C,a,D,b) = CΔxa + DΔtb

to data ((Δx(i),Δy(i)),e(i)) from a numerical experiment3. In our test we randomly perturb the e function to produce the data set.

>>> def

...

...

...

...

error\_model(p, x): C,a,D,b=p dx,dt=x

e = C\*dx\*\*a + D\*dt\*\*b return e

3 A typical application is fitting a convergence estimate for a numerical method for solving partial differential equations with space cell size Δx and time step size Δt.

...

>>> data = []

* >>> import random; random.seed(11)
* >>> C = 1; a = 2; D = 2; b = 1; p = (C, a, D, b)
* >>> dx = 0.5; dt = 1.0
* >>> for i in range(7): # create 7 data points  
     
  dx/=2; dt/=2  
   e = error\_model(p, (dx, dt))  
   e += random.gauss(0, 0.01\*e) # make some noise in e data.append( ((dx,dt), e) )
* >>> from Scientific.Functions.LeastSquares import leastSquaresFit
* >>> p\_guess = (1, 2, 2, 1) # exact guess... (if no noise)
* >>> r = leastSquaresFit(error\_model, p\_guess, data)
* >>> r[0] # fitted parameter values  
     
  [1.0864630262152011, 2.0402214672667118, 1.9767714371137151, 0.99937257343868868]  
     
  >>> r[1] # quality of fit 8.2409274338033922e-06  
     
  The results are reasonably accurate.  
     
  Statistical Data Analysis. The ScientificPython package also support some simple statistical data analysis, as exemplified by the code below:  
     
  from numpy import random  
   import Scientific.Statistics as S  
   data = random.normal(1.0, 0.5, 100000)  
   mean = S.mean(data)  
   stdev = S.standardDeviation(data)  
   median = S.median(data)  
   skewness = S.skewness(data)  
   print ’mean=%.2f standard deviation=%.2f skewness=%.1f ’\  
     
  ’median=%.2f’ % (mean, stdev, skewness, median)

The documentation of the Scientific.Statistics module contains a few more

functions for analysis. Histogram computations are also possible:

from Scientific.Statistics.Histogram import Histogram

h = Histogram(data, 50) # use 50 bins between min & max samples h.normalize() # make probabilities in histogram

The histogram can easily be plotted:

from scitools.easyviz import \* plot(h.getBinIndices(), h.getBinCounts())

You can run the src/py/intro/ScientificPython.py script to see what the resulting graphs look like.

4.4.2 The SciPy Package

The SciPy package [14], primarily developed by Eric Jones, Travis Oliphant, and Pearu Peterson, is an impressive and rapidly developing environment for

scientific computing with Python. It extends ScientificPython significantly, but also has some overlap. The SciPy tutorial provides a good example- oriented overview of the capabilities of the package. The forthcoming exam- ples on applying SciPy are meant as an appetizer for the reader to go through the SciPy tutorial in detail.

SciPy might require some efforts in the installation on Unix, see Ap- pendix A.1.5. The source code of the SciPy Python modules provides a good source of documentation, foremost in terms of carefully written doc strings, but also in terms of clean code. You can either browse the source code di- rectly or get the function signatures and doc strings formatted by pydoc or the help function in the Python shell.

Help Functionality. SciPy has a nice built-in help functionality. If you have done the recommended

from scipy import \*

then you can write info(mod) or info(mod.name) for getting the documen- tation of a module mod, or a function or class name in mod. For many SciPy modules the standard help utility drowns the user in information (mainly because of all the imported names in SciPy modules), but the info function provides just the doc string.

Studying the source code of a function is sometimes a necessary way to obtain documentation, especially about how arguments are treated and what the return values really are. SciPy has a function source which displays the source code of an object, e.g., source(mod.name).

Special Mathematical Functions. The scipy.special module contains a wide range of special mathematical functions: Airy functions, elliptic functions and integrals, Bessel functions, gamma and related functions, error func- tions, Fresnel integrals, Legendre functions, hyper-geometric functions, Math- ieu functions, spheroidal wave functions, and Kelvin functions. Run inside a Python shell from scipy import special and then info(special) to see a listing of all available functions.

Just as an example, let us print the first four zeros of the Bessel function J3:

>>> from scipy.special import jn\_zeros

>>> jn\_zeros(3, 4)

array([ 6.3801619 , 9.76102313, 13.01520072, 16.22346616])

SciPy is well equipped with doc strings so it is easy to figure out which functions to call and what the arguments are.

Integration. SciPy has interfaces to the classical QUADPACK Fortran li- brary from Netlib [25] for numerical computations of integrals. A simple illustration is

>>> from scipy import integrate >>> def myfunc(x):

return sin(x)

>>> result, error = integrate.quad(myfunc, 0, pi) >>> result, error

(2.0, 2.2204460492503131e-14)

The quad function can take lots of additional arguments (error tolerances among other things). The underlying Fortran library requires the function to be integrated to take one argument only, but SciPy often allows additional arguments represented as a tuple/list args (this is actually a feature of F2PY when wrapping the Fortran code). For example,

>>> def myfunc(x, a, b): return a + b\*sin(x)

>>> p=0; q=1

>>> integrate.quad(myfunc, 0, pi, args=(p,q), epsabs=1.0e-9) (2.0, 2.2204460492503131e-14)

There are also functions for various types of Gauss quadrature.

ODE Solvers. SciPy’s integrate module makes use of the widely used ODEPACK Fortran software from Netlib [25] for solving ordinary differential equations (ODEs). The integrate.odeint function applies the LSODA For- tran routine as solver. There is also a base class IntegratorBase which can be subclassed to add new ODE solvers (see documentation in ode.py). The only method in this hierarchy at the time of the current writing is the VODE integrator from Netlib.

Let us implement the oscillator code from Chapter 2.3 in SciPy. The 2nd-order ODE must be written as a first-order system

y ̇ 0 = y 1 , ( 4 . 8 ) y ̇ 1 = ( A s i n ( ω t ) − b y 1 − c f ( y 0 ) ) / m ( 4 . 9 )

We have here used (y0, y1) as unknowns rather than the more standard math- ematical notation (y1,y2), because we in the code will work with lists or NumPy arrays being indexed from 0.

The following class does the job:

class Oscillator:

"""Implementation of the oscillator code using SciPy.""" def \_\_init\_\_(self, \*\*kwargs):

"""Initialize parameters from keyword arguments.""" self.p = {’m’: 1.0, ’b’: 0.7, ’c’: 5.0, ’func’: ’y’,

’A’: 5.0, ’w’: 2\*pi, ’y0’: 0.2,

’tstop’: 30.0, ’dt’: 0.05} self.p.update(kwargs)

def scan(self):

"""

Read parameters from standard input in the same sequence as the F77 oscillator code.

"""

for name in ’m’, ’b’, ’c’, ’func’, ’A’, ’w’, \

’y0’, ’tstop’, ’dt’:

if name == ’func’: # expect string

self.p[’func’] = sys.stdin.readline().strip() else:

self.p[name] = float(sys.stdin.readline())

def solve(self):

"""Solve ODE system."""

# mapping: name of f(y) to Python function for f(y): self.\_fy = {’y’: lambda y: y, ’siny’: lambda y: sin(y),

’y3’: lambda y: y - y\*\*3/6.0} # set initial conditions:

self.y0 = [self.p[’y0’], 0.0]

# call SciPy solver:

from scitools.numpyutils import seq

self.t = seq(0, self.p[’tstop’], self.p[’dt’])

from scipy.integrate import odeint self.yvec = odeint(self.f, self.y0,

self.y = self.yvec[:,0] # y(t)

# write t and y(t) to sim.dat file: f = open(’sim.dat’, ’w’)

for y, t in zip(self.y, self.t):

f.write(’%g %g\n’ % (t, y)) f.close()

def f(self, y, t):

"""Right-hand side of 1st-order ODE

A, w, b, c, m = [p[k] for k in ’A’,

f = self.\_fy[self.p[’func’]]

return [y[1], (A\*cos(w\*t) - b\*y[1] - c\*f(y[0]))/m]

The code should be straightforward, perhaps with the exception of self.\_fy. This dictionary is introduced as a mapping between the name of the spring function f(y) and the corresponding Python function. The details of the arguments and return values of odeint can be obtained from the doc string (just type help(odeint) inside a Python shell).

Testing class Oscillator against the 2nd-order Runge-Kutta integrator implemented in the oscillator program can be done as follows:

def test\_Oscillator(dt=0.05): s = Oscillator(m=5, dt=dt) t1 = os.times()

s.solve()

t2 = os.times()

print ’CPU time of odeint:’, t2[0]-t1[0] + t2[1]-t1[1]

# compare with the oscillator program:

cmd = ’./simviz1.py -noscreenplot -case tmp1’

for option in s.p: # construct command-line options

cmd += ’ -’+option + ’ ’ + str(s.p[option]) t3 = os.times()

os.system(cmd)

t4 = os.times()

print ’CPU time of oscillator:’, t4[2]-t3[2] + t4[3]-t3[3]

# plot:

from scitools.filetable import readfile

t, y = readfile(os.path.join(’tmp1’,’sim.dat’))

from scitools.easyviz import \*

plot(t, y, ’r-’, s.t, s.y, ’b-’, legend=(’RK2’, ’LSODE’)) hardcopy(’tmp.ps’)

The CPU measurements show that LSODA and oscillator are about equally fast when the difference in solutions is visually negligible (see Figure 4.4). Note that LSODA probably applies a different time step internally than what we specify. Information on the numerical details of the integration can be obtained by setting a parameter full\_output:

self.yvec, self.info = odeint(self.f, self.y0, self.t, full\_output=True)

The self.info dictionary is a huge collection of data. From the other result parameter, the array self.info[’hu’], we can extract the time step sizes actually used inside the integrator. For Δt = 0.01 the time step varied from 0.00178 to 0.043. This shows that LSODA is capable of taking longer steps, but requires more internal computations, so the overall work becomes roughly equivalent to a constant step-size 2nd-order Runge-Kutta algorithm for this particular test case.

Fortunately, these code segments show how compact and convenient nu- merical computing can be in Python. In this ODE example the performance is optimal too, so we definitely face an environment based on “the best of all worlds”.

Fig. 4.4. Comparison of the 2nd-order Runge-Kutta method in oscillator and the LSODA Fortran routine (from SciPy) for Δt = 0.05 (left figure) and Δt = 0.01 (right figure).

Random Variables and Statistics. SciPy has a module stats, which offers lots of functions for drawing random numbers from a variety of distributions and computing empirical statistics. An overview is provided by info(stats), while more detailed information can be gained by running info on individual functions. The stats module also imports the Python interface RPy to the statistical computing environment R (if R and RPy are installed) and thereby allows Python data to be analyzed by the very rich functionality in R.

Linear Algebra. SciPy extends the linear algebra functionality of NumPy significantly through its linalg module. The SciPy tutorial lists the syntax for finding the determinant of a matrix, solving linear systems, computing the in- verse and the pseudo-inverse of a matrix, performing linear least squares com- putations, decomposition of matrices (Cholesky, QR, Schur), finding eigen- values and eigenvectors, calculating the singular value decomposition, and computing norms (check in particular the definitions of the norms - they may be different from what you intuitively assume). The functions in the linalg module call up LAPACK and ATLAS (if SciPy is built with these packages) and therefore provides very efficient implementation and tuning of the linear algebra algorithms.

Optimization and Root Finding. SciPy’s optimize module interfaces the well-known Fortran package MINPACK from Netlib [25] for optimization problems. MINPACK offers minimization and nonlinear least squares algo- rithms with and without gradient information. The optimize module also has routines for simulated annealing and for finding zeros of functions. The tutorial contains several examples to get started.

Interpolation. The interpolate module offers linear interpolation of one- dimensional data, plus an interface to the classical Fortran package FIT- PACK from Netlib [25] for spline interpolation of one- and two-dimensional data. There is also a signal processing toolbox. The tutorial contains several examples on spline computations and filtering.

4.4.3 The Python–Matlab Interface

A Python module pymat makes it possible to send NumPy arrays directly to Matlab and perform computations or visualizations in Matlab. The module is simple to use as there are only five functions to be aware of:

– open for opening a Matlab session,

– close for closing the session,

– eval for evaluating a Matlab command,

– put for sending a matrix to Matlab, and

– get for extracting a matrix from the Matlab session.

Here is a simple example, where we create x coordinates in Python and let Matlab compute y = sin(x) and plot the (x, y) points:

import pymat

x = linspace(0, 4\*math.pi, 401)

m = pymat.open()

pymat.put(m, ’x’, x);

pymat.eval(m, ’y = sin(x)’)

pymat.eval(m, ’plot(x,y)’)

y = pymat.get(m, ’y’) # get values from Matlab

import time; time.sleep(4) # wait 4s before killing the plot... pymat.close(m) # Matlab terminates

There is also a module mlabwrap (see link in doc.html) which makes all Matlab commands directly available in Python.

4.4.4 Symbolic Computing in Python

There are several useful packages for symbolic computing in Python. The most comprehensive, SAGE (see link from doc.html), is a complete environ- ment for symbolic and numerical computing, using an extension of Python as interface and programming language. The SAGE package contains a lot of Python packages and interfaces to many large, high-quality, mathemati- cal software systems. For example, SAGE is packed with NumPy and SciPy, and SAGE allows you to use Python to access Magma, Maple, Mathemat- ica, MATLAB, and MuPAD, and the free programs Axiom, GAP, GP/PARI, Macaulay2, Maxima, Octave, and Singular. A very nice feature is the abil- ity to create notebooks combining code, graphics, and mathematical type- setting in reports. SAGE has a wide range of mathematical objects (rings, fields, groups, etc.) for supporting research in pure mathematics. Although the symbolic computing support is very powerful and versatile in SAGE, the package aims at mathematicians and may therefore appear as considerably more complicated to understand and use than the tools mentioned below. We refer to the SAGE tutorial for an introduction to the package. SAGE is usually simple to install and therefore constitutes a smart way of getting many Python packages installed on your computer.

Swiginac (see link in doc.html) is a SWIG-based Python interface to the very efficient GiNaC C++ library for symbolic computing. That is, to use Swiginac you need to install GiNaC. Pyginac (see link in doc.html) is an alternative to Swiginac, which applies Boost.Python to interface the GiNaC library. This package is at the time of this writing in an alpha state. Another interesting package under very active development is SymPy (see link in doc.html), which is written in pure Python and therefore trivial to install. SymPy is also included in the SAGE distribution. Below we illustrate the simple use of SymPy and Swiginac.

SymPy. Contrary to common symbolic computing systems such as Maple and Mathematica, mathematical symbols must in SymPy be declared as

Symbol(’x’), symbol(’y’), etc. Then, mathematical expressions remain sym- bolic expressions. Here is a sample session:

>>> from sympy import \* >>> x = Symbol(’x’)

>>> f = cos(acos(x)) >>> f

cos(acos(x))

>>> sin(x).series(x, 4)

x - 1/6\*x\*\*3 + O(x\*\*4)

>>> dcos = diff(cos(2\*x), x)

>>> dcos

-2\*sin(2\*x)

>>> dcos.subs(x, pi).evalf() # x=pi, float evaluation 0

>>> I = integrate(log(x), x)

>>> print I

-x + x\*log(x)

The SymPy tutorial, reached from the SymPy homepage, has many more examples.

Using the StringFunction type developed in Chapter 12.2.1, one can eas- ily turn expressions from SymPy into ordinary Python functions which are as fast as if the string expressions had been hardcoded in the normal way we write Python functions. Let us demonstrate how we can use SymPy to differentiate

􏰎 􏰃x−m􏰄2􏰏 f(x;t,m,σ,A,a,ω) = Aexp − 2σ e−at sin(2πωx)

with respect to x twice and turn the symbolic formula into a fast Python function:

def make\_symbols(\*args):

return [Symbol(s) for s in args]

a, A, omega, sigma, m, t = \

make\_symbols(’a’, ’A’, ’omega’, ’sigma’, ’m’, ’t’)

f = A\*exp(-((x-m)/(2\*sigma))\*\*2)\*exp(-a\*t)\*sin(2\*pi\*omega\*x) prms = {’A’: 1, ’a’: 0.1, ’m’: 1, ’sigma’: 1,

’omega’: 1, ’t’: 0.2}

ddf\_formula = diff(f, x, 2)

ddf = StringFunction(ddf\_formula, \*\*prms) print ddf\_formula

x = 0.1

print ’\nddf(x=%g) = %g’ % (x, ddf(x))

The output (split manually into several lines) becomes

-1/2\*A\*sigma\*\*(-2)\*exp(-a\*t - 1/4\*sigma\*\*(-2)\*(x - m)\*\*2)\*\ sin(2\*pi\*omega\*x) - 4\*A\*pi\*\*2\*omega\*\*2\*exp(-a\*t - 1/4\*\

sigma\*\*(-2)\*(x - m)\*\*2)\*sin(2\*pi\*omega\*x) + (1/16)\*A\*\ sigma\*\*(-4)\*(-2\*m + 2\*x)\*\*2\*exp(-a\*t - 1/4\*sigma\*\*(-2)\*\ (x - m)\*\*2)\*sin(2\*pi\*omega\*x) - pi\*A\*omega\*sigma\*\*(-2)\*\ (-2\*m + 2\*x)\*exp(-a\*t - 1/4\*sigma\*\*(-2)\*(x - m)\*\*2)\*\ cos(2\*pi\*omega\*x)

ddf(x=0.1) = -18.8372

Swiginac. Both SAGE and SymPy have seemingly borrowed naming con- ventions from GiNaC and Swiginac, so the syntax differences between the three packages are small. Here is a sample session:

>>> from swiginac import \* >>> x = symbol(’x’)

>>> cos(acos(x))

x

>>> series(sin(x), x==0,4) # 0th to 4th term 1\*x+(-1/6)\*x\*\*3+Order(x\*\*4)

>>> dcos = diff(cos(2\*x), x)

>>> dcos

-2\*sin(2\*x)

>>> dcos.subs(x==Pi).evalf() # x=pi, float evaluation 0

>>> # integrate log(x) from x=1 to x=2:

>>> I = integrate(x, 1, 2, log(x))

>>> I.evalf()

0.38629436097734410374

Regarding the last integration example, GiNaC can only integrate polynomi- als symbolically, so 􏰊 2 ln x dx is here integrated numerically. We refer to the

1

Swiginac tutorial for more examples.

4.4.5 Some Useful Python Modules

Below is a list of some modules and packages for numerical computing with Python. A more complete list of available modules can be obtained from either the “Math” and “Graphics” sections of The Vaults of Parnassus or the “Scientific/Engineering” section of the PyPI page. Both Vaults of Parnassus and PyPI may be reached from the doc.html webpage.

* – Biggles: Curve plotting based on GNU plotutils.
* – CAGE: A fairly generic and complete cellular automata engine.
* – crng, rv: A collection of high-quality random number generators imple- mented in C.
* – DISLIN: Curve and surface plotting.
* – disipyl: Object-oriented interface to DISLIN.
* – ELLIPT2D: 2D finite element solver for elliptic equations.
* – FIAT: A new way of evaluating finite element basis functions.

– FiPy: tools for finite volume programming.

– fraction.py: Fraction arithmetics.

– Gato: Visualization of algorithms on graph structures.

– GDChart: Simple curve plotting and bar charts.

– gdmodule: Interface to the GD graphics drawing library.

– GGobi: Visualization of high-dimensional data.

– Gimp-Python: Tools for writing GIMP plug-ins in Python.

– GMPY: General Multiprecision PYthon module.

– pygrace.py: Interface to the Grace curveplotting program.

– pyIDL.py: Interface to the IDL system.

– Matplotlib: High-quality curve plotting with Matlab-like syntax.

– MatPy: Matlab/Octave-style expressions for matrix computations.

– MayaVi: Simple-to-use 3D visualization toolkit based on Vtk.

– Mlabwrap: Interface to all Matlab commands.

– MMTK: Molecular simulation toolkit.

– NURBS: Non-uniform rational B-splines.

– PIL: Image processing library.

– Pivy: Interface to the Coin (OpenInventor) 3D graphics library.

– pyacad: Combination of Python and Autocad.

– pycdf: Flexible reading of netCDF files.

– PyGlut: Interface to the OpenGL Utility Toolkit (GLUT).

– PyOpenGL: Interface to OpenGL.

– PyePiX: Interface to ePix for creating LATEX graphics.

– Pygame: Modules for multimedia, games, and visualization.

– PyGeo: Visualization of 3D dynamic geometries.

– PyGiNaC: Interface to the GiNaC C++ library for symbolic computing. – PyLab: Matlab compatible commands for computing and plotting.

– PYML: Interface to Mathematica.

– PyMOL: Molecular modeling toolkit.

– Py-OpenDX: Interface to the OpenDX data visualization system.

– PyQwt: Curve plotting widget a la BLT for use with PyQt.

– Pyscript: Programming of high-quality PostScript graphics.

– Pysparse: Sparse matrices and solvers with Python interface.

– PySPG: Run another code with varying input parameters.

* – Python Frame Buffer: Simple-to-use interactive drawing.
* – PythonPlot: Tkinter-based curve plotting program.
* – PyTables: Interface to HDF5 data storage tools.
* – PyX: TEX-like Python interface to PostScript drawing/plotting.
* – RPy: Interface to the R (S-PLUS) statistical computing environment.
* – Signaltools: Signal processing functionality a la Matlab.
* – SimPy: Discrete event simulation.
* – Unum: Unit conversions and calculations.
* – Uncertainties: Arithmetics for numbers with errors.
* – VPython: easy-to-use animation of 3D objects.
* – ZOE: Simple OpenGL based graphics engine.