A\_Primer\_Scientific\_Programming\_Python\_5E\_Hans\_c05

Array Computing and Curve Plotting

A list object is handy for storing tabular data, such as a sequence of objects or a table

of objects. An array is very similar to a list, but less flexible and computationally

much more efficient. When using the computer to perform mathematical calcula-

tions, we often end up with a huge amount of numbers and associated arithmetic

operations. Storing numbers in lists may in such contexts lead to slow programs,

while arrays can make the programs run much faster. This is crucial for many

advanced applications of mathematics in industry and science, where computer pro-

grams may run for hours and days, or even weeks. Any clever idea that reduces the

execution time by some factor is therefore paramount.

However, one can argue that programmers of mathematical software have tradi-

tionally paid too much attention to efficiency and “clever” program constructs. The

resulting software often becomes very hard to maintain and extend. In this book

we advocate a focus on clear, well-designed, and easy-to-understand programs that

work correctly. Thereafter, one can start thinking about optimization for speed.

Fortunately, arrays contribute to clear code, correctness and speed – all at once.

This chapter gives an introduction to arrays: how they are created and what they

can be used for. Array computing usually ends up with a lot of numbers. It may be

very hard to understand what these numbers mean by just looking at them. Since

the human is a visual animal, a good way to understand numbers is to visualize

them. In this chapter we concentrate on visualizing curves that reflect functions of

one variable; i.e., curves of the form y D f .x/. A synonym for curve is graph,

and the image of curves on the screen is often called a plot. We will use arrays to

store the information about points along the curve. In a nutshell, array computing

demands visualization and visualization demands arrays.

All program examples in this chapter can be found as files in the folder

src/plot1.

1

http://tinyurl.com/pwyasaa/plot

© Springer-Verlag Berlin Heidelberg 2016

H.P. Langtangen, A Primer on Scientific Programming with Python,

Texts in Computational Science and Engineering 6, DOI 10.1007/978-3-662-49887-3\_5

227228

5 Array Computing and Curve Plotting

5.1 Vectors

This section gives a brief introduction to the vector concept, assuming that you

have heard about vectors in the plane and maybe vectors in space before. This

background will be valuable when we start to work with arrays and curve plotting.

5.1.1 The Vector Concept

Some mathematical quantities are associated with a set of numbers. One example is

a point in the plane, where we need two coordinates (real numbers) to describe the

point mathematically. Naming the two coordinates of a particular point as x and y,

it is common to use the notation .x; y/ for the point. That is, we group the numbers

inside parentheses. Instead of symbols we might use the numbers directly: .0; 0/

and .1:5; 2:35/ are also examples of coordinates in the plane.

A point in three-dimensional space has three coordinates, which we may name

x1 , x2 , and x3 . The common notation groups the numbers inside parentheses:

.x1 ; x2 ; x3 /. Alternatively, we may use the symbols x, y, and z, and write the

point as .x; y; z/, or numbers can be used instead of symbols.

From high school you may have a memory of solving two equations with

two unknowns. At the university you will soon meet problems that are formu-

lated as n equations with n unknowns. The solution of such problems contains

n numbers that we can collect inside parentheses and number from 1 to n:

.x1 ; x2 ; x3 ; : : : ; xn1 ; xn /.

Quantities such as .x; y/, .x; y; z/, or .x1 ; : : : ; xn / are known as vectors in math-

ematics. A visual representation of a vector is an arrow that goes from the origin to

a point. For example, the vector .x; y/ is an arrow that goes from .0; 0/ to the point

with coordinates .x; y/ in the plane. Similarly, .x; y; z/ is an arrow from .0; 0; 0/

to the point .x; y; z/ in three-dimensional space.

Mathematicians found it convenient to introduce spaces with higher dimension

than three, because when we have a solution of n equations collected in a vector

.x1 ; : : : ; xn /, we may think of this vector as a point in a space with dimension n, or

equivalently, an arrow that goes from the origin .0; : : : ; 0/ in n-dimensional space

to the point .x1 ; : : : ; xn /. Figure 5.1 illustrates a vector as an arrow, either starting

at the origin, or at any other point. Two arrows/vectors that have the same direction

and the same length are mathematically equivalent.

We say that .x1 ; : : : ; xn / is an n-vector or a vector with n components. Each

of the numbers x1 , x2 , : : : is a component or an element. We refer to the first

component (or element), the second component (or element), and so forth.

A Python program may use a list or tuple to represent a vector:

v1 = [x, y]

# list of variables

v2 = (-1, 2)

# tuple of numbers

v3 = (x1, x2, x3)

# tuple of variables

from math import exp

v4 = [exp(-i\*0.1) for i in range(150)]5.1 Vectors

229

4

vector (2,3)

vector (2,3)

3

2

1

0

–1

–1

0

1

2

3

4

Fig. 5.1 A vector .2; 3/ visualized in the standard way as an arrow from the origin to the point

.2; 3/, and mathematically equivalently, as an arrow from .1; 12 / (or any point .a; b/) to .3; 3 12 / (or

.a C 2; b C 3/)

While v1 and v2 are vectors in the plane and v3 is a vector in three-dimensional

space, v4 is a vector in a 150-dimensional space, consisting of 150 values of the

exponential function. Since Python lists and tuples have 0 as the first index, we

may also in mathematics write the vector .x1 ; x2 / as .x0 ; x1 /. This is not at all

common in mathematics, but makes the distance from a mathematical description

of a problem to its solution in Python shorter.

It is impossible to visually demonstrate how a space with 150 dimensions looks

like. Going from the plane to three-dimensional space gives a rough feeling of what

it means to add a dimension, but if we forget about the idea of a visual perception

of space, the mathematics is very simple: going from a 4-dimensional vector to

a 5-dimensional vector is just as easy as adding an element to a list of symbols or

numbers.

5.1.2

Mathematical Operations on Vectors

Since vectors can be viewed as arrows having a length and a direction, vectors are

extremely useful in geometry and physics. The velocity of a car has a magnitude

and a direction, so has the acceleration, and the position of a point in the car is also

a vector. An edge of a triangle can be viewed as a line (arrow) with a direction and

length.

In geometric and physical applications of vectors, mathematical operations on

vectors are important. We shall exemplify some of the most important operations

on vectors below. The goal is not to teach computations with vectors, but more

to illustrate that such computations are defined by mathematical rules. Given two

vectors, .u1 ; u2 / and .v1 ; v2 /, we can add these vectors according to the rule:

.u1 ; u2 / C .v1 ; v2 / D .u1 C v1 ; u2 C v2 / :

(5.1)230

5 Array Computing and Curve Plotting

We can also subtract two vectors using a similar rule:

.u1 ; u2 / .v1 ; v2 / D .u1 v1 ; u2 v2 / :

(5.2)

A vector can be multiplied by a number. This number, called a below, is usually

denoted as a scalar:

a .v1 ; v2 / D .av1 ; av2 / :

(5.3)

The inner product, also called dot product, or scalar product, of two vectors is

a number:

.u1 ; u2 / .v1 ; v2 / D u1 v1 C u2 v2 :

(5.4)

(From high school mathematics and physics you might recall that the inner or dot

product also can be expressed as the product of the lengths of the two vectors mul-

tiplied by the cosine of the angle between them, but we will not make use of that

formula. There is also a cross product defined for 2-vectors or 3-vectors, but we do

not list the cross product formula here.)

The length of a vector is defined by

jj.v1 ; v2 /jj D

p

.v1 ; v2 / .v1 ; v2 / D

q

v12 C v22 :

(5.5)

The same mathematical operations apply to n-dimensional vectors as well. In-

stead of counting indices from 1, as we usually do in mathematics, we now count

from 0, as in Python. The addition and subtraction of two vectors with n compo-

nents (or elements) read

.u0 ; : : : ; un1 / C .v0 ; : : : ; vn1 / D .u0 C v0 ; : : : ; un1 C vn1 /;

.u0 ; : : : ; un1 / .v0 ; : : : ; vn1 / D .u0 v0 ; : : : ; un1 vn1 / :

(5.6)

(5.7)

Multiplication of a scalar a and a vector .v0 ; : : : ; vn1 / equals

.av0 ; : : : ; avn1 / :

(5.8)

The inner or dot product of two n-vectors is defined as

.u0 ; : : : ; un1 / .v0 ; : : : ; vn1 / D u0 v0 C C un1 vn1 D

n1

X

uj vj :

(5.9)

j D0

Finally, the length jjvjj of an n-vector v D .v0 ; : : : ; vn1 / is

p

1

2

2

.v0 ; : : : ; vn1 / .v0 ; : : : ; vn1 / D v02 C v12 C C vn1

0

1 12

n1

X

D@

vj2 A :

j D0

(5.10)5.1 Vectors

231

5.1.3 Vector Arithmetics and Vector Functions

In addition to the operations on vectors in Sect. 5.1.2, which you might recall from

high school mathematics, we can define other operations on vectors. This is very

useful for speeding up programs. Unfortunately, the forthcoming vector operations

are hardly treated in textbooks on mathematics, yet these operations play a signif-

icant role in mathematical software, especially in computing environment such as

MATLAB, Octave, Python, and R.

Applying a mathematical function of one variable, f .x/, to a vector is defined

as a vector where f is applied to each element. Let v D .v0 ; : : : ; vn1 / be a vector.

Then

f .v/ D .f .v0 /; : : : ; f .vn1 // :

For example, the sine of v is

sin.v/ D .sin.v0 /; : : : ; sin.vn1 // :

It follows that squaring a vector, or the more general operation of raising the vector

to a power, can be defined as applying the operation to each element:

b

v b D .v0b ; : : : ; vn1

/:

Another operation between two vectors that arises in computer programming of

mathematics is the “asterisk” multiplication, defined as

u v D .u0 v0 ; u1 v1 ; : : : ; un1 vn1 / :

(5.11)

Adding a scalar to a vector or array can be defined as adding the scalar to each

component. If a is a scalar and v a vector, we have

a C v D .a C v0 ; : : : ; a C vn1 / :

A compound vector expression may look like

v 2 cos.v/ e v C 2 :

(5.12)

How do we calculate this expression? We use the normal rules of mathematics,

working our way, term by term, from left to right, paying attention to the fact that

powers are evaluated before multiplications and divisions, which are evaluated prior

to addition and subtraction. First we calculate v 2 , which results in a vector we may

call u. Then we calculate cos.v/ and call the result p. Then we multiply u p to

get a vector which we may call w. The next step is to evaluate e v , call the result q,

followed by the multiplication w q, whose result is stored as r. Then we add r C 2

to get the final result. It might be more convenient to list these operations after each

other:

u D v2

p D cos.v/

w Dup232

5 Array Computing and Curve Plotting

q D ev

r Dwq

s Dr C2

Writing out the vectors u, w, p, q, and r in terms of a general vector v D

.v0 ; : : : ; vn1 / (do it!) shows that the result of the expression (5.12) is the vector

2

.v02 cos.v0 /e v0 C 2; : : : ; vn1

cos.vn1 /e vn1 C 2/ :

That is, component no. i in the result vector equals the number arising from apply-

ing the formula (5.12) to vi , where the \* multiplication is ordinary multiplication

between two numbers.

We can, alternatively, introduce the function

f .x/ D x 2 cos.x/e x C 2

and use the result that f .v/ means applying f to each element in v. The result is

the same as in the vector expression (5.12).

In Python programming it is important for speed (and convenience too) that

we can apply functions of one variable, like f .x/, to vectors. What this means

mathematically is something we have tried to explain in this subsection. Doing Ex-

ercises 5.5 and 5.6 may help to grasp the ideas of vector computing, and with more

programming experience you will hopefully discover that vector computing is very

useful. It is not necessary to have a thorough understanding of vector computing in

order to proceed with the next sections.

Arrays are used to represent vectors in a program, but one can do more with

arrays than with vectors. Until Sect. 5.8 it suffices to think of arrays as the same as

vectors in a program.

5.2 Arrays in Python Programs

This section introduces array programming in Python, but first we create some lists

and show how arrays differ from lists.

5.2.1

Using Lists for Collecting Function Data

Suppose we have a function f .x/ and want to evaluate this function at a number of

x points x0 ; x1 ; : : : ; xn1 . We could collect the n pairs .xi ; f .xi // in a list, or we

could collect all the xi values, for i D 0; : : : ; n 1, in a list and all the associated

f .xi / values in another list. The following interactive session demonstrates how to

create these three types of lists:

>>> def f(x):

...

return x\*\*3

...

# sample function5.2 Arrays in Python Programs

233

>>> n = 5

# no of points along the x axis

>>> dx = 1.0/(n-1)

# spacing between x points in [0,1]

>>> xlist = [i\*dx for i in range(n)]

>>> ylist = [f(x) for x in xlist]

>>> pairs = [[x, y] for x, y in zip(xlist, ylist)]

Here we have used list comprehensions for achieving compact code. Make sure that

you understand what is going on in these list comprehensions (if not, try to write

the same code using standard for loops and appending new list elements in each

pass of the loops).

The list elements consist of objects of the same type: any element in pairs is

a list of two float objects, while any element in xlist or ylist is a float. Lists

are more flexible than that, because an element can be an object of any type, e.g.,

mylist = [2, 6.0, ’tmp.pdf’, [0,1]]

Here mylist holds an int, a float, a string, and a list. This combination of di-

verse object types makes up what is known as heterogeneous lists. We can also

easily remove elements from a list or add new elements anywhere in the list. This

flexibility of lists is in general convenient to have as a programmer, but in cases

where the elements are of the same type and the number of elements is fixed, arrays

can be used instead. The benefits of arrays are faster computations, less memory

demands, and extensive support for mathematical operations on the data. Because

of greater efficiency and mathematical convenience, arrays will be used to a large

extent in this book. The great use of arrays is also prominent in other program-

ming environments such as MATLAB, Octave, and R, for instance. Lists will be

our choice instead of arrays when we need the flexibility of adding or removing

elements or when the elements may be of different object types.

5.2.2

Basics of Numerical Python Arrays

An array object can be viewed as a variant of a list, but with the following assump-

tions and features:

All elements must be of the same type, preferably integer, real, or complex num-

bers, for efficient numerical computing and storage.

The number of elements must be known when the array is created.

Arrays are not part of standard Python – one needs an additional package called

Numerical Python, often abbreviated as NumPy. The Python name of the pack-

age, to be used in import statements, is numpy.

With numpy, a wide range of mathematical operations can be done directly on

complete arrays, thereby removing the need for loops over array elements. This

is commonly called vectorization

Arrays with one index are often called vectors. Arrays with two indices are used

as an efficient data structure for tables, instead of lists of lists. Arrays can also

have three or more indices.234

5 Array Computing and Curve Plotting

We have two remarks to the above list. First, there is actually an object type called

array in standard Python, but this data type is not so efficient for mathematical

computations, and we will not use it in this book. Second, the number of elements

in an array can be changed, but at a substantial computational cost.

The following text lists some important functionality of NumPy arrays. A more

comprehensive treatment is found in the excellent NumPy Tutorial, NumPy User

Guide, NumPy Reference, Guide to NumPy, and NumPy for MATLAB Users, all

accessible at scipy.org2.

The standard import statement for Numerical Python reads

import numpy as np

To convert a list r to an array, we use the array function from numpy:

a = np.array(r)

To create a new array of length n, filled with zeros, we write

a = np.zeros(n)

The array elements are of a type that corresponds to Python’s float type. A sec-

ond argument to np.zeros can be used to specify other element types, e.g., int.

A similar function,

a = np.zeros\_like(c)

generates an array of zeros where the length is that of the array c and the element

type is the same as those in c. Arrays with more than one index are treated in

Sect. 5.8.

Often one wants an array to have n elements with uniformly distributed values

in an interval Œp; q. The numpy function linspace creates such arrays:

a = np.linspace(p, q, n)

Array elements are accessed by square brackets as for lists: a[i]. Slices also

work as for lists, for example, a[1:-1] picks out all elements except the first and

the last, but contrary to lists, a[1:-1] is not a copy of the data in a. Hence,

b = a[1:-1]

b[2] = 0.1

will also change a[3] to 0.1. A slice a[i:j:s] picks out the elements starting

with index i and stepping s indices at the time up to, but not including, j. Omitting

i implies i=0, and omitting j implies j=n if n is the number of elements in the array.

For example, a[0:-1:2] picks out every two elements up to, but not including, the

last element, while a[::4] picks out every four elements in the whole array.

2

http://scipy.org5.2 Arrays in Python Programs

235

Remarks on importing NumPy

The statement

import numpy as np

with subsequent prefixing of all NumPy functions and variables by np, has

evolved as a standard syntax in the Python scientific computing community.

However, to make Python programs look closer to MATLAB and ease the tran-

sition to and from that language, one can do

from numpy import \*

to get rid of the prefix (this is evolved as the standard in interactive Python

shells). This author prefers mathematical functions from numpy to be written

without the prefix to make the formulas as close as possible to the mathematics.

So, f .x/ D sinh.x 1/ sin.wt/ would be coded as

from numpy import sinh, sin

def f(x):

return sinh(x-1)\*sin(w\*t)

or one may take the less recommended lazy approach from numpy import \*

and fill up the program with a lot of functions and variables from numpy.

5.2.3

Computing Coordinates and Function Values

With these basic operations at hand, we can continue the session from the previous

section and make arrays out of the lists xlist and ylist:

>>> import numpy as np

>>> x2 = np.array(xlist)

# turn list xlist into array x2

>>> y2 = np.array(ylist)

>>> x2

array([ 0. , 0.25, 0.5 , 0.75, 1. ])

>>> y2

array([ 0.

, 0.015625, 0.125

, 0.421875, 1.

])

Instead of first making a list and then converting the list to an array, we can compute

the arrays directly. The equally spaced coordinates in x2 are naturally computed by

the np.linspace function. The y2 array can be created by np.zeros, to ensure

that y2 has the right length len(x2), and then we can run a for loop to fill in all

elements in y2 with f values:

>>> n = len(xlist)

>>> x2 = np.linspace(0, 1, n)

>>> y2 = np.zeros(n)236

5 Array Computing and Curve Plotting

>>> for i in xrange(n):

...

y2[i] = f(x2[i])

...

>>> y2

array([ 0.

, 0.015625,

0.125

,

0.421875,

1.

])

Note that we here in the for loop have used xrange instead of range. The former

is faster for long loops because it avoids generating and storing a list of integers,

it just generates the values one by one. Hence, we prefer xrange over range for

loops over long arrays. In Python version 3.x, range is the same as xrange.

Creating an array of a given length is frequently referred to as allocating the

array. It simply means that a part of the computer’s memory is marked for being

occupied by this array. Mathematical computations will often fill up most of the

computer’s memory by allocating long arrays.

We can shorten the previous code by creating the y2 data in a list comprehension,

but list comprehensions produce lists, not arrays, so we need to transform the list

object to an array object:

>>> x2 = np.linspace(0, 1, n)

>>> y2 = np.array([f(xi) for xi in x2])

Nevertheless, there is a much faster way of computing y2 as the next paragraph

explains.

5.2.4

Vectorization

Loops over very long arrays may run slowly. A great advantage with arrays is that

we can get rid of the loops and apply f directly to the whole array:

>>> y2 = f(x2)

>>> y2

array([ 0.

,

0.015625,

0.125

,

0.421875,

1.

])

The magic that makes f(x2) work builds on the vector computing concepts from

Sect. 5.1.3. Instead of calling f(x2) we can equivalently write the function formula

x2\*\*3 directly.

The point is that numpy implements vector arithmetics for arrays of any dimen-

sion. Moreover, numpy provides its own versions of mathematical functions like

cos, sin, exp, log, etc., which work for array arguments and apply the mathemat-

ical function to each element. The following code, computes each array element

separately:

from math import sin, cos, exp

import numpy as np

x = np.linspace(0, 2, 201)

r = np.zeros(len(x))

for i in xrange(len(x)):

r[i] = sin(np.pi\*x[i])\*cos(x[i])\*exp(-x[i]\*\*2) + 2 + x[i]\*\*25.2 Arrays in Python Programs

237

while here is a corresponding code that operates on arrays directly:

r = np.sin(np.pi\*x)\*np.cos(x)\*np.exp(-x\*\*2) + 2 + x\*\*2

Many will prefer to see such formulas without the np prefix:

from numpy import sin, cos, exp, pi

r = sin(pi\*x)\*cos(x)\*exp(-x\*\*2) + 2 + x\*\*2

An important thing to understand is that sin from the math module is different from

the sin function provided by numpy. The former does not allow array arguments,

while the latter accepts both real numbers and arrays.

Replacing a loop like the one above, for computing r[i], by a vector/array ex-

pression like sin(x)\*cos(x)\*exp(-x\*\*2) + 2 + x\*\*2, is called vectorization.

The loop version is often referred to as scalar code. For example,

import numpy as np

import math

x = np.zeros(N); y = np.zeros(N)

dx = 2.0/(N-1) # spacing of x coordinates

for i in range(N):

x[i] = -1 + dx\*i

y[i] = math.exp(-x[i])\*x[i]

is scalar code, while the corresponding vectorized version reads

x = np.linspace(-1, 1, N)

y = np.exp(-x)\*x

We remark that list comprehensions,

x = array([-1 + dx\*i for i in range(N)])

y = array([np.exp(-xi)\*xi for xi in x])

result in scalar code because we still have explicit, slow Python for loops operating

on scalar quantities. The requirement of vectorized code is that there are no explicit

Python for loops. The loops required to compute each array element are performed

in fast C or Fortran code in the numpy package.

Most Python functions intended for a scalar argument x, like

def f(x):

return x\*\*4\*exp(-x)

automatically work for an array argument x:

x = np.linspace(-3, 3, 101)

y = f(x)238

5 Array Computing and Curve Plotting

provided that the exp function in the definition of f accepts an array argument. This

means that exp must have been imported as from numpy import \* or explicitly

as from numpy import exp. One can, of course, prefix exp as in np.exp, at the

loss of a less attractive mathematical syntax in the formula.

When a Python function f(x) works for an array argument x, we say that the

function f is vectorized. Provided that the mathematical expressions in f involve

arithmetic operations and basic mathematical functions from the math module, f

will be automatically vectorized by just importing the functions from numpy instead

of math. However, if the expressions inside f involve if tests, the code needs

a rewrite to work with arrays. Section 5.4.1 presents examples where we have to do

special actions in order to vectorize functions.

Vectorization is very important for speeding up Python programs that perform

heavy computations with arrays. Moreover, vectorization gives more compact code

that is easier to read. Vectorization is particularly important for statistical simula-

tions, as demonstrated in Chap. 8.

5.3

Curve Plotting

Visualizing a function f .x/ is done by drawing the curve y D f .x/ in an xy

coordinate system. When we use a computer to do this task, we say that we plot the

curve. Technically, we plot a curve by drawing straight lines between n points on

the curve. The more points we use, the smoother the curve appears.

Suppose we want to plot the function f .x/ for a x b. First we pick out n x

coordinates in the interval Œa; b, say we name these x0 ; x1 ; : : : ; xn1 . Then we eval-

uate yi D f .xi / for i D 0; 1; : : : ; n 1. The points .xi ; yi /, i D 0; 1; : : : ; n 1,

now lie on the curve y D f .x/. Normally, we choose the xi coordinates to be

equally spaced, i.e.,

ba

:

xi D a C ih; h D

n1

If we store the xi and yi values in two arrays x and y, we can plot the curve by

a command like plot(x,y).

Sometimes the names of the independent variable and the function differ from x

and f , but the plotting procedure is the same. Our first example of curve plotting

demonstrates this fact by involving a function of t.

5.3.1 MATLAB-Style Plotting with Matplotlib

The standard package for curve plotting in Python is Matplotlib. We first exemplify

a usage of this package that is very similar with how you plot in MATLAB as many

readers will have MATLAB knowledge of will need to operate MATLAB at some

point.

A basic plot Let us plot the curve y D t 2 exp.t 2 / for t values between 0 and

3. First we generate equally spaced coordinates for t, say 51 values (50 intervals).

Then we compute the corresponding y values at these points, before we call the

plot(t,y) command to make the curve plot. Here is the complete program:5.3 Curve Plotting

239

from numpy import \*

from matplotlib.pyplot import \*

def f(t):

return t\*\*2\*exp(-t\*\*2)

t = linspace(0, 3, 51)

y = zeros(len(t))

for i in xrange(len(t)):

y[i] = f(t[i])

# 51 points between 0 and 3

# allocate y with float elements

plot(t, y)

show()

In this program we pre-allocate the y array and fill it with values, element by ele-

ment, in a Python loop. Alternatively, we may operate on the whole t array at once,

which yields faster and shorter code:

y = f(t)

To include the plot in electronic documents, we need a hardcopy of the figure in

PDF, PNG, or another image format. The savefig function saves the plot to files

in various image formats:

savefig(’tmp1.pdf’) # produce PDF

savefig(’tmp1.png’) # produce PNG

The filename extension determines the format: .pdf for PDF and .png for PNG.

Figure 5.2 displays the resulting plot.

Fig. 5.2 A simple plot in PDF format (Matplotlib)240

5 Array Computing and Curve Plotting

Fig. 5.3 A single curve with label, title, and axis adjusted (Matplotlib)

Decorating the plot The x and y axes in curve plots should have labels, here t

and y, respectively. Also, the curve should be identified with a label, or legend as it

is often called. A title above the plot is also common. In addition, we may want to

control the extent of the axes (although most plotting programs will automatically

adjust the axes to the range of the data). All such things are easily added after the

plot command:

plot(t, y)

xlabel(’t’)

ylabel(’y’)

legend([’t^2\*exp(-t^2)’])

axis([0, 3, -0.05, 0.6])

# [tmin, tmax, ymin, ymax]

title(’My First Matplotlib Demo’)

savefig(’tmp2.pdf’)

show()

Removing the show() call prevents the plot from being shown on the screen, which

is advantageous if the program’s purpose is to make a large number of plots in PDF

or PNG format (you do not want all the plot windows to appear on the screen and

then kill all of them manually). This decorated plot is displayed in Fig. 5.3.

Plotting multiple curves A common plotting task is to compare two or more

curves, which requires multiple curves to be drawn in the same plot. Suppose we

want to plot the two functions f1 .t/ D t 2 exp.t 2 / and f2 .t/ D t 4 exp.t 2 /. We

can then just issue two plot commands, one for each function. To make the syntax

resemble MATLAB, we call hold(’on’) after the first plot command to indicate

that subsequent plot commands are to draw the curves in the first plot.

def f1(t):

return t\*\*2\*exp(-t\*\*2)5.3 Curve Plotting

241

Fig. 5.4 Two curves in the same plot (Matplotlib)

def f2(t):

return t\*\*2\*f1(t)

t = linspace(0, 3, 51)

y1 = f1(t)

y2 = f2(t)

plot(t, y1, ’r-’)

hold(’on’)

plot(t, y2, ’bo’)

xlabel(’t’)

ylabel(’y’)

legend([’t^2\*exp(-t^2)’, ’t^4\*exp(-t^2)’])

title(’Plotting two curves in the same plot’)

show()

In these plot commands, we have also specified the line type: r- means red (r)

line (-), while bo means a blue (b) circle (o) at each data point. Figure 5.4 shows the

result. The legends for each curve is specified in a list where the sequence of strings

correspond to the sequence of plot commands. Doing a hold(’off’) makes the

next plot command create a new plot.

Placing several plots in one figure We may also put plots together in a figure with

r rows and c columns of plots. The subplot(r,c,a) does this, where a is a row-

wise counter for the individual plots. Here is an example with two rows of plots,

and one plot in each row, (see Fig. 5.5):

figure() # make separate figure

subplot(2, 1, 1)

t = linspace(0, 3, 51)

y1 = f1(t)

y2 = f2(t)242

5 Array Computing and Curve Plotting

Fig. 5.5 Example on two plots in one figure (Matplotlib)

plot(t, y1, ’r-’, t, y2, ’bo’)

xlabel(’t’)

ylabel(’y’)

axis([t[0], t[-1], min(y2)-0.05, max(y2)+0.5])

legend([’t^2\*exp(-t^2)’, ’t^4\*exp(-t^2)’])

title(’Top figure’)

subplot(2, 1, 2)

t3 = t[::4]

y3 = f2(t3)

plot(t, y1, ’b-’, t3, y3, ’ys’)

xlabel(’t’)

ylabel(’y’)

axis([0, 4, -0.2, 0.6])

legend([’t^2\*exp(-t^2)’, ’t^4\*exp(-t^2)’])

savefig(’tmp4.pdf’)

show()

The figure() call creates a new plot window on the screen.

All of the examples above on plotting with Matplotlib are collected in the file

mpl\_pylab\_examples.py.5.3 Curve Plotting

5.3.2

243

Matplotlib; Pyplot Prefix

The Matplotlib developers do not promote the plotting style we exemplified above.

Instead, they recommend to prefix plotting commands by the matplotlib.pyplot

module and also prefix array computing commands to demonstrate that they come

from Numerical Python:

import numpy as np

import matplotlib.pyplot as plt

The plot in Fig. 5.3 can typically be obtained by prefixing the previously shown

plotting commands with plt:

plt.plot(t, y)

plt.legend([’t^2\*exp(-t^2)’])

plt.xlabel(’t’)

plt.ylabel(’y’)

plt.axis([0, 3, -0.05, 0.6])

# [tmin, tmax, ymin, ymax]

plt.title(’My First Matplotlib Demo’)

plt.show()

plt.savefig(’tmp2.pdf’) # produce PDF

Instead of giving plot data and legends separately, it is more common to write

plt.plot(t, y, label=’t^2\*exp(-t^2)’)

However, in this book we shall stick to the legend command since this makes the

transition to/from MATLAB easier.

Figure 5.4 can be produced by

def f1(t):

return t\*\*2\*np.exp(-t\*\*2)

def f2(t):

return t\*\*2\*f1(t)

t = np.linspace(0, 3, 51)

y1 = f1(t)

y2 = f2(t)

plt.plot(t, y1, ’r-’)

plt.plot(t, y2, ’bo’)

plt.xlabel(’t’)

plt.ylabel(’y’)

plt.legend([’t^2\*exp(-t^2)’, ’t^4\*exp(-t^2)’])

plt.title(’Plotting two curves in the same plot’)

plt.savefig(’tmp3.pdf’)

plt.show()

Putting multiple plots in a figure follows the same set-up with subplot as previ-

ously shown, except that commands are prefixed by plt. The complete example,

along with the codes listed above, are found in the file mpl\_pyplot\_examples.py.244

5 Array Computing and Curve Plotting

Once you have created a basic plot, there are numerous possibilities for fine-

tuning the figure, i.e., adjusting tick marks on the axis, inserting text, etc. The

Matplotlib website is full of instructive examples on what you can do with this

excellent package.

5.3.3 SciTools and Easyviz

Matplotlib has become the de facto standard for curve plotting in Python, but there

are several other alternative packages, especially if we also consider plotting of

2D/3D scalar and vector fields. Python has interfaces to many leading visualiza-

tion packages: MATLAB, Gnuplot, Grace, OpenDX, and VTK. Even basic plotting

with these packages has very different syntax, and deciding what package and syn-

tax to go with was and still is a challenge. As a response to this challenge, Easyviz

was created to provide a common uniform interface to all the mentioned visualiza-

tion packages (including Matplotlib). The syntax of this interface was made very

close to that of MATLAB, since most scientists and engineers have experience with

MATLAB or most probably will be using it in some context. (In general, the Python

syntax used in the examples in this book is constructed to ease the transition to and

from MATLAB.)

Easyviz is part of the SciTools package, which consists of a set of Python tools

building on Numerical Python, ScientificPython, the comprehensive SciPy environ-

ment, and other packages for scientific computing with Python. SciTools contains

in particular software related to the book [13] and the present text. Installation is

straightforward as described on the web page https://github.com/hplgit/scitools.

Importing SciTools and Easyviz A standard import of SciTools is

from scitools.std import \*

The advantage of this statement is that it, with a minimum of typing, imports a lot

of useful modules for numerical Python programming: Easyviz for MATLAB-

style plotting, all of numpy (from numpy import \*), all of scipy (from scipy

import \*) if available, the StringFunction tool (see Sect. 4.3.3), many mathe-

matical functions and tools in SciTools, plus commonly applied modules such as

sys, os, and math. The imported standard mathematical functions (sqrt, sin,

asin, exp, etc.) are from numpy.lib.scimath and deal transparently with real

and complex input/output (as the corresponding MATLAB functions):

>>> from scitools.std import \*

>>> a = array([-4., 4])

>>> sqrt(a)

# need complex output

array([ 0.+2.j, 2.+0.j])

>>> a = array([16., 4])

>>> sqrt(a)

# can reduce to real output

array([ 4., 2.])

The inverse trigonometric functions have different names in math and numpy,

a fact that prevents an expression written for scalars, using math names, to be5.3 Curve Plotting

245

immediately valid for arrays. Therefore, the from scitools.std import \* ac-

tion also imports the names asin, acos, and atan for the numpy or scipy names

arcsin, arccos, and arctan functions, to ease vectorization of mathematical ex-

pressions involving inverse trigonometric functions.

The downside of the “star import” from scitools.std is twofold. First, it fills

up your program or interactive session with the names of several hundred functions.

Second, when using a particular function, you do not know the package it comes

from. Both problems are solved by doing an import of the type used in Sect. 5.3.2:

import scitools.std as st

import numpy as np

All of the SciTools and Easyviz functions must then be prefixed by st. Although

the numpy functions are available through the st prefix, we recommend using the

np prefix to clearly see where functionality comes from.

Since the Easyviz syntax for plotting is very close to that of MATLAB, it is also

very close to the syntax of Matplotlib shown earlier. This will be demonstrated in

the forthcoming examples. The advantage of using Easyviz is that the underlying

plotting package, used to create the graphics and known as a backend, can trivially

be replaced by another package. If users of your Python software have not installed

a particular visualization package, the software can still be used with another al-

ternative (which might be considerably easier to install). By default, Easyviz now

employs Matplotlib for plotting. Other popular alternatives are Gnuplot and MAT-

LAB. For 2D/3D scalar and vector fields, VTK is a popular backend for Easyviz.

We shall next redo the curve plotting examples from Sect. 5.3.1 using Easyviz

syntax.

A basic plot Plotting the curve y D t 2 exp.t 2 / for t 2 Œ0; 3, using 31 equally

spaced points (30 intervals) is performed by like this:

from scitools.std import \*

def f(t):

return t\*\*2\*exp(-t\*\*2)

t = linspace(0, 3, 31)

y = f(t)

plot(t, y, ’-’)

To save the plot in a file, we use the savefig function, which takes the filename as

argument:

savefig(’tmp1.pdf’) # produce PDF

savefig(’tmp1.eps’) # produce PostScript

savefig(’tmp1.png’) # produce PNG

The filename extension determines the format: .pdf for PDF, .ps or .eps for

PostScript, and .png for PNG. A synonym for the savefig function is hardcopy.246

5 Array Computing and Curve Plotting

What if the plot window quickly disappears?

On some platforms, some backends may result in a plot that is shown in just

a fraction of a second on the screen before the plot window disappears (the Gnu-

plot backend on Windows machines and the Matplotlib backend constitute two

examples). To make the window stay on the screen, add

raw\_input(’Press the Return key to quit: ’)

at the end of the program. The plot window is killed when the program termi-

nates, and this statement postpones the termination until the user hits the Return

key.

Decorating the plot Let us plot the same curve, but now with a legend, a plot title,

labels on the axes, and specified ranges of the axes:

from scitools.std import \*

def f(t):

return t\*\*2\*exp(-t\*\*2)

t = linspace(0, 3, 31)

y = f(t)

plot(t, y, ’-’)

xlabel(’t’)

ylabel(’y’)

legend(’t^2\*exp(-t^2)’)

axis([0, 3, -0.05, 0.6])

# [tmin, tmax, ymin, ymax]

title(’My First Easyviz Demo’)

Easyviz has also introduced a more Pythonic plot command where all the plot

properties can be set at once through keyword arguments:

plot(t, y, ’-’,

xlabel=’t’,

ylabel=’y’,

legend=’t^2\*exp(-t^2)’,

axis=[0, 3, -0.05, 0.6],

title=’My First Easyviz Demo’,

savefig=’tmp1.pdf’,

show=True)

With show=False one can avoid the plot window on the screen and just make

the plot file.

Note that we in the curve legend write t square as t^2 (LATEX style) rather than

t\*\*2 (program style). Whichever form you choose is up to you, but the LATEX form

sometimes looks better in some plotting programs (Matplotlib and Gnuplot are two

examples).

Plotting multiple curves Next we want to compare the two functions f1 .t/ D

t 2 exp.t 2 / and f2 .t/ D t 4 exp.t 2 /. Writing two plot commands after each other5.3 Curve Plotting

247

makes two separate plots. To make the second curve appear together with the first

one, we need to issue a hold(’on’) call after the first plot command. All subse-

quent plot commands will then draw curves in the same plot, until hold(’off’)

is called.

from scitools.std import \*

def f1(t):

return t\*\*2\*exp(-t\*\*2)

def f2(t):

return t\*\*2\*f1(t)

t = linspace(0, 3, 51)

y1 = f1(t)

y2 = f2(t)

plot(t, y1, ’r-’)

hold(’on’)

plot(t, y2, ’b-’)

xlabel(’t’)

ylabel(’y’)

legend(’t^2\*exp(-t^2)’, ’t^4\*exp(-t^2)’)

title(’Plotting two curves in the same plot’)

savefig(’tmp3.pdf’)

The sequence of the multiple legends is such that the first legend corresponds to the

first curve, the second legend to the second curve, and so forth.

Instead of separate calls to plot and the use of hold(’on’), we can do every-

thing at once and just send several curves to plot:

plot(t, y1, ’r-’, t, y2, ’b-’, xlabel=’t’, ylabel=’y’,

legend=(’t^2\*exp(-t^2)’, ’t^4\*exp(-t^2)’),

title=’Plotting two curves in the same plot’,

savefig=’tmp3.pdf’)

Throughout this book, we very often make use of this type of compact plot

command, which also only requires an import of the form from scitools.std

import plot.

Changing backend Easyviz applies Matplotlib for plotting by default, so the re-

sulting figures so far will be similar to those of Fig. 5.2–5.4.

However, we can use other backends (plotting packages) for creating the graph-

ics. The specification of what package to use is defined in a configuration file (see

the heading Setting Parameters in the Configuration File in the Easyviz documen-

tation), or on the command line:

Terminal

Terminal> python myprog.py --SCITOOLS\_easyviz\_backend gnuplot248

5 Array Computing and Curve Plotting

Plotting two curves in the same plot

0.6

t2exp(-t2)

4

2

t exp(-t )

0.5

y

0.4

0.3

0.2

0.1

0

0

0.5

1

1.5

t

2

2.5

3

Fig. 5.6 Two curves in the same plot (Gnuplot)

Now, the plotting commands in myprog.py will make use of Gnuplot to create the

graphics, with a slightly different result than that created by Matplotlib (compare

Figs. 5.4 and 5.6). A nice feature of Gnuplot is that the line types are automatically

changed if we save a figure to file, such that the lines are easily distinguishable in

a black-and-white plot. With Matplotlib one has to carefully set the line types to

make them effective on a grayscale.

Placing several plots in one figure Finally, we redo the example from Sect. 5.3.1

where two plots are combined into one figure, using the subplot command:

figure()

subplot(2, 1, 1)

t = linspace(0, 3, 51)

y1 = f1(t)

y2 = f2(t)

plot(t, y1, ’r-’, t, y2, ’bo’, xlabel=’t’, ylabel=’y’,

legend=(’t^2\*exp(-t^2)’, ’t^4\*exp(-t^2)’),

axis=[t[0], t[-1], min(y2)-0.05, max(y2)+0.5],

title=’Top figure’)

subplot(2, 1, 2)

t3 = t[::4]

y3 = f2(t3)

plot(t, y1, ’b-’, t3, y3, ’ys’,

xlabel=’t’, ylabel=’y’,

axis=[0, 4, -0.2, 0.6],

legend=(’t^2\*exp(-t^2)’, ’t^4\*exp(-t^2)’))

savefig(’tmp4.pdf’)

Note that figure() must be used if you want a program to make different plot

windows on the screen: each figure() call creates a new, separate plot.

All of the Easyviz examples above are found in the file easyviz\_examples.py.

We remark that Easyviz is just a thin layer of code providing access to the most5.3 Curve Plotting

249

common plotting functionality for curves as well as 2D/3D scalar and vector fields.

Fine-tuning of plots, e.g., specifying tick marks on the axes, is not supported, simply

because most of the curve plots in the daily work can be made without such func-

tionality. For fine-tuning the plot with special commands, you need to grab an object

in Easyviz that communicates directly with the underlying plotting package used to

create the graphics. With this object you can issue package-specific commands and

do whatever the underlying package allows you do. This is explained in the Easyviz

manual3 , which also comes up by running pydoc scitools.easyviz. As soon as

you have digested the very basics of plotting, you are strongly recommend to read

through the curve plotting part of the Easyviz manual.

5.3.4

Making Animations

A sequence of plots can be combined into an animation on the screen and stored in

a video file. The standard procedure is to generate a series of individual plots and

to show them in sequence to obtain an animation effect. Plots store in files can be

combined to a video file.

Example The function

1=2 1

f .xI m; s/ D .2/

s

1 x m 2

exp

2

s

is known as the Gaussian function or the probability density function of the normal

(or Gaussian) distribution. This bell-shaped function is wide for large s and peak-

formed for small s, see Fig. 5.7. The function is symmetric around x D m (m D 0

in the figure). Our goal is to make an animation where we see how this function

evolves as s is decreased. In Python we implement the formula above as a function

f(x, m, s).

2

s=0.2

s=1

s=2

1.8

1.6

1.4

1.2

1

0.8

0.6

0.4

0.2

0

-6

-4

-2

0

2

Fig. 5.7 Different shapes of a Gaussian function

3

https://scitools.googlecode.com/hg/doc/easyviz/easyviz.html

4

6250

5 Array Computing and Curve Plotting

The animation is created by varying s in a loop and for each s issue a plot

command. A moving curve is then visible on the screen. One can also make a video

that can be played as any other computer movie using a standard movie player. To

this end, each plot is saved to a file, and all the files are combined together using

some suitable tool to be explained later. Before going into programming detail there

is one key point to emphasize.

Keep the extent of axes fixed during animations!

The underlying plotting program will normally adjust the axis to the maximum

and minimum values of the curve if we do not specify the axis ranges explicitly.

For an animation such automatic axis adjustment is misleading – any axis range

must be fixed to avoid a jumping axis.

The relevant values for the y axis range in the present example is the minimum

and maximum value of f . The minimum value is zero, while the maximum value

appears for x D m and increases with decreasing s. The range of the y axis must

therefore be Œ0; f .mI m; min s/.

The function f is defined for all 1 < x < 1, but the function value is very

small already 3s away from x D m. We may therefore limit the x coordinates to

Œm 3s; m C 3s.

Animation in Easyviz We start with using Easyviz for animation since this is

almost identical to making standard static plots, and you can choose the plotting

engine you want to use, say Gunplot or Matplotlib. The Easyviz recipe for animat-

ing the Gaussian function as s goes from 2 to 0.2 looks as follows.

from scitools.std import sqrt, pi, exp, linspace, plot, movie

import time

def f(x, m, s):

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

m = 0

s\_min = 0.2

s\_max = 2

x = linspace(m -3\*s\_max, m + 3\*s\_max, 1000)

s\_values = linspace(s\_max, s\_min, 30)

# f is max for x=m; smaller s gives larger max value

max\_f = f(m, m, s\_min)

# Show the movie on the screen

# and make hardcopies of frames simultaneously.

counter = 0

for s in s\_values:

y = f(x, m, s)

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

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

savefig=’tmp%04d.png’ % counter)

counter += 1

#time.sleep(0.2) # can insert a pause to control movie speed5.3 Curve Plotting

251

Note that the s values are decreasing (linspace handles this automatically if

the start value is greater than the stop value). Also note that we, simply because

we think it is visually more attractive, let the y axis go from 0.1 although the

f function is always greater than zero. The complete code is found in the file

movie1.py.

Notice

It is crucial to use the single, compound plot command shown above, where

axis, labels, legends, etc., are set in the same call. Splitting up in individual calls

to plot, axis, and so forth, results in jumping curves and axis. Also, when

visualizing more than one animated curve at a time, make sure you send all data

to a single plot command.

Remark on naming plot files

For each frame (plot) in the movie we store the plot in a file, with the purpose of

combining all the files to an ordinary video file. The different files need different

names such that various methods for listing the files will list them in the correct

order. To this end, we recommend using filenames of the form tmp0001.png,

tmp0002.png, tmp0003.png, etc. The printf format 04d pads the integers with

zeros such that 1 becomes 0001, 13 becomes 0013 and so on. The expression

tmp\*.png will now expand (by an alphabetic sort) to a list of all files in proper

order.

Without the padding with zeros, i.e., names of the form tmp1.png, tmp2.png,

. . . , tmp12.png, etc., the alphabetic order will give a wrong sequence of frames

in the movie. For instance, tmp12.png will appear before tmp2.png.

Basic animation in Matplotlib Animation is Matplotib requires more than a loop

over a parameter and making a plot inside the loop. The set-up that is closest to

standard static plots is shown first, while the newer and more widely used tool

FuncAnimation is explained afterwards.

The first part of the program, where we define f, x, s\_values, and so forth, is

the same regardless of the animation technique. Therefore, we concentrate on the

graphics part here:

import matplotlib.pyplot as plt

...

# Make a first plot

plt.ion()

y = f(x, m, s\_max)

lines = plt.plot(x, y)

plt.axis([x[0], x[-1], -0.1, max\_f])

plt.xlabel(’x’)

plt.ylabel(’f’)

# Show the movie, and make hardcopies of frames simulatenously

counter = 0

for s in s\_values:

y = f(x, m, s)

lines[0].set\_ydata(y)252

5 Array Computing and Curve Plotting

plt.legend([’s=%4.2f’ % s])

plt.draw()

plt.savefig(’tmp\_%04d.png’ % counter)

counter += 1

The plt.ion() call is important, so is the first plot, where we grab the result of the

plot command, which is a list of Matplotlib’s Line2D objects. The idea is then to

update the data via lines[0].set\_ydata and show the plot via plt.draw() for

each frame. For multiple curves we must update the y data for each curve, e.g.,

lines = plot(x, y1, x, y2, x, y3)

for parameter in parameters:

y1 = ...

y2 = ...

y3 = ...

for line, y in zip(lines, [y1, y2, y3]):

line.set\_ydata(y)

plt.draw()

The file movie1\_mpl1.py contains the complete program for doing animation with

native Matplotlib syntax.

Using FuncAnimation in Matplotlib The recommended approach to animation

in Matplotlib is to use the FuncAnimation tool:

import matplotlib.pyplot as plt

from matplotlib.animation import animation

anim = animation.FuncAnimation(

fig, frame, all\_args, interval=150, init\_func=init, blit=True)

Here, fig is the plt.figure() object for the current figure, frame is a user-

defined function for plotting each frame, all\_args is a list of arguments for frame,

interval is the delay in ms between each frame, init\_func is a function called

for defining the background plot in the animation, and blit=True speeds up the

animation. For frame number i, FuncAnimation will call frame(all\_args[i]).

Hence, the user’s task is mostly to write the frame function and construct the

all\_args arguments.

After having defined m, s\_max, s\_min, s\_values, and max\_f as shown earlier,

we have to make a first plot:

fig = plt.figure()

plt.axis([x[0], x[-1], -0.1, max\_f])

lines = plt.plot([], [])

plt.xlabel(’x’)

plt.ylabel(’f’)

Notice that we save the return value of plt.plot in lines such that we can con-

veniently update the data for the curve(s) in each frame.5.3 Curve Plotting

253

The function for defining a background plot draws an empty plot in this example:

def init():

lines[0].set\_data([], [])

return lines

# empty plot

The function that defines the individual plots in the animation basically computes

y from f and updates the data of the curve:

def frame(args):

frame\_no, s, x, lines = args

y = f(x, m, s)

lines[0].set\_data(x, y)

return lines

Multiple curves can be updated as shown earlier.

We are now ready to call animation.FuncAnimation:

anim = animation.FuncAnimation(

fig, frame, all\_args, interval=150, init\_func=init, blit=True)

A common next action is to make a video file, here in the MP4 format with 5

frames per second:

anim.save(’movie1.mp4’, fps=5)

# movie in MP4 format

Finally, we must plt.show() as always to watch any plots on the screen.

The video making requires additional software on the computer, such as ffmpeg,

and can fail. One gets more control over the potentially fragile movie making pro-

cess by explicitly saving plots to file and explicitly running movie making programs

like ffmeg later. Such programs are explained in Sect. 5.3.5.

The complete code showing the basic use of FuncAnimation is available in

movie1\_FuncAnimation.py. There is also a MATLAB Animation Tutorial4 with

more basic information, plus a set of animation examples on http://matplotlib.org/

examples.

Remove old plot files!

We strongly recommend removing previously generated plot files before a new

set of files is made. Otherwise, the movie may get old and new files mixed up.

The following Python code removes all files of the form tmp\*.png:

import glob, os

for filename in glob.glob(’tmp\*.png’):

os.remove(filename)

These code lines should be inserted at the beginning of programs or functions

performing animations.

4

http://jakevdp.github.io/blog/2012/08/18/matplotlib-animation-tutorial/254

5 Array Computing and Curve Plotting

Instead of deleting the individual plotfiles, one may store all plot files in a sub-

folder and later delete the subfolder. Here is a suitable code segment:

import shutil, os

subdir = ’temp’

# subfolder name for plot files

if os.path.isdir(subdir): # does the subfolder already exist?

shutil.rmtree(subdir) # delete the whole folder

os.mkdir(subdir)

# make new subfolder

os.chdir(subdir)

# move to subfolder

# ... perform all the plotting, make movie ...

os.chdir(os.pardir)

# optional: move up to parent folder

Note that Python and many other languages use the word directory instead of folder.

Consequently, the name of functions dealing with folders have a name containing

dir for directory.

5.3.5 Making Videos

Suppose we have a set of frames in an animation, saved as plot files tmp\_\*.png.

The filenames are generated by the printf syntax ’tmp\_%04d.png’ % i, using

a frame counter i that goes from 0 to some value. The corresponding files are

then tmp\_0000.png, tmp\_0001.png, tmp\_0002.png, and so on. Several tools

can be used to create videos in common formats from the individual frames in the

plot files.

Animated GIF file The ImageMagick5 software suite contains a program convert

for making animated GIF files:

Terminal

Terminal> convert -delay 50 tmp\_\*.png movie.gif

The delay between frames, here 50, is measured in units of 1/100 s. The resulting

animated GIF file movie.gif can be viewed by another program in the ImageMag-

ick suite: animate movie.gif, but the most common way of displaying animated

GIF files is to include them in web pages. Writing the HTML code

<img src="movie.gif">

in some file with extension .html and loading this file into a web browser will play

the movie repeatedly. You may try this out online6 .

MP4, Ogg, WebM, and Flash videos The modern video formats that are best

suited for being displayed in web browsers are MP4, Ogg, WebM, and Flash. The

program ffmpeg, or the almost equivalent avconv, is a common tool to create such

movies. Creating a flash video is done by

5

6

http://www.imagemagick.org/

http://hplgit.github.io/scipro-primer/video/gaussian.html5.3 Curve Plotting

255

Terminal

Terminal> ffmpeg -i tmp\_%04d.png -r 5 -vcodec flv movie.flv

The -i option specifies the printf string that was used to make the names of the

individual plot files, -r specifies the number of frames per second, here 5, -vcodec

is the video codec for Flash, which is called flv, and the final argument is the name

of the video file. On Debian Linux systems, such as Ubuntu, you use the avconv

program instead of ffmpeg.

Other formats are created in the same way, but we need to specify the codec and

use the right extension in the video file:

Format

Flash

MP4

Webm

Ogg

Codec and filename

-vcodec flv movie.flv

-vcodec libx264 movie.mp4

-vcodec libvpx movie.webm

-vcodec libtheora movie.ogg

Video files are normally trivial to play in graphical file browser: double lick the

filename or right-click and choose a player. On Linux systems there are several

players that can be run from the command line, e.g., vlc, mplayer, gxine, and

totem.

It is easy to create the video file from a Python program since we can run any

operating system command in (e.g.) os.system:

cmd = ’convert -delay 50 tmp\_\*.png movie.gif’

os.system(cmd)

It might happen that your downloaded and installed version of ffmpeg fails to

generate videos in some of the mentioned formats. The reason is that ffmpeg de-

pends on many other packages that may be missing on your system. Getting ffmpeg

to work with the libx264 codec for making MP4 files is often challenging. On

Debian-based Linux systems, such as Ubuntu, the installation procedure at the time

of this writing goes like

Terminal

Terminal> sudo apt-get install lib-avtools libavcodec-extra-53 \

libx264-dev

5.3.6 Curve Plots in Pure Text

Sometimes it can be desirable to show a graph in pure ASCII text, e.g., as part

of a trial run of a program included in the program itself, or a graph that can be

illustrative in a doc string. For such purposes we have slightly extended a module

by Imri Goldberg (aplotter.py) and included it as a module in SciTools. Running

pydoc on scitools.aplotter describes the capabilities of this type of primitive

plotting. Here we just give an example of what it can do:256

5 Array Computing and Curve Plotting

>>> import numpy as np

>>> x = np.linspace(-2, 2, 81)

>>> y = np.exp(-0.5\*x\*\*2)\*np.cos(np.pi\*x)

>>> from scitools.aplotter import plot

>>> plot(x, y)

|

-+1

// |\

/

| \

/

|

\

/

|

\

/

|

\

/

|

\

/

|

\

/

|

\

-------\

/

|

\

---+-------\-----------------/---------+--------\-----------------/

-2

\

/

|

\

/

\

/

|

\

//

\

/

|

\

/

\

/

|

\

//

\

/

|

\

/

\

//

|

\-

//

----

-0.63

---/

5.4 Plotting Difficulties

The previous examples on plotting functions demonstrate how easy it is to make

graphs. Nevertheless, the shown techniques might easily fail to plot some functions

correctly unless we are careful. Next we address two types of difficult functions:

piecewisely defined functions and rapidly varying functions.

5.4.1 Piecewisely Defined Functions

A piecewisely defined function has different function definitions in different in-

tervals along the x axis. The resulting function, made up of pieces, may have

discontinuities in the function value or in derivatives. We have to be very care-

ful when plotting such functions, as the next two examples will show. The problem

is that the plotting mechanism draws straight lines between coordinates on the func-

tion’s curve, and these straight lines may not yield a satisfactory visualization of the

function. The first example has a discontinuity in the function itself at one point,

while the other example has a discontinuity in the derivative at three points.

Example: The Heaviside function Let us plot the Heaviside function

(

H.x/ D

0; x < 0

1; x 05.4 Plotting Difficulties

257

The most natural way to proceed is first to define the function as

def H(x):

return (0 if x < 0 else 1)

The standard plotting procedure where we define a coordinate array x and call y =

H(x) will not work for array arguments x, of reasons to be explained in Sect. 5.5.2.

However, we may use techniques from that chapter to create a function Hv(x) that

works for array arguments. Even with such a function we face difficulties with

plotting it.

Since the Heaviside function consists of two flat lines, one may think that we do

not need many points along the x axis to describe the curve. Let us try with nine

points:

x = np.linspace(-10, 10, 9)

from scitools.std import plot

plot(x, Hv(x), axis=[x[0], x[-1], -0.1, 1.1])

However, so few x points are not able to describe the jump from 0 to 1 at x D 0, as

shown by the solid line in Fig. 5.8 (left). Using more points, say 50 between 10

and 10,

x2 = np.linspace(-10, 10, 50)

plot(x, Hv(x), ’r’, x2, Hv(x2), ’b’,

legend=(’5 points’, ’50 points’),

axis=[x[0], x[-1], -0.1, 1.1])

makes the curve look better. However, the step is still not strictly vertical. More

points will improve the situation. Nevertheless, the best is to draw two flat lines

directly: from .10; 0/ to .0; 0/, then to .0; 1/ and then to .10; 1/:

plot([-10, 0, 0, 10], [0, 0, 1, 1],

axis=[x[0], x[-1], -0.1, 1.1])

The result is shown in Fig. 5.8 (right).

Some will argue that the plot of H.x/ should not contain the vertical line from

.0; 0/ to .0; 1/, but only two horizontal lines. To make such a plot, we must draw

two distinct curves, one for each horizontal line:

plot([-10,0], [0,0], ’r-’, [0,10], [1,1], ’r-’,

axis=[x[0], x[-1], -0.1, 1.1])

Observe that we must specify the same line style for both lines (curves), otherwise

they would by default get different colors on the screen or different line types in

a hardcopy of the plot. We remark, however, that discontinuous functions like H.x/

are often visualized with vertical lines at the jumps, as we do in Fig. 5.8b.258

5 Array Computing and Curve Plotting

Fig. 5.8 Plot of the Heaviside function using 9 equally spaced x points (left) and with a double

point at x D 0 (right)

Example: A hat function Let us plot the hat function N.x/, shown as the solid

line in Fig. 5.9. This function is a piecewise linear function. The implementation of

N.x/ must use if tests to locate where we are along the x axis and then evaluate

the right linear piece of N.x/. A straightforward implementation with plain if

tests does not work with array arguments x, but Sect. 5.5.3 explains how to make

a vectorized version Nv(x) that works for array arguments as well. Anyway, both

the scalar and the vectorized versions face challenges when it comes to plotting.

A first approach to plotting could be

x = np.linspace(-2, 4, 6)

plot(x, Nv(x), ’r’, axis=[x[0], x[-1], -0.1, 1.1])

1

0.8

0.6

0.4

0.2

0

-2

-1

0

1

2

3

4

Fig. 5.9 Plot of a hat function. The solid line shows the exact function, while the dashed line

arises from using inappropriate points along the x axis5.4 Plotting Difficulties

259

This results in the dashed line in Fig. 5.9. What is the problem? The problem lies in

the computation of the x vector, which does not contain the points x D 1 and x D 2

where the function makes significant changes. The result is that the hat is flattened.

Making an x vector with all critical points in the function definitions, x D 0; 1; 2,

provides the necessary remedy, either

x = np.linspace(-2, 4, 7)

or the simple

x = [-2, 0, 1, 2, 4]

Any of these x alternatives and a plot(x, Nv(x)) will result in the solid line in

Fig. 5.9, which is the correct visualization of the N.x/ function.

As in the case of the Heaviside function, it is perhaps best to drop using vec-

torized evaluations and just draw straight lines between the critical points of the

function (since the function is linear):

x = [-2, 0, 1, 2, 4]

y = [N(xi) for xi in x]

plot(x, y, ’r’, axis=[x[0], x[-1], -0.1, 1.1])

5.4.2

Rapidly Varying Functions

Let us now visualize the function f .x/ D sin.1=x/, using 10 and 1000 points:

def f(x):

return sin(1.0/x)

from scitools.std import linspace, plot

x1 = linspace(-1, 1, 10)

x2 = linspace(-1, 1, 1000)

plot(x1, f(x1), label=’%d points’ % len(x))

plot(x2, f(x2), label=’%d points’ % len(x))

The two plots are shown in Fig. 5.10. Using only 10 points gives a completely

wrong picture of this function, because the function oscillates faster and faster as we

approach the origin. With 1000 points we get an impression of these oscillations,

but the accuracy of the plot in the vicinity of the origin is still poor. A plot with

100000 points has better accuracy, in principle, but the extremely fast oscillations

near the origin just drowns in black ink (you can try it out yourself).

Another problem with the f .x/ D sin.1=x/ function is that it is easy to define

an x vector containing x D 0, such that we get division by zero. Mathematically,

the f .x/ function has a singularity at x D 0: it is difficult to define f .0/, so

one should exclude this point from the function definition and work with a domain

x 2 Œ1; [ Œ; 1, with chosen small.

The lesson learned from these examples is clear. You must investigate the func-

tion to be visualized and make sure that you use an appropriate set of x coordinates

along the curve. A relevant first step is to double the number of x coordinates260

5 Array Computing and Curve Plotting

11

0.80.8

0.60.6

0.40.4

0.20.2

00

-0.2-0.2

-0.4-0.4

-0.6-0.6

-0.8-0.8

-1

-1

-1

-0.5

0

0.5

1

-1

-0.5

0

0.5

1

Fig. 5.10 Plot of the function sin.1=x/ with 10 points (left) and 1000 points (right)

and check if this changes the plot. If not, you probably have an adequate set of x

coordinates.

5.5 More Advanced Vectorization of Functions

So far we have seen that vectorization of a Python function f(x) implementing

some mathematical function f .x/ seems trivial: f(x) works right away with an

array argument x and, in that case, returns an array where f is applied to each

element in x. When the expression for f .x/ is given in terms of a string and the

StringFunction tool is used to generate the corresponding Python function f(x),

one extra step must be performed to vectorize the Python function. This step is

explained in Sect. 5.5.1.

The described vectorization works well as long as the expression f .x/ is a math-

ematical formula without any if test. As soon as we have if tests (conditional

mathematical expressions) the vectorization becomes more challenging. Some use-

ful techniques are explained through two examples in Sects. 5.5.2 and 5.5.3. The

described techniques are considered advanced material and only necessary when

the time spent on evaluating a function at a very large set of points needs to be

significantly decreased.

5.5.1

Vectorization of StringFunction Objects

The StringFunction object from scitools.std can convert a formula, available

as a string, to a callable Python function (see Sect. 4.3.3). However, the function

cannot work with array arguments unless we explicitly tell the StringFunction

object to do so. The recipe is very simple. Say f is some StringFunction object.

To allow array arguments we are required to call f.vectorize(globals()) once:

from numpy import \*

x = linspace(0, 1, 30)

# f(x) will in general not work

f.vectorize(globals())

values = f(x)

# f works with array arguments5.5 More Advanced Vectorization of Functions

261

It is important that you import everything from numpy (or scitools.std) before

calling f.vectorize, exactly as shown.

You may take the f.vectorize call as a magic recipe. Still, some readers

want to know what problem f.vectorize solves. Inside the StringFunction

module we need to have access to mathematical functions for expressions like

sin(x)\*exp(x) to be evaluated. These mathematical functions are by default

taken from the math module and hence they do not work with array arguments.

If the user, in the main program, has imported mathematical functions that work

with array arguments, these functions are registered in a dictionary returned from

globals(). By the f.vectorize call we supply the StringFunction module

with the user’s global namespace so that the evaluation of the string expression can

make use of the mathematical functions for arrays from the user’s program. Un-

less you use np.sin(x)\*np.cos(x) etc. in the string formulas, make sure you

do a from numpy import \* so that the function names are defined without any

prefix.

Even after calling f.vectorize(globals()), a StringFunction object may

face problems with vectorization. One example is a piecewise constant function

as specified by a string expression ’1 if x > 2 else 0’. Section 5.5.2 explains

why if tests fail for arrays and what the remedies are.

5.5.2

Vectorization of the Heaviside Function

We consider the widely used Heaviside function defined by

(

H.x/ D

0; x < 0

1; x 0

The most compact way if implementing this function is

def H(x):

return (0 if x < 0 else 1)

Trying to call H(x) with an array argument x fails:

>>> def H(x): return (0 if x < 0 else 1)

...

>>> import numpy as np

>>> x = np.linspace(-10, 10, 5)

>>> x

array([-10., -5.,

0.,

5., 10.])

>>> H(x)

...

ValueError: The truth value of an array with more than

one element is ambiguous. Use a.any() or a.all()

The problem is related to the test x < 0, which results in an array of boolean values,

while the if test needs a single boolean value (essentially taking bool(x < 0)):262

5 Array Computing and Curve Plotting

>>> b = x < 0

>>> b

array([ True, True, False, False, False], dtype=bool)

>>> bool(b) # evaluate b in a boolean context

...

ValueError: The truth value of an array with more than

one element is ambiguous. Use a.any() or a.all()

>>> b.any() # True if any element in b is True

True

>>> b.all() # True if all elements in b are True

False

The any and all calls do not help us since we want to take actions element by

element depending on whether x[i] < 0 or not.

There are four ways to find a remedy to our problems with the if x < 0 test:

(i) we can write an explicit loop for computing the elements, (ii) we can use a tool

for automatically vectorize H(x), (iii) we can mix boolean and floating-point calcu-

lations, or (iv) we can manually vectorize the H(x) function. All four methods will

be illustrated next.

Loop The following function works well for arrays if we insert a simple loop over

the array elements (such that H(x) operates on scalars only):

def H\_loop(x):

r = np.zeros(len(x))

for i in xrange(len(x)):

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

return r

# Example:

x = np.linspace(-5, 5, 6)

y = H\_loop(x)

Automatic vectorization Numerical Python contains a method for automatically

vectorizing a Python function H(x) that works with scalars (pure numbers) as x

argument:

import numpy as np

H\_vec = np.vectorize(H)

The H\_vec(x) function will now work with vector/array arguments x. Unfortu-

nately, such automatically vectorized functions runs at a fairly slow speed compared

to the implementations below (see the end of Sect. 5.5.3 for specific timings).

Mixing boolean and floating-point calculations It appears that a very simple so-

lution to vectorizing the H(x) function is to implement it as

def H(x):

return x >= 05.5 More Advanced Vectorization of Functions

263

The return value is now a bool object, not an int or float as we would math-

ematically expect to be the proper type of the result. However, the bool object

works well in both scalar and vectorized operations as long as we involve the re-

turned H(x) in some arithmetic expression. The True and False values are then

interpreted as 1 and 0. Here is a demonstration:

>>> x = np.linspace(-1, 1, 5)

>>> H(x)

array([False, False, True, True, True], dtype=bool)

>>> 1\*H(x)

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

>>> H(x) - 2

array([-2, -2, -1, -1, -1])

>>>

>>> x = 0.2

# test scalar argument

>>> H(x)

True

>>> 1\*H(x)

1

>>> H(x) - 2

-1

If returning a boolean value is considered undesirable, we can turn the bool object

into the proper type by

def H(x):

r = x >= 0

if isinstance(x, (int,float)):

return int(r)

elif isinstance(x, np.ndarray):

return np.asarray(r, dtype=np.int)

Manual vectorization By manual vectorization we normally mean translating the

algorithm into a set of calls to functions in the numpy package such that no loops are

visible in the Python code. The last version of the H(x) is a manual vectorization,

but now we shall look at a more general technique when the result is not necessarily

0 or 1. In general, manual vectorization is non-trivial and requires knowledge of and

experience with the underlying library for array computations. Fortunately, there is

a simple numpy recipe for turning functions of the form

def f(x):

if condition:

r = <expression1>

else:

r = <expression2>

return r

into vectorized form:

def f\_vectorized(x):

x1 = <expression1>

x2 = <expression2>264

5 Array Computing and Curve Plotting

r = np.where(condition, x1, x2)

return r

The np.where function returns an array of the same length as condition, whose

element no. i equals x1[i] if condition[i] is True, and x2[i] otherwise. With

Python loops we can express this principle as

def my\_where(condition, x1, x2):

r = np.zeros(len(condition))

# result

for i in xrange(condition):

r[i] = x1[i] if condition[i] else x2[i]

return r

The x1 and x2 variables can be pure numbers too in the call to np.where.

In our case we can use the np.where function as follows:

def Hv(x):

return np.where(x < 0, 0.0, 1.0)

Instead of using np.where we can apply boolean indexing. The idea is that an

array a allows to be indexed by an array b of boolean values: a[b]. The result a[b]

is a new array with all the elements a[i] where b[i] is True:

>>> a

array([ 0. ,

2.5,

5. ,

>>> b = a > 5

>>> b

array([False, False, False,

>>> a[b]

array([ 7.5, 10. ])

7.5,10. ])

True,True], dtype=bool)

We can assign new values to the elements in a where b is True:

>>> a[b]

array([ 7.5, 10. ])

>>> a[b] = np.array([-10, -20], dtype=np.float)

>>> a

array([ 0. ,

2.5,

5. , -10. , -20. ])

>>> a[b] = -4

>>> a

array([ 0. , 2.5, 5. , -4. , -4. ])

To implement the Heaviside function, we start with an array of zeros and then

assign 1 to the elements where x >= 0:

def Hv(x):

r = np.zeros(len(x), dtype=np.int)

r[x >= 0] = 1

return r5.5 More Advanced Vectorization of Functions

5.5.3

265

Vectorization of a Hat Function

We now turn the attention to the hat function N.x/ defined by

8

0;

x<0

ˆ

ˆ

ˆ

< x;

0x<1

N.x/ D

ˆ

ˆ 2 x; 1 x < 2

:̂

0;

x 2

The corresponding Python implementation N(x) is

def N(x):

if x < 0:

return 0.0

elif 0 <= x < 1:

return x

elif 1 <= x < 2:

return 2 - x

elif x >= 2:

return 0.0

Unfortunately, this N(x) function does not work with array arguments x, because

the boolean expressions, like x < 0, are arrays and they cannot yield a single True

or False value for the if tests, as explained in Sect. 5.5.2.

The simplest remedy is to use np.vectorize from Sect. 5.5.2:

N\_vec = np.vectorize(N)

It is then important that N(x) returns float and not int values, otherwise the

vectorized version will produce int values and hence be incorrect.

A manual rewrite, yielding a faster vectorized function, is more demanding than

for the Heaviside function because we now have multiple branches in the if test.

One sketch is to replace

if condition1:

r = <expression1>

elif condition2:

r = <expression2>

elif condition3:

r = <expression3>

else:

r = <expression4>

by

x1 = <expression1>

x2 = <expression2>

x3 = <expression3>

x4 = <expression4>

r = np.where(condition1, x1, x4)

r = np.where(condition2, x2, r)

r = np.where(condition3, x3, r)

# initialize with "else" expr.266

5 Array Computing and Curve Plotting

Alternatively, we can use boolean indexing. Assuming that <expressionX> is

some expression involving an array x and coded as a Python function fX(x) (X is

1, 2, 3, or 4), we can write

r = f4(x)

r[condition1] = f1(x[condition1])

r[condition2] = f2(x[condition2])

r[condition3] = f2(x[condition3])

Specifically, when the function for scalar arguments x reads

def N(x):

if x < 0:

return 0.0

elif 0 <= x < 1:

return x

elif 1 <= x < 2:

return 2 - x

elif x >= 2:

return 0.0

a vectorized attempt would be

def Nv(x):

r = np.where(x < 0,

0.0,

r = np.where(0 <= x < 1, x,

r = np.where(1 <= x < 2, 2-x,

r = np.where(x >= 2,

0.0,

return r

0.0)

r)

r)

r)

The first and last line are not strictly necessary as we could just start with a zero

vector (making the insertion of zeros for the first and last condition a redundant

operation).

However, any condition like 0 <= x < 1, which is equivalent to 0 <= x and

x < 1, does not work because the and operator does not work with array ar-

guments. Fortunately, there is a simple solution to this problem: the function

logical\_and in numpy. A working Nv function must apply logical\_and instead

in each condition:

def Nv1(x):

condition1 = x < 0

condition2 = np.logical\_and(0 <= x, x < 1)

condition3 = np.logical\_and(1 <= x, x < 2)

condition4 = x >= 2

r = np.where(condition1, 0.0, 0.0)

r = np.where(condition2, x,

r)

r = np.where(condition3, 2-x, r)

r = np.where(condition4, 0.0, r)

return r

With boolean indexing we get the alternative form5.6 More on Numerical Python Arrays

267

def Nv2(x):

condition1 = x < 0

condition2 = np.logical\_and(0 <= x, x < 1)

condition3 = np.logical\_and(1 <= x, x < 2)

condition4 = x >= 2

r = np.zeros(len(x))

r[condition1] = 0.0

r[condition2] = x[condition2]

r[condition3] = 2-x[condition3]

r[condition4] = 0.0

return r

Again, the first and last assignment to r can be omitted in this special case where

we start with a zero vector.

The file hat.py implements four vectorized versions of the N(x) function:

N\_loop, which is a plain loop calling up N(x) for each x[i] element in the array

x; N\_vec, which is the result of automatic vectorization via np.vectorize; the

Nv1 function shown above, which uses the np.where constructions; and the Nv2

function, which uses boolean indexing. With a length of x of 1,000,000, the results

on my computer (MacBook Air 11”, 2 1.6GHz Intel CPU, running Ubuntu 12.04

in a VMWare virtual machine) became 4.8 s for N\_loop, 1 s N\_vec, 0.3 s for Nv1,

and 0.08 s for Nv2. Boolean indexing is clearly the fastest method.

5.6

More on Numerical Python Arrays

This section lists some more advanced but useful operations with Numerical Python

arrays.

5.6.1

Copying Arrays

Let x be an array. The statement a = x makes a refer to the same array as x.

Changing a will then also affect x:

>>> import numpy as np

>>> x = np.array([1, 2, 3.5])

>>> a = x

>>> a[-1] = 3 # this changes x[-1] too!

>>> x

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

Changing a without changing x requires a to be a copy of x:

>>> a = x.copy()

>>> a[-1] = 9

>>> a

array([ 1., 2.,

>>> x

array([ 1., 2.,

9.])

3.])268

5.6.2

5 Array Computing and Curve Plotting

In-Place Arithmetics

Let a and b be two arrays of the same shape. The expression a += b means a = a

+ b, but this is not the complete story. In the statement a = a + b, the sum a +

b is first computed, yielding a new array, and then the name a is bound to this new

array. The old array a is lost unless there are other names assigned to this array. In

the statement a += b, elements of b are added directly into the elements of a (in

memory). There is no hidden intermediate array as in a = a + b. This implies

that a += b is more efficient than a = a + b since Python avoids making an extra

array. We say that the operators +=, \*=, and so on, perform in-place arithmetics in

arrays.

Consider the compound array expression

a = (3\*x\*\*4 + 2\*x + 4)/(x + 1)

The computation actually goes as follows with seven hidden arrays for storing in-

termediate results:

r1 = x\*\*4

r2 = 3\*r1

r3 = 2\*x

r4 = r2 + r3

r5 = r4 + 4

r6 = x + 1

r7 = r5/r6

a = r7

With in-place arithmetics we can get away with creating three new arrays, at a cost

of a significantly less readable code:

a = x.copy()

a \*\*= 4

a \*= 3

a += 2\*x

a += 4

a /= x + 1

The three extra arrays in this series of statement arise from copying x, and comput-

ing the right-hand sides 2\*x and x+1.

Quite often in computational science and engineering, a huge number of arith-

metics is performed on very large arrays, and then saving memory and array allo-

cation time by doing in-place arithmetics is important.

The mix of assignment and in-place arithmetics makes it easy to make unin-

tended changes of more than one array. For example, this code changes x:

a = x

a += y

since a refers to the same array as x and the change of a is done in-place.5.6 More on Numerical Python Arrays

269

5.6.3 Allocating Arrays

We have already seen that the np.zeros function is handy for making a new array

a of a given size. Very often the size and the type of array elements have to match

another existing array x. We can then either copy the original array, e.g.,

a = x.copy()

and fill elements in a with the right new values, or we can say

a = np.zeros(x.shape, x.dtype)

The attribute x.dtype holds the array element type (dtype for data type), and

x.shape is a tuple with the array dimensions. The variable a.ndim holds the num-

ber of dimensions.

Sometimes we may want to ensure that an object is an array, and if not, turn it

into an array. The np.asarray function is useful in such cases:

a = np.asarray(a)

Nothing is copied if a already is an array, but if a is a list or tuple, a new array with

a copy of the data is created.

5.6.4

Generalized Indexing

Section 5.2.2 shows how slices can be used to extract and manipulate subarrays.

The slice f:t:i corresponds to the index set f, f+i, f+2\*i, ... up to, but not

including, t. Such an index set can be given explicitly too: a[range(f,t,i)].

That is, the integer list from range can be used as a set of indices. In fact, any

integer list or integer array can be used as index:

>>> a = np.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

# same as a[2:8:3] = -2

>>> a

array([ 1., 10., -2.,

4.,

5., -2., 10., 10.])

We can also use boolean arrays to generate an index set. The indices in the set

will correspond to the indices for which the boolean array has True values. This

functionality allows expressions like a[x<m]. Here are two examples, continuing

the previous interactive session:270

5 Array Computing and Curve Plotting

>>> 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.])

>>> # Replace elements where a is 10 by the first

>>> # elements from another array/list:

>>> a[a == 10] = [10, 20, 30, 40, 50, 60, 70]

>>> a

array([ 1., 10., 20.,

4.,

5., 30., 40., 50.])

Generalized indexing using integer arrays or lists is important for vectorized

initialization of array elements. The syntax for generalized indexing of higher-

dimensional arrays is slightly different, see Sect. 5.8.2.

5.6.5 Testing for the Array Type

Inside an interactive Python shell you can easily check an object’s type using the

type function (see Sect. 1.5.2). In case of a Numerical Python array, the type name

is ndarray:

>>> a = np.linspace(-1, 1, 3)

>>> a

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

>>> type(a)

<type ’numpy.ndarray’>

Sometimes you need to test if a variable is an ndarray or a float or int. The

isinstance function can be used this purpose:

>>> isinstance(a, np.ndarray)

True

>>> isinstance(a, (float,int))

False

# float or int?

A typical use of isinstance and type to check on object’s type is shown next.

Example: Vectorizing a constant function Suppose we have a constant function,

def f(x):

return 2

This function accepts an array argument x, but will return a float while a vector-

ized version of the function should return an array of the same shape as x where

each element has the value 2. The vectorized version can be realized as

def fv(x):

return np.zeros(x.shape, x.dtype) + 25.6 More on Numerical Python Arrays

271

The optimal vectorized function would be one that works for both a scalar and an

array argument. We must then test on the argument type:

def f(x):

if isinstance(x, (float, int)):

return 2

elif isinstance(x, np.ndarray):

return np.zeros(x.shape, x.dtype) + 2

else:

raise TypeError\

(’x must be int, float or ndarray, not %s’ % type(x))

5.6.6

Compact Syntax for Array Generation

There is a special compact syntax r\_[f:t:s] for the linspace function:

>>> a = r\_[-5:5:11j]

>>> print a

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

# same as linspace(-5, 5, 11)

0.

1.

2.

3.

4.

5.]

Here, 11j means 11 coordinates (between 5 and 5, including the upper limit 5).

That is, the number of elements in the array is given with the imaginary number

syntax.

5.6.7 Shape Manipulation

The shape attribute in array objects holds the shape, i.e., the size of each dimension.

A function size returns the total number of elements in an array. Here are a few

equivalent ways of changing the shape of an array:

>>> a = np.linspace(-1, 1, 6)

>>> print a

[-1. -0.6 -0.2 0.2 0.6 1. ]

>>> a.shape

(6,)

>>> a.size

6

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

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

# alternative

>>> a.shape

(2, 3)

>>> print a

[[-1. -0.6 -0.2]

[ 0.2 0.6 1. ]]

>>> a.size

# total no of elements

6

>>> len(a)

# no of rows

2

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

# reset shape

Note that len(a) always returns the length of the first dimension of an array.272

5 Array Computing and Curve Plotting

5.7 High-Performance Computing with Arrays

Programs with lots of array calculations may soon consume much time and memory,

so it may quickly become crucial to speed up calculations and use as little memory

as possible. The main technique for speeding up array calculations is vectorization,

i.e., avoiding explicit loops in Python over array elements. To save memory usage,

one needs to understand when arrays get allocated and avoid this by in-place ar-

ray arithmetics. You should review Sect. 5.6.2 about array allocation and in-place

arithmetics before reading on.

Example: axpy Our computational case study concerns the famous “axpy” opera-

tion: r D ax C y, where a is a number and x and y are arrays. All implementations

and the associated experimentation are found in the file hpc\_axpy.py.

5.7.1

Scalar Implementation

A naive loop implementation of the “axpy” operation ax C y reads

def axpy\_loop\_newr(a, x, y):

r = np.zeros\_like(x)

for i in range(r.size):

r[i] = a\*x[i] + y[i]

return r

Classical implementations overwrite y by ax C y: y

ax C y, but we shall

make implementations where we either can overwrite y or place ax C y in another

array. The function above creates a new array for the result.

Rather than allocating the array inside the function, we can put that burden on

the user and provide a result array r as input:

def axpy\_loop(a, x, y, r):

for i in range(r.size):

r[i] = a\*x[i] + y[i]

return r

The advantage of this version is that we can either overwrite y by ax C y or store

ax C y in a separate array:

# Classical axpy

y = axpy\_loop(a, x, y, y)

# Store axpy result in separate array

r = np.zeros\_like(x)

r = axpy\_loop(a, x, y, r)

Python functions return output data

The call

r = axpy\_loop(a, x, y, r)5.7 High-Performance Computing with Arrays

273

can equally well be written

axpy\_loop(a, x, y, r)

This is the typical coding style in Fortran, C, or C++ (where r is then transferred

as a reference or pointer to the array data). In Python, there is no need for the

function axpy\_loop to return r, because the assignment to r[i] inside the loop

changes all elements of the r. The array r will therefore be modified after calling

axpy\_loop(a, x, y, r) anyway. However, it is a good convention in Python

that all input data to a function are arguments and all output data are returned.

With

r = axpy\_loop(a, x, y, r)

we clearly see that r is both input and output.

5.7.2

Vectorized Implementation

The vectorized implementation of the “axpy” operation reads

def axpy1(a, x, y):

r = a\*x + y

return r

Note that the result is placed in a new array arrising from the operation a\*x+y.

The speed up of vectorization is significant, see Fig. 5.11 (made by the function

effect\_of\_vec in the file hpc\_axpy.py).

Temporary arrays are needed in the vectorized implementation. One would ex-

pect that a\*x must be calculated and stored in a temporary array, call it r1, and

then r1 + y must be evaluated and stored in another allocated array r, which is

returned. It appears that a\*x + y only needs the allocation of one array, the one

to be returned (we have investigated the memory consumption in detail using the

memory\_profiler module). Anyway, repeated calls to axpy1 with large arrays

lead to an allocation of a new large array in each call.

5.7.3

Memory-Saving Implementation

Applications with large arrays should avoid unnecessary allocation of temporary ar-

rays and instead reuse pre-allocated arrays. Suppose we have allocated an array for

the result r = ax + y once and for all. We can pass the r array to the computing

function as in the axpy\_loop function above and use the memory in r for interme-

diate calculation. In vectorized code, this requires use of in-place array arithmetics

(see Sect. 5.6.2)

In-place arithmetics for doing r = a\*x + y in a pre-allocated array r will first

copy all elements of x into r, then perform elementwise multiplication by a, and

finally elementwise addition of y:274

5 Array Computing and Curve Plotting

Fig. 5.11 Improved efficiency of vectorizing the “axpy” operation as function of array length

def axpy2(a, x, y, r):

r[:] = x

r \*= a

r += y

return r

Note that r[:] = x inserts the elements of x into r. A mathematically equivalent

construction, r = x.copy(), allocates a new object and fills it with the values of

x before the name r refers to this object. The array r supplied as argument to the

function is then lost, and the returned array is another object.

We can perform repeated calls to the axpy2 function without any extra memory

allocation. This can be proved by a small code snippet where we use id(r) to see

the unique identity of the r array. If this identity remains constant through calls, we

always reuse the pre-allocated r array:

r = np.zeros\_like(x)

print id(r)

for i in range(10):

r = axpy2(a, x, y, r)

print id(r)

The output prints the same number, proving that it is physically the same object we

feed as r to axpy2 that is also returned.

We can equally well drop returning r and utilize that the changes made by in-

place arithmetics is always reflected in the r array we allocate outside the function:5.7 High-Performance Computing with Arrays

275

def axpy3(a, x, y, r):

r[:] = x

r \*= a

r += y

The call can now just be

axpy3(a, x, y, r)

However, as emphasized in Sect. 5.7.1, in Python we usually return the output data

(because this is no extra cost, only references to objects are physically transferred

back to the calling code).

5.7.4

Analysis of Memory Usage

The module memory\_profiler is very useful for analyzing the memory usage of

every statement in a program. The module is installed by

Terminal

Terminal> sudo pip install memory\_profiler

Each function we want to analyze must have (the decorator) @profile at the line

above, e.g.,

@profile

def axpy1(a, x, y:

r = a\*x + y

return r

We can then run a program with name axpy.py by

Terminal

Terminal> python -m memory\_profiler axpy.py

The arrays must be quite large to see a significant increase in memory usage. With

10,000,000 elements in each array we get output like

Line #

Mem usage

Increment

Line Contents

================================================

1 251.977 MiB

0.000 MiB

@profile

2

def axpy1(a, x, y:

3 328.273 MiB

76.297 MiB

r = a\*x + y

4 328.273 MiB

0.000 MiB

return r276

5 Array Computing and Curve Plotting

Line #

Mem usage

Increment

Line Contents

================================================

6 251.977 MiB

0.000 MiB

@profile

7

def axpy2(a, x, y, r):

8 251.977 MiB

0.000 MiB

r[:] = x

9 251.977 MiB

0.000 MiB

r \*= a

10 251.977 MiB

0.000 MiB

r += y

11 251.977 MiB

0.000 MiB

return r

This demonstrates the larger memory consumption of axpy1 compared with axpy2.

5.7.5 Analysis of the CPU Time

The module line\_profiler can time each line of a program. Installation is easily

done by sudo pip install line\_profiler. As for module\_profiler de-

scribed in the previous section, also line\_profiler requires each function to be

analyzed to have (the decorator) @profile at the line above the function. The

module is installed together with an analysis script kernprof that we use to run the

program:

Terminal

Terminal> kernprof -l -v axpy.py

With an array length of 500,000 we get output like

Total time: 0.014291 s

Function: axpy1 at line 1

Line # Hits

Time Per Hit

% Time Line Contents

==============================================================

1

@profile

2

def axpy1(a, x, y):

3

3

14283

4761.0

99.9

r = a\*x + y

4

3

8

2.7

0.1

return r

Total time: 0.004382 s

Function: axpy2 at line 6

Line # Hits Time Per Hit

% Time Line Contents

==============================================================

6

@profile

7

def axpy2(a, x, y, r):

8

3

1981

660.3

45.2

r[:] = x

9

3

1258

419.3

28.7

r \*= a

10

3

1138

379.3

26.0

r += y

11

3

5

1.7

0.1

return r

Total time: 1.38674 s

Function: axpy\_loop at line 265.8 Higher-Dimensional Arrays

277

Line # Hits

Time Per Hit % Time Line Contents

==============================================================

26

@profile

27

def axpy\_loop(a, x, y, r):

28

1500006 449747

0.3

32.4

for i in range(r.size):

29

1500003 936985

0.6

67.6

r[i] = a\*x[i] + y[i]

30

3

10

3.3

0.0

return r

We see that the administration of the for loop takes 1/3 of the total cost of the loop.

Both line\_profiler and memory\_profiler are very useful tools for spotting

inefficient constructions in a code.

5.8Higher-Dimensional Arrays

5.8.1Matrices and Arrays

Vectors appeared when mathematicians needed to calculate with a list of numbers.

When they needed a table (or a list of lists in Python terminology), they invented

the concept of matrix (singular) and matrices (plural). A table of numbers has the

numbers ordered into rows and columns. One example is

3

2

0 12 1 5

7

6

4 1 1 1 0 5

11 5

5 2

This table with three rows and four columns is called a 3 4 matrix (mathematicians

may not like this sentence, but it suffices for our purposes). If the symbol A is

associated with this matrix, Ai;j denotes the number in row number i and column

number j . Counting rows and columns from 0, we have, for instance, A0;0 D 0 and

A2;3 D 2. We can write a general m n matrix A as

3

2

A0;n1

A0;0

7

6

::

::

::

7

6

:

:

:

5

4

Am1;0 Am1;n1

Matrices can be added and subtracted. They can also be multiplied by a scalar

(a number), and there is a concept of length or size. The formulas are quite similar

to those presented for vectors, but the exact form is not important here.

We can generalize the concept of table and matrix to array, which holds quanti-

ties with in general d indices. Equivalently we say that the array has rank d . For

d D 3, an array A has elements with three indices: Ap;q;r . If p goes from 0 to np 1,

q from 0 to nq 1, and r from 0 to nr 1, the A array has np nq nr elements in

total. We may speak about the shape of the array, which is a d -vector holding the

number of elements in each “array direction”, i.e., the number of elements for each

index. For the mentioned A array, the shape is .np ; nq ; nr /.

The special case of d D 1 is a vector, and d D 2 corresponds to a matrix. When

we program we may skip thinking about vectors and matrices (if you are not so278

5 Array Computing and Curve Plotting

familiar with these concepts from a mathematical point of view) and instead just

work with arrays. The number of indices corresponds to what is convenient in the

programming problem we try to solve.

5.8.2 Two-Dimensional Numerical Python Arrays

Consider a nested list table of two-pairs [C, F] (see Sect. 2.4) constructed by

>>> Cdegrees = [-30 + i\*10 for i in range(3)]

>>> Fdegrees = [9./5\*C + 32 for C in Cdegrees]

>>> table = [[C, F] for C, F in zip(Cdegrees, Fdegrees)]

>>> print table

[[-30, -22.0], [-20, -4.0], [-10, 14.0]]

This nested list can be turned into an array,

>>> table2 = np.array(table)

>>> print table2

[[-30. -22.]

[-20. -4.]

[-10. 14.]]

>>> type(table2)

<type ’numpy.ndarray’>

We say that table2 is a two-dimensional array, or an array of rank 2.

The table list and the table2 array are stored very differently in memory. The

table variable refers to a list object containing three elements. Each of these ele-

ments is a reference to a separate list object with two elements, where each element

refers to a separate float object. The table2 variable is a reference to a single ar-

ray object that again refers to a consecutive sequence of bytes in memory where the

six floating-point numbers are stored. The data associated with table2 are found

in one chunk in the computer’s memory, while the data associated with table are

scattered around in memory. On today’s machines, it is much more expensive to

find data in memory than to compute with the data. Arrays make the data fetching

more efficient, and this is major reason for using arrays. However, this efficiency

gain is only present for very large arrays, not for a 3 2 array.

Indexing a nested list is done in two steps, first the outer list is indexed, giving

access to an element that is another list, and then this latter list is indexed:

>>> table[1][0]

-20

# table[1] is [-20,4], whose index 0 holds -20

This syntax works for two-dimensional arrays too:

>>> table2[1][0]

-20.0

but there is another syntax that is more common for arrays:5.8 Higher-Dimensional Arrays

279

>>> table2[1,0]

-20.0

A two-dimensional array reflects a table and has a certain number of rows and

columns. We refer to rows as the first dimension of the array and columns as the

second dimension. These two dimensions are available as table2.shape:

>>> table2.shape

(3, 2)

Here, 3 is the number of rows and 2 is the number of columns.

A loop over all the elements in a two-dimensional array is usually expressed as

two nested for loops, one for each index:

>>> for i in range(table2.shape[0]):

...

for j in range(table2.shape[1]):

...

print ’table2[%d,%d] = %g’ % (i, j, table2[i,j])

...

table2[0,0] = -30

table2[0,1] = -22

table2[1,0] = -20

table2[1,1] = -4

table2[2,0] = -10

table2[2,1] = 14

An alternative (but less efficient) way of visiting each element in an array with any

number of dimensions makes use of a single for loop:

>>> for index\_tuple, value in np.ndenumerate(table2):

...

print ’index %s has value %g’ % \

...

(index\_tuple, table2[index\_tuple])

...

index (0,0) has value -30

index (0,1) has value -22

index (1,0) has value -20

index (1,1) has value -4

index (2,0) has value -10

index (2,1) has value 14

In the same way as we can extract sublists of lists, we can extract subarrays of

arrays using slices.

table2[0:table2.shape[0], 1]

array([-22., -4., 14.])# 2nd column (index 1)

>>> table2[0:, 1]

array([-22., -4.,# same

14.])>>> table2[:, 1]

array([-22., -4.,14.])

# same

To illustrate array slicing further, we create a bigger array:280

5 Array Computing and Curve Plotting

>>> t = np.linspace(1, 30, 30).reshape(5, 6)

>>> t

array([[ 1.,

2.,

3.,

4.,

5.,

6.],

[ 7.,

8.,

9., 10., 11., 12.],

[ 13., 14., 15., 16., 17., 18.],

[ 19., 20., 21., 22., 23., 24.],

[ 25., 26., 27., 28., 29., 30.]])

>>> t[1:-1:2, 2:]

array([[ 9., 10.,

[ 21., 22.,

11.,

23.,

12.],

24.]])

To understand the slice, look at the original t array and pick out the two rows

corresponding to the first slice 1:-1:2,

[ 7.,

[ 19.,

8.,

20.,

9.,

21.,

10.,

22.,

11.,

23.,

12.]

24.]

Among the rows, pick the columns corresponding to the second slice 2:,

[ 9.,

[ 21.,

10.,

22.,

11.,

23.,

12.]

24.]

Another example is

>>> t[:-2, :-1:2]

array([[ 1.,

3.,

[ 7.,

9.,

[ 13., 15.,

5.],

11.],

17.]])

Generalized indexing as described for one-dimensional arrays in Sect. 5.6.4 requires

a more comprehensive syntax for higher-dimensional arrays. Say we want to extract

a subarray of t that consists of the rows with indices 0 and 3 and the columns with

indices 1 and 2:

>>> t[np.ix\_([0,3], [1,2])]

array([[ 2.,

3.],

[ 20., 21.]])

>>> t[np.ix\_([0,3], [1,2])] = 0

>>> t

array([[ 1.,

0.,

0.,

4.,

[ 7.,

8.,

9., 10.,

[ 13., 14., 15., 16.,

[ 19.,

0.,

0., 22.,

[ 25., 26., 27., 28.,

5.,

11.,

17.,

23.,

29.,

6.],

12.],

18.],

24.],

30.]])

Recall that slices only gives a view to the array, not a copy of the values:

>>> a = t[1:-1:2, 1:-1]

>>> a

array([[ 8.,

9., 10.,

[ 0.,

0., 22.,

11.],

23.]])5.8 Higher-Dimensional Arrays

>>> a[:,:] = -99

>>> a

array([[-99., -99., -99., -99.],

[-99., -99., -99., -99.]])

>>> t # is t changed to? yes!

array([[ 1.,

0.,

0.,

4.,

5.,

[ 7., -99., -99., -99., -99.,

[ 13., 14., 15., 16., 17.,

[ 19., -99., -99., -99., -99.,

[ 25., 26., 27., 28., 29.,

281

6.],

12.],

18.],

24.],

30.]])

5.8.3 Array Computing

The operations on vectors in Sect. 5.1.3 can quite straightforwardly be extended to

arrays of any dimension. Consider the definition of applying a function f .v/ to

a vector v: we apply the function to each element vi in v. For a two-dimensional

array A with elements Ai;j , i D 0; : : : ; m, j D 0; : : : ; n, the same definition yields

f .A/ D .f .A0;0 /; : : : ; f .Am1;0 /; f .A1;0 /; : : : ; f .Am1;n1 // :

For an array B with any rank, f .B/ means applying f to each array entry.

The asterisk operation from Sect. 5.1.3 is also naturally extended to arrays: AB

means multiplying an element in A by the corresponding element in B, i.e., element

.i; j / in A B is Ai;j Bi;j . This definition naturally extends to arrays of any rank,

provided the two arrays have the same shape.

Adding a scalar to an array implies adding the scalar to each element in the

array. Compound expressions involving arrays, e.g., exp.A2 / A C 1, work as

for vectors. One can in fact just imagine that all the array elements are stored after

each other in a long vector (this is actually the way the array elements are stored

in the computer’s memory), and the array operations can then easily be defined in

terms of the vector operations from Sect. 5.1.3.

Remark Readers with knowledge of matrix computations may get confused by

the meaning of A2 in matrix computing and A2 in array computing. The former is

a matrix-matrix product, while the latter means squaring all elements of A. Which

rule to apply, depends on the context, i.e., whether we are doing linear algebra

or vectorized arithmetics. In mathematical typesetting, A2 can be written as AA,

while the array computing expression A2 can be alternatively written as A A. In

a program, A\*A and A\*\*2 are identical computations, meaning squaring all elements

(array arithmetics). With NumPy arrays the matrix-matrix product is obtained by

dot(A, A). The matrix-vector product Ax, where x is a vector, is computed by

dot(A, x). However, with matrix objects (see Sect. 5.8.4) A\*A implies the mathe-

matical matrix multiplication AA.

We shall leave this subject of notational confusion between array computing and

linear algebra here since this book will not further understanding and the confusion

is seldom serious in program code if one has a good overview of the mathematics

that is to be carried out.282

5 Array Computing and Curve Plotting

5.8.4 Matrix Objects

This section only makes sense if you are familiar with basic linear algebra and the

matrix concept. The arrays created so far have been of type ndarray. NumPy

also has a matrix type called matrix or mat for one- and two-dimensional arrays.

One-dimensional arrays are then extended with one extra dimension such that they

become matrices, i.e., either a row vector or a column vector:

>>> import numpy as np

>>> x1 = np.array([1, 2, 3], float)

>>> x2 = np.matrix(x1)

# or mat(x1)

>>> x2

# row vector

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

>>> x3 = mat(x1).T

# transpose = column vector

>>> x3

matrix([[ 1.],

[ 2.],

[ 3.]])

>>> type(x3)

<class ’numpy.matrixlib.defmatrix.matrix’>

>>> isinstance(x3, np.matrix)

True

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 = np.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

>>> y2

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

>>> y3 = A\*x3

>>> y3

matrix([[ 1.],

[ 2.],

[ 3.]])

# identity matrix

# vector-matrix product

# matrix-vector product

One should note here that the multiplication operator between standard ndarray

objects is quite different!

Readers who are familiar with MATLAB, or intend to use Python and MATLAB

together, should seriously think about programming with matrix objects instead

of ndarray objects, because the matrix type behaves quite similar to matrices5.9 Some Common Linear Algebra Operations

283

and vectors in MATLAB. Nevertheless, matrix cannot be used for arrays of larger

dimension than two.

5.9

Some Common Linear Algebra Operations

Python has strong support for numerical linear algebra, much like the functional-

ity found in MATLAB. Some of the most widely used operations are exemplified

below.

5.9.1

Inverse, Determinant, and Eigenvalues

We start with showing how to find the inverse and the determinant of a matrix, and

how to compute the eigenvalues and eigenvectors:

>>> import numpy as np

>>> A = np.array([[2, 0], [0, 5]], dtype=float)

>>> np.linalg.inv(A) # inverse matrix

array([[ 0.5, 0. ],

[ 0. , 0.2]])

>>> np.linalg.det(A)

9.9999999999999982

# determinant

>>> eig\_values, eig\_vectors = np.linalg.eig(A)

>>> eig\_values

array([ 2., 5.])

>>> eig\_vectors

array([[ 1., 0.],

[ 0., 1.]])

The eigenvectors are normalized to have unit lengths.

5.9.2

Products

The np.dot function is used for scalar or dot product as well as matrix-vector and

matrix-matrix products between array objects:

>>> a = np.array([4, 0])

>>> b = np.array([0, 1])

>>> np.dot(A, a)

# matrix vector product

array([ 8., 0.])

>>> np.dot(a, b)

# dot product between vectors

0

>>>

>>> B = np.ones((2, 2)) # 2x2 matrix with 1’s

>>> np.dot(A, B)

# matrix-matrix product

array([[ 2., 2.],

[ 5., 5.]])284

5 Array Computing and Curve Plotting

Note that using the matrix class instead of plain arrays (see Sect. 5.8.4) allows

\* to be used as operator for matrix-vector and matrix-matrix products.

The cross product a b, between vectors a and b of length 3, is computed by

>>> np.cross([1, 1, 1], [0, 0, 1])

array([ 1, -1, 0])

Finding the angle between vectors a and b,

D cos1

ab

;

jjajj jjbjj

goes like

>>> np.arccos(np.dot(a, b)/(np.linalg.norm(a)\*np.linalg.norm(b)))

1.5707963267948966

5.9.3 Norms

Various norms of matrices and vectors are well supported by NumPy. Some com-

mon examples are

>>> np.linalg.norm(A)

5.3851648071345037

>>> np.sqrt(np.sum(A\*\*2))

5.3851648071345037

>>> np.linalg.norm(a)

4.0

# Frobenius norm for matrices

# Frobenius norm: direct formula

# l2 norm for vectors

See pydoc numpy.linalg.norm for information on other norms.

5.9.4 Sum and Extreme Values

The sum of all elements or of the elements in a particular row or column is computed

by np.sum:

>>> np.sum(B)

2.0

>>> B.sum()

2.0

>>> np.sum(B, axis=0)

array([ 4., -2.])

>>> np.sum(B, axis=1)

array([ 3., -1.])

# sum of all elements

# sum of all elements; alternative syntax

# sum over index 0 (rows)

# sum over index 1 (columns)

The maximum or minimum value of an array is also often needed:5.9 Some Common Linear Algebra Operations

>>> np.max(B)

3.0

>>> B.max()

3.0

>>> np.min(B)

-4.0

>>> np.abs(B).min()

1.0

285

# max over all elements

# max over all elements, alt. syntax

# min over all elements

# min absolute value

A very frequent application of computing the minimum absolute value occurs in

test functions where we want to verify a result, e.g., that AA1 D I , where I is the

identity matrix. We then want to check the smallest absolute value in AA1 I :

>>> I = np.eye(2)

# identity matrix of size 2

>>> I

array([[ 1., 0.],

[ 0., 1.]])

>>> np.abs(np.dot(A, np.linalg.inv(A)) - I).max()

0.0

Never use == when testing real numbers!

It could be tempting to test AA1 D I using the syntax

>>> np.dot(A, np.linalg.inv(A)) == np.eye(2)

array([[ True, True],

[ True, True]], dtype=bool)

but there are two major problems with this test:

1. the result is a boolean matrix, not suitable for an if test

2. using == for matrices with float elements may fail because of rounding errors

The second problem must be solved by computing differences and comparing

them against small tolerances, as we did above. Here is an example where ==

fails:

>>> A = np.array([[4, 0], [0, 49]], dtype=float)

>>> np.dot(A, np.linalg.inv(A)) == np.eye(2)

array([[ True, True],

[ True, False]], dtype=bool)

(1.0/49\*49 is not exactly 1 because of rounding errors.)

The first problem is solved by using the C.all(), which returns one boolean

variable True if all elements in the boolean array C are True, otherwise it returns

False, as in the case above:

>>> (np.dot(A, np.linalg.inv(A)) == np.eye(2)).all()

False286

5 Array Computing and Curve Plotting

5.9.5 Indexing

Indexing an element is done by A[i,j]. A row or column is extracted as

>>> A[0,:] # first row

array([ 2., 0.])

>>> A[:,1] # second column

array([ 0., 5.])

NumPy also supports multiple values for the indices via the np.ix\_ function. Here

is an example where we grab row 0 and 2, then column 1:

>>> C = np.array([[1,2,3],[4,5,6],[7,8,9]])

>>> C[np.ix\_([0,2], [1])] # row 0 and 2, then column 1

array([[2],

[8]])

You can also use the colon notation to pick out other parts of a matrix. If C is

a 3 5-matrix,

C[1:3, 0:4]

gives a sub-matrix consisting of the two rows of C after the first, and the first four

columns of C (recall that the upper limits, here 3 and 4, are not included).

Readers familiar with MATLAB should note that the indexing may be a bit un-

expected when referring to parts of a matrix: writing C[[0, 2], [0, 2]] one

would expect entries residing in rows/columns 0 and 2, but that behavior requires

in Python the np.ix\_ command:

>>> C = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

>>> C[np.ix\_([0, 2], [0, 2])]

[[1 3]

[7 9]]

>>> # Grab row 0, 2, then column 0 from row 0 and column 2 from row 2

>>> C[[0, 2], [0, 2]]

[1 9]

5.9.6 Transpose and Upper/Lower Triangular Parts

The transpose of a matrix B is obtained by B.T:

>>> B = np.array([[1, 2], [3, -4]], dtype=float)

>>> B.T

# the transpose

array([[ 1., 3.],

[ 2., -4.]])

NumPy has rich functionality for doing operations on array objects. For exam-

ple, one can strip down a matrix to its upper or lower triangular parts:5.9 Some Common Linear Algebra Operations

287

>>> np.triu(B) # upper triangular part of B

array([[ 1., 2.],

[ 0., -4.]])

>>> np.tril(B) # lower triangular part of B

array([[ 1., 0.],

[ 3., -4.]])

5.9.7 Solving Linear Systems

The perhaps most frequent operation in linear algebra is the solution of sys-

tems of linear algebraic equations: Ax D b, where A is a coefficient matrix,

b is a given right-hand side vector, and x is the solution vector. The function

np.linalg.solve(A, b) does the job:

>>> A = np.array([[1, 2], [-2, 2.5]])

>>> x = np.array([-1, 1], dtype=float)

>>> b = np.dot(A, x)

>>> np.linalg.solve(A, b)

array([-1., 1.])

5.9.8

# pick a solution

# find right-hand side

# will this compute x?

Matrix Row and Column Operations

Implementing Gaussian elimination constitutes a good pedagogical example on

how to perform row and column operations on a matrix. Some needed function-

ality is

A[[i, j]] = A[[j, i]]

A[i] \*= k

A[j] += k\*A[i]

# swap rows i and j

# multiply row i by a constant k

# add row i, multiplied by k, to row j

With these operations, Gaussian elimination is programmed as follows.

m, n = shape(A)

for j in range(n - 1):

for i in range(j + 1, m):

A[i,j:] -= (A[i,j]/A[j,j])\*A[j,j:]

Note the special syntax j:, which refers to indices from j and up to the end of the

array. More generally, when referring to an array a with length n, the following are

equivalent:

a[0:n]

a[:n]

a[0:]

a[:]288

5 Array Computing and Curve Plotting

In the code for Gaussian elimination, we first eliminate the entries below the diago-

nal in the first column, by adding a scaled version of the first row to the other rows.

Then the same procedure is applied for the second row, and so on. The result is an

upper triangular matrix. The code can fail if some of the entries A[j,j] become

zero along the way. To avoid this, we can swap rows when the problem arises. The

following code implements the idea and will not fail, even if some of the columns

are zero.

def Gaussian\_elimination(A):

rank = 0

m, n = np.shape(A)

i = 0

for j in range(n):

p = np.argmax(abs(A[i:m,j]))

if p > 0: # swap rows

A[[i,p+i]] = A[[p+i, i]]

if A[i,j] != 0:

# j is a pivot column

rank += 1

for r in range(i+1, m):

A[r,j:] -= (A[r,j]/A[i,j])\*A[i,j:]

i += 1

if i > m:

break

return A, rank

Note that we stick to the habit of returning all results from a function, here the

modified matrix A and its rank.

5.9.9 Computing the Rank of a Matrix

The rank of a matrix equals the number of pivot columns after Gaussian elimination.

The variable rank counts these in the code above.

Due to rounding errors, the computed rank may be higher than the actual rank:

the rounding errors may imply that A[i,j] != 0 is true, even if Gaussian elim-

ination performed in exact arithmetics gives exactly zero. Such situations can be

avoided by replacing if A[i,j] !=0: with if abs(A[i,j]) > tol:, where

tol is some small tolerance.

A more reliable way to compute the rank is to compute the singular value de-

composition of A, and check how many of the singular values that are larger than

a threshold epsilon:

>>> A = np.array([[1, 2.01], [2.01, 4.0401]])

>>> U, s, V = np.linalg.svd(A) # s are the singular values of A

# abs(s) > tol gives an array with True and False values

# s.nonzero() lists indices k so that s[k] != 0

>>> shape((abs(s) > tol).nonzero())[1] # rank

1

>>> A, rank = Gaussian\_elimination(A)

>>> rank

25.9 Some Common Linear Algebra Operations

289

If you use a tolerance check on the form if abs(A[i,j]) > 1E-10: in the func-

tion Gaussian\_elimination, the code will say that the rank is 1, which is the

correct value also found by using the singular value decomposition.

It is known that the determinant is nonzero if and only if the rank equals the

number of rows/columns. For the matrix A we used above, the determinant should

thus be 0, but also here roundoff errors come into play:

>>> A = np.array([[1, 2.01], [2.01, 4.0401]])

>>> A[0, 0]\*A[1, 1] - A[0, 1]\*A[1, 0]

8.881784197e-16

>>> np.linalg.det(A)

8.92619311799e-16

Using our own Gaussian elimination function for computing the rank is less

efficient than calling NumPy’s singular value decomposition. Here are timings for

a random 100 100-matrix:

>>> A = np.random.uniform(0, 1, (100, 100))

>>> %timeit U, s, V = np.linalg.svd(A)

100 loops, best of 3: 3.7 ms per loop

>>> %timeit A, rank = Gaussian\_elimination(A)

100 loops, best of 3: 22.3 ms per loop

5.9.10

Symbolic Linear Algebra

SymPy supports symbolic computations also for linear algebra operations. We may

create a matrix and find its inverse and determinant:

>>> import sympy as sym

>>> A = sym.Matrix([[2, 0], [0, 5]])

>>> A\*\*-1

# the inverse

Matrix([

[1/2,

0],

[ 0, 1/5]])

>>> A.inv() # the inverse

Matrix([

[1/2,

0],

[ 0, 1/5]])

>>> A.det()

10

# the determinant

Note that the entries in the inverse matrix are rational numbers (sym.Rational

objects to be precise).

Eigenvalues can also be computed exactly:

>>> A.eigenvals()

{2: 1, 5: 1}290

5 Array Computing and Curve Plotting

The output is a dictionary meaning here that 2 is an eigenvalue with multiplicity

1 and 5 is an eigenvalue with multiplicity 1. It is more convenient to have the

eigenvalues in a list:

>>> e = list(A.eigenvals().keys())

>>> e

[2, 5]

Eigenvector computations have a somewhat complicated output:

>>> A.eigenvects()

[(2, 1, [Matrix([

[1],

[0]])]), (5, 1, [Matrix([

[0],

[1]])])]

The output is a list of three-tuples, one for each eigenvalue and eigenvector.

The three-tuple contains the eigenvalue, its multiplicity, and the eigenvector as

a sym.Matrix object. To isolate the first eigenvector, we can index the list and

tuple:

>>> v1 = A.eigenvects()[0][2]

>>> v1

Matrix([

[1],

[0]])

The vector is a sym.Matrix object with two indices. To extract the vector elements

in a plain list, we can do this:

>>> v1 = [v1[i,0] for i in range(v1.shape[0])]

>>> v1

[1, 0]

The following code extracts all eigenvectors as a list of 2-lists, which may be a con-

venient data structure for the eigenvectors:

>>> v = [[t[2][0][i,0] for i in range(t[2][0].shape[0])]

for t in A.eigenvects()]

>>> v

[[1, 0], [0, 1]]

The norm of a matrix or vector is an exact expression:

>>> A.norm()

sqrt(29)

>>> a = sym.Matrix([1, 2])

>>> a

# vector [1, 2]5.9 Some Common Linear Algebra Operations

291

Matrix([

[1],

[2]])

>>> a.norm()

sqrt(5)

The matrix-vector product and the dot product between vectors are done like

this:

>>> A\*a

Matrix([

[ 2],

[10]])

>>> b = sym.Matrix([2, -1])

>>> a.dot(b)

0

# matrix\*vector

# vector [2, -1]

Solving linear systems exactly is also possible:

>>> x = sym.Matrix([-1, 1])/2

>>> x

Matrix([

[-1/2],

[ 1/2]])

>>> b = A\*x

>>> x = A.LUsolve(b) # does it compute x?

>>> x

# x is a matrix object

Matrix([

[-1/2],

[ 1/2]])

Sometimes one wants to convert x to a plain numpy array with float values:

>>> x = np.array([float(x[i,0].evalf())

for i in range(x.shape[0])])

>>> x

array([-0.5, 0.5])

Exact row operations can be done as exemplified here:

>>> A[1,:] + 2\*A[0,:]

Matrix([[4, 5]])

# [0,5] + 2\*[2,0]

We refer to the online SymPy linear algebra tutorial7 for more information.

7

http://docs.sympy.org/dev/tutorial/matrices.html292

5.10

5 Array Computing and Curve Plotting

Plotting of Scalar and Vector Fields

Visualization of scalar and vector fields in Python is commonly done using Mat-

plotlib or Mayavi. Both packages support basic visualization of 2D scalar and

vector fields, but Mayavi offers more advanced three-dimensional visualization

techniques, especially for 3D scalar and vector fields.

One can also use SciTools for visualizing 2D scalar and vector fields, using either

Matplotlib, Gnuplot, or VTK as plotting engines, but this topic is omitted from the

present book. However, for fast visualization of large 2D scalar fields, Gnuplot is

a viable tool, and the SciTools interface offers a convenient MATLAB-style set of

commands to operate Gnuplot.

To exemplify visualization of scalar and vector fields with Matplotlib and

Mayavi, we use a common set of examples. A scalar function of x and y is vi-

sualized either as a flat two-dimensional plot with contour lines of the field, or

as a three-dimensional surface where the height of the surface corresponds to the

function value of the field. In the latter case we also add a three-dimensional

parameterized curve to the plot.

To illustrate plotting of vector fields, we simply plot the gradient of the scalar

field, together with the scalar field. Our convention for variable names goes as

follows:

x, y for one-dimensional coordinates along each axis direction.

xv, yv for the corresponding vectorized coordinates in a 2D.

u, v for the components of a vector field at points corresponding to xv, yv.

The following sections contain more mathematical details on the various scalar and

vector fields we aim to plot.

5.10.1 Installation

Previously in the book we have explained how to obtain Matplotlib for various

platforms. To obtain Mayavi on Ubuntu platforms you can write

Terminal

pip install mayavi --upgrade

For Mac OS X and Windows, we recommend using Anaconda. To obtain Mayavi

for Anaconda you can write

Terminal

conda install mayavi5.10 Plotting of Scalar and Vector Fields

5.10.2

293

Surface Plots

We consider the 2D scalar field defined by

h.x; y/ D

h0

1 C x RCy

2

2

2

:

(5.13)

h.x; y/ may model the height of an isolated circular mountain, h being the height

above sea level, while x and y are Cartesian coordinates on the earth’s surface, h0

the height of the mountain, and R the radius of the mountain. Since mountains

are actually quite flat (or more precisely, their heights are small compared to the

horizontal extent), we use meter as length unit for vertical distances (z direction)

and km as length unit for horizontal distances (x and y coordinates). Prior to all

code below we have initialized h0 and R with the following values: h0 D 2277 m

and R D 4 km.

Grid for 2D scalar fields Before we can plot h.x; y/, we need to create a rectan-

gular grid in the xy plane with all the points used for plotting. Regardless of which

plotting package we will use later on, the grid can be made as follows:

x = y = np.linspace(-10., 10., 41)

xv, yv = np.meshgrid(x, y, indexing=’ij’, sparse=False)

hv = h0/(1 + (xv\*\*2+yv\*\*2)/(R\*\*2))

The grid is based on equally spaced coordinates x and y in the interval Œ10; 10

km. Note the mysterious extra parameters to meshgrid here, which are needed

in order for the coordinates to have the right order such that the arithmetics in the

expression for hv becomes correct. The expression computes the surface value at

the 41 41 grid points in one vectorized operation.

A surface plot of a 2D scalar field h.x; y/ is a visualization of the surface z D

h.x; y/ in three-dimensional space. Most plotting packages have functions which

can be used to create surface plots of 2D scalar fields. These can be either wireframe

plots, where only lines connecting the grid points are drawn, or plots where the faces

of the surface are colored. In Fig. 5.12 we have shown two such plots of the surface

h.x; y/. Section 5.11.1 presents the code which generates these plots.

5.10.3 Parameterized Curve

To illustrate the plotting of three-dimensional parameterized curves, we consider

a trajectory that represents a circular climb to the top of the mountain:

t

t

cos.t/ i C 10 1

sin.t/ j

r.t/ D 10 1

2

2

h0

k:

C

2

1 C 100.1tR=.2//

2

(5.14)294

5 Array Computing and Curve Plotting

Fig. 5.12 Two different plots of a mountain. The right plot also shows a trajectory to the top of

the mountain

Here i , j , and k denote the unit vectors in the x-, y-, and z-directions, respectively.

The coordinates of r.t/ can be produced by

s = np.linspace(0, 2\*np.pi, 100)

curve\_x = 10\*(1 - s/(2\*np.pi))\*np.cos(s)

curve\_y = 10\*(1 - s/(2\*np.pi))\*np.sin(s)

curve\_z = h0/(1 + 100\*(1 - s/(2\*np.pi))\*\*2/(R\*\*2))

The parameterized curve is shown together with the surface h.x; y/ in the right plot

in Fig. 5.12.

5.10.4 Contour Lines

Contour lines are lines defined by the implicit equation h.x; y/ D C , where C is

some constant representing the contour level. Normally, we let C run over some

equally spaced values, and very often, the plotting program computes the C values.

To distinguish contours, one often associates each contour level C with its own

color.

Figure 5.13 shows different ways contour lines can be used to visualize the

surface h.x; y/. The first and last plot are visualizations utilizing two spatial dimen-

sions. The first draws a small set of contour lines only, while the last one displays

the surface as an image, whose colors reflect the values of the field, or equivalently,

the height of the surface. The third plot actually combines three different types of

contours, each type corresponding to keeping a coordinate constant and projecting

the contours on a “wall”. The code used to generate these plots is presented in

Sect. 5.11.2.

5.10.5 The Gradient Vector Field

The gradient vector field rh of a 2D scalar field h.x; y/ is defined by

rh D

@h

@h

iC

j:

@x

@y

(5.15)5.10 Plotting of Scalar and Vector Fields

295

Fig. 5.13 Different types of contour plots of a 2D scalar field in two and three dimensions

One learns in vector calculus that the gradient points in the direction where h in-

creases most, and that the gradients are orthogonal to the contour lines. This is

something we can easily illustrate by creating 2D plots of the contours and the gra-

dient field. A challenge in making such plots is to get the right arrow lengths so

that the arrows are well visible, but they do not collide and make a cluttered visual

impression. Since the arrows are drawn at each point in a 2D grid, one way of

controlling the number of arrows is to control the resolution of the grid.

So, let us create a grid with 20 instead of 40 intervals in the horizontal directions:

x2 = y2 = np.linspace(-10.,10.,11)

x2v, y2v = np.meshgrid(x2, y2, indexing=’ij’, sparse=False)

h2v = h0/(1 + (x2v\*\*2 + y2v\*\*2)/(R\*\*2)) # h on coarse grid

The gradient vector field of h.x; y/ can now be computed using the function

np.gradient:

dhdx, dhdy = np.gradient(h2v) # dh/dx, dh/dy

The gradient field (5.15) together with the contours appear in Fig. 5.14, from which

the orthogonality can be easily seen. Section 5.11.3 explains the code needed to

make this plot.296

5 Array Computing and Curve Plotting

Fig. 5.14 Gradient field with contour plot

5.11 Matplotlib

We import any visualization package under the name plt, so for Matplotlib the

import is done by

import matplotlib.pyplot as plt

When creating two-dimensional plots of scalar and vector fields, we shall make use

of a Matplotlib Axes object, named ax and made by

fig = plt.figure(1)

ax = fig.gca()

# Get current figure

# Get current axes

For three-dimensional visualization, we need the following alternative lines:

from mpl\_toolkits.mplot3d import Axes3D

fig = plt.figure(1)

ax = fig.gca(projection=’3d’)

5.11.1 Surface Plots

The Matplotlib functions for producing surface plots of 2D scalar fields are

ax.plot\_wireframe and ax.plot\_surface. The first one produces a wire-5.11 Matplotlib

297

frame plot, and the second one colors the surface. The following code uses the

functions to produce the plots shown in Fig. 5.12, once the grid has been defined as

in Sect. 5.10.2, and the coordinates of the parameterized curve have been computed

as in Sect. 5.10.3.

fig = plt.figure(1)

ax = fig.gca(projection=’3d’)

ax.plot\_wireframe(xv, yv, hv, rstride=2, cstride=2)

# Simple plot of mountain and parametric curve

fig = plt.figure(2)

ax = fig.gca(projection=’3d’)

from matplotlib import cm

ax.plot\_surface(xv, yv, hv, cmap=cm.coolwarm,

rstride=1, cstride=1)

# add the parametric curve. linewidth controls the width of the curve

ax.plot(curve\_x, curve\_y, curve\_z, linewidth=5)

Recall that a final plt.show() command is necessary to force Matplotlib to show

a plot on the screen.

Note that the second plot in this figure is drawn using a finer grid. This is con-

trolled with the rstride and cstride parameters, which sets the number of grid

lines in each direction. Setting one of these to 1 means that a grid line is drawn for

every value in the grid in the corresponding direction, and setting to 2 means that

a grid line will be drawn for every two values in the grid. You will normally need

to experiment with such parameters to get a visually attractive plot.

A surface with colors reflecting the height of the surface needs specification of

a color map, which is a mapping between function values and colors. Above we

applied the common coolwarm scheme which goes from blue (“cool” color for

minimum values) to red (“warm” color for maximum values). There are lots of

colormaps to choose from, and you have to experiment to find appropriate choices

according to your taste and to the problem at hand.

To the latter plot we also added the parameterized curve r.t/, defined by (5.14),

using the command plot. The attribute linewidth is increased here in order to

make the curve thicker and more visible. By default, Matplotlib adds plots to each

other without any need for plt.hold(’on’), although such a command can indeed

be used.

5.11.2

Contour Plots

The following code exemplifies different types of contour plots. The first two

plots (default two-dimensional and three-dimensional contour plots) are shown in

Fig. 5.13. The next four plots appear in Fig. 5.15. Note that, when we asked Mat-

plotlib to plot 10 contours, the response was, surprisingly, 9 contour lines, where

one of the contours was incomplete. This kind of behavior may also be found in

other plotting packages (such as MATLAB): the package will do its best to plot

the requested number of complete contour lines, but there is no guarantee that this

number is achieved exactly.298

5 Array Computing and Curve Plotting

Fig. 5.15 Some other contour plots with Matplotlib: 10 contour lines (upper left), 10 black contour

lines (upper right), specified contour levels (lower left), and labeled levels (lower right)

# Default two-dimensional contour plot with 7 colored lines

fig = plt.figure(3)

ax = fig.gca()

ax.contour(xv, yv, hv)

plt.axis(’equal’)

# Default three-dimensional contour plot

fig = plt.figure(4)

ax = fig.gca(projection=’3d’)

ax.contour(xv, yv, hv)

# Plot of mountain and contour lines projected on the

# coordinate planes

fig = plt.figure(5)

ax = fig.gca(projection=’3d’)

ax.plot\_surface(xv, yv, hv, cmap=cm.coolwarm,

rstride=1, cstride=1)

# zdir is the projection axis

# offset is the offset of the projection plane

ax.contour(xv, yv, hv, zdir=’z’, offset=-1000, cmap=cm.coolwarm)

ax.contour(xv, yv, hv, zdir=’x’, offset=-10,

cmap=cm.coolwarm)

ax.contour(xv, yv, hv, zdir=’y’, offset=10,

cmap=cm.coolwarm)

# View the contours by displaying as an image

fig = plt.figure(6)

ax = fig.gca()

ax.imshow(hv)5.12 Mayavi

299

# 10 contour lines (equally spaced contour levels)

fig = plt.figure(7)

ax = fig.gca()

ax.contour(xv, yv, hv, 10)

plt.axis(’equal’)

# 10 black (’k’) contour lines

fig = plt.figure(8)

ax = fig.gca()

ax.contour(xv, yv, hv, 10, colors=’k’)

plt.axis(’equal’)

# Specify the contour levels explicitly as a list

fig = plt.figure(9)

ax = fig.gca()

levels = [500., 1000., 1500., 2000.]

ax.contour(xv, yv, hv, levels=levels)

plt.axis(’equal’)

# Add labels with the contour level for each contour line

fig = plt.figure(10)

ax = fig.gca()

cs = ax.contour(xv, yv, hv)

plt.clabel(cs)

plt.axis(’equal’)

5.11.3 Vector Field Plots

The code for plotting the gradient field (5.15) together with contours goes as ex-

plained below, once the grid has been defined as in Sect. 5.10.5. The corresponding

plot is shown in Fig. 5.14.

fig = plt.figure(11)

ax = fig.gca()

ax.quiver(x2v, y2v, dhdx, dhdy, color=’r’,

angles=’xy’, scale\_units=’xy’)

ax.contour(xv, yv, hv)

plt.axis(’equal’)

5.12 Mayavi

Mayavi is an advanced, free, easy to use, scientific data visualizer, with an emphasis

on three-dimensional visualization techniques. The package is written in Python,

and uses the Visualization Toolkit (VTK) in C++ for rendering graphics. Since

VTK can be configured with different backends, so can Mayavi. Mayavi is cross

platform and runs on most platforms, including Mac OS X, Windows, and Linux.

The web page http://docs.enthought.com/mayavi/mayavi/ collects pointers to all

relevant documentation of Mayavi. We shall primarily deal with the mayavi.mlab

module, which provides a simple interface to plotting of 2D scalar and vector fields

with commands that mimic those of MATLAB. Let us import this module under

our usual name plt for a plotting package:

import mayavi.mlab as plt300

5 Array Computing and Curve Plotting

The official documentation of the mlab module is provided in two places, one for

the basic functionality8 and one for further functionality9 . Basic figure handling10 is

very similar to the one we know from Matplotlib. Just as for Matplotlib, all plotting

commands you do in mlab will go into the same figure, until you manually change

to a new figure.

5.12.1 Surface Plots

Mayavi has the functions mesh and surf for producing surface plots. These are

similar, but surf assumes an orthogonal grid, and uses this assumption to make

efficient data structures, while mesh makes no such assumptions on the grid. Here

we only use orthogonal grids and hence apply surf. The following code plots the

surface h.x; y/ in (5.13), as well as the parameterized curve r.t/ in (5.14). The

resulting graphics appears in Fig. 5.16.

# Create a figure with white background and black foreground

plt.figure(1, fgcolor=(.0, .0, .0), bgcolor=(1.0, 1.0, 1.0))

# ’representation’ sets type of plot, here a wireframe plot

plt.surf(xv, yv, hv, extent=(0,1,0,1,0,1),

representation=’wireframe’)

# Decorate axes (nb\_labels is the number of labels used

# in each direction)

plt.axes(xlabel=’x’, ylabel=’y’, zlabel=’z’, nb\_labels=5,

color=(0., 0., 0.))

# Decorate the plot with a title

plt.title(’h(x,y)’, size=0.4)

# Simple plot of mountain and parametric curve.

plt.figure(2, fgcolor=(.0, .0, .0), bgcolor=(1.0, 1.0, 1.0))

# Here, representation has default: colored surface elements

plt.surf(xv, yv, hv, extent=(0,1,0,1,0,1))

# Add the parametric curve. tube\_radius is the width of the

# curve (use ’extent’ for auto-scaling)

plt.plot3d(curve\_x, curve\_y, curve\_z, tube\_radius=0.2,

extent=(0,1,0,1,0,1))

plt.figure(3, fgcolor=(.0, .0, .0), bgcolor=(1.0, 1.0, 1.0))

# Use ’warp\_scale’ for vertical scaling

plt.surf(xv, yv, hv, warp\_scale=0.01, color=(.5, .5, .5))

plt.plot3d(curve\_x, curve\_y, 0.01\*curve\_z, tube\_radius=0.2)

surf can produce wireframe plots, as well as plots where the faces of the surface

are colored. The parameter representation controls this, as exemplified in the

first two plots. The first plot was also decorated with axes and a title.

The calls to plt.figure() take three parameters: First the usual index for the

plot, then two tuples of numbers , representing the RGB-values to be used for the

foreground (fgcolor) and the background (bgcolor). White and black are (1,1,1)

8

http://docs.enthought.com/mayavi/mayavi/auto/mlab\_helper\_functions.html

http://docs.enthought.com/mayavi/mayavi/auto/mlab\_other\_functions.html

10

http://docs.enthought.com/mayavi/mayavi/auto/mlab\_figure.html

95.12 Mayavi

301

Fig. 5.16 Surface plots produced with the surf function of Mayavi: The curve r.t / is also shown

in the two last plots

and (0,0,0), respectively. The foreground color is used for text and labels included

in the plot. The color attribute in plt.surf adjusts the surface so that it is colored

with small variations from the provided base color, here (.5, .5, .5).

The command plot3d is used to plot the curve r.t/. We have here increased the

attribute tube\_radius to make the curve thicker and more visible.

Mayavi does no auto-scaling of the axes by default (contrary to Matplotlib), so

if the magnitudes in the vertical and horizontal directions are very different, as they

are for h.x; y/, the plots may be very concentrated in one direction. We therefore

need to apply some auto-scaling procedure. In Fig. 5.16 two such procedures are

exemplified. In the first two plots the parameter extent is used. It tells Mayavi

to auto-scale the surface and curve to fit the contents described by the six listed

values (we will return to what these values mean when we have a more illustrating

example). Since the curve and the surface span different areas in space, we see

that they are auto-scaled differently in the second plot, with the undesired effect

that r.t/ is not drawn on the surface. The last plot has avoided this problem by

using the warp\_scale parameter for scaling the vertical direction. Not all Mayavi

functions accept this parameter. A remedy for this is to scale the z-coordinates

manually, as here exemplified in the last plot3d-call. As is seen, the curve is302

5 Array Computing and Curve Plotting

drawn correctly with respect to the surface in the last plot. In the following we will

use the warp\_scale parameter to avoid such auto-scaling problems.

Subplots The two plots in Fig. 5.16 were created as separate figures. One can also

create them as subplots within one figure:

plt.figure(4, fgcolor=(.0, .0, .0), bgcolor=(1.0, 1.0, 1.0))

plt.mesh(xv, yv, hv, extent=(0, 0.25, 0, 0.25, 0, 0.25),

colormap=’cool’)

plt.outline(plt.mesh(

xv, yv, hv,

extent=(0.375, 0.625, 0, 0.25, 0, 0.25),

colormap=’Accent’))

plt.outline(plt.mesh(

xv, yv, hv, extent=(0.75, 1, 0, 0.25, 0, 0.25),

colormap=’prism’), color=(.5, .5, .5))

The result is shown in Fig. 5.17. Three separate mesh commands are run, each

producing a new plot in the current figure. The commands use different values for

the colormap attribute to color the surface in different ways. When this attribute

is not provided, as in the code producing the two first plots in Fig. 5.16, a default

colormap is used.

The plt.outline command is used to create a frame around the subplots, and

as seen, we exemplify this possibility for the last two subplots, but not the first one.

We see that one of the two frames has a different color, obtained by setting the

color attribute of the plt.outline command.

From the computer code it is hopefully clear that the six values listed in extent

represent fractions of the cube (0,1,0,1,0,1), where the corresponding plots are

placed. The extents for the three plots are here defined such that they do not overlap.

Fig. 5.17 A plot with three subplots created with Mayavi5.12 Mayavi

5.12.2

303

Contour Plots

The following code exemplifies how one can produce contour plots with Mayavi.

The code is very similar to that of Matplotlib, but one difference is that the attribute

contours now can represent the number of levels, as well as the levels themselves.

The plots are shown in Fig. 5.18.

# Default contour plot plotted together with surf.

plt.figure(5, fgcolor=(.0, .0, .0), bgcolor=(1.0, 1.0, 1.0))

plt.surf(xv, yv, hv, warp\_scale=0.01)

plt.contour\_surf(xv, yv, hv, warp\_scale=0.01)

# 10 contour lines (equally spaced contour levels).

plt.figure(6, fgcolor=(.0, .0, .0), bgcolor=(1.0, 1.0, 1.0))

plt.contour\_surf(xv, yv, hv, contours=10, warp\_scale=0.01)

# 10 contour lines (equally spaced contour levels) together

# with surf. Black color for contour lines.

plt.figure(7, fgcolor=(.0, .0, .0), bgcolor=(1.0, 1.0, 1.0))

plt.surf(xv, yv, hv, warp\_scale=0.01)

plt.contour\_surf(xv, yv, hv, contours=10, color=(0., 0., 0.),

warp\_scale=0.01)

# Specify the contour levels explicitly as a list.

plt.figure(8, fgcolor=(.0, .0, .0), bgcolor=(1.0, 1.0, 1.0))

levels = [500., 1000., 1500., 2000.]

plt.contour\_surf(xv, yv, hv, contours=levels, warp\_scale=0.01)

# View the contours by displaying as an image.

plt.figure(9, fgcolor=(.0, .0, .0), bgcolor=(1.0, 1.0, 1.0))

plt.imshow(hv)

Note that there is no function in Mayavi which labels the contours.

Contour plots in Mayavi are shown in three-dimensional space, but you can ro-

tate and look at them from above if you want a two-dimensional plot. Their visual

appearance may be enhanced by also including the surface plot itself. We have

done this for the top and middle left plots in Fig. 5.18. There is a clear difference

in visual impression between these two plots: in the first one, default surface- and

contour coloring is used, resulting in less visible contours, but in the middle left

plot (plt.figure 6), we set black contours to make them better stand out.

5.12.3

Vector Field Plots

Mayavi supports only vector fields in three-dimensional space. We will therefore

visualize the two-dimensional gradient field (5.15) by adding a third component of

zero. The following code plots this gradient field together with the contours of h.

plt.figure(11, fgcolor=(.0, .0, .0), bgcolor=(1.0, 1.0, 1.0))

plt.contour\_surf(xv, yv, hv, contours=20, warp\_scale=0.01)304

5 Array Computing and Curve Plotting

Fig. 5.18 Some contour plots with Mayavi

# mode controls the style how vectors are drawn

# color controls the colors of the vectors

# scale\_mode=’none’ ensures that vectors are drawn with the same length

plt.quiver3d(x2v, y2v, 0.01\*h2v, dhdx, dhdy, np.zeros\_like(dhdx),

mode=’arrow’, color=(1,0,0), scale\_mode=’none’)

This will produce a 3D view, which we again can rotate to obtain a 2D view. The

result is shown in Fig. 5.19, which is similar to Fig. 5.14.

5.12.4

A 3D Scalar Field and Its Gradient Field

Mayavi has functionality for drawing contour surfaces of 3D scalar fields. Let us

consider the 3D scalar field

g.x; y; z/ D z h.x; y/:

A three-dimensional grid for g can be computed as follows.

x = y = np.linspace(-10.,10.,41)

z = np.linspace(0, 50, 41)

(5.16)5.12 Mayavi

305

Fig. 5.19 Gradient field with contour plot

xv, yv, zv = np.meshgrid(x, y, z,

sparse=False, indexing=’ij’)

hv = 0.01\*h0/(1 + (xv\*\*2+yv\*\*2)/(R\*\*2))

gv = zv - hv

The contours are now surfaces defined by the implicit equation g.x; y; z/ D C ,

corresponding to vertical shifts of the surface h.x; y/.

A corresponding vector field can be calculated:

rg D

@g

@g

@g

iC

jC

k:

@x

@y

@z

(5.17)

numpy’s gradient function can be used to compute a gradient vector field in 3D as

well, but you need a three-dimensional grid for the field as input. For the field

(5.16), the gradient field is computed as follows.

x2 = y2 = np.linspace(-10.,10.,5)

z2 = np.linspace(0, 50, 5)

x2v, y2v, z2v = np.meshgrid(x2, y2, z2,

indexing=’ij’, sparse=False)

h2v = 0.01\*h0/(1 + (x2v\*\*2 + y2v\*\*2)/(R\*\*2))

g2v = z2v - h2v

dhdx, dhdy, dhdz = np.gradient(g2v)

Again we have used a coarser grid for the vector field.

To visualize the field (5.16) and its gradient field together, we draw enough con-

tours, as we did in the 2D case in Fig. 5.14. The following code can be used.

plt.figure(12, fgcolor=(.0, .0, .0), bgcolor=(1.0, 1.0, 1.0))

# opacity controls how contours are visible through each other

plt.contour3d(xv, yv, zv, gv, contours=7, opacity=0.5)

# scale\_mode=’none’: vectors should not be scaled

plt.quiver3d(x2v, y2v, z2v, dhdx, dhdy, dhdz, mode=’arrow’,

scale\_mode=’none’, opacity=0.5)

The result is shown in Fig. 5.20.306

5 Array Computing and Curve Plotting

Fig. 5.20 The 3D scalar field (5.16) and its gradient field

This example demonstrates some of the challenges in plotting three-dimensional

vector fields. The vectors must not be too dense, and not too long. It is inevitable

that contours shadow one another. Fortunately, Mayavi supports an opacity setting,

which controls how contours are visible through each other. Visualizing a 3D scalar

field is clearly challenging, and we have only touched the subject.

5.12.5 Animations

It is straightforward to create animations with Mayavi. In the following code the

function h.x; y/ is scaled vertically, for different scaling constants between 0 and 1,

and each plot is saved in its own file. The files can then be combined to a standard

video file.

plt.figure(13, fgcolor=(.0, .0, .0), bgcolor=(1.0, 1.0, 1.0))

s = plt.surf(xv, yv, hv, warp\_scale=0.01)

for i in range(10):

# s.mlab\_source.scalars is a handle for the values of the surface,

# and is updated here

s.mlab\_source.scalars = hv\*0.1\*(i+1)

plt.savefig(’tmp\_%04d.png’ % i)5.13 Summary

5.13

307

Summary

5.13.1 Chapter Topics

This chapter has introduced computing with arrays and plotting curve data stored in

arrays. The Numerical Python package contains lots of functions for array comput-

ing, including the ones listed in the table below. Plotting has been done with tools

that closely resemble the syntax of MATLAB.

Construction

array(ld)

asarray(d)

zeros(n)

zeros(n, int)

zeros((m,n))

zeros\_like(x)

linspace(a,b,m)

a.shape

a.size

len(a)

a.dtype

a.reshape(3,2)

a[i]

a[i,j]

a[1:k]

a[1:10:3]

b = a.copy()

sin(a), exp(a), ...

c = concatenate((a, b))

c = where(cond, a1, a2)

isinstance(a, ndarray)

Meaning

copy list data ld to a numpy array

make array of data d (no data copy if already array)

make a float vector/array of length n, with zeros

make an int vector/array of length n with zeros

make a two-dimensional float array with shape (m,‘n‘)

make array of same shape and element type as x

uniform sequence of m numbers in Œa; b

tuple containing a’s shape

total no of elements in a

length of a one-dim. array a (same as a.shape[0])

the type of elements in a

return a reshaped as 3 2 array

vector indexing

two-dim. array indexing

slice: reference data with indices 1,2,...,k-1

slice: reference data with indices 1,4,7

copy an array

numpy functions applicable to arrays

c contains a with b appended

c[i] = a1[i] if cond[i], else c[i] = a2[i]

is True if a is an array

Array computing When we apply a Python function f(x) to a Numerical Python

array x, the result is the same as if we apply f to each element in x separately.

However, when f contains if statements, these are in general invalid if an array

x enters the boolean expression. We then have to rewrite the function, often by

applying the where function from Numerical Python.

Plotting curves Sections 5.3.1 and 5.3.2 provide a quick overview of how to plot

curves with the aid of Matplotlib. The same examples coded with the Easyviz

plotting interface appear in Sect. 5.3.3.

Making movies Each frame in a movie must be a hardcopy of a plot in PNG

format. These plot files should have names containing a counter padded with lead-

ing zeros. One example may be tmp\_0000.png, tmp\_0001.png, tmp\_0002.png.308

5 Array Computing and Curve Plotting

Having the plot files with names on this form, we can make an animated GIF movie

in the file movie.gif, with two frames per second, by

os.system(’convert -delay 50 tmp\_\*.png movie.gif’)

Alternatively, we may combine the plot files to a Flash video:

os.system(’ffmpeg -r 5 -i tmp\_%04d.png -vcodec flv movie.flv’)

Other formats can be made using other codecs, see Sect. 5.3.5.

Terminology The important topics in this chapter are

array computing

vectorization

plotting

animations

5.13.2 Example: Animating a Function

Problem In this chapter’s summarizing example we shall visualize how the tem-

perature varies downward in the earth as the surface temperature oscillates between

high day and low night values. One question may be: What is the temperature

change 10 m down in the ground if the surface temperature varies between 2 C in

the night and 15 C in the day?

Let the z axis point downwards, towards the center of the earth, and let z D 0

correspond to the earth’s surface. The temperature at some depth z in the ground

at time t is denoted by T .z; t/. If the surface temperature has a periodic variation

around some mean value T0 , according to

T .0; t/ D T0 C A cos.!t/;

one can find, from a mathematical model for heat conduction, that the temperature

at an arbitrary depth is

r

T .z; t/ D T0 C Ae

az

cos.!t az/;

aD

!

:

2k

(5.18)

The parameter k reflects the ground’s ability to conduct heat (k is called the thermal

diffusivity or the heat conduction coefficient).

The task is to make an animation of how the temperature profile in the ground,

i.e., T as a function of z, varies in time. Let ! correspond to a time period of 24

hours. The mean temperature T0 is taken as 10 C, and the maximum variation A

is assumed to be 10 C. The heat conduction coefficient k may be set as 1 mm2 =s

(which is 106 m2 =s in proper SI units).5.13 Summary

309

Solution To animate T .z; t/ in time, we need to make a loop over points in time,

and in each pass in the loop we must save a plot of T , as a function of z, to file. The

plot files can then be combined to a movie. The algorithm becomes

for ti D it, i D 0; 1; 2 : : : ; n:

– plot the curve y.z/ D T .z; ti /

– store the plot in a file

combine all the plot files into a movie

It can be wise to make a general animate function where we just feed in some

f .x; t/ function and make all the plot files. If animate has arguments for setting

the labels on the axis and the extent of the y axis, we can easily use animate also

for a function T .z; t/ (we just use z as the name for the x axis and T as the name

for the y axis in the plot). Recall that it is important to fix the extent of the y

axis in a plot when we make animations, otherwise most plotting programs will

automatically fit the extent of the axis to the current data, and the tick marks on

the y axis will jump up and down during the movie. The result is a wrong visual

impression of the function.

The names of the plot files must have a common stem appended with some frame

number, and the frame number should have a fixed number of digits, such as 0001,

0002, etc. (if not, the sequence of the plot files will not be correct when we specify

the collection of files with an asterisk for the frame numbers, e.g., as in tmp\*.png).

We therefore include an argument to animate for setting the name stem of the

plot files. By default, the stem is tmp\_, resulting in the filenames tmp\_0000.png,

tmp\_0001.png, tmp\_0002.png, and so forth. Other convenient arguments for the

animate function are the initial time in the plot, the time lag t between the plot

frames, and the coordinates along the x axis. The animate function then takes the

form

def animate(tmax, dt, x, function, ymin, ymax, t0=0,

xlabel=’x’, ylabel=’y’, filename=’tmp\_’):

t = t0

counter = 0

while t <= tmax:

y = function(x, t)

plot(x, y, ’-’,

axis=[x[0], x[-1], ymin, ymax],

title=’time=%2d h’ % (t/3600.0),

xlabel=xlabel, ylabel=ylabel,

savefig=filename + ’%04d.png’ % counter)

savefig(’tmp\_%04d.pdf’ % counter)

t += dt

counter += 1

The T .z; t/ function is easy to implement, but we need to decide whether the

parameters A, !, T0 , and k shall be arguments to the Python implementation of

T .z; t/ or if they shall be global variables. Since the animate function expects

that the function to be plotted has only two arguments, we must implement T .z; t/

as T(z,t) in Python and let the other parameters be global variables (Sects. 7.1.1

and 7.1.2 explain this problem in more detail and present a better implementation).

The T(z,t) implementation then reads310

5 Array Computing and Curve Plotting

def T(z, t):

# T0, A, k, and omega are global variables

a = sqrt(omega/(2\*k))

return T0 + A\*exp(-a\*z)\*cos(omega\*t - a\*z)

Suppose we plot T .z; t/ at n points for z 2 Œ0; D. We make such plots for

t 2 Œ0; tmax with a time lag t between the them. The frames in the movie are now

made by

# set T0, A, k, omega, D, n, tmax, dt

z = linspace(0, D, n)

animate(tmax, dt, z, T, T0-A, T0+A, 0, ’z’, ’T’)

We have here set the extent of the y axis in the plot as ŒT0 A; T0 C A, which is in

accordance with the T .z; t/ function.

The call to animate above creates a set of files with names of the form

tmp\_\*.png. Out of these files we can create an animated GIF movie or a video

in, e.g., Flash format by running operating systems commands with convert and

avconv (or ffmpeg):

os.system(’convert -delay 50 tmp\_\*.png movie.gif’)

os.system(’avconv -i tmp\_%04d.png -r 5 -vcodec flv movie.flv’)

See Sect. 5.3.5 for how to create videos in other formats.

It now remains to assign proper values to all the global variables in the pro-

gram: n, D, T0, A, omega, dt, tmax, and k. The oscillation period is 24 hours,

and ! is related to the period P of the cosine function by ! D 2=P (realize

that cos.t2=P / has period P ). We then express P D 24 h as 24 60 60 s and

compute ! as 2=P 7 105 s1 . The total simulation time can be 3 periods,

i.e., tmax D 3P . The T .z; t/ function decreases exponentially with the depth z so

there is no point in having the maximum depth D larger than the depth where T is

visually zero, say 0.001. We have that e aD D 0:001 when D D a1 ln 0:001, so

we can use this estimate in the program. The proper initialization of all parameters

can then be expressed as follows:

k = 1E-6

# thermal diffusivity (in m\*m/s)

P = 24\*60\*60.

# oscillation period of 24 h (in seconds)

omega = 2\*pi/P

dt = P/24

# time lag: 1 h

tmax = 3\*P

# 3 day/night simulation

T0 = 10

# mean surface temperature in Celsius

A = 10

# amplitude of the temperature variations in Celsius

a = sqrt(omega/(2\*k))

D = -(1/a)\*log(0.001) # max depth

n = 501

# no of points in the z direction

Note that it is very important to use consistent units. Here we express all units in

terms of meter, second, and Kelvin or Celsius.5.13 Summary

311

We encourage you to run the program heatwave.py to see the movie. The

hardcopy of the movie is in the file movie.gif. Figure 5.21 displays two snapshots

in time of the T .z; t/ function.

Fig. 5.21 Plot of the temperature T .z; t / in the ground for two different t values

Scaling In this example, as in many other scientific problems, it was easier to

write the code than to assign proper physical values to the input parameters in the

program. To learn about the physical process, here how heat propagates from the

surface and down in the ground, it is often advantageous to scale the variables in

the problem so that we work with dimensionless variables. Through the scaling

procedure we normally end up with much fewer physical parameters that must be

assigned values. Let us show how we can take advantage of scaling the present

problem.

Consider a variable x in a problem with some dimension. The idea of scaling

is to introduce a new variable xN D x=xc , where xc is a characteristic size of x.

Since x and xc have the same dimension, the dimension cancels in xN such that xN

is dimensionless. Choosing xc to be the expected maximum value of x, ensures

that xN 1, which is usually considered a good idea. That is, we try to have all

dimensionless variables varying between zero and one. For example, we can intro-

duce a dimensionless z coordinate: zN D z=D, and now zN 2 Œ0; 1. Doing a proper

scaling of a problem is challenging so for now it is sufficient to just follow the steps

below – and not worry why we choose a certain scaling.

In the present problem we introduce these dimensionless variables:

zN D z=D

T T0

TN D

A

Nt D !t

We now insert z D zN D and t D tN=! in the expression for T .z; t/ and get

T D T0 C Ae b zN cos.tN b zN /;

or

b D aD

T T0

TN .Nz ; tN/ D

D e b zN cos.tN b zN / :

A312

5 Array Computing and Curve Plotting

We see that TN depends on only one dimensionless parameter b in addition to the

independent dimensionless variables zN and tN. It is common practice at this stage of

the scaling to just drop the bars and write

T .z; t/ D e bz cos.t bz/ :

(5.19)

This function is much simpler to plot than the one with lots of physical parameters,

because now we know that T varies between 1 and 1, t varies between 0 and 2

for one period, and z varies between 0 and 1. The scaled temperature has only one

parameter b in addition to the independent variable. That is, the shape of the graph

is completely determined by b.

In our previous movie example,

we used specific values for D, !, and k, which

p

then implies a certain b D D !=.2k/ ( 6:9). However, we can now run different

b values and see the effect on the heat propagation. Different b values will in

our problems imply different periods of the surface temperature variation and/or

different heat conduction values in the ground’s composition of rocks. Note that

doubling ! and k leaves the same b – it is only the fraction !=k that influences the

value of b.

We can reuse the animate function also in the scaled case, but we need to make

a new T .z; t/ function and, e.g., a main program where b can be read from the

command line:

def T(z, t):

return exp(-b\*z)\*cos(t - b\*z)

# b is global

b = float(sys.argv[1])

n = 401

z = linspace(0, 1, n)

animate(3\*2\*pi, 0.05\*2\*pi, z, T, -1.2, 1.2, 0, ’z’, ’T’)

movie(’tmp\_\*.png’, encoder=’convert’, fps=2,

output\_file=’tmp\_heatwave.gif’)

os.system(’convert -delay 50 tmp\_\*.png movie.gif’)

Running the program, found as the file heatwave\_scaled.py, for different b

values shows that b governs how deep the temperature variations on the surface

z D 0 penetrate. A large b makes the temperature changes confined to a thin layer

close to the surface, while a small b leads to temperature variations also deep down

in the ground. You are encouraged to run the program with b D 2 and b D 20 to

experience the major difference, or just view the ready-made animations11 .

We can understand the results from a physical perspective. Think of increasing

!, which means reducing the oscillation period so we get a more rapid temperature

variation. To preserve the value of b we must increase k by the same factor. Since

a large k means that heat quickly spreads down in the ground, and a small k implies

the opposite, we see that more rapid variations at the surface requires a larger k to

more quickly conduct the variations down in the ground. Similarly, slow tempera-

ture variations on the surface can penetrate deep in the ground even if the ground’s

ability to conduct (k) is low.

11

http://hplgit.github.io/scipro-primer/video/heatwave.html5.14 Exercises

313

5.14 Exercises

Exercise 5.1: Fill lists with function values

Define

1

1 2

h.x/ D p e 2 x :

2

(5.20)

Fill lists xlist and hlist with x and h.x/ values for 41 uniformly spaced x coor-

dinates in Œ4; 4.

Hint You may adapt the example in Sect. 5.2.1.

Filename: fill\_lists.

Exercise 5.2: Fill arrays; loop version

The aim is to fill two arrays x and y with x and h.x/ values, respectively, where

h.x/ is defined in (5.20). Let the x values be as in Exercise 5.1. Create empty x and

y arrays and compute each element in x and y with a for loop.

Filename: fill\_arrays\_loop.

Exercise 5.3: Fill arrays; vectorized version

Vectorize the code in Exercise 5.2 by creating the x values using the linspace

function from the numpy package and by evaluating h.x/ for an array argument.

Filename: fill\_arrays\_vectorized.

Exercise 5.4: Plot a function

Make a plot of the function in Exercise 5.1 for x 2 Œ4; 4.

Filename: plot\_Gaussian.

Exercise 5.5: Apply a function to a vector

Given a vector v D .2; 3; 1/ and a function f .x/ D x 3 C xe x C 1, apply f to

each element in v. Then calculate by hand f .v/ as the NumPy expression v\*\*3 +

v\*exp(v) + 1 using vector computing rules. Demonstrate that the two results are

equal.

Filename: apply\_vecfunc.

Exercise 5.6: Simulate by hand a vectorized expression

Suppose x and t are two arrays of the same length, entering a vectorized expression

y = cos(sin(x)) + exp(1/t)

If x holds two elements, 0 and 2, and t holds the elements 1 and 1.5, calculate

by hand (using a calculator) the y array. Thereafter, write a program that mimics

the series of computations you did by hand (typically a sequence of operations of

the kind we listed in Sect. 5.1.3 – use explicit loops, but at the end you can use

Numerical Python functionality to check the results).

Filename: simulate\_vector\_computing.314

5 Array Computing and Curve Plotting

Exercise 5.7: Demonstrate array slicing

Create an array w with values 0; 0:1; 0:2; : : : ; 3. Write out w[:], w[:-2], w[::5],

w[2:-2:6]. Convince yourself in each case that you understand which elements of

the array that are printed.

Filename: slicing.

Exercise 5.8: Replace list operations by array computing

The data analysis problem in Sect. 2.6.2 is solved by list operations. Convert the

list to a two-dimensional array and perform the computations using array operations

(i.e., no explicit loops, but you need a loop to make the printout).

Filename: sun\_data\_vec.

Exercise 5.9: Plot a formula

Make a plot of the function y.t/ D v0 t 12 gt 2 for v0 D 10, g D 9:81, and

t 2 Œ0; 2v0 =g. Set the axes labels as time (s) and height (m).

Filename: plot\_ball1.

Exercise 5.10: Plot a formula for several parameters

Make a program that reads a set of v0 values from the command line and plots the

corresponding curves y.t/ D v0 t 12 gt 2 in the same figure, with t 2 Œ0; 2v0 =g for

each curve. Set g D 9:81.

Hint You need a different vector of t coordinates for each curve.

Filename: plot\_ball2.

Exercise 5.11: Specify the extent of the axes in a plot

Extend the program from Exercises 5.10 such that the minimum and maximum t

and y values are computed, and use the extreme values to specify the extent of the

axes. Add some space above the highest curve to make the plot look better.

Filename: plot\_ball3.

Exercise 5.12: Plot exact and inexact Fahrenheit-Celsius conversion formulas

A simple rule to quickly compute the Celsius temperature from the Fahrenheit de-

grees is to subtract 30 and then divide by 2: C D .F 30/=2. Compare this curve

against the exact curve C D .F 32/5=9 in a plot. Let F vary between 20 and

120.

Filename: f2c\_shortcut\_plot.

Exercise 5.13: Plot the trajectory of a ball

The formula for the trajectory of a ball is given by

f .x/ D x tan

1 gx 2

C y0 ;

2v02 cos2

(5.21)

where x is a coordinate along the ground, g is the acceleration of gravity, v0 is the

size of the initial velocity, which makes an angle with the x axis, and .0; y0 / is

the initial position of the ball.5.14 Exercises

315

In a program, first read the input data y0 , , and v0 from the command line. Then

plot the trajectory y D f .x/ for y 0.

Filename: plot\_trajectory.

Exercise 5.14: Plot data in a two-column file

The file src/plot/xy.dat12 contains two columns of numbers, corresponding to

x and y coordinates on a curve. The start of the file looks as this:

-1.0000

-0.9933

-0.9867

-0.9800

-0.9733

-0.0000

-0.0087

-0.0179

-0.0274

-0.0374

Make a program that reads the first column into a list x and the second column into

a list y. Plot the curve. Print out the mean y value as well as the maximum and

minimum y values.

Hint Read the file line by line, split each line into words, convert to float, and

append to x and y. The computations with y are simpler if the list is converted to

an array.

Filename: read\_2columns.

Remarks The function loadtxt in numpy can read files with tabular data (any

number of columns) and return the data in a two-dimensional array:

import numpy as np

# Read table of floats

data = np.loadtxt(’xy.dat’, dtype=np.float)

# Extract one-dim arrays from two-dim data

x = data[:,0] # column with index 0

y = data[:,1] # column with index 1

The present exercise asks you to implement a simplified version of loadtxt, but for

later loading of a file with tabular data into an array you will certainly use loadtxt.

Exercise 5.15: Write function data to file

We want to dump x and f .x/ values to a file, where the x values appear in the

first column and the f .x/ values appear in the second. Choose n equally spaced x

values in the interval Œa; b. Provide f , a, b, n, and the filename as input data on

the command line.

Hint You may use the StringFunction tool (see Sects. 4.3.3 and 5.5.1) to turn

the textual expression for f into a Python function. (Note that the program from

Exercise 5.14 can be used to read the file generated in the present exercise into

arrays again for visualization of the curve y D f .x/.)

Filename: write\_cml\_function.

12

http://tinyurl.com/pwyasaa/plot/xy.dat316

5 Array Computing and Curve Plotting

Exercise 5.16: Plot data from a file

The files density\_water.dat and density\_air.dat files in the folder src/

plot13 contain data about the density of water and air (respectively) for differ-

ent temperatures. The data files have some comment lines starting with # and some

lines are blank. The rest of the lines contain density data: the temperature in the

first column and the corresponding density in the second column. The goal of this

exercise is to read the data in such a file and plot the density versus the temperature

as distinct (small) circles for each data point. Let the program take the name of the

data file as command-line argument. Apply the program to both files.

Filename: read\_density\_data.

Exercise 5.17: Write table to file

Given a function of two parameters x and y, we want to create a file with a table

of function values. The left column of the table contains y values in decreasing

order as we go down the rows, and the last row contains the x values in increasing

order. That is, the first column and the last row act like numbers on an x and y

axis in a coordinate system. The rest of the table cells contains function values

corresponding to the x and y values for the respective rows and columns. For

example, if the function formula is x C 2y, x runs from 0 to 2 in steps of 0.5, and y

run from 1 to 2 in steps of 1, the table looks as follows:

2

1

0

-1

4 4.5

2 2.5

0 0.5

-2 -1.55 5.5

3 3.5

1 1.5

-1 -0.56

4

2

0

012

0.5

1.5

The task is to write a function

def write\_table\_to\_file(f, xmin, xmax, nx, ymin, ymax, ny,

width=10, decimals=None,

filename=’table.dat’):

where f is the formula, given as a Python function; xmin, xmax, ymin, and ymax are

the minimum and maximum x and y values; nx is the number of intervals in the x

coordinates (the number of steps in x direction is then (xmax-xmin)/nx); ny is the

number of intervals in the y coordinates; width is the width of each column in the

table (a positive integer); decimals is the number of decimals used when writing

out the numbers (None means no decimal specification), and filename is the name

of the output file. For example, width=10 and decimals=1 gives the output format

%10.1g, while width=5 and decimals=None implies %5g.

13

http://tinyurl.com/pwyasaa/plot5.14 Exercises

317

Here is a test function which you should use to verify the implementation:

def test\_write\_table\_to\_file():

filename = ’tmp.dat’

write\_table\_to\_file(f=lambda x, y: x + 2\*y,

xmin=0, xmax=2, nx=4,

ymin=-1, ymax=2, ny=3,

width=5, decimals=None,

filename=filename)

# Load text in file and compare with expected results

with open(filename, ’r’) as infile:

computed = infile.read()

expected = """\

2

4 4.5

5 5.5

6

1

2 2.5

3 3.5

4

0

0 0.5

1 1.5

2

-1

-2 -1.5

-1 -0.5

0

0 0.5

1 1.5

2"""

assert computed == expected

Filename: write\_table\_to\_file.

Exercise 5.18: Fit a polynomial to data points

The purpose of this exercise is to find a simple mathematical formula for how the

density of water or air depends on the temperature. The idea is to load density

and temperature data from file as explained in Exercise 5.16 and then apply some

NumPy utilities that can find a polynomial that approximates the density as a func-

tion of the temperature.

NumPy has a function polyfit(x, y, deg) for finding a best fit of a poly-

nomial of degree deg to a set of data points given by the array arguments x and

y. The polyfit function returns a list of the coefficients in the fitted polynomial,

where the first element is the coefficient for the term with the highest degree, and

the last element corresponds to the constant term. For example, given points in x

and y, polyfit(x, y, 1) returns the coefficients a, b in a polynomial a\*x + b

that fits the data in the best way. (More precisely, a line y D ax C b is a best fit

to the data points .xi ; yP

i /, i D 0; : : : ; n 1 if a and b are chosen to make the sum

2

of squared errors R D jn1

D0 .yj .axj C b// as small as possible. This approach

is known as least squares approximation to data and proves to be extremely useful

throughout science and technology.)

NumPy also has a utility poly1d, which can take the tuple or list of coefficients

calculated by, e.g., polyfit and return the polynomial as a Python function that

can be evaluated. The following code snippet demonstrates the use of polyfit and

poly1d:

coeff = polyfit(x, y, deg)

p = poly1d(coeff)

print p

# prints the polynomial expression

y\_fitted = p(x)

# computes the polynomial at the x points

# Use red circles for data points and a blue line for the polyn.

plot(x, y, ’ro’, x, y\_fitted, ’b-’,

legend=(’data’, ’fitted polynomial of degree %d’ % deg))318

5 Array Computing and Curve Plotting

a) Write a function fit(x, y, deg) that creates a plot of data in x and y arrays

along with polynomial approximations of degrees collected in the list deg as

explained above.

b) We want to call fit to make a plot of the density of water versus temperature

and another plot of the density of air versus temperature. In both calls, use

deg=[1,2] such that we can compare linear and quadratic approximations to

the data.

c) From a visual inspection of the plots, can you suggest simple mathematical for-

mulas that relate the density of air to temperature and the density of water to

temperature?

Filename: fit\_density\_data.

Exercise 5.19: Fit a polynomial to experimental data

Suppose we have measured the oscillation period T of a simple pendulum with

a mass m at the end of a massless rod of length L. We have varied L and

recorded the corresponding T value. The measurements are found in a file

src/plot/pendulum.dat14. The first column in the file contains L values and

the second column has the corresponding T values.

a) Plot L versus T using circles for the data points.

b) We shall assume that L as a function of T is a polynomial. Use the NumPy

utilities polyfit and poly1d, as explained in Exercise 5.18, to fit polynomials

of degree 1, 2, and 3 to the L and T data. Visualize the polynomial curves

together with the experimental data. Which polynomial fits the measured data

best?

Filename: fit\_pendulum\_data.

Exercise 5.20: Read acceleration data and find velocities

A file src/plot/acc.dat15 contains measurements a0 ; a1 ; : : : ; an1 of the accel-

eration of an object moving along a straight line. The measurement ak is taken at

time point tk D kt, where t is the time spacing between the measurements. The

purpose of the exercise is to load the acceleration data into a program and compute

the velocity v.t/ of the object at some time t.

In general, the acceleration a.t/ is related to the velocity v.t/ through v 0 .t/ D

a.t/. This means that

Zt

v.t/ D v.0/ C a. /d :

(5.22)

0

If a.t/ is only known at some discrete, equally spaced points in time, a0 ; : : : ; an1

(which is the case in this exercise), we must compute the integral in (5.22) numeri-

cally, for example by the Trapezoidal rule:

!

k1

X

1

1

ai ; 1 k n 1 :

a0 C ak C

(5.23)

v.tk / t

2

2

i D1

We assume v.0/ D 0 so that also v0 D 0.

14

15

http://tinyurl.com/pwyasaa/plot/pendulum.dat

http://tinyurl.com/pwyasaa/plot/acc.dat5.14 Exercises

319

Read the values a0 ; : : : ; an1 from file into an array, plot the acceleration versus

time, and use (5.23) to compute one v.tk / value, where t and k 1 are specified

on the command line.

Filename: acc2vel\_v1.

Exercise 5.21: Read acceleration data and plot velocities

The task in this exercise is the same as in Exercise 5.20, except that we now want to

compute v.tk / for all time points tk D kt and plot the velocity versus time. Now

only t is given on the command line, and the a0 ; : : : ; an1 values must be read

from file as in Exercise 5.20.

Hint Repeated use of (5.23) for all k values is very inefficient. A more efficient

formula arises if we add the area of a new trapezoid to the previous integral (see

also Sect. A.1.7):

Ztk

v.tk / D v.tk1 / C

1

a. /d v.tk1 / C t .ak1 C ak /;

2

(5.24)

tk1

for k D 1; 2; : : : ; n 1, while v0 D 0. Use this formula to fill an array v with

velocity values.

Filename: acc2vel.

Exercise 5.22: Plot a trip’s path and velocity from GPS coordinates

A GPS device measures your position at every s seconds. Imagine that the po-

sitions corresponding to a specific trip are stored as .x; y/ coordinates in a file

src/plot/pos.dat16 with an x and y number on each line, except for the first

line, which contains the value of s.

a) Plot the two-dimensional curve of corresponding to the data in the file.

Hint Load s into a float variable and then the x and y numbers into two arrays.

Draw a straight line between the points, i.e., plot the y coordinates versus the x

coordinates.

b) Plot the velocity in x direction versus time in one plot and the velocity in y

direction versus time in another plot.

Hint If x.t/ and y.t/ are the coordinates of the positions as a function of time,

we have that the velocity in x direction is vx .t/ D dx=dt, and the velocity in y

direction is vy D dy=dt. Since x and y are only known for some discrete times,

tk D ks, k D 0; : : : ; n 1, we must use numerical differentiation. A simple

(forward) formula is

vx .tk /

16

x.tkC1 / x.tk /

;

s

vy .tk /

http://tinyurl.com/pwyasaa/plot/pos.dat

y.tkC1 / y.tk /

;

s

k D 0; : : : ; n 2 :320

5 Array Computing and Curve Plotting

Compute arrays vx and vy with velocities based on the formulas above for vx .tk /

and vy .tk /, k D 0; : : : ; n 2.

Filename: position2velocity.

Exercise 5.23: Vectorize the Midpoint rule for integration

The Midpoint rule for approximating an integral can be expressed as

Zb

f .x/dx h

a

n

X

1

f .a h C ih/;

2

i D1

(5.25)

where h D .b a/=n.

a) Write a function midpointint(f, a, b, n) to compute Midpoint rule. Use

a plain Python for loop to implement the sum.

b) Make a vectorized implementation of the Midpoint rule where you compute the

sum by Python’s built-in function sum.

c) Make another vectorized implementation of the Midpoint rule where you com-

pute the sum by the sum function in the numpy package.

d) Organize the three implementations above in a module file midpoint\_vec.py.

Equip the module with one test function for verifying the three implementations.

R4

Use the integral 2 2xdx D 12 as test case since the Midpoint rule will integrate

such a linear integrand exactly.

e) Start IPython, import the functions from midpoint\_vec.py, define some

Python implementation of a mathematical function f .x/ to integrate, and use

the %timeit feature of IPython to measure the efficiency of the three alternative

implementations.

Hint The %timeit feature is described in Sect. H.8.1.

Filename: midpoint\_vec.

Remarks The lesson learned from the experiments in e) is that numpy.sum is much

more efficient than Python’s built-in function sum. Vectorized implementations

must always make use of numpy.sum to compute sums.

Exercise 5.24: Vectorize a function for computing the area of a polygon

The area of a polygon is given by (3.17) in Exercise 3.19. Vectorize this formula

such that there are no Python loops in the implementation. Make a test function that

compares the scalar implementation in the referred exercise with the new vectorized

implementation for some chosen polygons (the scalar version must then be available

in a module so that the function can be imported).

Pn1

Hint Observe that the formula x1 y2 C x2 y3 C C xn1 yn D

i D0 xi yi C1 is

the dot product of two vectors, x[:-1] and y[1:], which can be computed as

numpy.dot(x[:-1], y[1:]).

Filename: polygon\_area\_vec.5.14 Exercises

321

Exercise 5.25: Implement Lagrange’s interpolation formula

Imagine we have n C 1 measurements of some quantity y that depends on x:

.x0 ; y0 /; .x1 ; y1 /; : : : ; .xn ; yn /. We may think of y as a function of x and ask what y

is at some arbitrary point x not coinciding with any of the points x0 ; : : : ; xn . It is not

clear how y varies between the measurement points, but we can make assumptions

or models for this behavior. Such a problem is known as interpolation.

One way to solve the interpolation problem is to fit a continuous function that

goes through all the n C 1 points and then evaluate this function for any desired x.

A candidate for such a function is the polynomial of degree n that goes through all

the points. It turns out that this polynomial can be written

pL .x/ D

n

X

yk Lk .x/;(5.26)

x xi

:

xk xi(5.27)

kD0

where

Lk .x/ D

The

Q

n

Y

i D0;i ¤k

notation corresponds to

n

Y

P

, but the terms are multiplied. For example,

xi D x0 x1 xk1 xkC1 xn :

i D0;i ¤k

The polynomial pL .x/ is known as Lagrange’s interpolation formula, and the points

.x0 ; y0 /; : : : ; .xn ; yn / are called interpolation points.

a) Make functions p\_L(x, xp, yp) and L\_k(x, k, xp, yp) that evaluate

pL .x/ and Lk .x/ by (5.26) and (5.27), respectively, at the point x. The arrays

xp and yp contain the x and y coordinates of the n C 1 interpolation points,

respectively. That is, xp holds x0 ; : : : ; xn , and yp holds y0 ; : : : ; yn .

b) To verify the program, we observe that Lk .xk / D 1 and that Lk .xi / D 0 for

i ¤ k, implying that pL .xk / D yk . That is, the polynomial pL goes through

all the points .x0 ; y0 /; : : : ; .xn ; yn /. Write a function test\_p\_L(xp, yp) that

computes jpL .xk / yk j at all the interpolation points .xk ; yk / and checks that

the value is approximately zero. Call test\_p\_L with xp and yp corresponding

to 5 equally spaced points along the curve y D sin.x/ for x 2 Œ0; . Thereafter,

evaluate pL .x/ for an x in the middle of two interpolation points and compare

the value of pL .x/ with the exact one.

Filename: Lagrange\_poly1.

Exercise 5.26: Plot Lagrange’s interpolating polynomial

a) Write a function

def graph(f, n, xmin, xmax, resolution=1001):322

5 Array Computing and Curve Plotting

for plotting pL .x/ in Exercise 5.25, based on interpolation points taken from

some mathematical function f .x/ represented by the argument f. The argu-

ment n denotes the number of interpolation points sampled from the f .x/ func-

tion, and resolution is the number of points between xmin and xmax used

to plot pL .x/. The x coordinates of the n interpolation points can be uni-

formly distributed between xmin and xmax. In the graph, the interpolation points

.x0 ; y0 /; : : : ; .xn ; yn / should be marked by small circles. Test the graph func-

tion by choosing 5 points in Œ0; and f as sin x.

b) Make a module Lagrange\_poly2 containing the p\_L, L\_k, test\_p\_L, and

graph functions. The call to test\_p\_L described in Exercise 5.25 and the call

to graph described above should appear in the module’s test block.

Hint Section 4.9 describes how to make a module. In particular, a test block is ex-

plained in Sect. 4.9.3, test functions like test\_p\_L are demonstrated in Sect. 4.9.4

and also in Sect. 3.4.2, and how to combine test\_p\_L and graph calls in the test

block is exemplified in Sect. 4.9.5.

Filename: Lagrange\_poly2.

Exercise 5.27: Investigate the behavior of Lagrange’s interpolating

polynomials

Unfortunately, the polynomial pL .x/ defined and implemented in Exercise 5.25

can exhibit some undesired oscillatory behavior that we shall explore graphically

in this exercise. Call the graph function from Exercise 5.26 with f .x/ D jxj,

x 2 Œ2; 2, for n D 2; 4; 6; 10. All the graphs of pL .x/ should appear in the same

plot for comparison. In addition, make a new figure with results from calls to graph

for n D 13 and n D 20. All the code necessary for solving this exercise should

appear in some separate program file, which imports the Lagrange\_poly2 module

made in Exercise 5.26.

Filename: Lagrange\_poly2b.

Remarks The purpose of the pL .x/ function is to compute .x; y/ between some

given (often measured) data points .x0 ; y0 /; : : : ; .xn ; yn /. We see from the graphs

that for a small number of interpolation points, pL .x/ is quite close to the curve

y D jxj we used to generate the data points, but as n increases, pL .x/ starts to

oscillate, especially toward the end points .x0 ; y0 / and .xn ; yn /. Much research has

historically been focused on methods that do not result in such strange oscillations

when fitting a polynomial to a set of points.

Exercise 5.28: Plot a wave packet

The function

2

f .x; t/ D e .x3t / sin .3.x t//

(5.28)

describes for a fixed value of t a wave localized in space. Make a program that

visualizes this function as a function of x on the interval Œ4; 4 when t D 0.

Filename: plot\_wavepacket.

Exercise 5.29: Judge a plot

Assume you have the following program for plotting a parabola:5.14 Exercises

323

import numpy as np

x = np.linspace(0, 2, 20)

y = x\*(2 - x)

import matplotlib.pyplot as plt

plt.plot(x, y)

plt.show()

Then you switch to the function cos.18x/ by altering the computation of y to y

= cos(18\*pi\*x). Judge the resulting plot. Is it correct? Display the cos.18x/

function with 1000 points in the same plot.

Filename: judge\_plot.

Exercise 5.30: Plot the viscosity of water

The viscosity of water, , varies with the temperature T (in Kelvin) according to

.T / D A 10B=.T C / ;

(5.29)

where A D 2:414 105 Pa s, B D 247:8 K, and C D 140 K. Plot .T / for T

between 0 and 100 degrees Celsius. Label the x axis with ‘temperature (C)’ and the

y axis with ‘viscosity (Pa s)’. Note that T in the formula for must be in Kelvin.

Filename: water\_viscosity.

Exercise 5.31: Explore a complicated function graphically

The wave speed c of water surface waves depends on the length of the waves.

The following formula relates c to :

s

2h

g

4 2

c./ D

1Cs

:

(5.30)

tanh

2

g2

Here, g is the acceleration of gravity (9:81 m/s2 ), s is the air-water surface tension

(7:9 102 N/m), is the density of water (can be taken as 1000 kg/m3 ), and h is

the water depth. Let us fix h at 50 m. First make a plot of c./ (in m/s) for small

(0.001 m to 0.1 m). Then make a plot c./ for larger (1 m to 2 km.

Filename: water\_wave\_velocity.

Exercise 5.32: Plot Taylor polynomial approximations to sin x

The sine function can be approximated by a polynomial according to the following

formula:

n

X

x 2j C1

.1/j

:

(5.31)

sin x S.xI n/ D

.2j C 1/Š

j D0

The expression .2j C 1/Š is the factorial (math.factorial can compute this quan-

tity). The error in the approximation S.xI n/ decreases as n increases and in the

limit we have that limn!1 S.xI n/ D sin x. The purpose of this exercise is to

visualize the quality of various approximations S.xI n/ as n increases.

a) Write a Python function S(x, n) that computes S.xI n/. Use a straightfor-

ward approach where you compute each term as it stands in the formula, i.e.,324

5 Array Computing and Curve Plotting

.1/j x 2j C1 divided by the factorial .2j C 1/Š. (We remark that Exercise A.14

outlines a much more efficient computation of the terms in the series.)

b) Plot sin x on Œ0; 4 together with the approximations S.xI 1/, S.xI 2/, S.xI 3/,

S.xI 6/, and S.xI 12/.

Filename: plot\_Taylor\_sin.

Exercise 5.33: Animate a wave packet

Display an animation of the function f .x; t/ in Exercise 5.28 by plotting f as

a function of x on Œ6; 6 for a set of t values in Œ1; 1. Also make an animated

GIF file.

Hint A suitable resolution can be 1000 intervals (1001 points) along the x axis, 60

intervals (61 points) in time, and 6 frames per second in the animated GIF file. Use

the recipe in Sect. 5.3.4 and remember to remove the family of old plot files in the

beginning of the program.

Filename: plot\_wavepacket\_movie.

Exercise 5.34: Animate a smoothed Heaviside function

Visualize the smoothed Heaviside function H .x/, defined in 3.26), as an animation

where starts at 2 and then goes to zero.

Filename: smoothed\_Heaviside\_movie.

Exercise 5.35: Animate two-scale temperature variations

We consider temperature oscillations in the ground as addressed in Sect. 5.13.2.

Now we want to visualize daily and annual variations. Let A1 be the amplitude of

annual variations and A2 the amplitude of the day/night variations. Let also P1 D

365 days and P2 D 24 h be the periods of the annual and the daily oscillations. The

temperature at time t and depth z is then given by

T .z; t/ D T0 C A1 e a1 z sin.!1 t a1 z/ C A2 e a2 z sin.!2 t a2 z/;

(5.32)

where

!1 D 2P1 ;

!2 D 2P2 ;

r

!1

a1 D

;

2k

r

!2

:

a2 D

2k

Choose k D 106 m2 =s, A1 D 15 C, A2 D 7 C, and the resolution t as P2 =10.

Modify the heatwave.py program in order to animate this new temperature func-

tion.

Filename: heatwave2.5.14 Exercises

325

Remarks We assume in this problem that the temperature T equals the reference

temperature T0 at t D 0, resulting in a sine variation rather than the cosine variation

in (5.18).

Exercise 5.36: Use non-uniformly distributed coordinates for visualization

Watching the animation in Exercise 5.35 reveals that there are rapid oscillations in

a small layer close to z D 0. The variations away from z D 0 are much smaller

in time and space. It would therefore be wise to use more z coordinates close to

z D 0 than for larger z values. Given a set x0 < x1 < < xn of uniformly spaced

coordinates in Œa; b, we can compute new coordinates xN i , stretched toward x D a,

by the formula

x a s

i

xN i D a C .b a/

;

ba

for some s > 1. In the present example, we can use this formula to stretch the z

coordinates to the left.

a) Experiment with s 2 Œ1:2; 3 and few points (say 15) and visualize the curve

as a line with circles at the points so that you can easily see the distribution of

points toward the left end. Identify a suitable value of s.

b) Run the animation with no circles and (say) 501 points with the found s value.

Filename: heatwave2a.

Exercise 5.37: Animate a sequence of approximations to

Exercise 3.18 outlines an idea for approximating as the length of a polygon in-

side the circle. Wrap the code from that exercise in a function pi\_approx(N),

which returns the approximation to using a polygon with N C 1 equally dis-

tributed points. The task of the present exercise is to visually display the polygons

as a movie, where each frame shows the polygon with N C 1 points together with

the circle and a title reflecting the corresponding error in the approximate value of

. The whole movie arises from letting N run through 4; 5; 6; : : : ; K, where K is

some (large) prescribed value. Let there be a pause of 0.3 s between each frame in

the movie. By playing the movie you will see how the polygons move closer and

closer to the circle and how the approximation to improves.

Filename: pi\_polygon\_movie.

Exercise 5.38: Animate a planet’s orbit

A planet’s orbit around a star has the shape of an ellipse. The purpose of this ex-

ercise is to make an animation of the movement along the orbit. One should see

a small disk, representing the planet, moving along an elliptic curve. An evolving

solid line shows the development of the planet’s orbit as the planet moves and the

title displays the planet’s instantaneous velocity magnitude. As a test, run the spe-

cial case of a circle and verify that the magnitude of the velocity remains constant

as the planet moves.

Hint 1 The points .x; y/ along the ellipse are given by the expressions

x D a cos.!t/;

y D b sin.!t/;326

5 Array Computing and Curve Plotting

where a is the semi-major axis of the ellipse, b is the semi-minor axis, ! is an

angular velocity of the planet around the star, and t denotes time. One complete

orbit corresponds to t 2 Œ0; 2=!. Let us discretize time into time points tk D kt,

where t D 2=.! n/. Each frame in the movie corresponds to .x; y/ points along

the curve with t values t0 ; t1 ; : : : ; ti , i representing the frame number (i D 1; : : : ; n).

Hint 2 The velocity vector is

dx dy

;

/ D .! a sin.!t/; !b cos.!t//;

dt dt

p

and the magnitude of this vector becomes ! a2 sin2 .!t/ C b 2 cos2 .!t/.

Filename: planet\_orbit.

.

Exercise 5.39: Animate the evolution of Taylor polynomials

A general series approximation (to a function) can be written as

S.xI M; N / D

N

X

fk .x/ :

kDM

For example, the Taylor polynomial of degree N for e x equals S.xI 0; N / with

fk .x/ D x k =kŠ. The purpose of the exercise is to make a movie of how S.xI M; N /

develops and improves as an approximation as we add terms in the sum. That

is, the frames in the movie correspond to plots of S.xI M; M /, S.xI M; M C 1/,

S.xI M; M C 2/, : : :, S.xI M; N /.

a) Make a function

animate\_series(fk, M, N, xmin, xmax, ymin, ymax, n, exact)

for creating such animations. The argument fk holds a Python function imple-

menting the term fk .x/ in the sum, M and N are the summation limits, the next

arguments are the minimum and maximum x and y values in the plot, n is the

number of x points in the curves to be plotted, and exact holds the function

that S.x/ aims at approximating.

Hint Here is some more information on how to write the animate\_series func-

tion. The function must accumulate the fk .x/ terms in a variable s, and for each

k value, s is plotted against x together with a curve reflecting the exact function.

Each plot must be saved in a file, say with names tmp\_0000.png, tmp\_0001.png,

and so on (these filenames can be generated by tmp\_%04d.png, using an appropri-

ate counter). Use the movie function to combine all the plot files into a movie in

a desired movie format.

In the beginning of the animate\_series function, it is necessary to remove all

old plot files of the form tmp\_\*.png. This can be done by the glob module and

the os.remove function as exemplified in Sect. 5.3.4.5.14 Exercises

327

b) Call the animate\_series function for the Taylor series for sin x, where

fk .x/ D .1/k x 2kC1 =.2k C 1/Š, and x 2 Œ0; 13, M D 0, N D 40,

y 2 Œ2; 2.

c) Call the animate\_series function for the Taylor series for e x , where fk .x/ D

.x/k =kŠ, and x 2 Œ0; 15, M D 0, N D 30, y 2 Œ0:5; 1:4.

Filename: animate\_Taylor\_series.

Exercise 5.40: Plot the velocity profile for pipeflow

A fluid that flows through a (very long) pipe has zero velocity on the pipe wall and

a maximum velocity along the centerline of the pipe. The velocity v varies through

the pipe cross section according to the following formula:

v.r/ D

ˇ

2 0

1=n

n 1C1=n

r 1C1=n ;

R

nC1

(5.33)

where R is the radius of the pipe, ˇ is the pressure gradient (the force that drives the

flow through the pipe), 0 is a viscosity coefficient (small for air, larger for water

and even larger for toothpaste), n is a real number reflecting the viscous properties

of the fluid (n D 1 for water and air, n < 1 for many modern plastic materials), and

r is a radial coordinate that measures the distance from the centerline (r D 0 is the

centerline, r D R is the pipe wall).

a) Make a Python function that evaluates v.r/.

b) Plot v.r/ as a function of r 2 Œ0; R, with R D 1, ˇ D 0:02, 0 D 0:02, and

n D 0:1.

c) Make an animation of how the v.r/ curves varies as n goes from 1 and down

to 0.01. Because the maximum value of v.r/ decreases rapidly as n decreases,

each curve can be normalized by its v.0/ value such that the maximum value is

always unity.

Filename: plot\_velocity\_pipeflow.

Exercise 5.41: Plot sum-of-sines approximations to a function

Exercise 3.21 defines the approximation S.tI n/ to a function f .t/. Plot S.tI 1/,

S.tI 3/, S.tI 20/, S.tI 200/, and the exact f .t/ function in the same plot. Use T D

2.

Filename: sinesum1\_plot.

Exercise 5.42: Animate the evolution of a sum-of-sine approximation to

a function

First perform Exercise 5.41. A natural next step is to animate the evolution of

S.tI n/ as n increases. Create such an animation and observe how the discontinuity

in f .t/ is poorly approximated by S.tI n/, even when n grows large (plot f .t/ in

each frame). This is a well-known deficiency, called Gibb’s phenomenon, when

approximating discontinuous functions by sine or cosine (Fourier) series.

Filename: sinesum1\_movie.328

5 Array Computing and Curve Plotting

Exercise 5.43: Plot functions from the command line

For quickly getting a plot of a function f .x/ for x 2 Œxmin ; xmax it could be nice to

a have a program that takes the minimum amount of information from the command

line and produces a plot on the screen and saves the plot to a file tmp.png. The

usage of the program goes as follows:

Terminal

plotf.py "f(x)" xmin xmax

Plotting e 0:2x sin.2x/ for x 2 Œ0; 4 is then specified as

Terminal

plotf.py "exp(-0.2\*x)\*sin(2\*pi\*x)" 0 4\*pi

Write the plotf.py program with as short code as possible (we leave it to Ex-

ercise 5.44 to test for valid input).

Hint Make x coordinates from the second and third command-line arguments

and then use eval (or StringFunction from scitools.std, see Sects. 4.3.3

and 5.5.1) on the first argument.

Filename: plotf.

Exercise 5.44: Improve command-line input

Equip the program from Exercise 5.43 with tests on valid input on the command

line. Also allow an optional fourth command-line argument for the number of points

along the function curve. Set this number to 501 if it is not given.

Filename: plotf2.

Exercise 5.45: Demonstrate energy concepts from physics

The vertical position y.t/ of a ball thrown upward is given by y.t/ D v0 t 12 gt 2 ,

where g is the acceleration of gravity and v0 is the velocity at t D 0. Two important

physical quantities in this context are the potential energy, obtained by doing work

against gravity, and the kinetic energy, arising from motion. The potential energy is

defined as P D mgy, where m is the mass of the ball. The kinetic energy is defined

as K D 12 mv 2 , where v is the velocity of the ball, related to y by v.t/ D y 0 .t/.

Make a program that can plot P .t/ and K.t/ in the same plot, along with their

sum P C K. Let t 2 Œ0; 2v0 =g. Read m and v0 from the command line. Run

the program with various choices of m and v0 and observe that P C K is always

constant in this motion. (In fact, it turns out that P C K is constant for a large class

of motions, and this is a very important result in physics.)

Filename: energy\_physics.

Exercise 5.46: Plot a w-like function

Define mathematically a function that looks like the “w” character. Plot the func-

tion. Also write a formal test function that verifies the implementation.

Filename: plot\_w.5.14 Exercises

329

Fig. 5.22 Visualization of numerical integration rules, with the Midpoint rule to the left and the

Trapezoidal rule to the right. The filled areas illustrate the deviations in the approximation of the

area under the curve

Exercise 5.47: Plot a piecewise constant function

Consider the piecewise constant function defined in Exercise 3.32. Make a Python

function plot\_piecewise(data, xmax) that draws a graph of the function,

where data is the nested list explained in mentioned exercise and xmax is the

maximum x coordinate. Use ideas from Sect. 5.4.1.

Filename: plot\_piecewise\_constant.

Exercise 5.48: Vectorize a piecewise constant function

Consider the piecewise constant function defined in Exercise 3.32. Make a vec-

torized implementation piecewise\_constant\_vec(x, data, xmax) of such

a function, where x is an array.

Hint You can use ideas from the Nv1 function in Sect. 5.5.3. However, since the

number of intervals is not known, it is necessary to store the various intervals and

conditions in lists.

Filename: piecewise\_constant\_vec.

Remarks Plotting the array returned from piecewise\_constant\_vec faces the

same problems as encountered in Sect. 5.4.1. It is better to make a custom plotting

function that simply draws straight horizontal lines in each interval (Exercise 5.47).

Exercise 5.49: Visualize approximations in the Midpoint integration rule

Consider the midpoint rule for integration from Exercise 3.12. Use Matplotlib to

make an illustration of the midpoint rule as shown to the left in Fig. 5.22.

The f .x/ function used in Fig. 5.22 is

f .x/ D x.12 x/ C sin.x/;

x 2 Œ0; 10 :

Hint Look up the documentation of the Matplotlib function fill\_between and

use this function to create the filled areas between f .x/ and the approximating

rectangles.

Note that the fill\_between requires the two curves to have the same number

of points. For accurate visualization of f .x/ you need quite many x coordinates,330

5 Array Computing and Curve Plotting

and the rectangular approximation to f .x/ must be drawn using the same set of x

coordinates.

Filename: viz\_midpoint.

Exercise 5.50: Visualize approximations in the Trapezoidal integration rule

Redo Exercise 5.49 for the Trapezoidal rule from Exercise 3.11 to produce the graph

shown to the right in Fig. 5.22.

Filename: viz\_trapezoidal.

Exercise 5.51: Experience overflow in a function

We are give the mathematical function

v.x/ D

where

1 e x=

;

1 e 1=

is a parameter.

a) Make a Python function v(x, mu=1E-6, exp=math.exp) for calculating the

formula for v.x/ using exp as a possibly user-given exponential function. Let

the v function return the nominator and denominator in the formula as well as

the fraction.

b) Call the v function for various x values between 0 and 1 in a for loop, let mu be

1E-3, and have an inner for loop over two different exp functions: math.exp

and numpy.exp. The output will demonstrate how the denominator is subject to

overflow and how difficult it is to calculate this function on a computer.

c) Plot v.x/ for D 1; 0:01; 0:001 on Œ0; 1 using 10,000 points to see what the

function looks like.

d) Convert x and eps to a higher precision representation of real numbers, with the

aid of the NumPy type float96, before calling v:

import numpy

x = numpy.float96(x); mu = numpy.float96(e)

Repeat point b) with these type of variables and observe how much better re-

sults we get with float96 compared with the standard float value, which is

float64 (the number reflects the number of bits in the machine’s representation

of a real number).

e) Call the v function with x and mu as float32 variables and report how the

function now behaves.

Filename: boundary\_layer\_func1.

Remarks When an object (ball, car, airplane) moves through the air, there is a very,

very thin layer of air close to the object’s surface where the air velocity varies dra-

matically, from the same value as the velocity of the object at the object’s surface

to zero a few centimeters away. This layer is called a boundary layer. The physics

in the boundary layer is important for air resistance and cooling/heating of objects.

The change in velocity in the boundary layer is quite abrupt and can be modeled by5.14 Exercises

331

the functiion v.x/, where x D 1 is the object’s surface, and x D 0 is some distance

away where one cannot notice any wind velocity v because of the passing object

(v D 0). The wind velocity coincides with the velocity of the object at x D 1,

here set to v D 1. The parameter is very small and related to the viscosity of air.

With a small value of , it becomes difficult to calculate v.x/ on a computer. The

exercise demonstrates the difficulties and provides a remedy.

Exercise 5.52: Apply a function to a rank 2 array

Let A be the two-dimensional array

3

0

2 1

7

6

4 1 1 0 5

0

5

0

2

Apply the function f from Exercise 5.5 to each element in A. Then calculate the

result of the array expression A\*\*3 + A\*exp(A) + 1, and demonstrate that the

end result of the two methods are the same.

Filename: apply\_arrayfunc.

Exercise 5.53: Explain why array computations fail

The following loop computes the array y from x:

>>> import numpy as np

>>> x = np.linspace(0, 1, 3)

>>> y = np.zeros(len(x))

>>> for i in range(len(x)):

...

y[i] = x[i] + 4

However, the alternative loop

>>> for xi, yi in zip(x, y):

...

yi = xi + 5

leaves y unchanged. Why? Explain in detail what happens in each pass of this loop

and write down the contents of xi, yi, x, and y as the loop progresses.

Filename: find\_errors\_arraycomp.

Exercise 5.54: Verify linear algebra results

When we want to verify that a mathematical result is true, we often generate ma-

trices or vectors with random elements and show that the result holds for these

“arbitrary” mathematical objects. As an example, consider testing that A C B D

B C A for matrices A and B:

def test\_addition():

n = 4 # matrix size

A = matrix(random.rand(n, n))

B = matrix(random.rand(n, n))332

5 Array Computing and Curve Plotting

tol = 1E-14

result1 = A + B

result2 = B + A

assert abs(result1 - result2).max() < tol

Use this technique to write test functions for the following mathematical results:

1. .A C B/C D AC C BC

2. .AB/C D A.BC /

3. rankA D rankAT

4. det.AB/ D det A det B

5. The eigenvalues if A equals the eigenvalues of AT when A is square.

Filename: verify\_linalg.6

Dictionaries and Strings