Array Computing and Curve Plotting

Mojtaba Alaei

October 21, 2013

Content of the course

Using Lists for Collecting Function Data

```
>>>def f(x):
...return x**3  # sample function
...
>>>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)]
```

Lists are flexible

```
mylist = [2, 6.0, 'tmp.ps', [0,1]]
```

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

Basics of Numerical Python Arrays

```
import numpy as np
```

To convert a list r to an array:

```
a = np.array(r)
```

To create a new array of length n:

```
a = np.zeros(n)
```

An array to have n elements with uniformly distributed values in an interval [p, q]:

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

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:

```
b = a[1:-1]
b[2] = 0.1
```

will also change a[3] to 0.1

Basics of Numerical Python Arrays

a[i:j:s] ?
a[0:-1:2] ?
a[::4] ?

Vectorization

Vectorization:

Operations on whole arrays, instead of using Python for loops, is called vectorization and is very convenient and very efficient (and an important programming technique to master)

```
>>>x = np.linspace(0, 1, n)
>>>y = np.zeros(n)
>>>for i in xrange(n):
>>>...y[i] = f(x[i])
```

Instead of using loops, we can use vectrorization in numpy:

```
>>> y = f(x)
```

Vectorization

Use numpy for sin, exp, ... functions instead of math:

```
from math import sin, cos, exp
import numpy as np
r = np.zeros(len(x))
for i in xrange(len(x)):
    r[i] = sin(x[i])*cos(x[i])*exp(-x[i]**2) + 2 + x[i]**2
```

So to use vectorization we replace loop in the following way:

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

Or

```
from numpy import sin, cos, exp
r = sin(x)*cos(x)*exp(-x**2) + 2 + x**2
```

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., 9.])
>>> x
array([ 1., 2., 3.])
```

In-Place Arithmetics

The difference between a = a+b and a += b

- 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.
- 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.

In-Place Arithmetics

```
a = (3*x**4 + 2*x + 4)/(x + 1)
```

The computation actually goes as follows with seven hidden arrays for storing intermediate results:

```
1. r1 = x**4
2. r2 = 3*r1
3. r3 = 2*x
4. r4 = r2 + r3
5. r5 = r4 + 4
6. r6 = x + 1
7. r7 = r5/r6
8. 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
```

Allocating Arrays

To make an array with the size and the type of another existing array:

```
a = x.copy()
```

Or

```
a = np.zeros(x.shape, x.dtype)
```

To ensure that an object is an array:

```
a = np.asarray(a)
```

Generalized Indexing

The slice f:t:i corresponds to the index set f, f+i, f+2*i. Such an index set can be given explicitly too: a[range(f,t,i)]:

```
>>> 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:

```
>>> 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.])
```

Array Type: The N-dimensional array (ndarray)

A numerical Python array has a tpye which name is ndarray:

```
>>> a = np.linspace(-1, 1, 3)
>>> a
array([-1., 0., 1.])
>>> type(a)
<type 'numpy.ndarray'>
```

To learn about ndarray see the following link:
docs.scipy.org/doc/numpy/reference/arrays.ndarray.html

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.

```
>>> a = np.linspace(-1, 1, 6)
>>> a.shape
(6,)
>>> a.size
6
>>> a.shape = (2, 3)
>>> a.shape
(2, 3)
>>> a.size # total no of elements
6
>>> a.shape = (a.size,) # reset shape
>>> a = a.reshape(3, 2) # alternative
>>> len(a) # no of rows
3
```

Two-Dimensional Numerical Python Arrays

```
>>> 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]]
```

To convert to the an array:

```
>>> table2 = np.array(table)
>>> print table2
[[-30. -22.]
[-20. -4.]
[-10. 14.]]
>>> type(table2)
<type 'numpy.ndarray'>
```

Two-Dimensional Numerical Python Arrays

Access to an element in list:

```
>>> table[1][0] # table[1] is [-20,4], whose index 0 holds -20
-20
```

This syntax works for two-dimensional arrays too:

```
>>> table2[1][0]
-20.0
```

But there is another syntax which is more common for arrays:

```
>>> table2[1,0]
-20.0
```

Two-Dimensional Numerical Python Arrays

```
from numpy import empty, zeros
# Create arrays
# Creat an array with (M+1)*(M+1) zero elements
V1 = zeros([M+1, M+1], float)
# Creat a empty array with (M+1)*(M+1) dimension
V2 = empty([M+1, M+1], float)
```

Matrix Objects

NumPy also has a matrix type called matrix or mat for one- and two-dimensional arrays:

```
>>> 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 = x2.transpose()                   # column vector
>>> x3
matrix([[ 1.],
        [ 2.],
        [ 3.]])

>>> type(x3)
<class 'numpy.matrixlib.defmatrix.matrix'>
>>> isinstance(x3, np.matrix)
True
```

Matrix Objects

The matrix-matrix, vector-matrix, or matrix-vector product:

```
>>> A = eye(3)          # identity matrix, also we can use identity(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                # vector-matrix product
>>> y2
matrix([[ 1., 2., 3.]])
>>> y3 = A*x3                # matrix-vector product
>>> y3
matrix([[ 1.],
        [ 2.],
        [ 3.]])
```

Matrix Objects

One should note here that the multiplication operator between standard ndarray objects is quite different:

```
>>> A*x1                    # no matrix-array product!
Traceback (most recent call last):
...
ValueError: matrices are not aligned

>>> # try array*array product:
>>> A = (zeros(9) + 1).reshape(3,3)
>>> A
array([[ 1., 1., 1.],
       [ 1., 1., 1.],
       [ 1., 1., 1.]])
>>> A*x1                        # [A[0,:]*x1, A[1,:]*x1, A[2,:]*x1]
array([[ 1., 2., 3.],
       [ 1., 2., 3.],
       [ 1., 2., 3.]])
>>> B = A + 1
>>> A*B                         # element-wise product
array([[ 2., 2., 2.],
       [ 2., 2., 2.],
       [ 2., 2., 2.]])
>>> A = mat(A); B = mat(B)
>>> A*B                         # matrix-matrix product
matrix([[ 6., 6., 6.],
        [ 6., 6., 6.],
        [ 6., 6., 6.]])
```

Summary

`array(ld)`	copy list data ld to a numpy array
`asarray(d )`	make array of data d (copy **if** list, no copy **if** already array)
`zeros(n)`	make a float vector/array of length n, with
`zeros(n, int)`	make an int vector/array of length n with ze
`zeros((m,n))`	make a two-dimensional float array with shap
`zeros(x.shape, x.dtype)`	make array of same shape as x **and** same element data type
`linspace(a,b,m)`	uniform sequence of m numbers between a **and** (b **is** included **in** the sequence)
`a.shape`	tuple containing a's shape
`a.size`	total no of elements in a
`len(a)`	length of a one-dim. array a (same as a.shap
`a.reshape(3,2)`	return a reshaped as 2 x 3 array
`a[i]`	vector indexing
`a[i,j]`	two-dim. array indexing
`a[1:k]`	slice: reference data with indices 1,. . . ,
`a[1:8:3]`	slice: reference data with indices 1, 4,. .
`b = a.copy()`	copy an array
`sin(a), exp(a), ...`	numpy functions applicable to arrays
`c = concatenate(a, b)`	c contains a with b appended
`c = where(cond, a1, a2)`	c[i] = a1[i] if cond[i], else c[i] = a2[i]
`isinstance(a, ndarray)`	is True if a is an array

Quiz1

The matrix file "n_mmp_mat" is a density matrix with two 7×7 matrix with complex elements for spin up and down (Don't worry, you do not need to know what density matrix is). The file contains 7 columns and 28 rows. Each two numbers in a row shows a complex number (first one is real part and the second one is imaginary part). So the first 14 rows make a 7×7 comlex matrix (for spin up) and anthor 14 rows make an another 7×7 comlex matrix (for spin down).

Quiz:

By using numpy (you need to know about "loadtxt", "linalg.eig"), calculate eigenvectors and eigenvalues of the first "7×7 comlex matrix"

Further reading

- https://github.com/jrjohansson/scientific-python-lectures
- http://nbviewer.ipython.org/urls/raw.github.com/jrjohansson/scienti
  python-lectures/master/Lecture-2-Numpy.ipynb
- http://docs.scipy.org/doc/numpy/reference/arrays.ndarray.html
- http://docs.scipy.org/doc/numpy/reference/routines.array-creation.html
- http://docs.scipy.org/doc/numpy/reference/generated/numpy.matrix
- http://docs.scipy.org/doc/numpy/reference/arrays.ndarray.html

Curve Plotting: matplotlib.pylab

```
from matplotlib.pylab import *
def f(t):
    return t**2*exp(-t**2)
t = linspace(0, 3, 51)          # 51 points between 0 and 3
y = zeros(len(t))               # allocate y with float elements
y = f(t)                        # vectorization
plot(t, y)
show()
```

The from matplotlib.pylab import *
performs a from numpy import * import as well as an import of all Matplotlib commands that resemble Matlab-style syntax.

matplotlib.pylab: title, legend, axis, savefig

```
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('tmp1.eps')               # produce PostScript
savefig('tmp1.png')               # produce PNG
show()
```

Plotting Multiple Curves

```
from matplotlib.pylab 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-')
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.

Placing Several Plots in One Figure

```
from matplotlib.pylab import *

def f1(t):
    return t**2*exp(-t**2)
def f2(t):
    return t**2*f1(t)

figure() # make separate 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')
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)'])
title('Bottom figure')
savefig('tmp4.eps')
show()
```

Matplotlib; Pyplot

The Matplotlib developers do not promote the matplotlib.pylab interface. Instead, they recommend the matplotlib.pyplot module and prefix Numerical Python and Matplotlib functionality by short forms of their package names:

```
import numpy as np
import matplotlib.pyplot as plt
```

The commands in matplotlib.pyplot are similar to those in matplotlib.pylab.

```
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.eps')      # produce PostScript
```

Plotting data in a file

suppose we have a file,ld1.wfc, with 5 columns of data

```
#          r              3 S             2 P             2 S             1 S
  0.001831563    0.005941348    0.000094305    0.025734199    0.109245136
  0.001877930    0.006088921    0.000099117    0.026373394    0.111958793
  0.001925470    0.006240086    0.000104175    0.027028146    0.114738511
  0.001974213    0.006394925    0.000109489    0.027698818    0.117585831
  0.002024191    0.006553526    0.000115074    0.028385779    0.120502323
  0.002075433    0.006715974    0.000120943    0.029089406    0.123489593
  0.002127973    0.006882359    0.000127111    0.029810086    0.126549278
  0.002181843    0.007052772    0.000133592    0.030548211    0.129683050
  0.002237077    0.007227305    0.000140403    0.031304183    0.132892613
  0.002293709    0.007406052    0.000147560    0.032078411    0.136179710
  0.002351774    0.007589111    0.000155080    0.032871312    0.139546114
  0.002411310    0.007776579    0.000162983    0.033683312    0.142993637
  0.002472352    0.007968557    0.000171287    0.034514847    0.146524128
  0.002534940    0.008165147    0.000180013    0.035366358    0.150139468
  0.002599112    0.008366453    0.000189182    0.036238298    0.153841579
  0.002664909    0.008572582    0.000198816    0.037131127    0.157632419
  0.002732372    0.008783642    0.000208940    0.038045313    0.161513983
  0.002801542    0.008999742    0.000219576    0.038981335    0.165488305
  0.002872463    0.009220996    0.000230753    0.039939679    0.169557457
  0.002945180    0.009447518    0.000242496    0.040920842    0.173723551
  0.003019738    0.009679425    0.000254834    0.041925330    0.177988736
  0.003096183    0.009916835    0.000267798    0.042953655    0.182355202
  0.003174563    0.010159869    0.000281418    0.044006342    0.186825177
  0.003254928    0.010408651    0.000295728    0.045083923    0.191400931
  0.003337326    0.010663305    0.000310763    0.046186940    0.196084773
  0.003421811    0.010923958    0.000326558    0.047315946    0.200879052
  0.003508435    0.011190740    0.000343153    0.048471500    0.205786159
  0.003597251    0.011463783    0.000360587    0.049654174    0.210808522
  0.003688316    0.011743221    0.000378903    0.050864546    0.215948615
  0.003781686    0.012029189    0.000398145    0.052103206    0.221208947
  0.003877420    0.012321826    0.000418358    0.053370751    0.226592072
  0.003975578    0.012621271    0.000439593    0.054667791    0.232100582
  0.004076220    0.012927667    0.000461900    0.055994941    0.237737110
  0.004179410    0.013241160    0.000485332    0.057352828    0.243504331
```

Plotting data in a file

By using "loadtxt" we can convert the data to the array:

```
from numpy import loadtxt
from pylab import plot,xlabel,ylabel,xlim,show, legend
r, S3, P2, S2, S1  = loadtxt('ld1.wfc', unpack = True)
xlabel("r")
ylabel("Psi(r)")
xlim(0,10)
plot(r,S3,"g--")
plot(r,P2,"r--")
plot(r,S2,"b--")
plot(r,S1,"y")
legend(('3S', '2P', '2S', '1S'), shadow=True )
show()
```

matplotlib; logarithmic plot

From pylab examples: log_demo.py

```
import numpy as np
import matplotlib.pyplot as plt

plt.subplots_adjust(hspace=0.4)
t = np.arange(0.01, 20.0, 0.01)

# log y axis
plt.subplot(221)
plt.semilogy(t, np.exp(-t/5.0))
plt.title('semilogy')
plt.grid(True)

# log x axis
plt.subplot(222)
plt.semilogx(t, np.sin(2*np.pi*t))
plt.title('semilogx')
plt.grid(True)

# log x and y axis
plt.subplot(223)
plt.loglog(t, 20*np.exp(-t/10.0), basex=2)
plt.grid(True)
plt.title('loglog base 4 on x')

# with errorbars: clip non-positive values
ax = plt.subplot(224)
ax.set_xscale("log", nonposx='clip')
ax.set_yscale("log", nonposy='clip')

x = 10.0**np.linspace(0.0, 2.0, 20)
y = x**2.0
plt.errorbar(x, y, xerr=0.1*x, yerr=5.0+0.75*y)
ax.set_ylim(ymin=0.1)
ax.set_title('Errorbars go negative')


plt.show()
```

Further reading

- http://matplotlib.org/users/pyplot_tutorial.html
- http://matplotlib.org/gallery.html