Lezione 6

Bioinformatica

Mauro Ceccanti[‡] e Alberto Paoluzzi[†]

[†]Dip. Informatica e Automazione – Università “Roma Tre”
[‡]Dip. Medicina Clinica – Università “La Sapienza”

Contents

Installing NumPy

As with a lot of open-source software, the best way to fully exploit and contribute to Scipy is to compile it from source. This will guarantee you the latest stable releases and a better support from mailing-lists. However, this can be challenging, and the second best way to run Scipy is to use binaries

Binaries for Windows and MacOSX available !

NumPy: choose version 1.3.0
Official releases are on SourceForge download site for numpy

SciPy: choose version 0.7.1
Official releases are on SourceForge download site for scipy

Contents

Best way to learn

Browse within the Numpy Example List, with added documentation from doc strings and arguments specification for methods and functions of Numpy

Numpy Example List With Doc

EXAMPLE:
numpy.sin()

ALSO:
Tentative NumPy Tutorial

Quick Tour

- NumPy is a Python library for working with multidimensional arrays
- The main data type is an array
- An array is a set of elements, all of the same type, indexed by a vector of nonnegative integers.

Quick Tour

Arrays can be created in different ways:

```
>>> from numpy import *
>>> a = array( [ 10, 20, 30, 40 ] )
# create an array out of a list

>>> a
array([10, 20, 30, 40])

>>> b = arange( 4 )
# create an array of 4 integers, from 0 to 3

>>> b
array([0, 1, 2, 3])

>>> c = linspace(-pi,pi,3)
# create an array of 3 evenly spaced samples from -pi to
     pi

>>> c
array([-3.14159265,  0.        ,  3.14159265])
```

Quick Tour

New arrays can be obtained by operating with existing arrays:

```
>>> d = a+b**2          # elementwise operations

>>> d
array([10, 21, 34, 49])
```

Quick Tour

Arrays may have more than one dimension:

```
>>> x = ones( (3,4) )

>>> x
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])

>>> x.shape                # a tuple with the dimensions
(3, 4)
```

```
>>> y = zeros( (3,4) )

>>> y
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
```

EXERCISE: generate the $n \times n$ identity matrix in 3 different ways

Quick Tour

and you can change the dimensions of existing arrays:

```
>>> y = arange(12)

>>> y
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])

>>> y.shape = 3,4
# does not modify the total number of elements

>>> y
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

REMARK: The evaluation of a statement does not produce an output, whereas the evaluation of an expression returns its value

Quick Tour

It is possible to operate with arrays of different dimensions as long as they fit well (broadcasting):

```
>>> 3*a                 # multiply each element of a by 3
array([ 30,  60,  90, 120])

>>> a+y                 # sum a to each row of y
array([[10, 21, 32, 43],
       [14, 25, 36, 47],
       [18, 29, 40, 51]])
```

Quick Tour

Similar to Python lists, arrays can be indexed, sliced and iterated over.

```
>>> a[2:4] = -7,-3
# modify last two elements of a

>>> for i in a:      # iterate over a
...     print i
10
20
-7
-3
```

Quick Tour

When indexing more than one dimension, indices are separated by commas:

```
>>> x[1,2] = 20

>>> x[1,:]
    # x’s second row
array([ 1,  1, 20,  1])

>>> x[0] = a
    # change first row of x

>>> x
array([[10, 20, -7, -3],
       [ 1,  1, 20,  1],
       [ 1,  1,  1,  1]])
```

REMARK: Indexing and slicing allow access to array elements both in reading and in writing mode

Quick Tour

Arrays can be created in different ways:

```
>>> from numpy import *
>>> a = array( [ 10, 20, 30, 40 ] )
# create an array out of a list

>>> a
array([10, 20, 30, 40])

>>> b = arange( 4 )
# create an array of 4 integers, from 0 to 3

>>> b
array([0, 1, 2, 3])

>>> c = linspace(-pi,pi,3)
# create an array of 3 evenly spaced samples from −π to π

>>> c
array([-3.14159265,  0.        ,  3.14159265])
```

Contents

The multidimensional array class is called `ndarray`

Note that this is not the same as the Standard Python Library class array, which is only for one-dimensional arrays

The more important attributes of an ndarray object are:

ndarray.ndim	the number of axes (dimensions) of the array. In the Python world, the number of dimensions is often referred to as rank.
ndarray.shape	the dimensions of the array. This is a tuple of integers indicating the size of the array in each dimension. For a matrix with n rows and m columns, shape will be (n,m). The length of the shape tuple is therefore the rank, or number of dimensions, ndim.
ndarray.size	the total number of elements of the array. This is equal to the product of the elements of shape.

The multidimensional array class is called `ndarray`

ndarray.dtype	an object describing the type of the elements in the array. One can create or specify dtype's using standard Python types. NumPy provides a bunch of them, for example: bool_, character, int_, int8, int16, int32, int64, float_, float8, float16, float32, float64, complex_, complex64, object_.
ndarray.itemsize	the size in bytes of each element of the array. For example, an array of elements of type float64 has itemsize 8 (=64/8), while one of type complex32 has itemsize 4 (=32/8). It is equivalent to ndarray.dtype.itemsize.
ndarray.data	the buffer containing the actual elements of the array. Normally, we won't need to use this attribute because we will access to the elements in an array using indexing facilities.

An example

We define the following array:

```
>>> a = arange(10).reshape(2,5)
>>> a
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

We have just created an array object with a label a attached to it. The array a has several attributes –or properties. In Python, attributes of a specific object are denoted name_object.attribute. In our case:

- `a.shape` is `(2,5)`
- `a.ndim` is 2 (which is the length of `a.shape`)
- `a.size` is 10
- `a.dtype.name` is `int32`
- `a.itemsize` is 4, which means that an `int32` takes 4 bytes in memory.

Python Short Course

Download the “Lecture2: Numerical Python”, by Richard P. Muller at Caltech

Lecture2: Numerical Python

EXERCISE

Try to execute several simple examples (written in May, 2000) and adapt them to current version of Numpy

Contents

NumPy for Matlab Users

Look carefully at the first sections of the linked web page

NumPy for Matlab Users

Contents

Pytrsxge: geometric kernel of Plasm language

HPC (hierarchical polyhedral complex) is the name of the geometric type

```
from math import *
from pytrsxge import *

# mkpol (MaKe POLyhedron) in 2D
Plasm.View(Plasm.mkpol(2,[0,0, 1,0 ,1,1, 0,1],
    [[0,1,2],[2,0,3]] ))

# mkpol in 3D
Plasm.View(Plasm.mkpol(3,[0,0,0,1,0,0,1,1,0,0,1,0,
    0,0,1,1,0,1,1,1,1,0,1,1],[[0,1,2,3,4,5,6,7]]))

# example of structure
args = [Plasm.cube(0), Plasm.translate(Plasm.cube(1),
    3,1,1), Plasm.translate(Plasm.cube(2),3,1,2), Plasm.
    translate(Plasm.cube(3),3,1,3)]
Plasm.View(Plasm.Struct(args))
```

Introduction to `Pytrsxge` package

It is the Python porting of the geometric kernel `xge` of the Plasm Language

```
# apply a transformation matrix (homogeneous components
    in first row/col)
vmat=Matf([1,0,0,0, 0,1,0,1, 0,0,1,1, 0,0,0,1])
Plasm.View(Plasm.transform(Plasm.cube(3),vmat,vmat.
    invert()))

# scale an hpc
Plasm.View(Plasm.scale(Plasm.cube(3),Vecf(0.0,
    1.0,2.0,3.0)))

# translate an hpc
Plasm.View(Plasm.translate(Plasm.cube(3),Vecf(0.0,
    1.0,2.0,3.0)))

# rotate an hpc
Plasm.View(Plasm.Struct([Plasm.cube(3), Plasm.rotate(
    Plasm.cube(3), 3,1,2,pi)]))
```

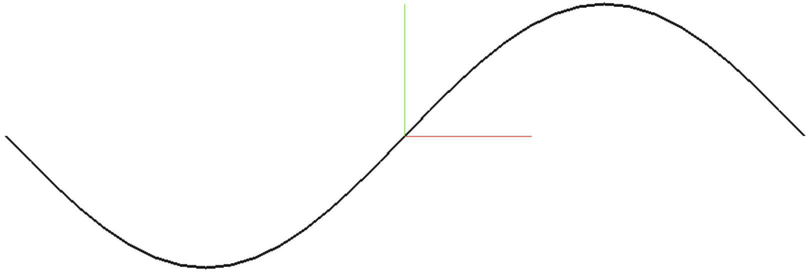
Graph of the `sin()` function

```
from numpy import *
from pytrsxge import *

c = linspace(-pi,pi,16)
points=array(zip(c,sin(c)))

def polyline (points):
    n,d = points.shape
    points.shape = n*d
    return Plasm.mkpol(d,
        points,zip(range(n
        -1),range(1,n)))

Plasm.View(polyline(points))
```

```
>>> print points
[[ -3.14159265e+00  -1.22464680e-16]
 [ -2.72271363e+00  -4.06736643e-01]
 [ -2.30383461e+00  -7.43144825e-01]
 [ -1.88495559e+00  -9.51056516e-01]
 [ -1.46607657e+00  -9.94521895e-01]
 [ -1.04719755e+00  -8.66025404e-01]
 [ -6.28318531e-01  -5.87785252e-01]
 [ -2.09439510e-01  -2.07911691e-01]
 [  2.09439510e-01   2.07911691e-01]
 [  6.28318531e-01   5.87785252e-01]
 [  1.04719755e+00   8.66025404e-01]
 [  1.46607657e+00   9.94521895e-01]
 [  1.88495559e+00   9.51056516e-01]
 [  2.30383461e+00   7.43144825e-01]
 [  2.72271363e+00   4.06736643e-01]
 [  3.14159265e+00   1.22464680e-16]]
>>> points.size
32
```

Graph of the `sin()` function

REMARK: every graphics object is a linear approximation of the smooth shape ...

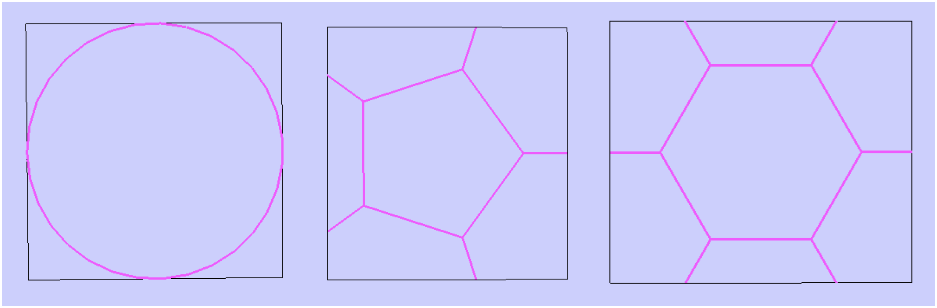


```
from numpy import *
from pytrsxge import *

Plasm.View(polyline(points))
```

The `pytrsxge` package is the Python porting of the Plasm language

Pentose and aromatic shape

REMARK: every graphics object is a linear approximation of the smooth shape ...



In organic chemistry, the structures of some rings of atoms are unexpectedly stable. Aromaticity is a chemical property in which a conjugated ring of unsaturated bonds, lone pairs, or empty orbitals exhibit a stabilization stronger than would be expected by the stabilization of conjugation alone. It can also be considered a manifestation of cyclic delocalization and of resonance. This is usually considered to be because electrons are free to cycle around circular arrangements of atoms, which are alternately single- and double-bonded to one another.
These bonds may be seen as a hybrid of a single bond and a double bond, each bond in the ring identical to every other. This commonly-seen model of aromatic rings, namely the idea that benzene was formed from a six-membered carbon ring with alternating single and double bonds (cyclohexatriene), was developed by Kekulé (see History section below). The model for benzene consists of two resonance forms, which corresponds to the double and single bonds' switching positions. Benzene is a more stable molecule than would be expected without accounting for charge delocalization. (From Wikipedia)

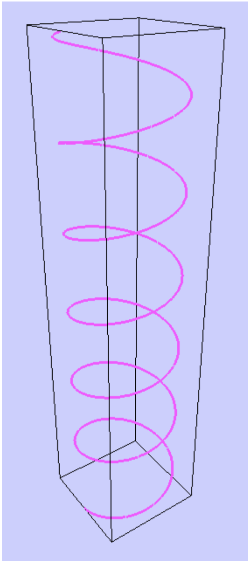
Graph of the `sin()` function

REMARK: every graphics object is a linear approximation of the smooth shape ...

```
from numpy import *
from pytrsxge import *

# circle
c = linspace(-pi,pi,6)
p = array( zip(cos(c), sin(c)) )
Plasm.View(polyline(p))

# pentagon with legs
a3 = map(polyline,  map(array, zip(p, 1.5*p)) )
Plasm.View(Plasm.Struct( a3 + [polyline(p)] ))

# hexagon  with legs
c = linspace(-pi,pi,7)
p = array( zip(cos(c), sin(c)) )
a3 = map(polyline,  map(array, zip(p, 1.5*p)) )
q = Plasm.Struct( a3 + [polyline(p)] )
Plasm.View(q)
```

Parametric helix curve

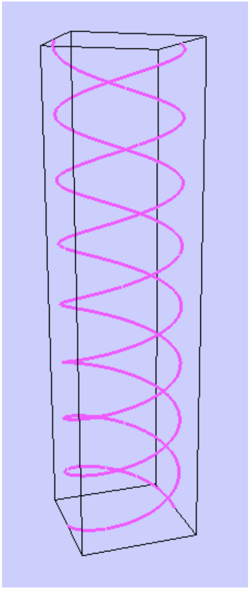
The ‘pitch’ is the (constant) distance between (any) pair of closest points



```
from numpy import *
from pytrsxge import *

def helixpoints(radius,pitch,nturns):
    c = linspace(0,2*pi*nturns,12*nturns)
    return array(zip( cos(c),sin(c), c*(pitch
        /(2*pi)) ))

def helix(radius,pitch,nturns):
    return polyline(helixPoints(radius,pitch,
        nturns))

Plasm.View(helix(1,1.5,6))
```

Doublehelix

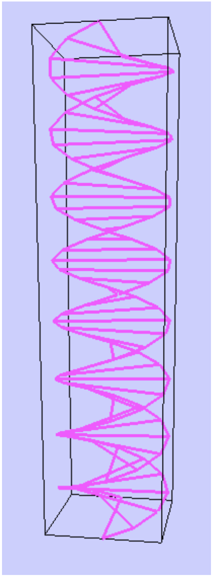
A polyline is a rotated copy of the other



```
def doubleHelix(radius,pitch,nturns):
    p = polyline(helixPoints(radius,pitch,
        nturns))
    q = Plasm.copy(p)
    two_hpc = [p, Plasm.rotate(q, 3,1,2,pi)]
    return Plasm.Struct(two_hpc)

Plasm.View(doubleHelix(1,2,4))
```

DNA structure

The `ukpol` function returns the vertices of its `hpc` argument



```
def dnaStructure(radius,pitch,nturns):
    p = helixPoints(radius,pitch,nturns)
    q = array(matrix(p) * matrix([[-1,0,0],
        [0,-1,0], [0,0,1]]))
    diameters = map(polyline, map(array, zip(p.
        tolist(),q.tolist())) )
    return Plasm.Struct( diameters + [polyline(p
        ),polyline(q)] )

Plasm.View(dnaStructure(1,2,4))
```

3D polylines

The backbone of the 2J5Y protein