

Lezione 6

Bioinformatica

Mauro Ceccanti^{†‡} e Alberto Paoluzzi[†]

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

Lab 01: Numerical Python

Installing NumPy

Numerical Python

Quick Tour

The Basics

NumPy for Matlab Users

Introduction to geometric module Pytrsxge



Contents

Lab 01: Numerical Python

Installing NumPy

Numerical Python

Quick Tour

The Basics

NumPy for Matlab Users

Introduction to geometric module Pytrsxge

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

Lab 01: Numerical Python

Installing NumPy

Numerical Python

Quick Tour

The Basics

NumPy for Matlab Users

Introduction to geometric module Pytsxge



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.



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

Arrays can be created in different ways:

```
1 >>> from numpy import *
2 >>> a = array( [ 10, 20, 30, 40 ] )
3 # create an array out of a list
4
5 >>> a
6 array([10, 20, 30, 40])
7
8 >>> b = arange( 4 )
9 # create an array of 4 integers, from 0 to 3
10
11 >>> b
12 array([0, 1, 2, 3])
13
14 >>> c = linspace(-pi,pi,3)
15 # create an array of 3 evenly spaced samples from -pi to
16 pi
17 >>> c
18 array([-3.14159265,  0.          ,  3.14159265])
```



Quick Tour

New arrays can be obtained by operating with existing arrays:

```
1 >>> d = a+b**2          # elementwise operations
2
3 >>> d
4 array([10, 21, 34, 49])
```



Quick Tour

Arrays may have more than one dimension:

```
1 >>> x = ones( (3,4) )
2
3 >>> x
4 array([[1., 1., 1., 1.],
5        [1., 1., 1., 1.],
6        [1., 1., 1., 1.]])
7
8 >>> x.shape           # a tuple with the dimensions
9 (3, 4)
```

```
1 >>> y = zeros( (3,4) )
2
3 >>> y
4 array([[ 0.,  0.,  0.,  0.],
5        [ 0.,  0.,  0.,  0.],
6        [ 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:

```
1 >>> y = arange(12)
2
3 >>> y
4 array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
5
6 >>> y.shape = 3,4
7 # does not modify the total number of elements
8
9 >>> y
10 array([[ 0,  1,  2,  3],
11         [ 4,  5,  6,  7],
12         [ 8,  9, 10, 11]])
```



Quick Tour

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

```
1 >>> 3*a                # multiply each element of a by 3
2 array([ 30,  60,  90, 120])
3
4 >>> a+y                # sum a to each row of y
5 array([[10, 21, 32, 43],
6        [14, 25, 36, 47],
7        [18, 29, 40, 51]])
```

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



Quick Tour

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

```
1 >>> a[2:4] = -7,-3
2 # modify last two elements of a
3
4 >>> for i in a:      # iterate over a
5     ...     print i
6 10
7 20
8 -7
9 -3
```

Quick Tour

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

```
1 >>> x[1,2] = 20
2
3 >>> x[1,:]
4     # x's second row
5 array([ 1,  1, 20,  1])
6
7 >>> x[0] = a
8     # change first row of x
9
10 >>> x
11 array([[10, 20, -7, -3],
12        [ 1,  1, 20,  1],
13        [ 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:

```
1 >>> from numpy import *
2 >>> a = array( [ 10, 20, 30, 40 ] )
3 # create an array out of a list
4
5 >>> a
6 array([10, 20, 30, 40])
7
8 >>> b = arange( 4 )
9 # create an array of 4 integers, from 0 to 3
10
11 >>> b
12 array([0, 1, 2, 3])
13
14 >>> c = linspace(-pi,pi,3)
15 # create an array of 3 evenly spaced samples from -π to π
16
17 >>> c
18 array([-3.14159265,  0.          ,  3.14159265])
```



Contents

Lab 01: Numerical Python

Installing NumPy

Numerical Python

Quick Tour

The Basics

NumPy for Matlab Users

Introduction to geometric module Pytrsxge



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`.



An example

We define the following array:

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



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.



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

Lab 01: Numerical Python

Installing NumPy

Numerical Python

Quick Tour

The Basics

NumPy for Matlab Users

Introduction to geometric module Pytrsxge



NumPy for Matlab Users

Look carefully at the first sections of the linked web page

NumPy for Matlab Users

Contents

Lab 01: Numerical Python

Installing NumPy

Numerical Python

Quick Tour

The Basics

NumPy for Matlab Users

Introduction to geometric module Pytrsxge



Pytrsxge: geometric kernel of Plasm language

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

```
1 from math import *
2 from pytrsxge import *
3
4 # mkpol (MaKe POLyhedron) in 2D
5 Plasm.View(Plasm.mkpol(2, [0,0, 1,0 ,1,1, 0,1],
6     [[0,1,2],[2,0,3]] ))
7
8 # mkpol in 3D
9 Plasm.View(Plasm.mkpol(3, [0,0,0,1,0,0,1,1,0,0,1,0,
10     0,0,1,1,0,1,1,1,1,0,1,1], [[0,1,2,3,4,5,6,7]]))
11
12 # example of structure
13 args = [Plasm.cube(0), Plasm.translate(Plasm.cube(1),
14     3,1,1), Plasm.translate(Plasm.cube(2),3,1,2), Plasm.
15     translate(Plasm.cube(3),3,1,3)]
16 Plasm.View(Plasm.Struct(args))
```

Introduction to Pytrsxge package

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

```
1 # apply a transformation matrix (homogeneous components
  in first row/col)
2 vmat=Matf([1,0,0,0, 0,1,0,1, 0,0,1,1, 0,0,0,1])
3 Plasm.View(Plasm.transform(Plasm.cube(3), vmat, vmat.
  invert()))
4
5 # scale an hpc
6 Plasm.View(Plasm.scale(Plasm.cube(3), Vecf(0.0,
  1.0,2.0,3.0)))
7
8 # translate an hpc
9 Plasm.View(Plasm.translate(Plasm.cube(3), Vecf(0.0,
  1.0,2.0,3.0)))
10
11 # rotate an hpc
12 Plasm.View(Plasm.Struct([Plasm.cube(3), Plasm.rotate(
  Plasm.cube(3), 3,1,2,pi)]))
```



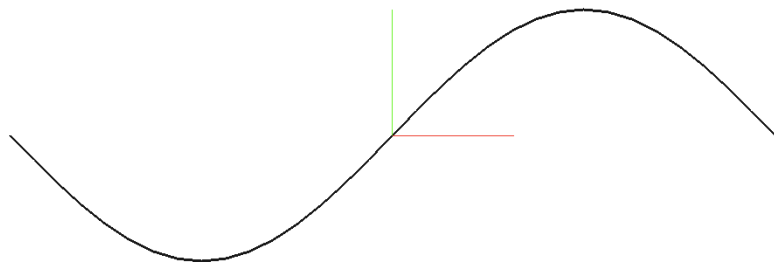
Graph of the `sin()` function

```
from numpy import * 1 >>> print points
from pytrsxge import * 2 [[ -3.14159265e+00 -1.22464680e-16]
3 [ -2.72271363e+00 -4.06736643e-01]
c = linspace(-pi,pi,16) 4 [ -2.30383461e+00 -7.43144825e-01]
points=array(zip(c, sin(c))) 5 [ -1.88495559e+00 -9.51056516e-01]
6 [ -1.46607657e+00 -9.94521895e-01]
def polyline (points): 7 [ -1.04719755e+00 -8.66025404e-01]
  n,d = points.shape 8 [ -6.28318531e-01 -5.87785252e-01]
  points.shape = n*d 9 [ -2.09439510e-01 -2.07911691e-01]
  return Plasm.mkpol(d, 10 [ 2.09439510e-01 2.07911691e-01]
  points, zip(range(n 11 [ 6.28318531e-01 5.87785252e-01]
  -1), range(1,n)) 12 [ 1.04719755e+00 8.66025404e-01]
13 [ 1.46607657e+00 9.94521895e-01]
Plasm.View(polyline(points)) 14 [ 1.88495559e+00 9.51056516e-01]
15 [ 2.30383461e+00 7.43144825e-01]
16 [ 2.72271363e+00 4.06736643e-01]
17 [ 3.14159265e+00 1.22464680e-16]]
18 >>> points.size
19 32
```



Graph of the `sin()` function

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



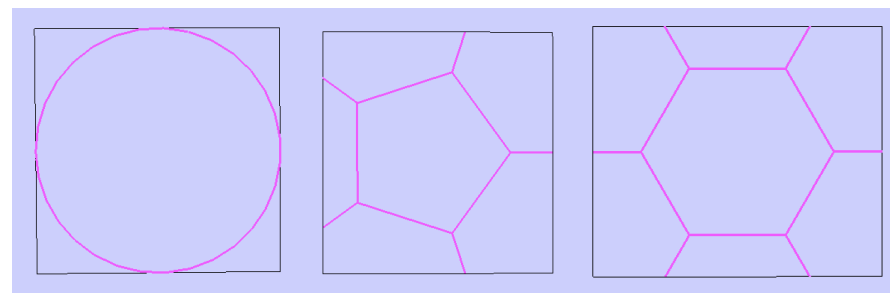
```
1 from numpy import *
2 from pytrsxge import *
3
4 Plasm.View(polyline(points))
```

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

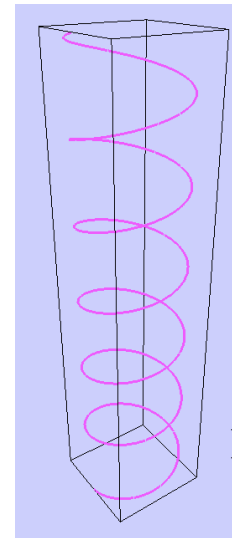
REMARK: every graphics object is a **linear approximation** of the smooth shape ...

```
1 from numpy import *
2 from pytrsxge import *
3
4 # circle
5 c = linspace(-pi,pi,6)
6 p = array( zip(cos(c), sin(c)) )
7 Plasm.View(polyline(p))
8
9 # pentagon with legs
10 a3 = map(polyline, map(array, zip(p, 1.5*p)) )
11 Plasm.View(Plasm.Struct( a3 + [polyline(p)] ))
12
13 # hexagon with legs
14 c = linspace(-pi,pi,7)
15 p = array( zip(cos(c), sin(c)) )
16 a3 = map(polyline, map(array, zip(p, 1.5*p)) )
17 q = Plasm.Struct( a3 + [polyline(p)] )
18 Plasm.View(q)
```



Parametric helix curve

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

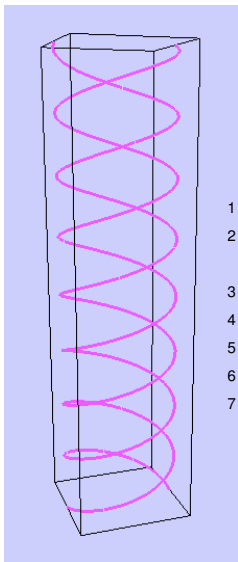


```
1 from numpy import *
2 from pytrsxge import *
3
4 def helixpoints(radius,pitch,nturns):
5     c = linspace(0,2*pi*nturns,12*nturns)
6     return array( zip( cos(c), sin(c), c*(pitch
7         / (2*pi)) ) )
8
9 def helix(radius,pitch,nturns):
10    return polyline( helixPoints( radius, pitch,
11    nturns))
12
13 Plasm.View(helix(1,1.5,6))
```



Doublehelix

A polyline is a rotated copy of the other

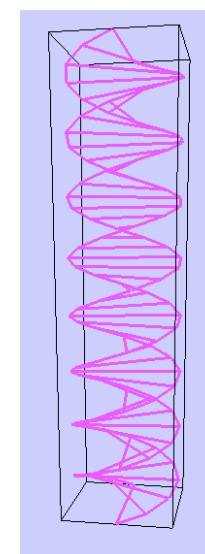


```
1 def doubleHelix(radius,pitch,nturns):
2     p = polyline( helixPoints( radius, pitch,
3     nturns))
4     q = Plasm.copy(p)
5     two_hpc = [p, Plasm.rotate(q, 3,1,2,pi)]
6     return Plasm.Struct( two_hpc)
7
8 Plasm.View(doubleHelix(1,2,4))
```



DNA structure

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

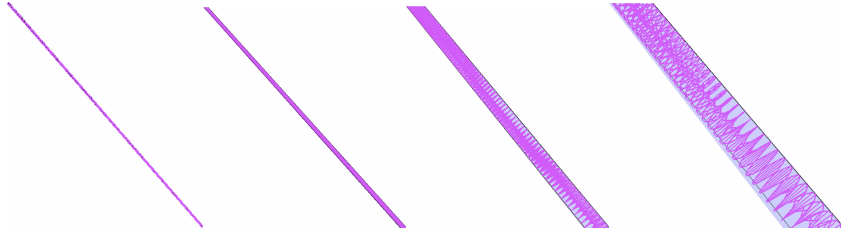


```
1 def dnaStructure(radius,pitch,nturns):
2     p = helixPoints( radius, pitch, nturns)
3     q = array( matrix(p) * matrix([[ -1, 0, 0],
4     [ 0, -1, 0], [ 0, 0, 1]]))
5     diameters = map(polyline, map(array, zip(p.
6     tolist(), q.tolist()))) )
7     return Plasm.Struct( diameters + [polyline(p
8     ), polyline(q)] )
9
10 Plasm.View(dnaStructure(1,2,4))
```



Deoxyribonucleic acid (DNA) fragment

Gene: The basic biological unit of heredity. A segment of DNA needed to contribute to a function.



left image: 10K base pairs
other images: subset enlargement



3D polylines

The backbone of the 2J5Y protein

