NumPy and SciPy

Shawn T. Brown
Director of Public Health Applications
Pittsburgh Supercomputing Center

What are NumPy and SciPy

- NumPy and SciPy are open-source add-on modules to Python that provide common mathematical and numerical routines in pre-compiled, fast functions.
- The NumPy (Numeric Python) package provides basic routines for manipulating large arrays and matrices of numeric data.
- The SciPy (Scientific Python) package extends the functionality of NumPy with a substantial collection of useful algorithms, like minimization, Fourier transformation, regression, and other applied mathematical techniques.

Installing NumPy and SciPy

- These packages are installed like all Python modules
- The packages can be downloaded from http://www.scipy.org/Downloads
- Can be built from source (required to have a valid C build chain… e.g. Windows you will need VS8.0 Express or MinGW installed)
- Precompiled binaries for all platforms.
- Linux, generally use package distribution systems (e.g. Ubuntu… apt-get install numpy)

Using NumPy and SciPy

- Import the modules into your program like most Python packages

```
>>> import numpy

>>> import numpy as np

>>> from numpy import *

>>> import scipy

>>> import scipy as sp

>>> from scipy import *
```

Introducing … the array

- The array is the basic, essential unit of NumPy
  - Designed to be accessed just like Python lists
  - All elements are of the same type
  - Ideally suited for storing and manipulating large numbers of elements

```
>>> a = np.array([1, 4, 5, 8], float)

>>> a
array([1., 4., 5., 8.])

>>> type(a)
<type 'numpy.ndarray'>

>>> a[:2]
Array([1., 4.])

>>> a[3]
8.0
```

Multi-dimensional Arrays

- Just like lists, an array can have multiple dimensions (obviously useful for matrices)

```
>>> a = np.array([1, 2, 3], [4, 5, 6], float)

>>> a
array([[1., 2., 3.],
       [4., 5., 6]])

>>> a[0,0]
1.0

>>> a[0,1]
2.0

>> a.shape
(2,3)
```

Multi-dimensional arrays cont.

- Arrays can be reshaped

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

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

  - Note, reshape creates a new array, not a modification of the original (in the case of above, the original **a** is gone.)

Care must be taken…

- One must (as with all Python objects) pay attention to what variables are actually being assigned

```
>>> a = np.array([1, 2, 3], float)
>>> b = a
>>> c = a.copy()

>>> a[0] = 0
>>> a
array([0., 2., 3.])
>>> b
array([0., 2., 3.])
>>> c
array([1., 2., 3.])
```

Other array operations

- Arrays can be turned into lists

```
>>> a = np.array([1, 2, 3],float)
>>> a.tolist()
[1.0, 2.0, 3.0]
>>> list(a)
[1.0, 2.0, 3.0]
```

- Arrays can be turned into a binary string

```
>>> a = array([1, 2, 3], float)
>>> s = a.tostring()
>>> s
'\x00\x00\x00\x00\x00\x00\xf0?\x00\x00\x00\x00\x00\x00\x
00@\x00\x00\x00\x00\x00\x00\x08@'
>>> np.fromstring(s)
array([ 1., 2., 3.])
```

Other array operations

- One can fill an array with a single value

```
>>> a = np.array([1, 2, 3],float)
>>> a
array([1.0, 2.0, 3.0])
>>> a.fill(0)
array([0.0, 0.0, 0.0])
```

- Arrays can be transposed easily

```
>>> a = np.array(range(6), float).reshape((2, 3))
>>> a
array([[ 0., 1., 2.],
       [ 3., 4., 5.]])
>>> a.transpose()
array([[ 0., 3.],
       [ 1., 4.],
       [ 2., 5.]])
```

Other array operations

- Arrays can be flatten (essentially reshaping to one dimension)

```
>>> np.array([[1, 2, 3], [4, 5, 6]], float)
>>> a
array([[ 1., 2., 3.],
       [ 4., 5., 6.]])
>>> a.flatten()
array([ 1., 2., 3., 4., 5., 6.])
```

array concatenation

- Combining arrays can be done through concatenation

```
>>> a = np.array([1,2], float)
>>> b = np.array([3,4,5,6], float)
>>> c = np.array([7,8,9], float)
>>> np.concatenate((a, b, c))
array([1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

array concatenation

- Multi-dimensional arrays can be concatenated along specific axis

```
>>> a = np.array([[1, 2], [3, 4]], float)
>>> b = np.array([[5, 6], [7, 8]], float)
>>> np.concatenate((a,b))
array([[ 1., 2.],
       [ 3., 4.],
       [ 5., 6.],
       [ 7., 8.]])
>>> np.concatenate((a,b), axis=0)
array([[ 1., 2.],
       [ 3., 4.],
       [ 5., 6.],
       [ 7., 8.]])
>>> np.concatenate((a,b), axis=1)
array([[ 1., 2., 5., 6.],
       [ 3., 4., 7., 8.]])
```

Other array operations

- The dimensionality of an array can be increased using **newaxis**

```
>>> a = np.array([1, 2, 3], float)
>>> a
array([1., 2., 3.])
>>> a[:,np.newaxis]
array([[ 1.],
       [ 2.],
       [ 3.]])
>>> a[:,np.newaxis].shape
(3,1)
>>> b[np.newaxis,:]
array([[ 1., 2., 3.]])
>>> b[np.newaxis,:].shape
(1,3)
```

Other ways to create arrays

```
>>> np.arange(5, dtype=float)
array([ 0., 1., 2., 3., 4.])
>>> np.arange(1, 6, 2, dtype=int)
array([1, 3, 5])

>>> np.ones((2,3), dtype=float)
array([[ 1., 1., 1.],
[ 1., 1., 1.]])
>>> np.zeros(7, dtype=int)
array([0, 0, 0, 0, 0, 0, 0])

>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)
>>> np.zeros_like(a)
array([[ 0., 0., 0.],
       [ 0., 0., 0.]])
>>> np.ones_like(a)
array([[ 1., 1., 1.],
       [ 1., 1., 1.]])
```

Other ways to create arrays

- Identity and eye matrices

```
>>> np.identity(4, dtype=float)
array([[ 1., 0., 0., 0.],
       [ 0., 1., 0., 0.],
       [ 0., 0., 1., 0.],
       [ 0., 0., 0., 1.]])

>>> np.eye(4, k=1, dtype=float)
array([[ 0., 1., 0., 0.],
       [ 0., 0., 1., 0.],
       [ 0., 0., 0., 1.],
       [ 0., 0., 0., 0.]])
```

Now the power of NumPy

- When using NumPy arrays, you now can perform math on these arrays inside your Python script trivially

```
>>> a = np.array([1,2,3], float)
>>> b = np.array([5,2,6], float)
>>> a + b
array([6., 4., 9.])
>>> a - b
array([-4., 0., -3.])
>>> a * b
array([5., 4., 18.])
>>> b / a
array([5., 1., 2.])
>>> a % b
array([1., 0., 3.])
>>> b**a
array([5., 4., 216.])
```

Array mathematics

- Operations are valid for any dimension array
  - Careful … element by element operations, not a matrix operation!

```
>>> a = np.array([[1,2], [3,4]], float)
>>> b = np.array([[2,0], [1,3]], float)
>>> a * b
array([[2., 0.], [3., 12.]])

>>> a = np.array([1,2,3], float)
>>> b = np.array([4,5], float)
>>> a + b
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: shape mismatch: objects cannot be broadcast
to a single shape
```

Tricky operations

- One must be careful when performing operations on different sized arrays

```
>>> a = np.array([[1, 2], [3, 4], [5, 6]], float)
>>> b = np.array([-1, 3], float)
>>> a
array([[ 1., 2.],
       [ 3., 4.],
       [ 5., 6.]])
>>> b
array([-1., 3.])
>>> a + b
array([[ 0., 5.],
       [ 2., 7.],
       [ 4., 9.]])
```

```
array([[-1., 3.],
       [-1., 3.],
       [-1., 3.]])
```

Tricky operations

- One can control this process with **newaxis**

```
>>> a = np.zeros((2,2), float)
>>> b = np.array([-1., 3.], float)
>>> a
array([[ 0., 0.],
       [ 0., 0.]])
>>> b
array([-1., 3.])
>>> a + b
array([[-1., 3.],
       [-1., 3.]])
>>> a + b[np.newaxis,:]
array([[-1., 3.],
       [-1., 3.]])
>>> a + b[:,np.newaxis]
array([[-1., -1.],
       [ 3., 3.]])
```

Math with arrays

- NumPy offers a large library of common mathematical functions that can be applied elementwise to arrays
  - Among these are: `abs`, `sign`, `sqrt`, `log`, `log10`, `exp`, `sin`, `cos`, `tan`, `arcsin`, `arccos`, `arctan`, `sinh`, `cosh`, `tanh`, `arcsinh`, `arccosh`, and `arctanh`
- Example

```
>>> a = np.array([1, 4, 9], float)
>>> np.sqrt(a)
array([ 1., 2., 3.])
```

Math with arrays

- NumPy also allows rounding of entire array

```
>>> a = np.array([1.1, 1.5, 1.9], float)
>>> np.floor(a)
array([ 1., 1., 1.])
>>> np.ceil(a)
array([ 2., 2., 2.])
>>> np.rint(a)
array([ 1., 2., 2.])
```

- NumPy also gives two very important constants

```
>>> np.pi
3.1415926535897931
>>> np.e
2.7182818284590451
```

Iterating over arrays

- Just like lists

```
>>> a = np.array([1, 4, 5], int)
>>> for x in a:
... print x
... <hit return>
1
4
5
```

- Iterating over multidimensional arrays

```
>>> a = np.array([[1, 2], [3, 4], [5, 6]], float)
>>> for x in a:
... print x
... <hit return>
[ 1. 2.]
[ 3. 4.]
[ 5. 6.]
```

Iterating over arrays

- Multiple assignment can also be used in array iteration

```
>>> a = np.array([[1, 2], [3, 4], [5, 6]], float)
>>> for (x, y) in a:
... print x * y
... <hit return>
2.0
12.0
30.0
```

Basic array Operations

- NumPy offers several operations on arrays that provide performance and convenience

```
>>> a = np.array([2, 4, 3], float)
>>> a.sum()
9.0
>>> a.prod()
24.0
>>> np.sum(a)
9.0
>>> np.prod(a)
24.0

>>> a = np.array([2, 1, 9], float)
>>> a.mean()
4.0
>>> a.var()
12.666666666666666
>>> a.std()
3.5590260840104371
```

Basic array Operations

- Multi dimensional arrays can also use these

```
>>> a = np.array([[0, 2], [3, -1], [3, 5]], float)
>>> a.mean(axis=0)
array([ 2., 2.])
>>> a.mean(axis=1)
array([ 1., 1., 4.])
>>> a.min(axis=1)
array([ 0., -1., 3.])
>>> a.max(axis=0)
array([ 3., 5.])
```

- Arrays can be sorted and clipped

```
>>> a = np.array([6, 2, 5, -1, 0], float)
>>> sorted(a)
[-1.0, 0.0, 2.0, 5.0, 6.0]

>>> a = np.array([6, 2, 5, -1, 0], float)
>>> a.clip(0, 5)
array([ 5., 2., 5., 0., 0.])
```

Comparison operations

- Boolean comparisons can be used to compare members elementwise on arrays of equal size

```
>>> a = np.array([1, 3, 0], float)
>>> b = np.array([0, 3, 2], float)
>>> a > b
array([ True, False, False], dtype=bool)
>>> a == b
array([False, True, False], dtype=bool)
>>> a <= b
array([False, True, True], dtype=bool)
>>> c = a > b
>>> c
array([ True, False, False], dtype=bool)
>>> a = np.array([1, 3, 0], float)
>>> a > 2
array([False, True, False], dtype=bool)
```

Array item select and manipulation

- A nice feature of arrays, on can “filter” them with an *array selector*

```
>>> a = np.array([[6, 4], [5, 9]], float)
>>> a >= 6
array([[ True, False],
       [False, True]], dtype=bool)
>>> a[a >= 6]
array([ 6., 9.])
```

Vector and Matrix Mathematics

- Perhaps the most powerful feature of NumPy is the vector and matrix operations
  - Provide compiled code performance similar to machine specific BLAS
- Performing a vector-vector, vector-matrix or matrix-matrix multiplication using **dot**

```
>>> a = np.array([1, 2, 3], float)
>>> b = np.array([0, 1, 1], float)
>>> np.dot(a, b)
5.0
```

Vector and Matrix Mathematics

- **dot** continued…

```
>>> a = np.array([[0, 1], [2, 3]], float)
>>> b = np.array([2, 3], float)
>>> c = np.array([[1, 1], [4, 0]], float)
>>> a
array([[ 0., 1.],
       [ 2., 3.]])
>>> np.dot(b, a)
array([ 6., 11.])
>>> np.dot(a, b)
array([ 3., 13.])
>>> np.dot(a, c)
array([[ 4., 0.],
       [ 14., 2.]])
>>> np.dot(c, a)
array([[ 2., 4.],
       [ 0., 4.]])
```

Vector and Matrix Mathematics

- Inner, outer and cross products of matrices and vectors also supported

```
>>> a = np.array([1, 4, 0], float)
>>> b = np.array([2, 2, 1], float)
>>> np.outer(a, b)
array([[ 2., 2., 1.],
       [ 8., 8., 4.],
       [ 0., 0., 0.]])
>>> np.inner(a, b)
10.0
>>> np.cross(a, b)
array([ 4., -1., -6.]
```

Vector and Matrix Mathematics

- NumPy also offers a number of built-in routines for linear algebra through the `linalg` submodule.

```
>>> a = np.array([[4, 2, 0], [9, 3, 7], [1, 2, 1]],
float)
>>> a
array([[ 4., 2., 0.],
       [ 9., 3., 7.],
       [ 1., 2., 1.]])
>>> np.linalg.det(a)
-53.999999999999993

>>> vals, vecs = np.linalg.eig(a)
>>> vals
array([ 9.  , 2.44948974, -2.44948974])
>>> vecs
array([[-0.3538921 , -0.56786837, 0.27843404],
       [-0.88473024, 0.44024287, -0.89787873],
       [-0.30333608, 0.69549388, 0.34101066]])
```

Vector and Matrix Mathematics

```
>>> b = np.linalg.inv(a)
>>> b
array([[ 0.14814815, 0.07407407, -0.25925926],
       [ 0.2037037 , -0.14814815, 0.51851852],
       [-0.27777778, 0.11111111, 0.11111111]])
>>> np.dot(a, b)
array([[ 1.00000000e+00, 5.55111512e-17, 2.22044605e-16],
       [ 0.00000000e+00, 1.00000000e+00, 5.55111512e-16],
       [ 1.11022302e-16, 0.00000000e+00, 1.00000000e+00]])


>>> a = np.array([[1, 3, 4], [5, 2, 3]], float)
>>> U, s, Vh = np.linalg.svd(a)
>>> U
array([[-0.6113829 , -0.79133492],
       [-0.79133492, 0.6113829 ]])
>>> s
array([ 7.46791327, 2.86884495])
>>> Vh
array([[-0.61169129, -0.45753324, -0.64536587],
       [ 0.78971838, -0.40129005, -0.46401635],
       [-0.046676 , -0.79349205, 0.60678804]])
```

Other things NumPy offers

- Polynomial Mathematics
- Statistical computations
- Full suite of pseudo-random number generators and operations
- Discrete Fourier transforms,
- more complex linear algebra operations
- size / shape / type testing of arrays,
- splitting and joining arrays, histograms
- creating arrays of numbers spaced in various ways
- creating and evaluating functions on grid arrays
- treating arrays with special (NaN, Inf) values
- set operations
- creating various kinds of special matrices
- evaluating special mathematical functions (e.g. Bessel functions)

- To learn more, consult the NumPy documentation at http://docs.scipy.org/doc/

SciPy functionality

- The SciPy module greatly extends the functionality of the NumPy routines.

```
>>> import scipy
SciPy imports all the functions from the NumPy
namespace, and in
addition provides:
Available subpackages
---------------------
odr                          --- Orthogonal Distance Regression [*]
misc                         --- Various utilities that don't have
                                 another home.
sparse.linalg.eigen.arpack   --- Eigenvalue solver using
                                 iterativemethods. [*]
fftpack                      --- Discrete Fourier Transform
                                 algorithms[*]
io                           --- Data input and output [*]
sparse.linalg.eigen.lobpcg   --- Locally Optimal Block Preconditioned
                                 Conjugate Gradient Method (LOBPCG) [*]
```

SciPy cont.

```
special                          --- Airy Functions[*]
lib.blas                         --- Wrappers to BLAS library [*]
sparse.linalg.eigen              --- Sparse Eigenvalue Solvers [*]
stats                            --- Statistical Functions [*]
lib                              --- Python wrappers to external
                                     libraries[*]
lib.lapack                       --- Wrappers to LAPACK library [*]
maxentropy                       --- Routines for fitting maximum
                                     entropy models [*]
integrate                        --- Integration routines [*]
ndimage                          --- n-dimensional image package [*]
linalg                           --- Linear algebra routines [*]
spatial                          --- Spatial data structures and
                                     algorithms [*]
interpolate                      --- InterpolationTools [*]
sparse.linalg                    --- Sparse Linear Algebra [*]
sparse.linalg.dsolve.umfpack     --- :Interface to the UMFPACK
                                     library: [*]
sparse.linalg.dsolve             --- Linear Solvers [*]
optimize                         --- Optimization Tools [*]
cluster                          --- Vector Quantization / Kmeans [*]
signal                           --- Signal Processing Tools [*]
sparse                           --- Sparse Matrices [*]
[*] - using a package requires explicit import (see pkgload)
```

Hands-On

- Goal – Show why using NumPy is from performance standpoint
- Task: Write a Python code that:
  - Accepts an arbitrary integer value as a command line argument (http://docs.python.org/2/library/optparse.html#module-optparse)
  - Create two multi-dimensional lists of random floating point variables of this dimension
  - Perform an explicit matrix-matrix multiply storing the result in another two-dimensional list
  - Perform the same task with NumPy, using the dot operation
  - Time the performance of these two approaches over a number of different dimensions (100, 500, and 1000)
  - Time the performance of just the NumPy approach for larger dimensions (5000,10000) and see how the performance scales.

Hands-on

- Time permitting
  - Look through the SciPy set of functionalities.
  - Choose a feature that aligns with your own research.
  - Write a Python program that implements this functionality on a kernel similar to your work.