Python Numpy (1)

Intro to multi-dimensional array & numerical linear algebra

Harry Lee

January 29, 2018

CEE 696

Table of contents

Introduction

From the last lecture

```
import numpy as np
ibound = np.ones((NLAY,NROW,NCOL),dtype=np.int32)
```

We have used numpy package and its array objects for MODFLOW model setup. Let's dig into them.

Why Numpy?

- the core library for scientific computing in Python.
- multi-dimensional array object
- math tools for working with these arrays
- interfaces to standard math libraries coded in a compiled language (written in C++ or Fortran) for speed

Have you used MATLAB or R?

Numpy for matlab users:

`http://www.numpy.org/devdocs/user/numpy-for-matlab-users.html`

Next slides..

- Array creation
- Array access/slicing
- Array operations

Numpy Example

```python
import numpy as np

a = np.array([1, 2, 3, 4])    # Create a "rank" 1 array
print(type(a))                # <class 'numpy.ndarray'>
print(a.shape)                # "(4,)"
print(a[0], a[1], a[2], a[3]) # "1 2 3 4"
a[1] = 4                      # Change an element
print(a)                      # "[1, 4, 3, 4]"

b = np.array([[1,2],[3,4]])    # a rank 2 array
print(b.shape)                     # "(2, 2)"
print(b[0, 0], b[0, 1], b[1, 0])   # "1 2 3"
```

Note that "rank" in python means the number of dimensions of an array while "rank" in linear algebra is the maximum number of linearly independent columns of a 2D matrix.

Anatomy of Numpy Array

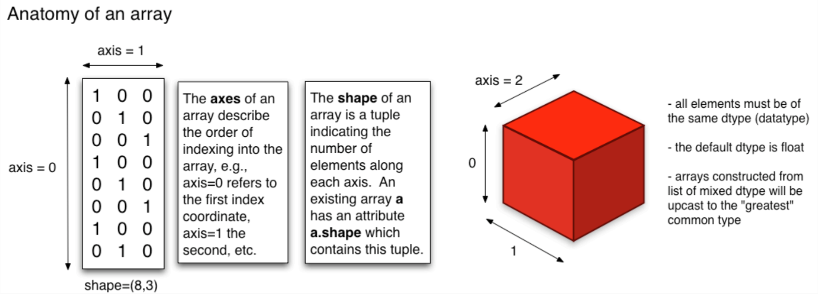

Figure 1: `http://pages.physics.cornell.edu/~myers/teaching/ComputationalMethods/python/arrays.html`

Numpy Array (1) - Creation

```
a = np.zeros((2,2))   # all zeros
print(a)              # [[ 0.  0.]
                      #  [ 0.  0.]]

b = np.ones((1,2))    # all ones
print(b)              # [[ 1.  1.]]

c = np.full((2,2), 3.) # constant array
print(c)               # [[ 3.  3.]
                       #  [ 3.  3.]]

d = np.eye(2)         # identity matrix
print(d)              # [[ 1.  0.]
                      #  [ 0.  1.]]

e = np.random.random((2,2)) # random array
print(e)
```

Numpy Array (2) - Data Type

```
x = np.array([1, 2])   # numpy will choose its datatype
print(x.dtype)         # datatype = int64

x = np.array([1.0, 2.0])
print(x.dtype)                  # datatype = float64

# for single-precision MODFLOW (see available executable
IBOUND = np.array([1, 2], dtype=np.int32)
print(x.dtype)

# one can use dtype = "d" for double-precision
# i.e, np.float64
HK = np.ones((100,100),'d')
```

Numpy Array (3) - Modification

```
a = np.array([[1,2],[3,4]])
b = np.array(a) # create a new array
c = a           # referencing
print(a)
print(b)
print(c)

a[0,0] = 10
print(a)
print(b)
print(c) # this is easy.. wait, what?
```

Reference/Shallow Copy vs. Deep Copy

This is one of the most confusing aspects for beginners. Be careful!

```
a = [1,2,3] # type(a) : list
b = a
c = a[:] # NOT for list with nested structure and np.arr
b[1] = 10
print(id(a),a)
print(id(b),b)
print(id(c),c)

x = np.array([1, 2, 3])
y = x
z = np.copy(x)
x[0] = 10
print(id(x),x)
print(id(y),y)
print(id(z),z)
```

Numpy Array (4) - Slice Notation CON'T

We use "slicing" to pull out the sub-array

a[start:end]
a[start:end:step]

Make sure the [:end] value represents the first value that is not in the selected slice.

```
# create an array
a = np.array([1,2,3,4,5,6,7,8,9,10])
a[:]   # a copy of the whole array
a[0:10] # = a[0:] = a[:10] = a[:] = a[::]
a[0:10:2] # = a[:10:2] = a[::2]

a[-1]    # last item in the array
a[-2:]   # last two items in the array
a[:-2]   # everything except the last two items
```

Numpy Arrays (5) - Slice Notation CON'T

```python
# create an array
a = np.array([[1,2,3], [4,5,6], [7,8,9]])

b = a[:2, 1:3]

# This is IMPORTANT!!
print(a)
b[0, 0] = 10      # b[0, 0] from a[0, 1]
print(a)          # print it.. wait, what?
```

A slice of an array is a "view" into a part of the original array. Thus, modifying it will change the original array as before. Be careful!

Numpy Array (6) - Slice Notation CON'T

```
a = np.array([[1,2,3], [4,5,6], [7,8,9]])
# integer index + slicing for lower dimensional array
row1 = a[1, :]    # Rank 1 view of the second row of a
# slicing for the same dimension
row2 = a[1:2, :]  # Rank 2 view of the second row of a

print(row1, row1.shape, row1.ndim)
print(row2, row2.shape, row2.ndim)
```

Make sure the dimension of your array is consistent with what you thought!

Numpy Array (7) - Element Access

```
# create a new array from which we will select elements
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])

print(a)

# an array of indices (for each row)
b = np.array([2, 1, 0, 1])

# print element from each row of a using the indices in
print(a[np.arange(4), b])

# even we can modify the values
a[np.arange(4), b] = a[np.arange(4), b] + 5
```

Numpy Array (8) - Element Access

```
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])

idx = (a > 2)   # find element > 2
                # return booleans

print(idx)

print(a[idx]) # return values greater than 2
              # with booleans

# in a single statement
print(a[a > 2])
```

Arrays (6) - Operations (1)

```
x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])
print(x+y)
print(np.add(x, y))

print(x-y)
print(np.subtract(x, y))

# make sure this is element-wise product
print(x*y)
print(np.multiply(x, y))

# make sure this is element-wise division
print(x/y)
print(np.divide(x, y))

print(np.sqrt(x))
```

Arrays (6) - operations (2)

```
x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11, 12])

# Inner product of vectors
print(v.dot(w))
print(np.dot(v, w))

# Matrix-vector product
print(x.dot(v))
print(np.dot(x, v))

# Matrix-matrix product
print(x.dot(y))
print(np.dot(x, y))
```

Linear Algebra

Solution to linear system

$$\mathsf{Ax} = \mathsf{b}$$

A is n by n matrix

b is $n \times 1$ vector

x is $n \times 1$ vector to solve

- numerical solution to PDE (partial differential equation) ex) MODFLOW
- optimization ex) quadratic programming

MODFLOW - Numerical Modeling (1)

In MODFLOW, water mass balance is enforced by summing the water fluxes $Q_{i,j,k}$ across each side of the cell and internal source/sinks:

$\sum Q_{i,j,k} = 0$ (for steady state condition, i.e., no time-related term)

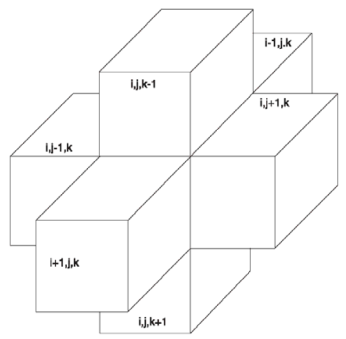

Figure 2: cell (i,j,k) configuration for mass balance equation (from Fig. 2-2 Harbaugh [2005])

With Darcy's law,

$$q_{i,j-1/2,k} = K_{i,j-1/2,k} \Delta c_i \Delta v_k \frac{(\phi_{i,j-1,k} - \phi_{i,j,k})}{\Delta r_{j-1/2}}$$

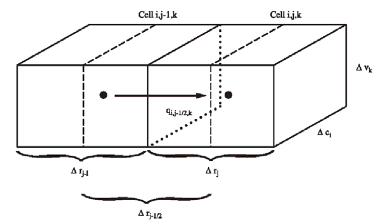

Figure 3: Flow into cell i,j,k from cell i,j-1,k (from Fig. 2-3 Harbaugh [2005])

Combining with mass balance equation $\sum Q_i = 0$ (for steady stead) for every cell will lead to the system of linear equations

$$\mathbf{A}\phi = \mathbf{f}$$

numpy.linalg.inv

```
# from FVM with K=1, dr,dc,dz = 1
# const. head = 10 m at the left
# no flow at the right
A = np.array([[1., 0., 0.],[-1., 2., -1.],[0., -1., 1.]])
f = np.array([[10],[0],[0]])

# inverse of A to compute h = np.dot(inv(A),f)
# NEVER do this in practice! because
# 1) it's expensive O(n^3)
# 2) poor numerical accuracy
invA = np.linalg.inv(A)
h = np.dot(invA,f)

print(h)                  # what do you expect?
print(invA)               #
print(np.dot(A,invA))     # is this np.eye?
print(np.dot(A,h) - f)    # satisfy mass balance?
```

numpy.linalg.solve

```
# so-called stiffness matrix
A = np.array([[1., 0., 0.],[-1., 2., -1.],[0., 1., -1.]]
# force/load vector
f = np.array([[10],[0],[0]])

# solution of Ah = f
h = np.linalg.solve(A,f)

print(h) # what do you expect?
print(np.dot(A,h) - f) # satisfy mass balance?
# how about constant head boundaries at both ends?
```

We will discuss advanced materials later (i.e., iterative approach as in PCG module of MODFLOW and eigen-decomposition)

Connection to Quadratic Function Optimization

$$f(x) = \frac{1}{2}\mathrm{x}^T\mathrm{A}\mathrm{x} - \mathrm{b}\mathrm{x}^T$$

- $\mathrm{A}\mathrm{x}^\star - \mathrm{b} = \mathrm{0}$ for necessary condition to optimal (local) solution $\mathrm{x}^\star$ to min/max f(x)
- Quadratic function is related to some energy. In fact, nature acts so as to minimize energy
- If a physical system is in a stable state of equilibrium, then the energy in that state should be minimal
- Thus, no wonder linear algebra is related to optimization!