CSC326 Array Programming Paradigm

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
1.0	2011-09		JZ

Contents

1 Agenda

- Array Programming Paradigm
- NumPy
- Array as collection
- Elementwise operations
- Reduction

2 Array Programming Language

- Native sequences are nice
  - But very general: element can be anything
  - Slow for large scale data and numerical computation
- Array Programming Paradigm
  - Everything is an array
  - No loops! (we already saw list comprehension)
- APL (A Programming Language)
  - Kenneth E. Iverson: Candian Computer Scientist
  - Turing Speech: "Notations as a Tool of Thoughts" (one of the most inspiring talks in CS)
  - Influenced spreadsheets, functional programming, and computer math packages
- Vector machine
  - Vector Machine
  - Each register is an array
  - Instructions operate on arrays
  - Seymour Cray: Father of Supercomputer
- Question: How to
  - Combine performance of C
  - Expressive Power of APL
  - Python as a language substrate
- NumPy: Multidimensional arrays!
  - Vector / Matrix
  - Photo:
  - Tables

3 NumPy Package

- Retrieving source

```
>wget url
```

- Unpack

```
>tar xvfz foo.tar.gz
```

- Installation
  - setup.py

```
>python setup.py build
>python setup.py install --user
```

- Ready to import

```
>python

>>>import numpy as np
.....


NumPy Type: ndarray
```

- Rank
  - Number of dimensions
- Axis
  - Each dimension
- Shape
  - tuple of integers indicating the size of the array in each dimension
- accessors
  - a.ndim: rank
  - a.shape: shape
  - a.size: total number of elements (prod of all elements of shape)
  - a.itemsize: number of bytes for each elements
  - a.dtype: data type of each element
  - a.data: actual data (do not use directly)

```
>>> from numpy  import *
>>> a = arange(10).reshape(2,5)
>>> a
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> a.shape
(2, 5)
>>> a.ndim
2
>>> a.dtype.name
'int32'
>>> a.itemsize
4
>>> a.size
10
```

4 Creating Array

- array function
  - Convert from sequences

```
>>> import numpy as np
>>> a = np.array( [2,3,4] )
>>> a
array([2, 3, 4])
>>> a.dtype
dtype('int32')
>>> b = array([1.2, 3.5, 5.1])
>>> b.dtype
dtype('float64')
```

- Or sequences of sequences . . .

```
>>> b = np.array( [ (1.5,2,3), (4,5,6) ] )
>>> b
array([[ 1.5,  2. ,  3. ],
       [ 4. ,  5. ,  6. ]])
```

- zeros/ones/empty function

```
>>> np.zeros( (3,4) )
array([[0.,  0.,  0.,  0.],
       [0.,  0.,  0.,  0.],
       [0.,  0.,  0.,  0.]])
>>> np.ones( (2,3,4), dtype=int16 )                # dtype can also be specified
array([[[ 1, 1, 1, 1],
        [ 1, 1, 1, 1],
        [ 1, 1, 1, 1]],
       [[ 1, 1, 1, 1],
        [ 1, 1, 1, 1],
        [ 1, 1, 1, 1]]], dtype=int16)

>>> np.empty( (2,3) )
array([[  3.73603959e-262,   6.02658058e-154,   6.55490914e-260],
       [  5.30498948e-313,   3.14673309e-307,   1.00000000e+000]])
```

- arange
  - recall range function ?

```
>>> np.arange( 10, 30, 5 )
array([10, 15, 20, 25])
>>> np.arange( 0, 2, 0.3 )                 # it accepts float arguments
array([ 0. ,  0.3,  0.6,  0.9,  1.2,  1.5,  1.8])
```

- linspace: more predicatable number of elements

```
>>> np.linspace( 0, 2, 9 )                 # 9 numbers from 0 to 2
array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ,  1.25,  1.5 ,  1.75,  2.  ])
>>> x = np.linspace( 0, 2*pi, 100 )        # useful to evaluate function at lots of points
>>> f = sin(x)
```

- random

```
>>> b = np.random.random((2,3))
>>> b
array([[ .69092703,  .8324276 ,  .0114541 ],
       [ .18679111,  .3039349 ,  .37600289]])
```

5 Changing Shape

- ravel

```
>>> a = np.array([[ 7.,  5.,  9.,  3.],
       [ 7.,  2.,  7.,  8.],
       [ 6.,  8.,  3.,  2.]])
>>> a.ravel() # flatten the array
array([ 7.,  5.,  9.,  3.,  7.,  2.,  7.,  8.,  6.,  8.,  3.,  2.])
>>> a.shape = (6, 2)
>>> a.transpose()
array([[ 7.,  9.,  7.,  7.,  6.,  3.],
       [ 5.,  3.,  2.,  8.,  8.,  2.]])
```

- resize
  - modify the array in place

```
>>> a
array([[ 7.,  5.],
       [ 9.,  3.],
       [ 7.,  2.],
       [ 7.,  8.],
       [ 6.,  8.],
       [ 3.,  2.]])
>>> a.resize((2,6))
>>> a
array([[ 7.,  5.,  9.,  3.,  7.,  2.],
       [ 7.,  8.,  6.,  8.,  3.,  2.]])
```

- reshape
  - returns another array with changed shape
  - -1 means the dimension is automatically calculated according to other dimensions

```
>>> a.reshape(3,-1)
array([[ 7.,  5.,  9.,  3.],
       [ 7.,  2.,  7.,  8.],
       [ 6.,  8.,  3.,  2.]])
```

6 Indexing and Slicing

- Just like list

```
>>> a = array([  0,   1,   8,  27,  64, 125, 216, 343, 512, 729])
>>> a[2]
8
>>> a[2:5]
array([ 8, 27, 64])
>>> a[:6:2] = -1000                              # modify elements in a
>>> a
array([-1000,     1, -1000,    27. -1000,   125,   216,   343,   512,   729])
>>> a[::-1]                                      # reversed a
array([  729,   512,   343,   216,   125, -1000,    27, -1000,     1, -1000])
>>> for i in a:
...         print i**(1/3.),
...
nan 1.0 nan 3.0 nan 5.0 6.0 7.0 8.0 9.0
```

- Tuple indexed (**NOT like list**)

```
>>> b = array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23],
       [30, 31, 32, 33],
       [40, 41, 42, 43]])
>>> b[2,3]
23
>>> b[:,1]                                       # the second column of b
array([ 1, 11, 21, 31, 41])
>>> b[1:3,:]                                     # the second and third row of b
array([[10, 11, 12, 13],
       [20, 21, 22, 23]])
```

- Missing indices

```
>>> b[-1]                                        # the last row. Equivalent to b[-1,:]
array([40, 41, 42, 43])
```

- Dots (. . .)
  - Means: as many as :
  - x[1,2,. . .] is equivalent to x[1,2,:,:,:],

– x[...,3] to x[:,:,:,:,3] and
– x[4,...,5,:] to x[4,:,:,5,:].

```
>>> c = np.array( [ [[  0,  1,  2],            # a 3d array (two stacked 2d arrays)
...                 [ 10, 12, 13]],
...
...                [[100,101,102],
...                 [110,112,113]] ] )
>>> c.shape
(2, 2, 3)
>>> c[1,...]                                # same as c[1,:,:] or c[1]
array([[100, 101, 102],
       [110, 112, 113]])
>>> c[...,2]                                # same as c[:,:,2]
array([[  2,  13],
       [102, 113]])
>>> c = np.array( [ [[  0,  1,  2],            # a 3d array (two stacked 2d arrays)
...                 [ 10, 12, 13]],
...
...                [[100,101,102],
...                 [110,112,113]] ] )
>>> c.shape
(2, 2, 3)
>>> c[1,...]                                # same as c[1,:,:] or c[1]
array([[100, 101, 102],
       [110, 112, 113]])
>>> c[...,2]                                # same as c[:,:,2]
array([[  2,  13],
       [102, 113]])
```

7 Enumeration

- Hierachical

```
for element in b:
    print element,
```

- Flat

```
for element in b.flat:
    print element,
```

8 Elementwise Operations

- So far similar to list
  – Seems just a convenience
  – Maybe more efficient in storage
  – But why bother?
  – compare

```
for i in range(len(a)) :
  c[i] = a[i] + b[i]
```

```
[ x + y for x, y in zip(a,b)]
```

```
a + b
```

- array op array
  - vector operation just like scalar operation
  - no loops
  - not even list comprehension

```
>>> a = np.array( [20,30,40,50] )
>>> b = np.arange( 4 )
>>> c = a-b
>>> c
array([20, 29, 38, 47])
```

- array op scalar

```
>>> b**2
array([0, 1, 4, 9])
>>> 10*sin(a)
array([ 9.12945251, -9.88031624,  7.4511316 , -2.62374854])
>>> a<35
array([True, True, False, False], dtype=bool)
```

9 Universal Functions

- Operators are nothing but functions
- Universal functions
- Unary
  - arccos/arccosh/arcsin/arcsinh/arctan/arctanh
  - cos/cosh/exp/log/log10/sin/sinh/sqrt/tan/tanh
  - ...
- Binary
  - add/subtract/multiply/divide
  - remainder
  - power
  - ...
  - Comparison
  - greater/less
  - ...

10 More indexing

- We saw indexing by
  - integers
  - slices
  - tuple of integers/slices
- Array of integers!
  - Gather operation

```
>>> a = np.arange(12)**2                       # the first 12 square numbers
>>> i = np.array( [ 1,1,3,8,5 ] )              # an array of indices
>>> a[i]                                       # the elements of a at the positions i
array([ 1,  1,  9, 64, 25])
```

```
>> palette = array( [ [0,0,0],                # black
...                       [255,0,0],           # red
...                       [0,255,0],           # green
...                       [0,0,255],           # blue
...                       [255,255,255] ] )    # white
>>> image = array( [ [ 0, 1, 2, 0 ],           # 2x4 image with color index entry
...                     [ 0, 3, 4, 0 ]  ] )
>>> palette[image]                            # 2x4 image with RGB entry
array([[[  0,   0,   0],
        [255,   0,   0],
        [  0, 255,   0],
        [  0,   0,   0]],
       [[  0,   0,   0],
        [  0,   0, 255],
        [255, 255, 255],
        [  0,   0,   0]]])
```

- Scatter operation

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

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

[NOTE] For repeat entries, take the value of last one.

- Array of booleans!
  - Pack operation

```
>>> a = np.arange(12).reshape(3,4)
>>> b = a > 4
>>> b                                    # b is a boolean with a's shape
array([[False, False, False, False],
       [False, True, True, True],
       [True, True, True, True]], dtype=bool)
>>> a[b]                                 # 1d array with the selected elements
array([ 5,  6,  7,  8,  9, 10, 11])
>>> a[b] = 0                             # All elements of 'a' higher than 4 become 0
>>> a
array([[0, 1, 2, 3],
       [4, 0, 0, 0],
       [0, 0, 0, 0]]
```

[NOTES] We have seen how elementwise operation help eliminate loops in code — what procedural construct does pack help eliminate?

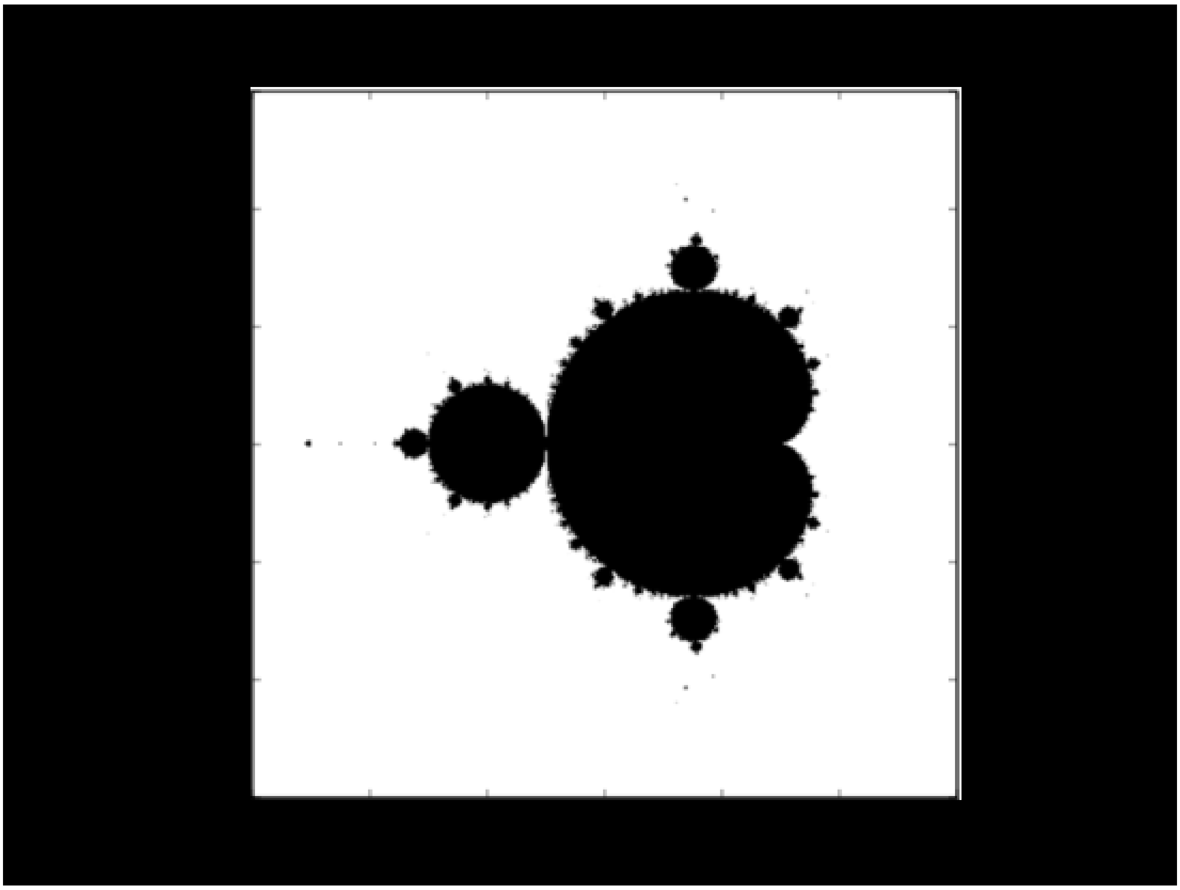
11 Broadcasting

- Elementwise function applies to arrays with matching shape
- What happens if they do not match?
  - We already see the case with array op scalar
- Rules
  - All inputs with smaller ndim will have 1s prepended in their shape
  - size of each dimension of output array will be the same as the maximum size of all inputs along that dimension
  - an input can be used if its shape in a dimension is either one or equal to the output size (maximum size)
  - if input size along a dimension is 1, first data entry will always be used (stride will be 0 in stepping)

12 Fractal Example

- Mandelbrot set
  - Given a complex number z, make a copy of the number (call it c), and then perform the following operation recursively:

```
z = z**2 + c
```



- May go to infinity - Refinement:
  - Any point z which, after 100 iterations, has a magnitude of greater than 10, belongs to the Mandelbrot set.
- Constructing a grid
  - 1D axis

```
>>> np.linspace(0, 1, num=5)
array([ 0.   ,  0.25,  0.5 ,  0.75,  1.  ])
```

```
>>> re = np.linspace(-2, 1, 1000)
>>> im = np.linspace(-1.5, 1.5, 1000)
```

- 2D grid

```
>>> x, y = np.meshgrid([1,2,3], [1,2])
>>> x
array([[1, 2, 3],
       [1, 2, 3]])
>>> y
array([[1, 1, 1],
       [2, 2, 2]])
```

```
>>> x, y = np.meshgrid(re, im)
```

- Complex grid: What's happening down there?

```
>>> z = x + 1j*y
>>> z.shape
(1000, 1000)
```

- Copies and Views
  - To save space and time, Python uses copies whenever possible:

```
# Create a new array
>>> x = np.array([1,2,3])
# View the first two elements and call it 'y'
>>> y = x[:2]
>>> y
array([1, 2])
# Modify the first element of 'y'
>>> y[0] = 3
# And note that 'x' has also changed!
>>> x
array([3, 2, 3])
```

- Use explicit copy if needed

```
>>> c = z.copy()
```

- Creating an empty image

```
>>> fractal = np.zeros(z.shape, dtype=np.uint8)
```

- All black pixels for now
  - Generate a fractal

```
for n in range(100):
    print "Iteration %d" % n
    z *= z
    z += c
```

- Remember z is a **grid** of complex numbers!

[NOTE] the **in-place** operator: Not exactly the same as z = z * 3

```
>>> mask = (np.abs(z) > 100)
>>> fractal[mask] = 255
```

[NOTE] Boolean array is used for indexing!

- Ploting

```
>>> import matplotlib.pyplot as plt
>>> plt.imshow(fractal)
>>> plt.show()
```

- Full listing
  - With color indicting how fast they escape to infinity

```
ITERATIONS = 100
DENSITY = 1000 # warning: execution speed decreases with square of DENSITY

x_min, x_max = -2, 1
y_min, y_max = -1.5, 1.5

x, y = np.meshgrid(np.linspace(x_min, x_max, DENSITY),
                   np.linspace(y_min, y_max, DENSITY))

c = x + 1j*y # complex grid
z = c.copy()
fractal = np.zeros(z.shape, dtype=np.uint8) + 255

for n in range(ITERATIONS):
    print "Iteration %d" % n

    # --- Uncomment to see different sets ---

    # Tricorn
    # z = z.conj()

    # Burning ship
    # z = abs(z.real) + 1j*abs(z.imag)

    # ---

    # Leave the lines below in place
    z *= z
    z += c

    mask = (fractal == 255) & (abs(z) > 10)
    fractal[mask] = 254 * n / float(ITERATIONS)

plt.imshow(np.log(fractal), cmap=plt.cm.hot,
           extent=(x_min, x_max, y_min, y_max))
plt.title('Mandelbrot Set')
plt.xlabel('Re(z)')
plt.ylabel('Im(z)')
plt.show()
```

13 Sum and Partial Sum

- Let's see some of Iverson's original notations (APL)
- from scalar to vector

```
iota 5

1 2 3 4 5
```

- sum: from vector to scalar

```
+/ iota 5
15
```

- partial sum: from vector to vector

```
+\ iota 5
1 3 6 10 15

+/ +\ iota 5
35
```

- reverse:

```
phi iota 5
5 4 3 2 1
```

- repeat:

```
5 rho 6
6 6 6 6 6 6
```

- What is this?

```
+/ 5 rho 6
30

5 X 6
30

+/ 5 rho 6 <-> 6 x 5
+/ iota N <-> ( (N+1) x N ) / 2
```

14 Reduction

- Suggestivity:
  - "A notation will be said to be suggestive if the forms of expressions arising in one set of problems suggest related expression which finds application in other problems"
- Sum can be generalized
  - **operator**: applies to an input function, produce an **derived** function
  - reduce and scan
- Applicable to any binary functions that are associative
  - multiply
  - and
  - or
  - min
  - max

```
5 rho 2
2 2 2 2 2
*/ 5 rho 2
32
```

Note
power is to times what times is to add!

```
5 rho 1
1 1 1 1 1
+\ 5 rho 1
1 2 3 4 5

iota 5
1 2 3 4 5
*/ iota 5
120
```

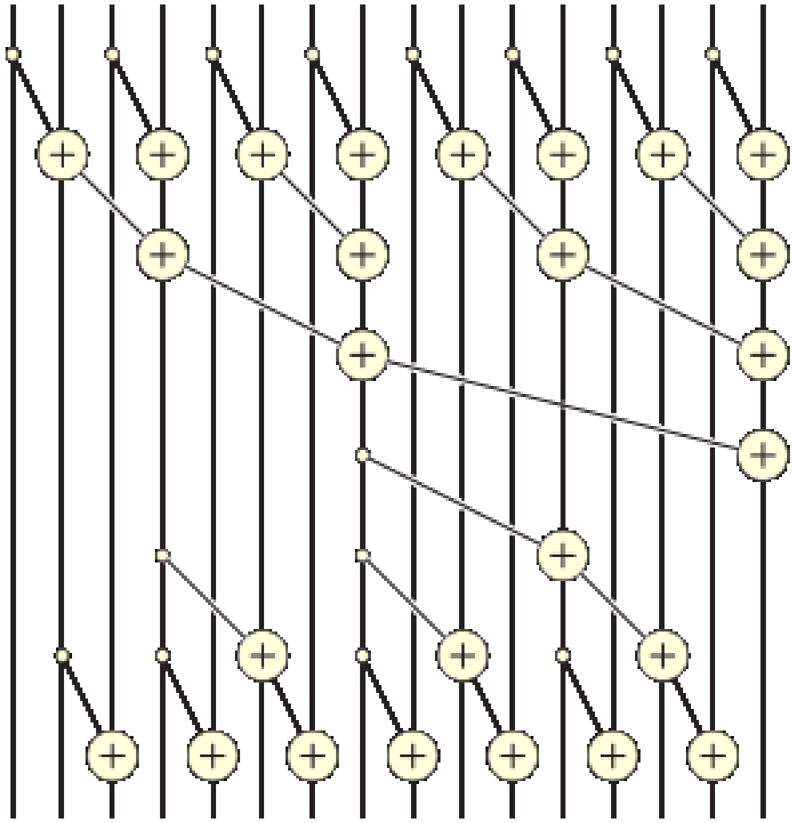
15 Reduction/Scan In Python

```
>>> a = np.arange(5)
>>> a
array( [0 1 2 3 4] )
>>> np.add.reduce(a)
10                      # that's 0 + 1 + 2 + 3 + 4
>>> np.add.accumulate(a)
array( [0 1 3 6 10] )
```

```
>>> a = np.array( [1]*10 )
>>> np.add.accumulate( a )
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

- Applicable to other universal functions too
  - np.multiply.reduce?
  - np.multiply.accumulate?

16 Parallel Reduction and Scan



17 Inner Product

- If P and Q are two vectors, then inner product + . x is

```
.....
P + . x Q <-> +/ P x Q
.....
```

  - pairwise multiplication produce intermediate vector
  - reduction over intermediate vector produce a scalar

- Applicable to any functions

```
P <- 2 3 5
Q <- 2 1 2
P +.x Q
17
P x.* Q
300
P min.+ Q
4
```

- What is Matrix Multiplication?
  - For 1-D arrays to inner product of vectors (without complex conjugation)

- For 2-D arrays it is equivalent to matrix multiplication
- For N dimensions it is a sum product over the last axis of a and the second-to-last of b

```
np.dot(a, b)[i,j,k,m] = np.sum(a[i,j,:] * b[k,:,m])
```

18 Directed Graph

- A set of nodes [QRST]
- A set of directed edges
- Connection matrix

```
0 0 0 0
0 0 0 1
1 0 0 0
1 0 0 0
```

- How to calculate
  - Out degree?
  - In degree?
  - Number of edges?
  - Related graph with direction reversed?
  - Immediately reachable neighbours?
  - Transitively reachable neighbours (transitive closure)?

19 Recap

- Think in collection
- Array programming concept
  - Shape and Layout
  - Indexing and Slicing
  - Scatter/Gather/Pack
  - Elementwise function
  - Reduce/scan/inner operators