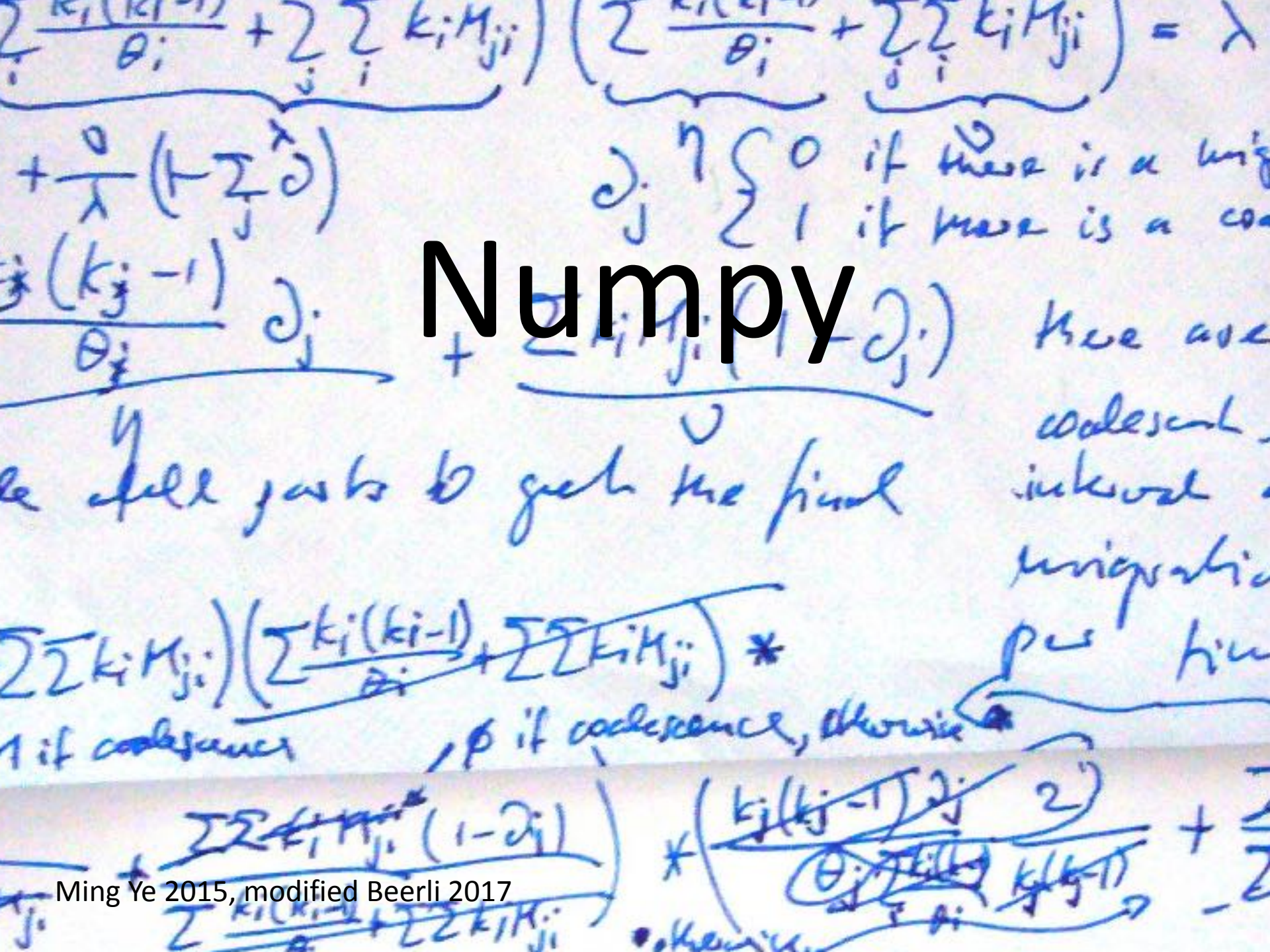
# Numpy



Ming Ye 2015, modified Beerli 2017

NumPy  is fundamental package for scientific computing with Python

- powerful N-dimensional array object
- sophisticated functions
- basic linear algebra functions
- basic Fourier transforms
- sophisticated random number capabilities
- tools for integrating Fortran code.
- tools for integrating C/C++ code.

Numpy provides

- extension package to Python for multi-dimensional arrays

- closer to hardware (efficiency)

- designed for scientific computation (convenience)
- Also known as array oriented computing

# Numpy Array

```
import numpy as np
a = np.array([0, 1, 2, 3])
print a
```

```
[0 1 2 3]
```

```
a = np.arange(10)  # 0 .. n-1
print a
```

```
[0 1 2 3 4 5 6 7 8 9]
```

```
b = np.arange(1, 9, 2) # start, end (exclusive), step
print b
```

```
[1 3 5 7]
```

```
c = np.linspace(0, 1, 6) # start, end, num-points
print c
```

```
[ 0.   0.2  0.4  0.6  0.8  1. ]
```

# Array size

```
print a
print a.shape
print len(a)
print a.size
print a.ndim
```

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

# 2 dimensional array

```
b = np.array([[0, 1, 2], [3, 4, 5]]) # 2 x 3 array
print b
print b.ndim
print b.shape
print len(b) # returns the size of the first dimension
```

```
[[0 1 2]
 [3 4 5]]
2
(2, 3)
2
```

# 3 dimensional array

```
b = np.array([[[0, 1, 2], [3, 4, 5]]])
print b
```

```
[[[0 1 2]
  [3 4 5]]]
```

```
print b.shape
```

```
(1, 2, 3)
```

```
b = np.array([[[0, 1, 2], [3, 4, 5]],[[0, 1, 2], [3, 4, 5]]])
print b
```

```
[[[0 1 2]
  [3 4 5]]

 [[0 1 2]
  [3 4 5]]]
```

```
print b.shape
```

```
(2, 2, 3)
```

# Array initialization

```
a=np.ones((2,2))
```

```
print a
```

```
[[ 1.  1.]
 [ 1.  1.]]
```

```
b = np.zeros((2, 2))
print b
```

```
[[ 0.  0.]
 [ 0.  0.]]
```

```
c = np.eye(2)
print c
```

```
[[ 1.  0.]
 [ 0.  1.]]
```

```
d = np.diag(np.array([1, 2, 3, 4]))
print d
```

```
[[1 0 0 0]
 [0 2 0 0]
 [0 0 3 0]
 [0 0 0 4]]
```

# 3 dimensional array

```
a=np.ones((2,3,4))
```

```
print a
```

```
[[[ 1.  1.  1.  1.]
  [ 1.  1.  1.  1.]
  [ 1.  1.  1.  1.]]

 [[ 1.  1.  1.  1.]
  [ 1.  1.  1.  1.]
  [ 1.  1.  1.  1.]]]
```

# Data type

Different data-types allow us to store data more compactly in memory, but most of the time we simply work with floating point numbers.

```
a = np.array([1, 2, 3])
print a.dtype
```

```
int64
```

```
b = np.array([1., 2., 3.])
print b.dtype
```

```
float64
```

# we can explicitly specify data

```
c = np.array([1, 2, 3], dtype=float)
print c.dtype
```

```
float64
```

# Other data types

```
#complex
d = np.array([1+2j, 3+4j, 5+6*1j])
print d.dtype

#boolian
e = np.array([True, False, False, True])
print e.dtype

#Strings
f = np.array(['Bonjour', 'Hello', 'Hallo'])
print f.dtype # <--- strings containing max. 7 letters
```

```
complex128
bool
|S7
```

# Slices in list vs array

```
a=range(5)
print a
```

```
[0, 1, 2, 3, 4]
```

```
b=a[1:3]
print b
```

```
[1, 2]
```

```
b[1]=10
```

```
print a
print b
```

```
[0, 1, 2, 3, 4]
[1, 10]
```

```
a=np.array(range(5))
print a
```

```
[0 1 2 3 4]
```

```
b=a[1:3]
print b
```

```
[1 2]
```

```
b[1]=10
```

```
print a
print b
```

```
[ 0  1 10  3  4]
[ 1 10]
```

# reshape

```
b = np.array([[0, 1, 2], [3, 4, 5]])
print b
```

```
[[0 1 2]
 [3 4 5]]
```

```
a=b.reshape(3,2)
print a
```

```
[[0 1]
 [2 3]
 [4 5]]
```

```
c=np.ravel(b)
print c
```

```
[0 1 2 3 4 5]
```

```
d=c.reshape(2,3)
print d
```

```
[[0 1 2]
 [3 4 5]]
```

**Ravel**: return 1-D array of input

unspecified (-1) value is inferred

```
a.reshape((2, -1))
print a
```

```
[[0 1]
 [2 3]
 [4 5]]
```

# Ravel vs Flatten

```
a = np.array([[1,2], [3,4]])
a.flatten()
print a
```

```
[[1 2]
 [3 4]]
```

```
print a.flatten()
```

```
[1 2 3 4]
```

```
b=np.ravel(a)
print b
```

```
[1 2 3 4]
```

Flatten returns a view

Ravle returns a copy

# Transpose for dimension shuffling

- If **A** is an $m \times n$ matrix then $\mathbf{A}^T$ is an $n \times m$ matrix

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

```
(2, 3)
```

```
b=a.transpose()
print b.shape
```

```
(3, 2)
```

```
print b
```

```
[[1 4]
 [2 5]
 [3 6]]
```

# Assignment

In [1]:
```
import numpy as np
```

In [30]:
```
a = np.array([[1,2,3],[4,5,6],[7,8,9]])
print a
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

In [32]:
```
b = a
b[0][0]=111
print a
print b
```

```
[[111   2   3]
 [  4   5   6]
 [  7   8   9]]
[[111   2   3]
 [  4   5   6]
 [  7   8   9]]
```

PROBLEMATIC!

# Assignment: TWO POSSIBLE SOLUTIONS

In [31]:
```
a = np.array([[1,2,3],[4,5,6],[7,8,9]])
b[:] = a
b[0][0]=111
print a
print b
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
[[111   2   3]
 [  4   5   6]
 [  7   8   9]]
```

In [33]:
```
a = np.array([[1,2,3],[4,5,6],[7,8,9]])
b = 1 * a
b[0][0]=111
print a
print b
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
[[111   2   3]
 [  4   5   6]
 [  7   8   9]]
```

# ndenumerate

- Multidimensional index iterator.
- Return an iterator yielding pairs of array coordinates and values.

```
a = np.array([[1, 2], [3, 4]])
for index, x in np.ndenumerate(a):
    print index, x
```

```
(0, 0) 1
(0, 1) 2
(1, 0) 3
(1, 1) 4
```

# Adding and multiplying element wise

```
a = np.array(range(12))
b = np.array(range(12))
c = a + b
print c
```

```
[ 0  2  4  6  8 10 12 14 16 18 20 22]
```

```
a = np.array(range(12))
b = np.array(range(12))
c = a * b
print c
```

```
[  0   1   4   9  16  25  36  49  64  81 100 121]
```

```
a = np.array([[1, 2], [3, 4]])
b = np.array([[1, 2], [3, 4]])
c=a * b
print c
```

```
[[ 1  4]
 [ 9 16]]
```

# Matrix multiplication

```
a = np.array([[1, 2], [3, 4]])
b = np.array([[1, 2], [3, 4]])
```

```
c=np.dot(a,b)
print c
```

```
[[ 7 10]
 [15 22]]
```

array

```
a = np.matrix([[1, 2], [3, 4]])
b = np.matrix([[1, 2], [3, 4]])
```

matrix

```
c= a*b
print c
```

```
[[ 7 10]
 [15 22]]
```

# Many operations work with arrays

- abs, cos, sin, ...
- accumulate, all
- append, arange, argsort, array_split, matrix,
- asmatrix (not copied)
- average, copy, cross,delete
- linear algebra: diagonal, dot, eye, fill, fix, flat (iterator), flatten, indices, inner, insert, item, linspace, mat, maximum, ndim, outer, pinv, repeat, reshape, shape, swapaxis, tile, tolist, transpose, zeros

# Converting numpy array to python list

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

```
[[1 2 3]
 [4 5 6]]
```

```
b=a.tolist()
print b
```

```
[[1, 2, 3], [4, 5, 6]]
```

# Regular loop vs vectorization

Example : 2D Lapacian equation

$$\frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2} = \psi_{xx} + \psi_{yy} = 0$$

Numerical solution using finite difference

$$T_{ij} = \frac{\gamma_y^2 (T_{i-1,j} + T_{i+1,j} + r_x (T_{i+1,j} - T_{i-1,j})) + \gamma_x^2 (T_{i,j-1} + T_{i,j+1} + r_y (T_{i,j+1} - T_{i,j-1}))}{2(\gamma_x^2 + \gamma_y^2)}$$

```
import time

nx=1000; ny=1000
u = np.zeros((nx, ny), 'd')
u[0,:]=.1; u[-1,:]=.1; u[:,0]=.2; u[:,-1]=.2

dx=1; dy=1; dx2=1; dy2=1
#regular loop
tic=time.clock()
for i in range(1, nx-1):
        for j in range(1, ny-1):
            u[i,j] = ((u[i-1, j] + u[i+1, j])*dy**2 + (u[i, j-1] + u[i, j+1])*dx**2)/(2.0*(dx**2 + dy**2))
toc=time.clock()
print toc-tic
#print u

###Vectorization
tic=time.clock()
dnr_inv = 0.5/(dx2 + dy2)
u[1:-1, 1:-1] = ((u[0:-2, 1:-1] + u[2:, 1:-1])*dy2 +(u[1:-1,0:-2] + u[1:-1, 2:])*dx2)*dnr_inv
toc=time.clock()
print toc-tic
#print u
```

# comparison

| | N=10 | N=100 | N=1000 |
|---|---|---|---|
| loop | 0.000897000000002 | 0.106809 | 6.922471 |
| Vectorization | 0.000724000000002 | 0.00067 | 0.023252 |

# Read and Write files

- Loadtxt : Construct an array from data in a text or binary file.

```
y = np.loadtxt(c, delimiter='\t')
```

- Savetxt: save as delimitated text

```
np.savetxt(path, A, delimiter='\t')
```

# Functions

- Functions with scalar arguments
  - scalar: float, int
- Functions with array arguments
  - automatically speedup

```
def expression(x,y):
    z = np.cos(x) * 3
    return x+np.sin(z+np.cos(y))
```

```
L1=[]
tic=time.clock()
for x in range(100):
    L1.append(expression(x, 2+x))
toc=time.clock()

print toc-tic

tic=time.clock()
x = np.array(range(100))
y = 2. + np.array(range(100))
L2=expression(x,y)
toc=time.clock()
print toc-tic

#print L1
#print L2
```

```
0.027195
0.000603000000005
```