2018

INTRODUCTION TO NUMERICAL ANALYSIS

Lecture 2-4: NumPy array & linear algebra (II)

Kai-Feng Chen
National Taiwan University

MATRICES OPERATIONS

- Up to now we have gone through the following operations with NumPy, mostly the operations are still "technical-wise":
  - Creation of array
  - Basic "element-wise" operations
  - Indexing and slicing
  - Reductions (e.g. sum(), mean(), std()...)
- We will start to discuss few more operations that you may have learned from the corresponding mathematics lectures:
  - Transpose
  - Determinant
  - Solving linear equations
  - Matrix inversion

TRANSPOSE

- It is almost trivial to produce a transposed array with NumPy. For example:

```
>>> a = np.arange(4).reshape((2,2))
>>> a
array([[0, 1],
       [2, 3]])
>>> np.transpose(a)
array([[0, 2],
       [1, 3]])
>>> a.T ⇐ a short cut
array([[0, 2],
       [1, 3]])
```

TRANSPOSE (II)

- The transpose does not work on 1D array:

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

- The transpose will not create a new array copy, instead, it creates a **view** of the original array.

```
>>> a = np.arange(4).reshape((2,2))
>>> b = a.T
>>> a is b
False
>>> np.may_share_memory(a,b)
True
```

TRANSPOSE (III)

- Let's test a basic transpose property:

$$(A \cdot B \cdot C)^T = C^T \cdot B^T \cdot A^T$$

```
(A*B*C)^T =
[[ 0.66778556  0.72427017  0.57545512]
 [ 0.56744982  0.56148092  0.4579648 ]
 [ 0.34617964  0.38819541  0.30567286]]
C^T * B^T * A^T =
[[ 0.66778556  0.72427017  0.57545512]
 [ 0.56744982  0.56148092  0.4579648 ]
 [ 0.34617964  0.38819541  0.30567286]]
...
```

```
import numpy as np

A = np.random.rand(9).reshape((3,3))
B = np.random.rand(9).reshape((3,3))
C = np.random.rand(9).reshape((3,3))

M = A.dot(B).dot(C).T
N = C.T.dot(B.T).dot(A.T)

print('(A*B*C)^T =\n',M)
print('C^T * B^T * A^T =\n',N)
print('Difference =\n',M-N)
```

l204-example-01.py

⇐ A,B,C are all 3x3 random matrix

⇐ Remember you have to use **dot()** rather than "*" in order to perform the matrix product!

DETERMINANT

- Calculation of the determinant (by its definition?):

$$|A| = \sum_{i=1}^{n} (-1)^{i+j} A_{ij} |R_{ij}|$$

If we simply pick up the first row and expands:

$$(+1)\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix} + (-1)\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix} \cdots\cdots$$

Residual matrix R_{ij}

R_{ij}

Let's implement such a calculation by ourselves!

DETERMINANT (II)

- Let's try a straightforward example implementation:

⇓ if it's 2x2, calculate the determinant directly.

⇓ buffer for R_{ij}

⇑ odd: –, even: +

```
def det_rec(A):
    if A.shape==(2,2): return A[0,0]*A[1,1]-A[0,1]*A[1,0]

    det = 0.
    reduced = np.zeros((A.shape[0]-1,A.shape[1]-1))

    for i in range(A.shape[1]):
        reduced[:,:i] = A[1:,:i]
        reduced[:,i:] = A[1:,i+1:]
        r = A[0,i]*det_rec(reduced)

        if i % 2==1: det -= r
        else:        det += r

    return det

T = np.array([[1.,1.,2.],[2.,1.,1.],[1.,2.,1.]])
print('T =\n',T)
print('|T| =',det_rec(T))
```

```
T =
[[ 1.  1.  2.]
 [ 2.  1.  1.]
 [ 1.  2.  1.]]
|T| = 4.0
```

l204-example-02.py (partial)

PROPERTIES OF DETERMINANT

- If you check your linear algebra text book, you may find the following properties easily —
  - Transpose: $|\mathbf{A}| = |\mathbf{A}^T|$
  - Product: $|\mathbf{AB}| = |\mathbf{A}|\,|\mathbf{B}|$
  - Row/column interchanges:
    Sign flipped after exchange any two rows/columns.
  - Removing factors:
    Apply a common factor $\boldsymbol{\lambda}$ to any row/column, resulting a determinant of $\boldsymbol{\lambda}|\mathbf{A}|$.
  - Identical rows or columns:
    If two identical rows/columns exist, $|\mathbf{A}|=\mathbf{0}$.
  - Adding a constant multiple of one row (column) to another:
    Determinant will not change if row/column operations applied.

PROPERTIES OF DETERMINANT (II)

- Let's quickly test some of these properties!
  - Transpose: $|\mathbf{A}| = |\mathbf{A}^T|$
  - Product: $|\mathbf{AB}| = |\mathbf{A}|\,|\mathbf{B}|$

```
|A| = -0.0148187224319
|A.T| = -0.0148187224319

|A*B| = 0.0013109818352
|A|*|B| = 0.0013109818352
```

```python
A = np.random.rand(25).reshape((5,5))
B = np.random.rand(25).reshape((5,5))

print('|A| =',det_rec(A))
print('|A.T| =',det_rec(A.T))

print('|A*B| =',det_rec(A.dot(B)))
print('|A|*|B| =‘,det_rec(A)*det_rec(B))
```

l204-example-02.py (partial)

PROPERTIES OF DETERMINANT (III)

- Adding a constant multiple of one row (column) to another: Determinant will not change if **row/column operations** applied.

```
A = np.random.rand(25).reshape((5,5))

print('|A| =',det_rec(A))
A[1,:] += A[0,:]*0.5      ⇐ adding the 1st row (times 0.5) to 2nd row
print('|A| =',det_rec(A))
```

l204-example-02.py (partial)

```
|A| = -0.014818722431 9
|A| = -0.014818722431 9
```

INDEED IT IS WORKING, BUT...

- Although python is not an efficient language for computing speed itself, have you tried to increase the size of array and see the how long it takes?

```
|A(6x6)| = 0.0400768615453
0.001898 sec.

|A(7x7)| = -0.00232927423832
0.011164 sec.

|A(8x8)| = 0.00320616779202
0.096507 sec.

|A(9x9)| = -0.00587454657049
0.820429 sec.

|A(10x10)| = 0.0419854509311
8.227578 sec.

|A(11x11)| = -0.107359595362
93.831300 sec.
```

```python
import timeit

def speed_test(n):
    A = np.random.rand(n**2).reshape((n,n))
    print('|A('+str(n)+'x'+str(n)+')| =',det_rec(A))

for n in range(2,12):
    t = timeit.timeit('speed_test('+str(n)+')',
    'from __main__ import speed_test',number=1)
    print('%.6f sec.\n' % t)
```

l204-example-02a.py (partial)

# of operations (multiplications) ~ ***N!***
So, for a 10x10 array takes ~ 3.6M operations.

A (MUCH) MORE EFFICIENT WAY

- Remember —
  - Determinant will not change if row / column operations applied.
  - Here comes the **Gaussian elimination**:

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix} \xrightarrow{\textit{Perform row operations until becoming a upper triangular matrix}} \begin{bmatrix} \boxed{A'_{11}} & A'_{12} & A'_{13} & A'_{14} \\ 0 & \boxed{A'_{22}} & A'_{23} & A'_{24} \\ 0 & 0 & \boxed{A'_{33}} & A'_{34} \\ 0 & 0 & 0 & \boxed{A'_{44}} \end{bmatrix}$$

Thus $|A| = A'_{11} \cdot A'_{22} \cdot A'_{33} \cdot A'_{44}$

Number of operations ~ ***$N^3/3$***

A rank 10x10 matrix ~ 300 operations

GAUSSIAN ELIMINATION

- An example implementation:

```
def det_gau(A):
    tmp = A.copy() ⇐ preserve the original array A
    for i in range(A.shape[0]):
        for j in range(i+1,A.shape[0]):
            scale = tmp[j,i]/tmp[i,i]
            tmp[j,:] -= tmp[i,:]*scale
    det = 1.
    for i in range(A.shape[0]):
        det *= tmp[i,i]
    return det
```

Gaussian elimination

l204-example-03.py (partial)

$$|A| = \prod_{i=1}^{N} A'_{ii}$$

$$\begin{array}{cc}
\mathtt{i} & \\
 & \\
\mathtt{j} & \\
 & \\
 &
\end{array}
\begin{array}{c}
\begin{bmatrix}
A_{11} & A_{12} & A_{13} & A_{14} \\
0 & A'_{22} & A'_{23} & A'_{24} \\
\boxed{0} & A'_{32} & A'_{33} & A'_{34} \\
A_{41} & A_{42} & A_{43} & A_{44}
\end{bmatrix} \\
\begin{array}{cccc} \mathtt{i} & & & \end{array}
\end{array}$$

$$\begin{array}{c}
 \\
\mathtt{i} \\
\mathtt{j} \\
 \\

\end{array}
\begin{array}{c}
\begin{bmatrix}
A_{11} & A_{12} & A_{13} & A_{14} \\
0 & A'_{22} & A'_{23} & A'_{24} \\
0 & \boxed{0} & A''_{33} & A''_{34} \\
0 & A'_{42} & A'_{43} & A'_{44}
\end{bmatrix} \\
\begin{array}{cccc} & \mathtt{i} & & \end{array}
\end{array}$$

$$\begin{array}{c}
 \\
 \\
\mathtt{i} \\
\mathtt{j} \\

\end{array}
\begin{array}{c}
\begin{bmatrix}
A_{11} & A_{12} & A_{13} & A_{14} \\
0 & A'_{22} & A'_{23} & A'_{24} \\
0 & 0 & A''_{33} & A''_{34} \\
0 & 0 & \boxed{0} & A'''_{44}
\end{bmatrix} \\
\begin{array}{cccc} & & \mathtt{i} & \end{array}
\end{array}$$

GAUSSIAN ELIMINATION (II)

- Let's do a speed test again –

```
|A(9x9)| = -0.00171469542351
0.000199 sec.

|A(10x10)| = 0.048346343439
0.000233 sec.

|A(11x11)| = 0.0149438621403
0.000274 sec.

|A(30x30)| = 10.2745647461
0.001835 sec.

|A(100x100)| = -2.6347483347e+24
0.016603 sec.

|A(300x300)| = -6.94240345784e+145
0.171189 sec.

|A(1000x1000)| = inf
2.424678 sec.
```

l204-example-03.py (output)

A much more efficient calculation of the same thing!

The Gaussian elimination requires much less multiplications. However, this is not the full story. In most of the cases, you will have to consider if your problem is CPU bound, or memory bound, or both. A full discussion of speed optimization is beyond the scope of this lecture.

WITH SCIPY LINEAR ALGEBRA ROUTINE

- Python usually is not a very efficient language in terms of computing speed. This is why the core part of SciPy/NumPy is usually written in a different language (or calling some existing high performance package).
- Indeed there is a built-in function **`linalg.det()`** for calculating the determinant coming with **SciPy**, and *it is very fast*:

```
>>> import numpy as np
>>> import scipy.linalg as linalg
>>> A = np.random.rand(100).reshape((10,10))
>>> linalg.det(A)
-0.0386860978752376
```

Note: there is a **`numpy.linalg`** as well, but with less functions you can find in **`scipy.linalg`**.

INTERMISSION

- You can test those untested properties of determinant, e.g.:
  - Sign flipped after exchange any two rows/columns.
  - Apply a common factor $\boldsymbol{\lambda}$ to any row/column, resulting a determinant of $\boldsymbol{\lambda}|\mathbf{A}|$.
  - If two identical rows/columns exist, $|\mathbf{A}|=\mathbf{0}$.
- Instead of those homemade functions **`det_rec()`** and **`det_gau()`** modify the l204-example-03.py and use the built-in function **`linalg.det()`** for calculating the determinant.
  And see how fast it runs up to a 1000x1000 square matrix!

SOLVING LINEAR EQUATIONS

- Solving of the linear equations, a trivial method?

$$\begin{bmatrix} A_{11} & A_{12} & \dots & A_{1N} \\ A_{21} & A_{22} & \dots & A_{2N} \\ \vdots & \vdots & & \vdots \\ A_{N1} & A_{N2} & \dots & A_{NN} \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{bmatrix}$$

diagonalize w/ row operations

$$\begin{bmatrix} A'_{11} & 0 & \dots & 0 \\ 0 & A'_{22} & \dots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & A'_{NN} \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} = \begin{bmatrix} b'_1 \\ b'_2 \\ \vdots \\ b'_N \end{bmatrix}$$

$$x_i = \frac{b'_i}{A'_{ii}}$$

This is the *Gaussian-Jordan elimination,* *as we learned from high schools?*

GAUSSIAN-JORDAN ELIMINATION

- An example implementation:

Actually, we can solve several columns in a single operation!

$$\begin{bmatrix} A_{11} & A_{12} & \dots & A_{1N} \\ A_{21} & A_{22} & \dots & A_{2N} \\ \vdots & \vdots & & \vdots \\ A_{N1} & A_{N2} & \dots & A_{NN} \end{bmatrix} \times \begin{bmatrix} x_{11} \dots x_{1M} \\ x_{21} \dots x_{2M} \\ \vdots \\ x_{N1} \dots x_{NM} \end{bmatrix} = \begin{bmatrix} b_{11} \dots b_{1M} \\ b_{21} \dots b_{2M} \\ \vdots \\ b_{N1} \dots b_{NM} \end{bmatrix}$$

```
def solve_gau1(A,b):

    tmp = A.copy()
    out = b.copy()

    for i in range(A.shape[0]):
        for j in range(A.shape[0]):
            if j==i: continue

            scale = tmp[j,i]/tmp[i,i]
            tmp[j,:] -= tmp[i,:]*scale
            out[j,:] -= out[i,:]*scale

    for i in range(A.shape[0]):
        out[i,:] /= tmp[i,i]

    return out
```

Gaussian-Jordan elimination

$$x_i = \frac{b'_i}{A'_{ii}}$$

l204-example-04.py (partial)

Size of A: NxN
b: NxM

GAUSSIAN-JORDAN ELIMINATION (II)

```
Matrix A =
[[ 0.4371515   0.63767654  0.95401163]
 [ 0.14512069  0.41424897  0.30485322]
 [ 0.07114006  0.40952833  0.07784141]]
Matrix b =
[[ 0.09021453]
 [ 0.78508836]
 [ 0.1562952 ]]
Matrix x =
[[-43.26069725]
 [  4.7089612 ]
 [ 16.77013009]]
Ax-b =
[[  2.33146835e-15]
 [  1.88737914e-15]
 [  3.33066907e-16]]
```

l204-example-04.py (output)

- Solve the linear equations as designed!

```
A = np.random.rand(9).reshape((3,3))
b = np.random.rand(3).reshape((3,1))

x = solve_gau1(A,b)

print('Matrix A =\n',A)
print('Matrix b =\n',b)
print('Matrix x =\n',x)
print('Ax-b =\n',A.dot(x)-b)
```

l204-example-04.py (partial)

OF OPERATIONS

Size of A: **N×N**

b: **N×M**

$$A \cdot x = b$$

- The cost of this **Gaussian-Jordan elimination** is roughly **~N³/2 + N²M** (counts on the multiplications of doubles)

  for matrix A *matrix b*

- A slightly improved method, **Gaussian elimination with back substitution** cost **~N³/3 + N²M/2 + N²M/2**

  for matrix A *matrix b* *back substitution*

- Another classical method, the **LU decomposition** has a similar operation counts **~N³/3 + N²M/2 + N²M/2**

  LU decomposition *forward substitution* *back substitution*

GAUSSIAN ELIMINATION + BACK SUBSTITUTION

- The idea is actually very simple. Let's performance the elimination up to the triangular matrix only:

$$\begin{bmatrix} A'_{11} & A'_{12} & A'_{13} & A'_{14} \\ 0 & A'_{22} & A'_{23} & A'_{24} \\ 0 & 0 & A'_{33} & A'_{24} \\ 0 & 0 & 0 & A'_{44} \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \end{bmatrix}$$

Solving $\boldsymbol{x_i}$ with a reversed order:

$$x_4 = b'_4 / A'_{44}$$

$$x_3 = [b'_3 - (A'_{34} x_4)] / A'_3 3$$

$$x_2 = [b'_2 - (A'_{24} x_4 + A'_{23} x_3)] / A'_{22}$$

$$x_1 = [b'_1 - (A'_{14} x_4 + A'_{13} x_3 + A'_{12} x_2)] / A'_{11}$$

One can already solve the linear equations without the full elimination!

GAUSSIAN ELIMINATION + BACK SUBSTITUTION (II)

- An example implementation (again):

```
def solve_gau2(A,b):

    tmp = A.copy()
    out = b.copy()

    for i in range(A.shape[0]):
        for j in range(i+1,A.shape[0]):

            scale = tmp[j,i]/tmp[i,i]
            tmp[j,:] -= tmp[i,:]*scale
            out[j,:] -= out[i,:]*scale

    for i in range(A.shape[0])[::-1]:
            for j in range(i+1,A.shape[0]):
                out[i,:] -= out[j,:]*tmp[i,j]
            out[i,:] /= tmp[i,i]

    return out
```

Gaussian elimination

Back substitution

l204-example-04a.py (partial)

Surely you can try this code and see if it is working or not.

LU DECOMPOSITION

- Since the triangular matrix can reduce the cost on the computation, one can actually decompose a matrix into the **"lower"** part and the **"upper"** part:

$$\overset{\textbf{L}}{\begin{bmatrix} \alpha_{11} & 0 & 0 & 0 \\ \alpha_{21} & \alpha_{22} & 0 & 0 \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & 0 \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \alpha_{44} \end{bmatrix}} \times \overset{\textbf{U}}{\begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ 0 & \beta_{22} & \beta_{23} & \beta_{24} \\ 0 & 0 & \beta_{33} & \beta_{34} \\ 0 & 0 & 0 & \beta_{44} \end{bmatrix}} = \overset{\textbf{A}}{\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix}}$$

$$L \times U = A$$

$$Ax = (L \cdot U)x = L(Ux) = b$$

$$L \cdot y = b$$

$$U \cdot x = y$$

Solve ***y*** & ***x*** with forward substitution & back substitution

LU DECOMPOSITION (II)

- How to do this decomposition?
  The diagonal elements are redundant; we could assign **$\alpha_{ii} = 1$** first

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ \alpha_{21} & 1 & 0 & 0 \\ \alpha_{31} & \alpha_{32} & 1 & 0 \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & 1 \end{bmatrix} \times \begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ 0 & \beta_{22} & \beta_{23} & \beta_{24} \\ 0 & 0 & \beta_{33} & \beta_{34} \\ 0 & 0 & 0 & \beta_{44} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix}$$

First solve $\boldsymbol{\beta_{11}}$;

Exact $\boldsymbol{\alpha_{i1}}$ with $\boldsymbol{\beta_{11}}$;

Exact $\boldsymbol{\beta_{i2}}$ with $\boldsymbol{\alpha_{i1}}$;

Exact $\boldsymbol{\alpha_{i2}}$ with $\boldsymbol{\beta_{i2}}$;

......

$$\beta_{ij} = A_{ij} - \sum_{k=1}^{i-1} \alpha_{ik}\beta_{kj}$$

$$\alpha_{ij} = \frac{1}{\beta_{jj}}(A_{ij} - \sum_{k=1}^{j-1} \alpha_{ik}\beta_{kj})$$

LU DECOMPOSITION: IMPLEMENTATION

$$tmp = \begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ \alpha_{21} & \beta_{22} & \beta_{23} & \beta_{24} \\ \alpha_{31} & \alpha_{32} & \beta_{33} & \beta_{34} \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \beta_{44} \end{bmatrix}$$

First solve $\boldsymbol{\beta}_{11}$;
Exact $\boldsymbol{\alpha}_{i1}$ with $\boldsymbol{\beta}_{11}$;
Exact $\boldsymbol{\beta}_{i2}$ with $\boldsymbol{\alpha}_{i1}$;
Exact $\boldsymbol{\alpha}_{i2}$ with $\boldsymbol{\beta}_{i2}$;
......

```
def solve_LU(A,b):

    tmp = A.copy()
    out = b.copy()

    for c in range(A.shape[1]):
        for r in range(A.shape[0]):
            if c>=r:
                tmp[r,c] -= (tmp[r,:r]*tmp[:r,c]).sum()⇐ solve for β
            else:
                tmp[r,c] -= (tmp[r,:c]*tmp[:c,c]).sum()⇐ solve for α
                tmp[r,c] /= tmp[c,c]
```

LU Decomposition

l204-example-05.py (partial)

LU DECOMPOSITION: IMPLEMENTATION (II)

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ \alpha_{21} & 1 & 0 & 0 \\ \alpha_{31} & \alpha_{32} & 1 & 0 \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & 1 \end{bmatrix} \times \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

Solve ***y*** with forward substitution

```
for i in range(A.shape[0]):
    for j in range(i):
        out[i,:] -= out[j,:]*tmp[i,j]
```

Forward substitution

```
for i in range(A.shape[0])[::-1]:
    for j in range(i+1,A.shape[1]):
        out[i,:] -= out[j,:]*tmp[i,j]
    out[i,:] /= tmp[i,i]
```

Back substitution

```
return out
```

l204-example-05.py (partial)

$$\begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ 0 & \beta_{22} & \beta_{23} & \beta_{24} \\ 0 & 0 & \beta_{33} & \beta_{34} \\ 0 & 0 & 0 & \beta_{44} \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}$$

Solve ***x*** with back substitution

SOLVE THE EQUATIONS WITH SCIPY ROUTINE

- The example codes shown above are just for a "demonstration" (and they are all very slow!).
- For your real work, it is much better to use the `linalg.solve()` function from SciPy:

```
import numpy as np
import scipy.linalg as linalg

A = np.random.rand(9).reshape((3,3))
b = np.random.rand(3).reshape((3,1))

x = linalg.solve(A,b)
```

l204-example-06.py (partial)

This `linalg.solve()` function actually calls to the `gesv()` function from the **LAPACK** package, which is written in Fortran. The implemented method is the **LU decomposition** and is very fast.

MATRIX INVERSION

- Well, Gaussian eliminations again:
  joint an identity matrix to the matrix A

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Diagonalize with row operations

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} A'_{11} & A'_{12} & A'_{13} & A'_{14} \\ A'_{21} & A'_{22} & A'_{23} & A'_{24} \\ A'_{31} & A'_{32} & A'_{33} & A'_{34} \\ A'_{41} & A'_{42} & A'_{43} & A'_{44} \end{bmatrix}$$

The joint identity matrix becomes the **inverse matrix of A**

MATRIX INVERSION (II)

- Actually, if we think carefully, the matrix inversion is not really different from solving of the linear equations:

$$A \cdot x = b \text{ given } b = \mathrm{I} \Rightarrow \text{solve } A \cdot x = \mathrm{I},\ x = A^{-1}$$

```python
A = np.random.rand(9).reshape((3,3))
I = np.eye(3)

Ainv = solve_gau2(A,I)

print('Matrix A =\n',A)
print('Matrix A^-1 =\n',Ainv)
print('A*A^-1 =\n',A.dot(Ainv))
```

l204-example-07.py (partial)

```
... ....
A*A^-1 =
[[  1.00000000e+00  -1.11022302e-16  -2.22044605e-16]
 [ -1.66533454e-16   1.00000000e+00   4.44089210e-16]
 [ -2.22044605e-16  -4.44089210e-16   1.00000000e+00]]
```

l204-example-07.py (output)

MATRIX INVERSION (III)

- Surely there is a built-in function to do this work, and no needs of allocating an identity matrix first — `linalg.inv()`:

```python
A = np.random.rand(9).reshape((3,3))
Ainv2 = linalg.inv(A)
print('A*A^-1 [calculated by linalg.inv()] =\n',A.dot(Ainv2))
```

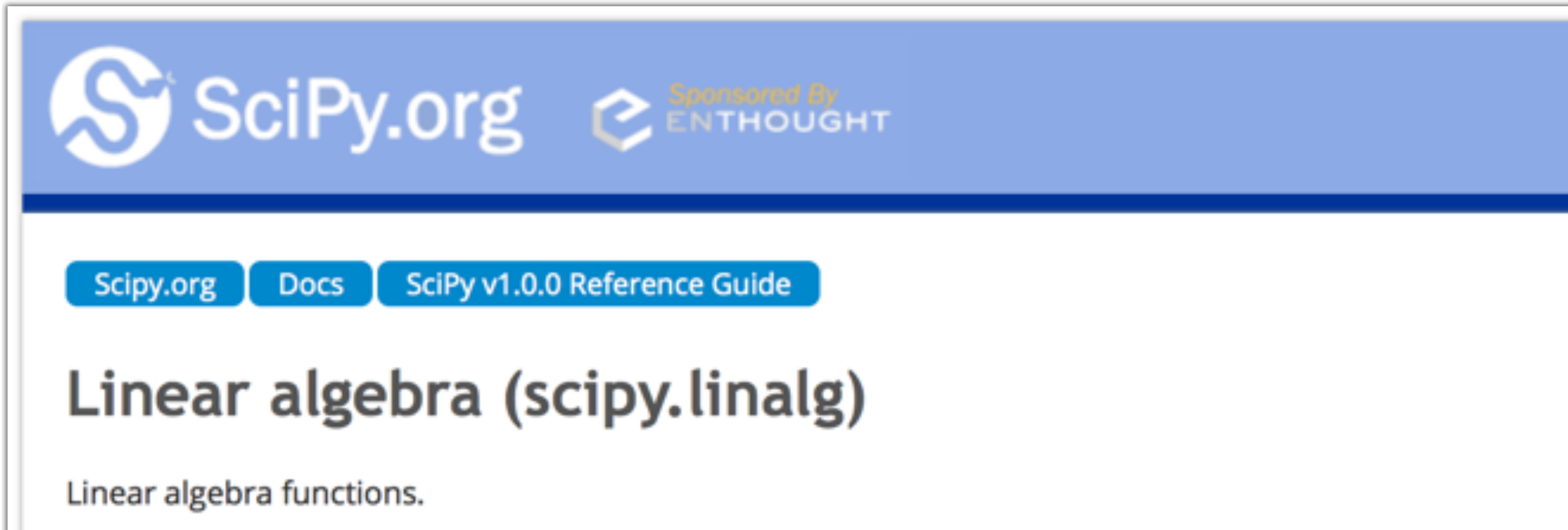
l204-example-07.py (partial)

```
... ....
A*A^-1 [calculated by linalg.inv()] =
[[  1.00000000e+00  -4.44089210e-16   0.00000000e+00]
 [  5.55111512e-17   1.00000000e+00  -1.11022302e-16]
 [  0.00000000e+00   0.00000000e+00   1.00000000e+00]]
```

l204-example-07.py (output)

COMMENTS

- Although we have partially introduced *"how things work"* and provided some example code, but please do not use those naive codes to do your real calculation work. The functions in SciPy (which is calling LAPACK/BLAS package) are much more efficient.
- It might be wise to take a look of the **`scipy.linalg`** reference manual, you'll see lots of functions available there: http://docs.scipy.org/doc/scipy/reference/linalg.html

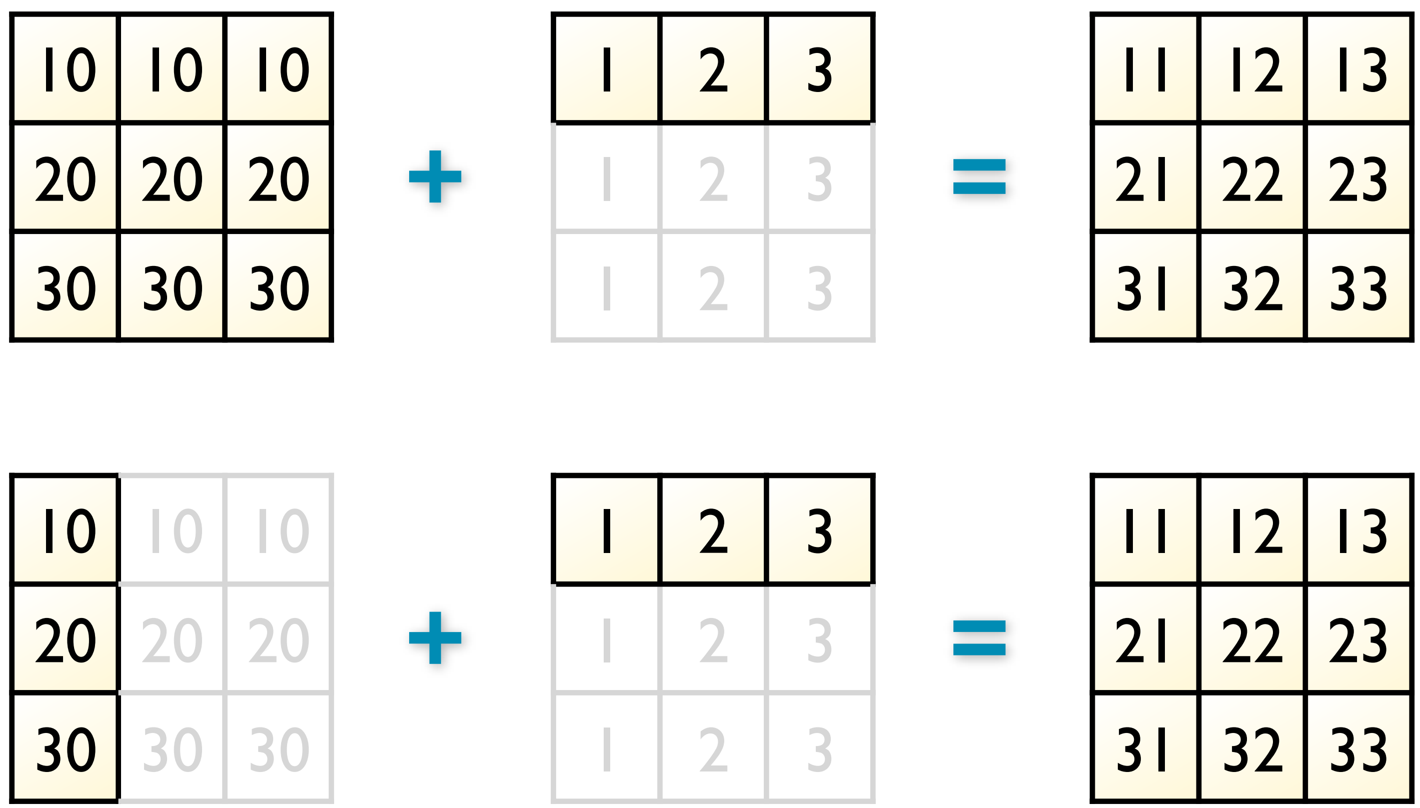


INTERMISSION

- ***In theory*** — both the Gaussian elimination with back substitution and the LU decomposition has a similar multiplication counts of **$\sim N^3/3 + N^2M/2 + N^2M/2$**.
- If **M** = 1 (just solve single vector of *b*), the operation counts is proportional to **$\sim N^3/3 + N^2$**. Try to see if the time spent (using the **`timeit`** module introduced earlier) is proportional to this counts or not.

MORE NUMPY OPERATIONS: BROADCASTING

- Here we will discuss few not-yet-covered NumPy array operations. One of them is **broadcasting**, which can be demonstrated as following:



There is more than one way to produce the resulting array!

BROADCASTING

- Example coding is given below:

```
>>> a = np.array([1,2,3])
>>> b = np.array([[10]*3,[20]*3,[30]*3])
>>> b
array([[10, 10, 10],
       [20, 20, 20],
       [30, 30, 30]])
>>> a+b
array([[11, 12, 13],
       [21, 22, 23],
       [31, 32, 33]])
>>> c = np.array([[10],[20],[30]])
>>> a+c
array([[11, 12, 13],
       [21, 22, 23],
       [31, 32, 33]])
```

⇐ an **1x3** array plus a **3x1** array, resulting a **3x3** array.

CONCATENATE

```
>>> a = np.ones((2,2))
>>> a
array([[ 1.,  1.],
       [ 1.,  1.]])
>>> b = np.zeros((2,2))
>>> b
array([[ 0.,  0.],
       [ 0.,  0.]])
>>> np.concatenate((a,b))
array([[ 1.,  1.],
       [ 1.,  1.],
       [ 0.,  0.],
       [ 0.,  0.]])
>>> np.concatenate((a,b),axis=1)
array([[ 1.,  1.,  0.,  0.],
       [ 1.,  1.,  0.,  0.]])
```

⇐ Note **concatenate()** function requires a tuple as the argument.

- This operation is kind of obvious, joint two arrays into one:

VSTACK/HSTACK

```
>>> a = np.ones((2,2))
>>> a
array([[ 1.,  1.],
       [ 1.,  1.]])
>>> b = np.zeros((2,2))
>>> b
array([[ 0.,  0.],
       [ 0.,  0.]])
>>> np.vstack((a,b))
array([[ 1.,  1.],
       [ 1.,  1.],
       [ 0.,  0.],
       [ 0.,  0.]])
>>> np.hstack((a,b))
array([[ 1.,  1.,  0.,  0.],
       [ 1.,  1.,  0.,  0.]])
```

⇐ Both **vstack()/hstack** function requires a tuple as the argument.

- The vstack and hstack work in a very similar way to the concatenate() as introduced earlier.

INDEXING ROUTINE

```
>>> a = np.ones((2,2))
>>> a
array([[ 1.,  1.],
       [ 1.,  1.]])
>>> b = np.zeros((2,2))
>>> b
array([[ 0.,  0.],
       [ 0.,  0.]])
>>> np.r_[a,b]
array([[ 1.,  1.],
       [ 1.,  1.],
       [ 0.,  0.],
       [ 0.,  0.]])
>>> np.c_[a,b]
array([[ 1.,  1.,  0.,  0.],
       [ 1.,  1.,  0.,  0.]])
```

⇐ Note **r_** and **c_** are not functions!

- You can even do the same thing with very short indexing routines!

SEARCHING IN DATA

- There are several convenient routines to do searching within the given array, for example, finding the index of the largest/smallest element.

```
>>> a = np.array([2,3,1,4,7,8,2,6])
>>> np.argmax(a)
5                    ⇐ find the index for the largest/smallest element
>>> np.argmin(a)
2
>>> np.argwhere(a>3) ⇐ find the index for the non-zero elements
array([[3],
       [4],
       [5],
       [7]])
```

SORTING DATA

- In NumPy there are also several convenient functions for sorting the data in the array. For example:

```
>>> a = np.array([2,3,1,4])
>>> b = np.sort(a) ⇐ sorted data stored in b
>>> a,b
(array([2, 3, 1, 4]), array([1, 2, 3, 4]))
>>> a.sort() ⇐ sorted data stored in a (in place sorting)
>>> a
array([1, 2, 3, 4])
```

```
>>> a = np.array([2,3,1,4])                                   2, 3, 1, 4
>>> np.argsort(a) ⇐ index of the sorting results:
array([2, 0, 1, 3])                                           2, 0, 1, 3
```

SAVE/LOAD THE ARRAY

- Actually this is one of the very useful functions — it is very common to load your data from somewhere (your experiment results, exported from some other program, etc.) as a NumPy array for further processing.

Anything after # is treated as a comment ⇒

```
# year    records      time
 2011     3830000000   0.23
 2012     5510000000   0.28
 2013     6900000000   0.22
 2014     8970000000   0.26
```

data.txt

```
>>> a = np.loadtxt('data.txt')
>>> a
array([[  2.01100000e+03,   3.83000000e+09,   2.30000000e-01],
       [  2.01200000e+03,   5.51000000e+09,   2.80000000e-01],
       [  2.01300000e+03,   6.90000000e+09,   2.20000000e-01],
       [  2.01400000e+03,   8.97000000e+09,   2.60000000e-01]])
>>> np.savetxt('temp.txt',a)
>>>
```

⇐ This will save the array to another file.

SAVE/LOAD THE ARRAY (II)

- The **CSV** (comma separated value) format can be imported as well. Just need to add a **comma**!
- It is easy to read the data from a spreadsheet program.

	A	B	C	D	E	F
1	Plant	Week 1	Week 5	Week 10	Week 15	
2	1	2.50	5.70	8.40	10.40	
3	2	1.90	5.50	8.80	11.00	
4	3	2.30	6.20	9.00	11.80	
5	4	2.20	5.30	8.70	10.40	
6	5	1.20	4.80	6.70	8.00	
7	6	2.60	5.60	8.80	10.90	
8	7	2.20	5.70	8.90	10.80	
9	8	2.40	6.20	9.20	11.50	
10	9	1.70	5.80	8.60	10.40	
11	10	1.90	5.90	9.00	9.50	
12						

```
>>> np.loadtxt('data.csv',delimiter=',',skiprows=1)
array([[  1. ,   2.5,   5.7,   8.4,  10.4],
       [  2. ,   1.9,   5.5,   8.8,  11. ],
       [  3. ,   2.3,   6.2,   9. ,  11.8],
       [  4. ,   2.2,   5.3,   8.7,  10.4],
       [  5. ,   1.2,   4.8,   6.7,   8. ],
       [  6. ,   2.6,   5.6,   8.8,  10.9],
       [  7. ,   2.2,   5.7,   8.9,  10.8],
       [  8. ,   2.4,   6.2,   9.2,  11.5],
       [  9. ,   1.7,   5.8,   8.6,  10.4],
       [ 10. ,   1.9,   5.9,   9. ,   9.5]])
```

Skip the first row, which is the header.

SAVE/LOAD THE ARRAY (III)

- Or, you may want to use NumPy format directly, ie. those files with **.npy** and **.npz**:

```
>>> a = np.ones((2,2))
>>> a
array([[ 1.,  1.],
       [ 1.,  1.]])
>>> np.save('test.npy',a)
>>> os.system('ls -l test.npy')
-rw-r--r--  1 kfjack  staff  112 Mar 26 10:04
test.npy                    ⇐ a file named 'test.npy' being generated
>>> b = np.load('test.npy')  ⇐ read it back
>>> b
array([[ 1.,  1.],
       [ 1.,  1.]])
```

SAVE/LOAD THE ARRAY (IV)

- With **savez()**, one can store multiple arrays at once!

```
>>> a = np.ones((2,2))
>>> a
array([[ 1.,  1.],
       [ 1.,  1.]])
>>> b = np.arange(5)
>>> b
array([0, 1, 2, 3, 4])
>>> np.savez('test.npz', alice=a, bob=b)
>>> data = np.load('test.npz')
>>> data
<numpy.lib.npyio.NpzFile object at 0x105389fd0>
>>> data['bob']
array([0, 1, 2, 3, 4])
```

⇐ using the argument to keep the name of arrays

IMAGE MANIPULATION

- It is also very common to load the image as an array and perform some further processing.

```
type: <class 'numpy.ndarray'>
shape: (600, 800, 3)
datatype: uint8
```

```python
import matplotlib.pyplot as plt

img = plt.imread('testimage.jpg')

print('type:',type(img))
print('shape:',img.shape)
print('datatype:',img.dtype)

plt.imshow(img)
plt.show()
```

l204-example-08.py

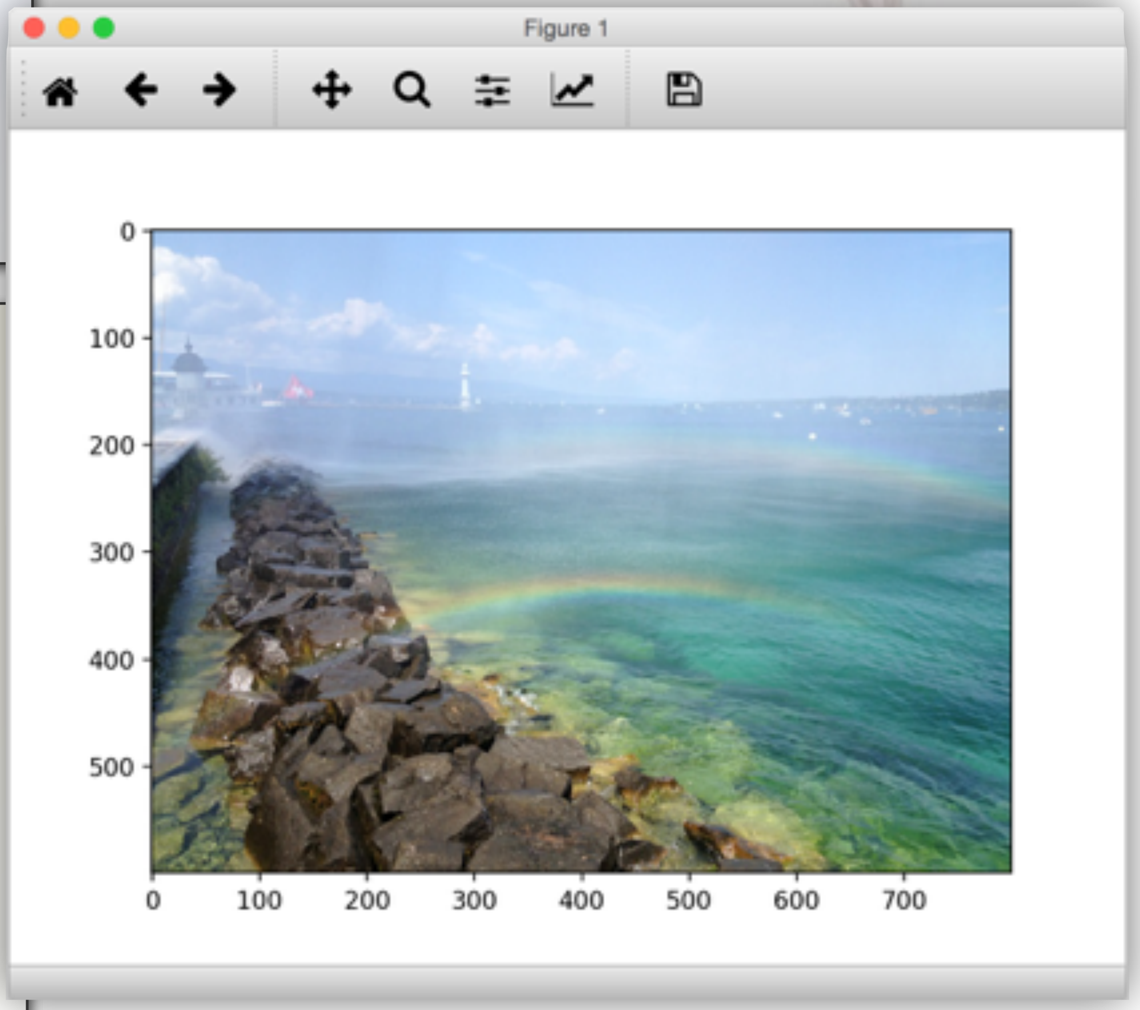


IMAGE MANIPULATION (II)

- The loaded image is stored in a NumPy array:

```
>>> img
array([[[161, 195, 230],
        [163, 197, 232],
        [164, 198, 233],
        ...,
        [162, 197, 237],
        [163, 198, 238],
        [165, 200, 240]],
       ...,
       [[ 86,  97,  80],
        [ 87,  98,  81],
        [ 91, 105,  82],
        ...,
        [ 68, 112,  59],
        [ 84, 129,  70],
        [ 99, 145,  81]]], dtype=uint8)
```

⇐ Every pixel has three integers **[R-G-B]**.

⇐ **8 bits** integer

With such an array, you can do:
1) image processing
2) extract information, etc.

IMAGE MANIPULATION (III)

```
plt.figure(figsize=(4, 9), dpi=80)

img_r = img.copy()
img_r[...,1] = 0
img_r[...,2] = 0

img_g = img.copy()
img_g[...,0] = 0
img_g[...,2] = 0

img_b = img.copy()
img_b[...,0] = 0
img_b[...,1] = 0

plt.subplot(3,1,1)
plt.imshow(img_r)

plt.subplot(3,1,2)
plt.imshow(img_g)

plt.subplot(3,1,3)
plt.imshow(img_b)

plt.show()
```

l204-example-09.py (partial)

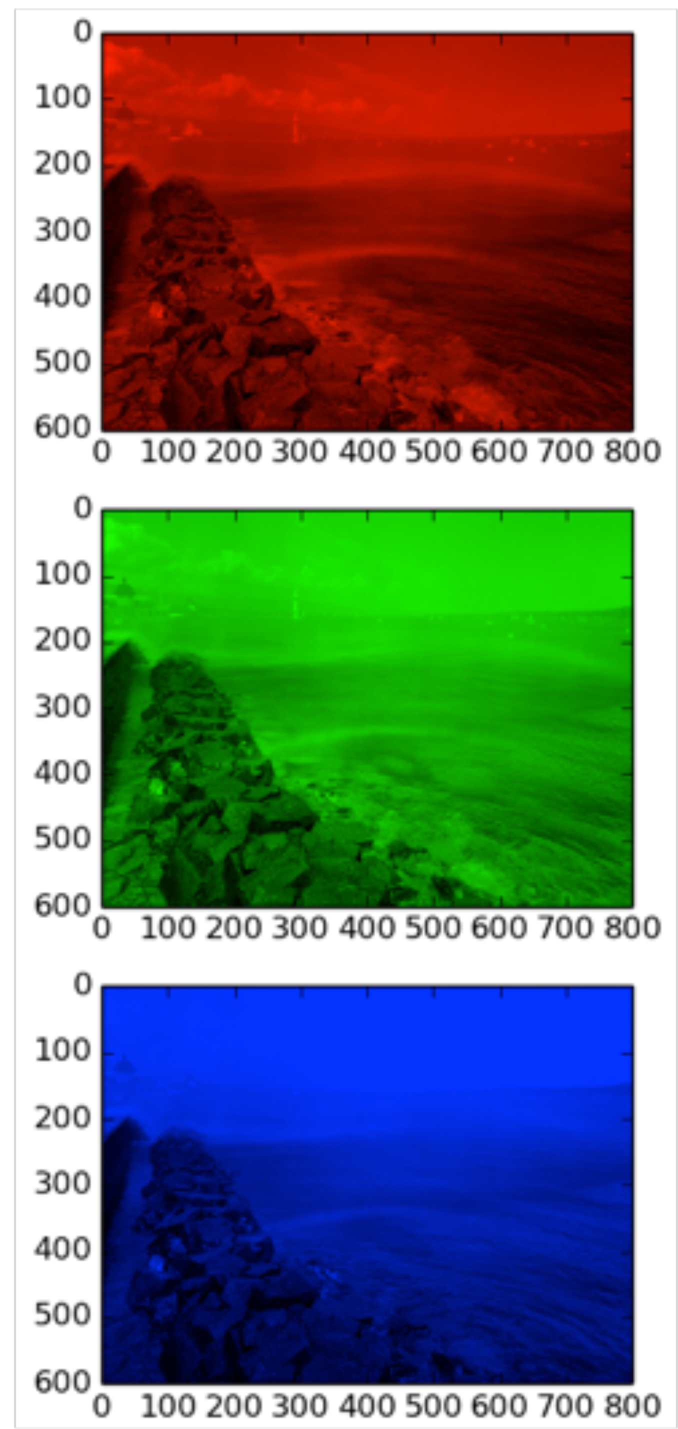
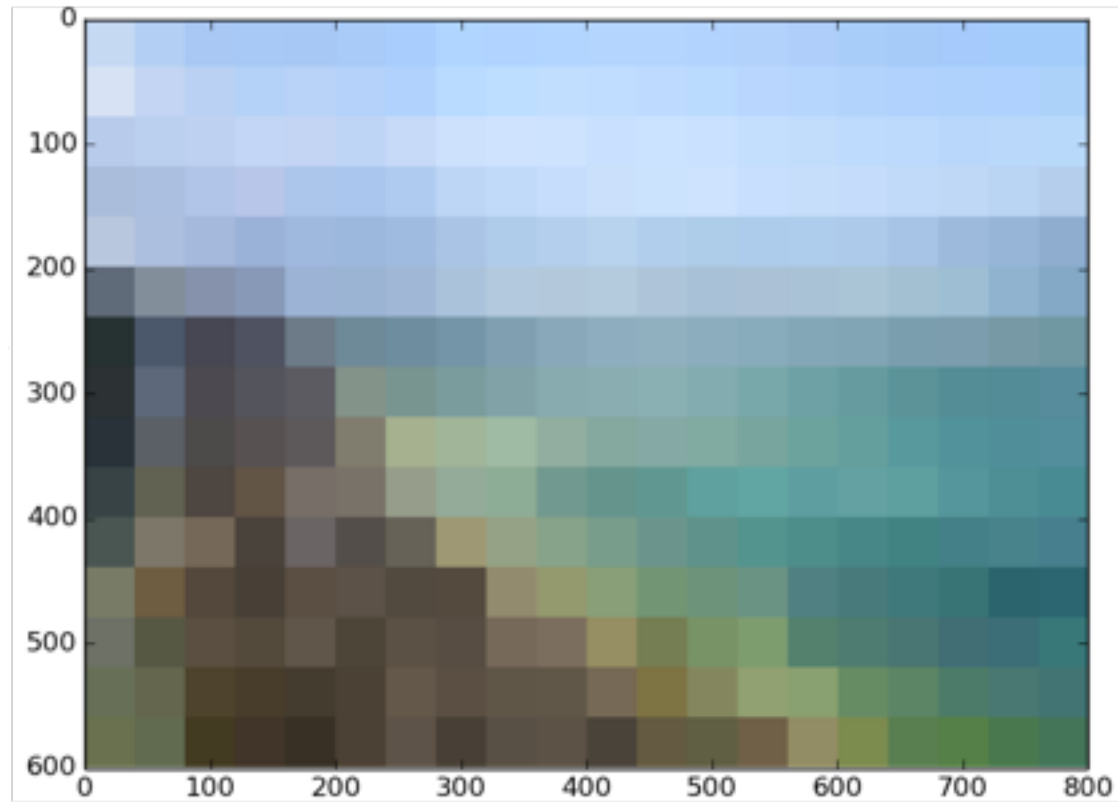


IMAGE MANIPULATION (IV)

- Making some mosaic is definitely very simple:



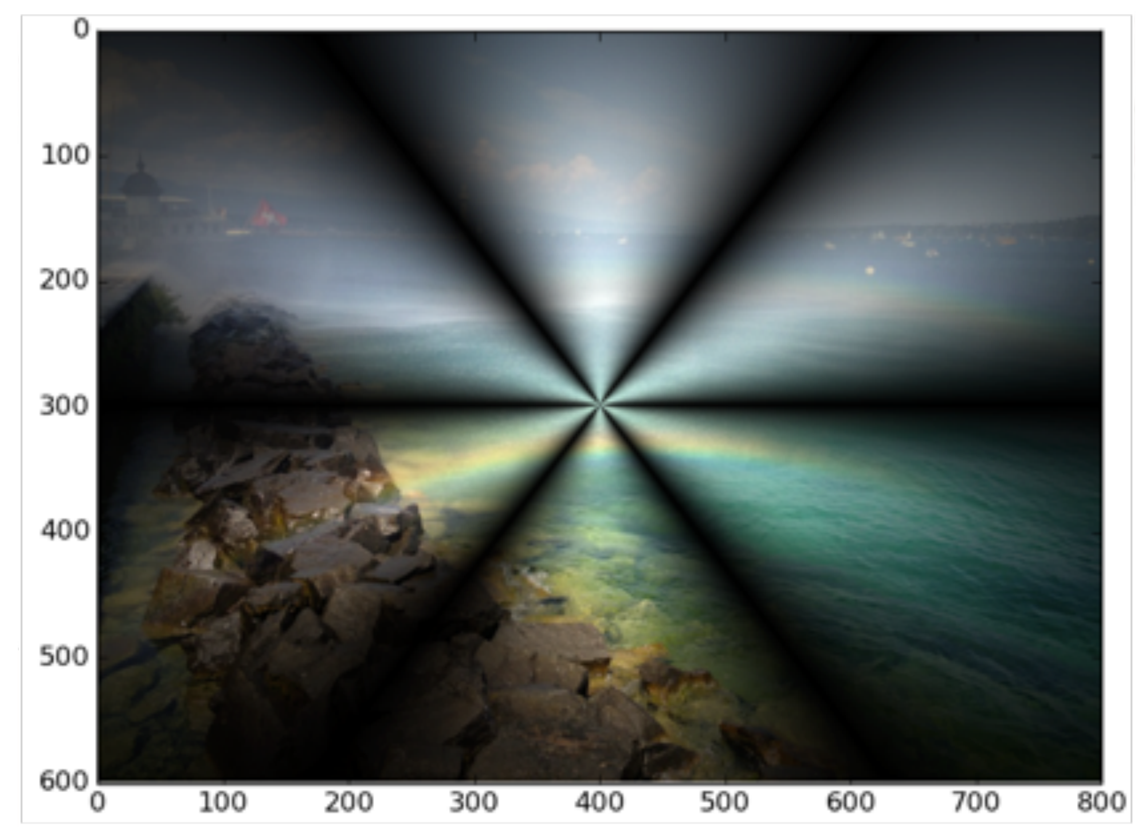

```
img = plt.imread('testimage.jpg')

# convert to [0-1] array
tmp = img.astype('float64')/255.

# make 40x40 mosaic
n = 40
for i in range(0,img.shape[0],n):
    for j in range(0,img.shape[1],n):
        tmp[i:i+n,j:j+n,:] = tmp[i:i+n,j:j+n,:].mean((0,1))

plt.imshow(tmp)
plt.show()
```

Average over first two dimensions (x-y)

l204-example-09a.py (partial)

IMAGE MANIPULATION (V)

- Some special(?) effects can be applied easily as well.




```
img = plt.imread('testimage.jpg')

tmp = img.astype('float64')/255.

x = np.linspace(-np.pi, np.pi, img.shape[1])
y = np.linspace(-np.pi, np.pi, img.shape[0])
xv, yv = np.meshgrid(x, y)
zv = abs(np.sin(np.arctan2(yv,xv)*3.))*np.exp(-0.1*(yv**2+xv**2))

for i in range(3):
    tmp[:,:,i] *= zv
    tmp[:,:,i] /= tmp[:,:,i].max()

plt.imshow(tmp)
plt.show()
```

l204-example-09b.py (partial)

COMMENTS

- Actually there are many online resources for image manipulation/processing with NumPy array.
- There is already a package named **ndimage** under SciPy, and there are many functions to do image processing:
  - http://docs.scipy.org/doc/scipy/reference/tutorial/ndimage.html
- **Scikit-image** is a more advanced package that can be used, see:
  - http://scipy-lectures.github.io/packages/scikit-image/
- Some more advanced tools can be found in **OpenCV**.
- If you are interested in this kind of software package, it is nice to go through some of these documents/tutorials!

HANDS-ON SESSION

- **Practice 01:**
  Given matrix **A** and **B** are 10x10 random matrices (generate it with `numpy.random.rand()`), test the following properties with `linalg.inv()` and `linalg.det()` functions:
  - $|\mathbf{A}^{-1}| = |\mathbf{A}|^{-1}$.
  - $(\mathbf{A}^{\mathrm{T}})^{-1} = (\mathbf{A}^{-1})^{\mathrm{T}}$.
  - $(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\,\mathbf{A}^{-1}$.

HANDS-ON SESSION

- **Practice 02:**
  Processing the test image with only 8 colors (pick up the closest one instead of the original color):
  (R,G,B) = (0,0,0),
  (0,0,255),
  (0,255,0),
  (255,0,0),
  (255,0,255),
  (255,255,0),
  (0,255,255),
  (255,255,255)