Futuristic Computing Platform for Science and Engineering: Python + NumPy/SciPy/Matplotlib

Yung-Yu Chen

yyc@solvcon.net

Python Hsinchu User Group

September 19, 2011

Python for Scientific and Engineering Computing

Python shows the traits of a futuristic computing platform for scientific and engineering applications:

- Productive.
  - Concise syntax for rapid development.
  - Versatile in built-in and 3rd-party libraries.
  - Short time to delivery.
- Robust and maintainable.
  - Deliver correct and easy-to-be-maintained code.
- Reasonably fast.
  - Avoid performance penalties.
  - I/O intensive: Pure Python is sufficient.
  - Computational intensive: Easy to be optimized by using C or Fortran.

Basic Python Scientific Tool Chain

- NumPy: `http://numpy.scipy.org/`
  - The core part is N-dimensional arrays, which are defined in `numpy.ndarray`.
  - The N-dimensional arrays serve as the foundation for all scientific computing tasks in Python.
- SciPy: `http://www.scipy.org/`
  - Provide calculation functionalities for scientific computing tasks, including: Linear algebra, equation solving, ODE integration, interpolation, special functions, Fourier transform, signal processing, etc.
- Matplotlib: `http://matplotlib.sourceforge.net/`
  - Two-dimensional plotting: Line, polar, bar, X-Y, pie, etc.

Python + NumPy + SciPy + MPL = Better MxxLAB

- The basic layer: (i) Array operation (NumPy), (ii) calculation tools (SciPy), and (iii) visualization/presentation (Matplotlib).
- Advanced tools:
  - Complementary functionalities to SciPy: SciKits (`http://scikits.appspot.com/`).
  - Symbolic mathematics: SymPy (`http://sympy.org/`).
  - Statistical computing: statlib (`http://code.google.com/p/python-statlib/`).
  - General-purpose GPU computing: PyCUDA and PyOpenCL.

Python Facilitates Software Engineering

- The language provides abundant armament for developing maintainable code.
  - Multi-paradigm: object-oriented, procedural, and functional.
  - Built-in containers: `dict`, `tuple`, `list`, `set`, etc.
  - Meta-classing.
- Production tools:
  - Documentation: docutils and sphinx.
  - Unit tests: built-in framework and `nosetests`.
  - Continuous integration: buildbot.

Optimization by Using C

- Use Python constructs.
  - The operations of the built-in containers are implemented in C.
  - `numpy.ndarray` supports many operations written in pre-compiled C code.
- Write C code yourself.
  - Develop a C extension.
  - Use Cython.
  - Write C and then load by using `ctypes`.
  - Write C++ and then interface by using swig or boost.python.

NumPy

- Installation.
  - Debian/Ubuntu: `apt-get install python-numpy`
  - Windows: Enthought Python Distribution (EPD).
- Every task starts with making a `ndarray` object.
  - `import numpy as np`
  - From existing data: `arr = np.array(list_or_tuple)`
  - Create uniform data: `arr = np.zeros((dim1, dim2, dim3))`
  - Allocate memory: `arr = np.empty((dim1, dim2), dtype='float64')`

Calculate Values of an Array

```
>>> import numpy as np
>>> arr = np.array([1, 2, 3], dtype='float64')
>>> brr = np.array([4, 5, 6], dtype='float64')
>>> crr = arr + brr # create a new object.
>>> print crr
[ 5.  7.  9.]
>>> print id(arr), id(brr), id(crr)
44843680 44845952 44855776
>>> arr *= 2  # modify the current object.
>>> print arr
[ 2.  4.  6.]
>>> print id(arr)
44843680
```

Array Manipulation

```
>>> mat = np.zeros((3,3), dtype='float64')
>>> mat[1,2] = 3
>>> print mat
[[ 0.  0.  0.]
 [ 0.  0.  3.]
 [ 0.  0.  0.]]
>>> mtt = mat.T # transpose.
>>> print mtt
[[ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  3.  0.]]
>>> mtt[0,1] = 5
>>> print mat
[[ 0.  0.  0.]
 [ 5.  0.  3.]
 [ 0.  0.  0.]]
```

SciPy

- Installation.
  - Debian/Ubuntu: `apt-get install python-scipy`
  - Windows: Enthought Python Distribution (EPD).
- SciPy is a collection of tools.
- Each sub-package should be imported individually.

Find Determinant

$$A = \begin{pmatrix} 1 & 1 & 4 \\ 1 & -1 & 3 \\ 2 & 0 & 1 \end{pmatrix} \quad \Rightarrow \quad \det(A) = 12$$

```
>>> import numpy as np
>>> from scipy import linalg
>>> A = np.array([[1, 1, 4], [1, -1, 3],
... [2, 0, 1]], dtype='float64')
>>> print linalg.det(A)
12.0
```

Solve Eigen Problem

$$A = \begin{pmatrix} 1 & 1 & 4 \\ 1 & -1 & 3 \\ 4 & 3 & 5 \end{pmatrix}$$

$$A\mathbf{v} = \lambda\mathbf{v} \Rightarrow A - \lambda I = 0$$

where λ is the eigenvalue and $\mathbf{v}$ is the eigenvector of the square matrix A.

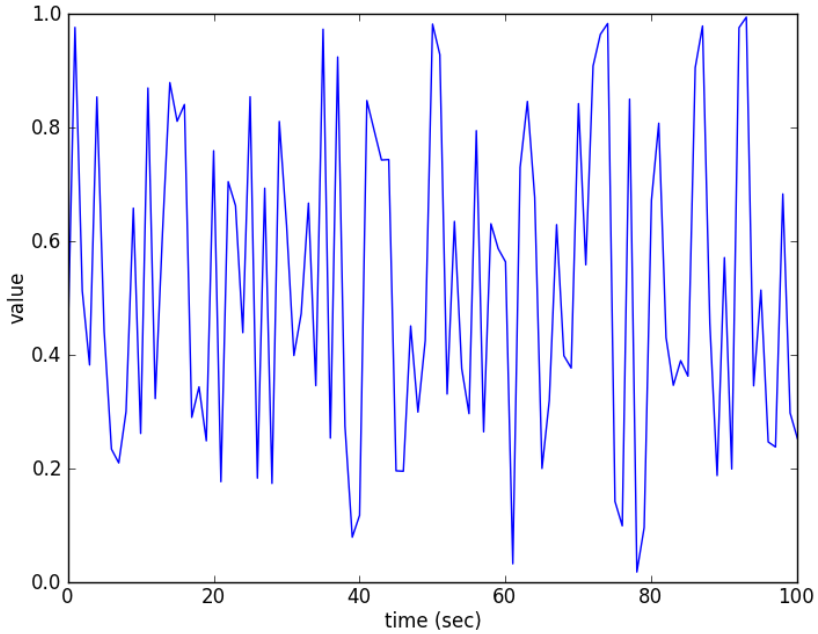
```
>>> import numpy as np
>>> from scipy import linalg
>>> A = np.array([[1, 1, 4], [1, -1, 3],
... [4, 3, 5]], dtype='float64')
>>> eval, evec = linalg.eig(A)
>>> print eval
[ 8.47722558+0.j -1.00000000+0.j -2.47722558+0.j]
```

Matplotlib

- Installation.
  - Debian/Ubuntu: `apt-get install python-scipy`
  - Windows: Enthought Python Distribution (EPD).
- SciPy is a collection of tools.
- Each sub-package should be imported individually.

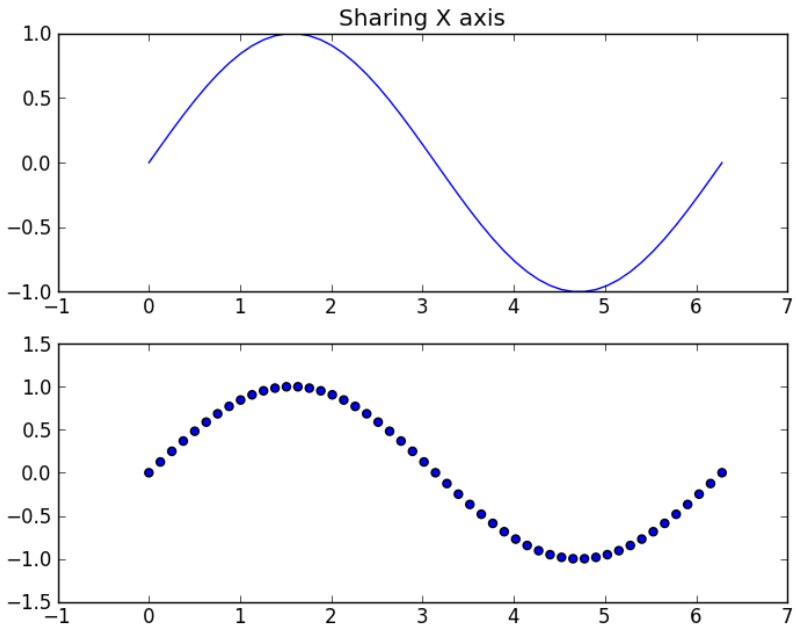
Line Plot

```
import numpy as np
import pylab
arr = np.arange(101,
    dtype='float64')
val = np.random_sample(
    arr.shape)
pylab.plot(arr, val)
pylab.xlabel('time (sec)')
pylab.ylabel('value')
pylab.show()
```



Sub-Plot

```
import numpy as np
import pylab
x = np.arange(51,
    dtype='float64'
    )/50*2*np.pi
y = np.sin(x)
f, ax = pylab.subplots(2,
    sharex=True)
ax[0].plot(x, y)
ax[0].set_title(
    'Sharing X axis')
ax[1].scatter(x, y)
pylab.show()
```



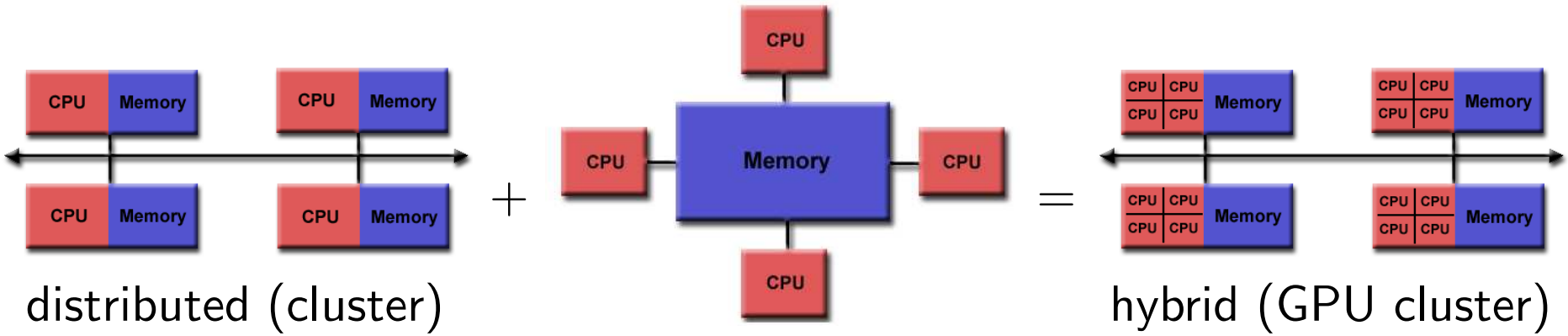
Documentation

- NumPy and SciPy: `http://docs.scipy.org/doc/`
  - NumPy book: `http://docs.scipy.org/doc/numpy/user/`
  - NumPy reference: `http://docs.scipy.org/doc/numpy/reference/`
  - SciPy reference: `http://docs.scipy.org/doc/scipy/reference/`
  - Cookbook: `http://www.scipy.org/Cookbook`
- Matplotlib: `http://matplotlib.sf.net/`
  - Examples: `http://matplotlib.sf.net/examples/index.html`

Python for High-Performance Computing

- New hardware boosts computing power.
  - 1970-1990: Vector machines (Cray computers).
  - 1990-200x: Clusters with MPI (10,000+ cores).
  - Now: Many-core technology, e.g., general-purpose GPU clusters.
- Quickly-evolving computer systems need agile programming for HPC software.

Hybrid Parallel Computing



distributed (cluster) shared (vector/GPU) hybrid (GPU cluster)

- Definition: Simultaneously use distributed-memory (e.g., MPI) and shared-memory (e.g., CUDA/OpenCL) parallel computing.
  - The difference in parallel computing models requires using disparate programming tools.
- A single language is not sufficient to model the hybrid system.
  - None of C, C++, and Fortran alone can be productive.
  - Software structure needs to be organized to accommodate the computing models.

Example: Solving Conservation Laws

- Three challenges for the next-generation solvers of conservation laws.
  - Challenge 1: HPC by routinely using 1,000+ cores and GPGPU computing.
  - Challenge 2: I/O and post-processing large data; the bottleneck is the laborious work flow for post-processing the results.
  - Challenge 3: Code reuse for long life cycle.
- Python can address the challenges.
  - NumPy is the foundation.
  - Replace the hotspots with C or CUDA code.

Conclusions

- De facto Python tool chain for scientific and engineering computing: NumPy + SciPy + Matplotlib.
  - NumPy: The core of array operations.
  - SciPy: Scientific calculations.
  - Matplotlib: Two-dimensional plotting.
- Python extends conventional scientific programming to use modern software-engineering practices.
- Python combined with C or CUDA: A productive approach to the best practice of HPC.
- Download the source files of the slides:

  `https://bitbucket.org/yungyuc/talk/src/tip/20110919_pysci/`