Scientific computing platforms at PGI / JCNS

Jan 21, 2016

PGI-1 / IAS-1  |  Scientific Visualization Workshop  |  Josef Heinen
Outline

✓ Introduction
✓ Python distributions
✓ The SciPy stack
✓ Julia as an alternative to Python
✓ Jupyter notebook
✓ Development environments
✓ Best practices for using Python and Julia
✓ An Overview of software developed by the “Scientific IT Systems” group
Introduction

Why should I use Python?

✓ general purpose programming language
✓ REPL (Read – Eval – Print – Loop)
✓ suited for interactive work or quick prototyping
✓ dynamically typed powerful data structures, e.g. dictionaries
✓ simple and elegant Object-Oriented Programming scheme
✓ huge standard library and thousands of add-on packages
✓ user community support
Python distributions

Which version should I use and install?

- UNIX and OS X systems include a system installation of Python (/usr)
- locally installed Python 2.7.11 (/usr/local)
  ➟ maintained by “Scientific IT–Systems” group for both Linux and OS X
- Anaconda 2.4.1 (Continuum Analytics) with Python 2.7.11 / 3.5.1 support (/usr/local/anaconda2)
  ➟ completely free Python distribution, installed on all Linux and OS X machines
  - create virtual environments (virtualenv, pyenv)
Python distributions

Please use the locally installed Python or Anaconda distributions:

```
% python-select
Please use one of the following environments:
    local       - locally installed Python 2.7
    anaconda\root - Anaconda Python 2.7
    py3k         - Anaconda Python 3.5
    pypy         - Python 2.7.10 interpreter and just-in-time compiler
```

```
% python-select local
Python 2.7.11
```

```
% python-select anaconda
Python 2.7.11 :: Anaconda 2.4.1 (x86_64)
```

```
% python-select py3k
Python 3.5.1 :: Anaconda 2.4.1 (x86_64)
```
The SciPy stack

Powerful modules for science, technology, engineering and mathematics (STEM)

NumPy
SciPy
SymPy
pandas

IPython
The SciPy stack

… more components:

Mayavi2

Anaconda

OpenGL

OpenCL

NVIDIA CUDA
Batteries included

✓ NumPy — package for numerical computation
✓ SciPy — collection of numerical algorithms and specific toolboxes
✓ Matplotlib — popular plotting package
✓ GR framework — combined 2D/3D visualization framework
✓ Pandas — provides high-performance, easy to use data structures
✓ SymPy — symbolic mathematics and computer algebra
✓ IPython — rich interactive interface (including IPython notebook)
✓ Mayavi2 — 3D visualization framework, based on VTK
✓ h5py, PyTables — managing hierarchical datasets (HDF5)
Julia as an alternative to Python

✓ very similar to Python – familiar syntax
✓ better performance, achieved using just-in-time (JIT) compilation
✓ multiple dispatch, based on one or more dynamic (run-time) types
✓ builtin parallel programming and distributed arrays
✓ IPython notebook support (IJulia)
✓ integrated package ecosystem
✓ Python language interop: PyCall.jl
Jupyter notebook

... is a web application for creating and sharing documents containing

✓ code, e.g. Python or Julia
✓ scientific equations
✓ visualizations, images or videos
✓ explanatory text (markdown syntax)

... can be used with local interpreter kernels or on remote notebook servers (github, nbviewer)
Jupyter notebook

Quick reference

```
jupyter notebook -generate-config
jupyter kernelspec list
```

→ Probably add kernel specifications (kernel.json):

```json
{
  "display_name": "Julia 0.4.3",
  "argv": [
    "/usr/local/Applications/Julia-0.4.3.app/Contents/Resources/julia/bin/julia",
    "-i",
    "-F",
    "/usr/local/lib/julia/v0.4/IIulia/src/kernel.jl",
    "{connection_file}"
  ],
  "language": "julia"
}
```
% jupyter notebook specgram.ipynb
# Jupyter notebook

<table>
<thead>
<tr>
<th>File</th>
<th>Edit</th>
<th>View</th>
<th>Insert</th>
<th>Cell</th>
<th>Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>New Notebook</td>
<td>Cut Cells</td>
<td>Toggle Header</td>
<td>Insert Cell Above</td>
<td>Run Cells</td>
<td>Interrupt</td>
</tr>
<tr>
<td>Open …</td>
<td>Copy Cells</td>
<td>Toggle Toolbar</td>
<td>Insert Cell Below</td>
<td>Run Cells and Select Below</td>
<td>Restart</td>
</tr>
<tr>
<td>Make a Copy …</td>
<td>Paste Cells Above</td>
<td>Cell Toolbar</td>
<td>Run Cells and Insert Below</td>
<td>Run All</td>
<td>Restart &amp; Clear Output</td>
</tr>
<tr>
<td>Rename …</td>
<td>Paste Cells Below</td>
<td></td>
<td>Run All Above</td>
<td>Run All Above</td>
<td>Restart &amp; Run All</td>
</tr>
<tr>
<td>Save and Checkpoint</td>
<td>Delete Cells</td>
<td></td>
<td>Run All Below</td>
<td>Run All Below</td>
<td>Reconnect</td>
</tr>
<tr>
<td>Revert to Checkpoint</td>
<td>Undo Delete Cells</td>
<td></td>
<td>Cell Type</td>
<td>All Output</td>
<td>Change kernel</td>
</tr>
<tr>
<td>Print Preview</td>
<td>Split Cell</td>
<td></td>
<td>Current Outputs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Download as</td>
<td>Merge Cell Above</td>
<td></td>
<td>All Output</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trusted Notebook</td>
<td>Merge Cell Below</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Close and Halt</td>
<td>Move Cell Up</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Move Cell Down</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Edit Notebook Metadata</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Find and Replace</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Download Formats**

- IPython Notebook (.ipynb)
- Python (.py)
- HTML (.html)
- Markdown (.md)
- reST (.rst)
- PDF via LaTeX (.pdf)
Simple spectral analysis

An illustration of the Discrete Fourier Transform using windowing, to reveal the frequency content of a sound signal.

\[ X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi}{N}kn} \quad k = 0, \ldots, N - 1 \]

We begin by loading a datafile using the wave package:

```
In [1]: import wave

wf = wave.open("test_mono.wav")
data = wf.readframes(800000)
samples = len(data)/2
wf.close()
```

```
In [2]: import numpy as np

FS = 44100.0      # Sampling frequency
t = np.linspace(0, samples/FS, samples)
amplitudes = np.fromstring(data, dtype=np.short)
```
And we can easily view its spectral structure using matplotlib's built-in `specgram` routine:

```python
In [3]:
    # matplotlib inline
    from matplotlib import pyplot as plt

    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
    ax1.plot(t, amplitudes); ax1.set_title('Raw audio signal')
    ax2.specgram(amplitudes); ax2.set_title('Spectrogram'); ax2.set_xticks(range(0, 25000, 5000))

Out[3]:
    [matplotlib.axis.XTick at 0x10912ba90>,
     <matplotlib.axis.XTick at 0x10912b950>,
     <matplotlib.axis.XTick at 0x109162b50>,
     <matplotlib.axis.XTick at 0x109149a10>,
     <matplotlib.axis.XTick at 0x109140ad0>]
```

![Raw audio signal](image)

![Spectrogram](image)
Markdown syntax

*italics* and _italics_

**bold** and __bold__

superscript\(^2^\)

~~strikethrough~~

inline equation: $A = \pi*r^{2}$

image: ![](path/to/image.png)

horizontal rule (or slide break): 

```markdown
> block quote
```

# Header 1

## Header 2

### Header 3

#### Header 4

##### Header 5

###### Header 6

* unordered list

  * item 2

  + sub-item 1

  + sub-item 2

1. ordered list

  1. item 2

  + sub-item 1

  + sub-item 2

endash: --

emdash: ---

ellipsis: ...

<table>
<thead>
<tr>
<th>Table Header</th>
<th>Second Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table Cell</td>
<td>Cell 2</td>
</tr>
<tr>
<td>Cell 3</td>
<td>Cell 4</td>
</tr>
</tbody>
</table>
Jupyter notebook

Demos
Development tools

You can use your favorite editor and start Python in a shell. But the impatient user should choose a more convenient environment:
% ipython

```python
In [1]: import scipy.linalg

In [2]: scipy.linalg.solve
scipy.linalg.solve
scipy.linalg.solve_banded
scipy.linalg.solve_circulant
scipy.linalg.solve_continuous_are
scipy.linalg.solve_discrete_are
scipy.linalg.solve_discrete_lyapunov
scipy.linalg.solve_lyapunov
scipy.linalg.solve_sylvester
scipy.linalg.solve_toeplitz
scipy.linalg.solve_triangular
scipy.linalg.solvev_banded
```
% jupyter qtconsole

Jupyter QtConsole 4.1.1
Python 2.7.11 (default, Dec 8 2015, 07:52:51)
Type "copyright", "credits" or "license" for more information.

IPython 4.0.2 -- An enhanced Interactive Python.
? -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help -> Python's own help system.
object? -> Details about 'object', use 'object??' for extra details.
%guiref -> A brief reference about the graphical user interface.

In [1]: import gr

In [2]: gr.polyline(x, y)

Signature: gr.polyline(x, y)
Docstring:
Draw a polyline using the current line attributes, starting from the
first data point and ending at the last data point.

**Parameters:**

`x`:
A list containing the X coordinates

`y`:
A list containing the Y coordinates

The values for `x` and `y` are in world coordinates. The attributes that
control the appearance of a polyline are linetype, linewidth and color
index.

File: /usr/local/gr/lib/python/gr/__init__.py
Type: function
% spyder

def create_fractal(min_x, max_x, min_y, max_y, image, iters):
    height = image.shape[0]
    width = image.shape[1]
    pixel_size_x = (max_x - min_x) / width
    pixel_size_y = (max_y - min_y) / height
    for x in range(width):
        real = min_x + x * pixel_size_x
        for y in range(height):
            imag = min_y + y * pixel_size_y
            color = mandel(real, imag, iters)
            image[y, x] = color

setviewport

setviewport(xmin, xmax, ymin, ymax):

setviewport defines a rectangular subspace of normalized device coordinates.

Parameters:

- xmin:
  The left horizontal coordinate of the viewport.
- xmax:
  The right horizontal coordinate of the viewport (0 <= xmin < xmax <= 1).
- ymin:
  The bottom vertical coordinate of the viewport.
- ymax:
  The top vertical coordinate of the viewport (0 <= ymin < ymax <= 1).

setviewport defines the rectangular portion of the Normalized Device Coordinate (NDC) space to be associated with the specified normalization transformation. The NDC viewport and World Coordinate (WC) window define the normalization transformation through which all output primitives pass. The WC window is mapped onto the rectangular NDC viewport which, in turn, mapped onto the display surface of the open and active workstation, in device coordinates.
PyCharm.app
Best practices for using Python and Julia

✓ small examples can be very helpful for beginners – and there are many

✓ let the system find your executable by looking through the PATH variable:

  #!/path/to/NAME → #!/usr/bin/env NAME

✓ only import methods you really need:

  from module import *

✓ don’t rewrite stuff in the standard library again

✓ use packages that support Python 3 and Julia 0.4+ respectively
An Overview of software developed by the “Scientific IT Systems” group

✓ **GR framework** – a universal framework for cross-platform visualization applications for creating publication quality 2D graphs and/or complex 3D scenes (GR3)

✓ **GR3 sample applications**

✓ **PyMolDyn** – a molecule viewer based on GR3 capable of computing molecular cavities
GR demos

\[
\zeta(s) := \sum_{n=1}^{\infty} \frac{1}{n^s}
\]

\[
\sigma = \Re(s) > 1
\]

\[
\zeta(s) := \frac{1}{\Gamma(s)} \int_0^\infty \frac{x^{s-1}}{e^x - 1} \, dx
\]
GR3 sample applications
PyMolDyn demo
Future workshop topics …

✓ GR framework tutorial

✓ introduction

✓ convenience MATLAB–like layers for Python (gr.pygr) and Julia (GR.jl, Plots.jl)

✓ using GR3 (Florian Rhiem)

✓ Julia tutorial: “Julia for Python programmers”
Questions