Python in Scientific Computing

An Introduction

Fernando Pérez

<Fernando.Perez@colorado.edu>

Applied Mathematics, U. of Colorado at Boulder

NCAR

August 22, 2007

Outline

Outline

Outline

Outline

FORTRAN, C and C++

Caveat: C++ suffers less from some of these problems, but it has other issues.

- Tools from a time when CPU time was more expensive than human time.
- Low-level:
  - Primitive data types (no good strings, sets, hash tables, ...).
  - Manual memory management: bugs, bugs, bugs. Hard ones.
  - Slow edit/compile/test cycle.
- Clumsy access to visualization, quick profiling, text processing, ...
- No interactive facilities – scientific work is inherently exploratory.
- Object Orientation?
  - Not a silver bullet, but a very good model for many scientific tasks.
  - Non-existent in (old) FORTRAN 2, difficult and alien in C++.

However!

- They deliver excellent performance.
- Millions of LOC in existing scientific libraries (LAPACK, BLAS, ...)

FORTRAN, C and C++

Caveat: C++ suffers less from some of these problems, but it has other issues.

- Tools from a time when CPU time was more expensive than human time.
- Low-level:
  - Primitive data types (no good strings, sets, hash tables, . . .).
  - Manual memory management: bugs, bugs, bugs. Hard ones.
  - Slow edit/compile/test cycle.
- Clumsy access to visualization, quick profiling, text processing, . . .
- No interactive facilities - scientific work is inherently exploratory.
- Object Orientation?
  - Not a silver bullet, but a very good model for many scientific codes.
  - Non-existent in (old) FORTRAN & C, difficult and subtle in C++.

However!

- They deliver excellent performance.
- Millions of LOC in existing scientific libraries (LAPACK, BLAS, . . .)

FORTRAN, C and C++

Caveat: C++ suffers less from some of these problems, but it has other issues.

- Tools from a time when CPU time was more expensive than human time.
- Low-level:
  - Primitive data types (no good strings, sets, hash tables, . . .).
  - Manual memory management: bugs, bugs, bugs. Hard ones.
  - Slow edit/compile/test cycle.
- Clumsy access to visualization, quick profiling, text processing, . . .
- No interactive facilities - scientific work is inherently exploratory.
- Object Orientation?
  - Not a silver bullet, but a very good model for many scientific codes.
  - Non-existent in (old) FORTRAN & C, difficult and subtle in C++.

However!

- They deliver excellent performance.
- Millions of LOC in existing scientific libraries (LAPACK, BLAS, . . .)

FORTRAN, C and C++

Caveat: C++ suffers less from some of these problems, but it has other issues.

- Tools from a time when CPU time was more expensive than human time.
- Low-level:
  - Primitive data types (no good strings, sets, hash tables, . . .).
  - Manual memory management: bugs, bugs, bugs. Hard ones.
  - Slow edit/compile/test cycle.
- Clumsy access to visualization, quick profiling, text processing, . . .
- No interactive facilities - scientific work is inherently exploratory.
- Object Orientation?
  - Not a silver bullet, but a very good model for many scientific codes.
  - Non-existent in (old) FORTRAN & C, difficult and subtle in C++.

However!

- They deliver excellent performance.
- Millions of LOC in existing scientific libraries (LAPACK, BLAS, . . .)

FORTRAN, C and C++

Caveat: C++ suffers less from some of these problems, but it has other issues.

- Tools from a time when CPU time was more expensive than human time.
- Low-level:
  - Primitive data types (no good strings, sets, hash tables, . . .).
  - Manual memory management: bugs, bugs, bugs. Hard ones.
  - Slow edit/compile/test cycle.
- Clumsy access to visualization, quick profiling, text processing, . . .
- No interactive facilities - scientific work is inherently exploratory.
- Object Orientation?
  - Not a silver bullet, but a very good model for many scientific codes.
  - Non-existent in (old) FORTRAN & C, difficult and subtle in C++.

However!

- They deliver excellent performance.
- Millions of LOC in existing scientific libraries (LAPACK, BLAS, . . .)

FORTRAN, C and C++

Caveat: C++ suffers less from some of these problems, but it has other issues.

- Tools from a time when CPU time was more expensive than human time.
- Low-level:
  - Primitive data types (no good strings, sets, hash tables, . . .).
  - Manual memory management: bugs, bugs, bugs. Hard ones.
  - Slow edit/compile/test cycle.
- Clumsy access to visualization, quick profiling, text processing, . . .
- No interactive facilities - scientific work is inherently exploratory.
- Object Orientation?
  - Not a silver bullet, but a very good model for many scientific codes.
  - Non-existent in (old) FORTRAN & C, difficult and subtle in C++.

However!

- They deliver excellent performance.
- Millions of LOC in existing scientific libraries (LAPACK, BLAS, . . .)

FORTRAN, C and C++

Caveat: C++ suffers less from some of these problems, but it has other issues.

- Tools from a time when CPU time was more expensive than human time.
- Low-level:
  - Primitive data types (no good strings, sets, hash tables, . . .).
  - Manual memory management: bugs, bugs, bugs. Hard ones.
  - Slow edit/compile/test cycle.
- Clumsy access to visualization, quick profiling, text processing, . . .
- No interactive facilities - scientific work is inherently exploratory.
- Object Orientation?
  - Not a silver bullet, but a very good model for many scientific codes.
  - Non-existent in (old) FORTRAN & C, difficult and subtle in C++.

However!

- They deliver excellent performance.
- Millions of LOC in existing scientific libraries (LAPACK, BLAS, . . .)

FORTRAN, C and C++

Caveat: C++ suffers less from some of these problems, but it has other issues.

- Tools from a time when CPU time was more expensive than human time.
- Low-level:
  - Primitive data types (no good strings, sets, hash tables, . . .).
  - Manual memory management: bugs, bugs, bugs. Hard ones.
  - Slow edit/compile/test cycle.
- Clumsy access to visualization, quick profiling, text processing, . . .
- No interactive facilities - scientific work is inherently exploratory.
- Object Orientation?
  - Not a silver bullet, but a very good model for many scientific codes.
  - Non-existent in (old) FORTRAN & C, difficult and subtle in C++.

However!

- They deliver excellent performance.
- Millions of LOC in existing scientific libraries (LAPACK, BLAS, . . .)

Higher level tools in the last decade

- Mathematica and Maple: I won't address symbolic computing in detail.
- IDL and Matlab: extremely popular tools in science and engineering.
  - Great interactivity, visualization, and extensive libraries.
  - Unpleasant languages for large-scale programs and non-numerical tasks.
  - Expensive/proprietary: lock-in.
  - Often considered 'prototyping' tools: this leads to a lot of code rewriting.
- A common approach (I've been there): mix and match multiple tools:
  - FORTRAN, C, C++ programs...
  - driven by Bash/sh/ksh/csh/Perl scripts...
  - which feed them input and take their output...
  - and pass them to Gnuplot, Grace, OpenDX, etc.
- Many different syntaxes: huge context switching overhead!

Higher level tools in the last decade

- Mathematica and Maple: I won't address symbolic computing in detail.
- IDL and Matlab: extremely popular tools in science and engineering.
  - Great interactivity, visualization, and extensive libraries.
  - Unpleasant languages for large-scale programs and non-numerical tasks.
  - Expensive/proprietary: lock-in.
  - Often considered 'prototyping' tools: this leads to a lot of code rewriting.
- A common approach (I've been there): mix and match multiple tools:
  - FORTRAN, C, C++ programs ...
  - driven by Bash/awk/sed/Perl scripts ...
  - which feed them input and talk to their outputs ...
  - and pass them to Gnuplot, Grace, OpenDX, etc.
- Many different syntaxes: huge context switching overhead!

Higher level tools in the last decade

- Mathematica and Maple: I won't address symbolic computing in detail.
- IDL and Matlab: extremely popular tools in science and engineering.
  - Great interactivity, visualization, and extensive libraries.
  - Unpleasant languages for large-scale programs and non-numerical tasks.
  - Expensive/proprietary: lock-in.
  - Often considered 'prototyping' tools: this leads to a lot of code rewriting.
- A common approach (I've been there): mix and match multiple tools:
  - FORTRAN, C, C++ programs ...
  - driven by Bash/awk/sed/Perl scripts ...
  - which feed them input and take their outputs ...
  - and pass them to Gnuplot, Grace, OpenDX, etc.
- Many different syntaxes: huge context switching overhead!

Higher level tools in the last decade

- Mathematica and Maple: I won't address symbolic computing in detail.
- IDL and Matlab: extremely popular tools in science and engineering.
  - Great interactivity, visualization, and extensive libraries.
  - Unpleasant languages for large-scale programs and non-numerical tasks.
  - Expensive/proprietary: lock-in.
  - Often considered 'prototyping' tools: this leads to a lot of code rewriting.
- A common approach (I've been there): mix and match multiple tools:
  - FORTRAN, C, C++ programs ...
  - driven by Bash/awk/sed/Perl scripts ...
  - which feed them input and take their outputs ...
  - and pass them to Gnuplot, Grace, OpenDX, etc.
- Many different syntaxes: huge context switching overhead!

Higher level tools in the last decade

- Mathematica and Maple: I won't address symbolic computing in detail.
- IDL and Matlab: extremely popular tools in science and engineering.
  - Great interactivity, visualization, and extensive libraries.
  - Unpleasant languages for large-scale programs and non-numerical tasks.
  - Expensive/proprietary: lock-in.
  - Often considered 'prototyping' tools: this leads to a lot of code rewriting.
- A common approach (I've been there): mix and match multiple tools:
  - FORTRAN, C, C++ programs ...
  - driven by Bash/awk/sed/Perl scripts ...
  - which feed them input and take their outputs ...
  - and pass them to Gnuplot, Grace, OpenDX, etc.
- Many different syntaxes: huge context switching overhead!

Higher level tools in the last decade

- Mathematica and Maple: I won't address symbolic computing in detail.
- IDL and Matlab: extremely popular tools in science and engineering.
  - Great interactivity, visualization, and extensive libraries.
  - Unpleasant languages for large-scale programs and non-numerical tasks.
  - Expensive/proprietary: lock-in.
  - Often considered 'prototyping' tools: this leads to a lot of code rewriting.
- A common approach (I've been there): mix and match multiple tools:
  - FORTRAN, C, C++ programs ...
  - driven by Bash/awk/sed/Perl scripts ...
  - which feed them input and take their outputs ...
  - and pass them to Gnuplot, Grace, OpenDX, etc.
- Many different syntaxes: huge context switching overhead!

Higher level tools in the last decade

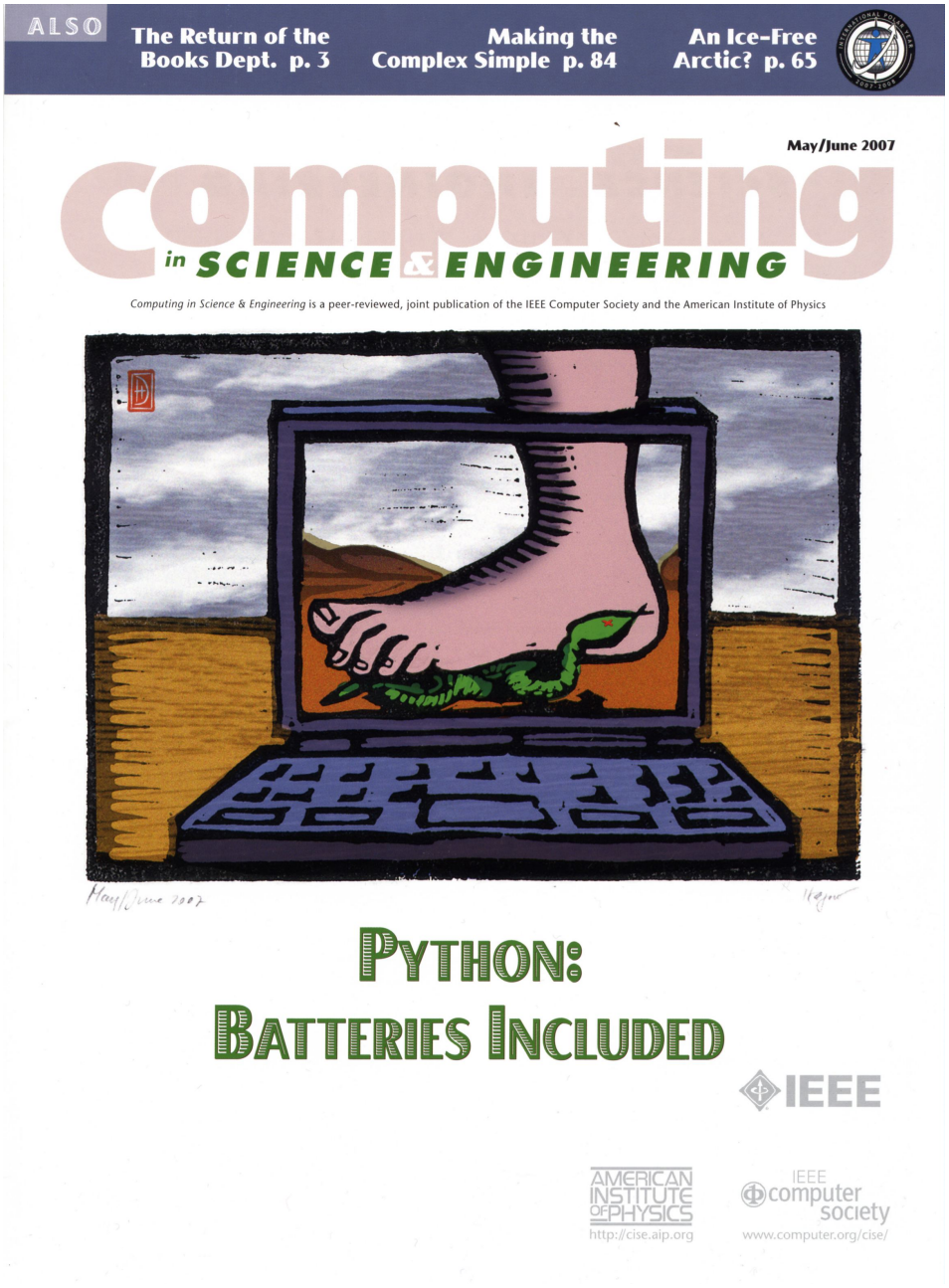
- Mathematica and Maple: I won't address symbolic computing in detail.
- IDL and Matlab: extremely popular tools in science and engineering.
  - Great interactivity, visualization, and extensive libraries.
  - Unpleasant languages for large-scale programs and non-numerical tasks.
  - Expensive/proprietary: lock-in.
  - Often considered 'prototyping' tools: this leads to a lot of code rewriting.
- A common approach (I've been there): mix and match multiple tools:
  - FORTRAN, C, C++ programs ...
  - driven by Bash/awk/sed/Perl scripts ...
  - which feed them input and take their outputs ...
  - and pass them to Gnuplot, Grace, OpenDX, etc.
- Many different syntaxes: huge context switching overhead!

Higher level tools in the last decade

- Mathematica and Maple: I won't address symbolic computing in detail.
- IDL and Matlab: extremely popular tools in science and engineering.
  - Great interactivity, visualization, and extensive libraries.
  - Unpleasant languages for large-scale programs and non-numerical tasks.
  - Expensive/proprietary: lock-in.
  - Often considered 'prototyping' tools: this leads to a lot of code rewriting.
- A common approach (I've been there): mix and match multiple tools:
  - FORTRAN, C, C++ programs ...
  - driven by Bash/awk/sed/Perl scripts ...
  - which feed them input and take their outputs ...
  - and pass them to Gnuplot, Grace, OpenDX, etc.
- Many different syntaxes: huge context switching overhead!

Python is growing in scientific computing

Entire May/June 2007 issue of CiSE devoted to Python



Python in this context

- A bytecode-interpreted language (VMs in C, Java and .NET).
- Interactive interpreter provided.
- Free (BSD license), highly portable (Linux, OSX, Solaris, Windows, ...).
- Extremely readable syntax ("executable pseudo-code").
- Simple: non-professional programmers can become (and remain) proficient with a very small effort (c.f. C++).
- Clean object oriented model, but not mandatory.
- Rich built-in types: lists, sets, dictionaries (hash tables), strings, ...
- Very comprehensive standard library (batteries included):
  - Text processing, networking protocols, threading, GUIs, ...
- Standard libraries for IDL/Matlab-like arrays (Numpy).
- Easy to wrap existing C, C++ and FORTRAN codes:
  - Python bindings for a vast amount of third-party libraries (scientific or not).
  - This addresses the speed issues (native python is slow).

Python in this context

- A bytecode-interpreted language (VMs in C, Java and .NET).
- Interactive interpreter provided.
- Free (BSD license), highly portable (Linux, OSX, Solaris, Windows, ...).
- Extremely readable syntax ("executable pseudo-code").
- Simple: non-professional programmers can become (and remain) proficient with a very small effort (c.f. C++).
- Clean object oriented model, but not mandatory.
- Rich built-in types: lists, sets, dictionaries (hash tables), strings, ...
- Very comprehensive standard library (batteries included):
  - Text processing, networking protocols, threading, GUIs, ...
- Standard libraries for IDL/Matlab-like arrays (Numpy).
- Easy to wrap existing C, C++ and FORTRAN codes:
  - Python bindings for a vast amount of third-party libraries (scientific or not).
  - This addresses the speed issues (native python is slow).

Python in this context

- A bytecode-interpreted language (VMs in C, Java and .NET).
- Interactive interpreter provided.
- Free (BSD license), highly portable (Linux, OSX, Solaris, Windows, ...).
- Extremely readable syntax ("executable pseudo-code").
- Simple: non-professional programmers can become (and remain) proficient with a very small effort (c.f. C++).
- Clean object oriented model, but not mandatory.
- Rich built-in types: lists, sets, dictionaries (hash tables), strings, ...
- Very comprehensive standard library (batteries included):
  - Text processing, networking protocols, threading, GUIs, ...
- Standard libraries for IDL/Matlab-like arrays (Numpy).
- Easy to wrap existing C, C++ and FORTRAN codes:
  - Python bindings for a vast amount of third-party libraries (scientific or not).
  - This addresses the speed issues (native python is slow).

Python in this context

- A bytecode-interpreted language (VMs in C, Java and .NET).
- Interactive interpreter provided.
- Free (BSD license), highly portable (Linux, OSX, Solaris, Windows, ...).
- Extremely readable syntax ("executable pseudo-code").
- Simple: non-professional programmers can become (and remain) proficient with a very small effort (c.f. C++).
- Clean object oriented model, but not mandatory.
- Rich built-in types: lists, sets, dictionaries (hash tables), strings, ...
- Very comprehensive standard library (batteries included):
  - Text processing, networking protocols, threading, GUIs, ...
- Standard libraries for IDL/Matlab-like arrays (Numpy).
- Easy to wrap existing C, C++ and FORTRAN codes:
  - Python bindings for a vast amount of third-party libraries (scientific or not).
  - This addresses the speed issues (native python is slow).

Python in this context

- A bytecode-interpreted language (VMs in C, Java and .NET).
- Interactive interpreter provided.
- Free (BSD license), highly portable (Linux, OSX, Solaris, Windows, ...).
- Extremely readable syntax ("executable pseudo-code").
- Simple: non-professional programmers can become (and remain) proficient with a very small effort (c.f. C++).
- Clean object oriented model, but not mandatory.
- Rich built-in types: lists, sets, dictionaries (hash tables), strings, ...
- Very comprehensive standard library (batteries included):
  - Text processing, networking protocols, threading, GUIs, ...
- Standard libraries for IDL/Matlab-like arrays (Numpy).
- Easy to wrap existing C, C++ and FORTRAN codes:
  - Python bindings for a vast amount of third-party libraries (scientific or not).
  - This addresses the speed issues (native python is slow).

Python in this context

- A bytecode-interpreted language (VMs in C, Java and .NET).
- Interactive interpreter provided.
- Free (BSD license), highly portable (Linux, OSX, Solaris, Windows, ...).
- Extremely readable syntax ("executable pseudo-code").
- Simple: non-professional programmers can become (and remain) proficient with a very small effort (c.f. C++).
- Clean object oriented model, but not mandatory.
- Rich built-in types: lists, sets, dictionaries (hash tables), strings, ...
- Very comprehensive standard library (batteries included):
  - Text processing, networking protocols, threading, GUIs, ...
- Standard libraries for IDL/Matlab-like arrays (Numpy).
- Easy to wrap existing C, C++ and FORTRAN codes:
  - Python bindings for a vast amount of third-party libraries (scientific or not).
  - This addresses the speed issues (native python is slow).

Python in this context

- A bytecode-interpreted language (VMs in C, Java and .NET).
- Interactive interpreter provided.
- Free (BSD license), highly portable (Linux, OSX, Solaris, Windows, ...).
- Extremely readable syntax ("executable pseudo-code").
- Simple: non-professional programmers can become (and remain) proficient with a very small effort (c.f. C++).
- Clean object oriented model, but not mandatory.
- Rich built-in types: lists, sets, dictionaries (hash tables), strings, ...
- Very comprehensive standard library (batteries included):
  - Text processing, networking protocols, threading, GUIs, ...
- Standard libraries for IDL/Matlab-like arrays (Numpy).
- Easy to wrap existing C, C++ and FORTRAN codes:
  - Python bindings for a vast amount of third-party libraries (scientific or not).
  - This addresses the speed issues (native python is slow).

Python in this context

- A bytecode-interpreted language (VMs in C, Java and .NET).
- Interactive interpreter provided.
- Free (BSD license), highly portable (Linux, OSX, Solaris, Windows, ...).
- Extremely readable syntax ("executable pseudo-code").
- Simple: non-professional programmers can become (and remain) proficient with a very small effort (c.f. C++).
- Clean object oriented model, but not mandatory.
- Rich built-in types: lists, sets, dictionaries (hash tables), strings, ...
- Very comprehensive standard library (batteries included):
  - Text processing, networking protocols, threading, GUIs, ...
- Standard libraries for IDL/Matlab-like arrays (Numpy).
- Easy to wrap existing C, C++ and FORTRAN codes:
  - Python bindings for a vast amount of third-party libraries (scientific or not).
  - This addresses the speed issues (native python is slow).

Python in this context

- A bytecode-interpreted language (VMs in C, Java and .NET).
- Interactive interpreter provided.
- Free (BSD license), highly portable (Linux, OSX, Solaris, Windows, ...).
- Extremely readable syntax ("executable pseudo-code").
- Simple: non-professional programmers can become (and remain) proficient with a very small effort (c.f. C++).
- Clean object oriented model, but not mandatory.
- Rich built-in types: lists, sets, dictionaries (hash tables), strings, ...
- Very comprehensive standard library (batteries included):
  - Text processing, networking protocols, threading, GUIs, ...
- Standard libraries for IDL/Matlab-like arrays (Numpy).
- Easy to wrap existing C, C++ and FORTRAN codes:
  - Python bindings for a vast amount of third-party libraries (scientific or not).
  - This addresses the speed issues (native python is slow).

Python in this context

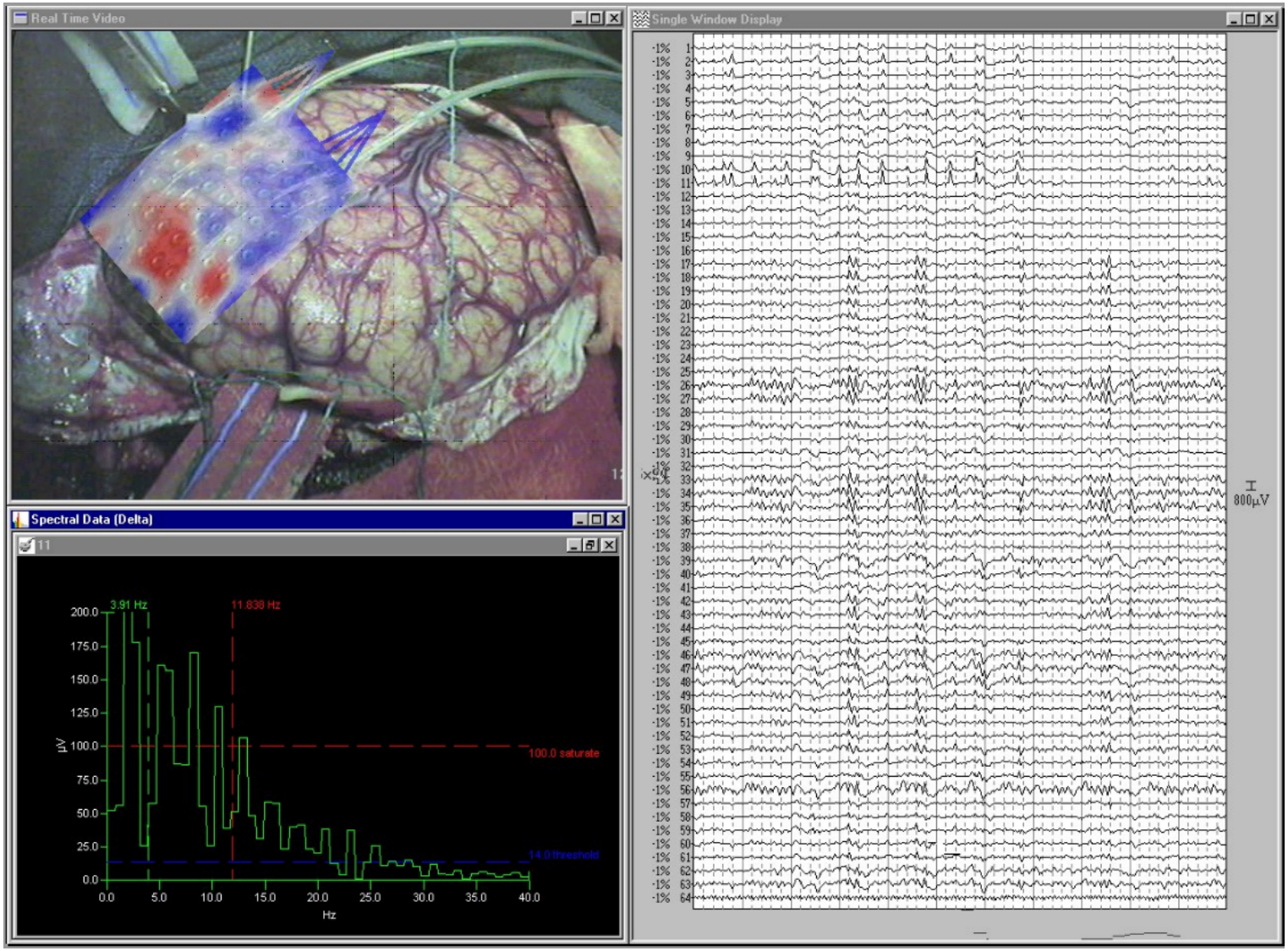
- A bytecode-interpreted language (VMs in C, Java and .NET).
- Interactive interpreter provided.
- Free (BSD license), highly portable (Linux, OSX, Solaris, Windows, ...).
- Extremely readable syntax ("executable pseudo-code").
- Simple: non-professional programmers can become (and remain) proficient with a very small effort (c.f. C++).
- Clean object oriented model, but not mandatory.
- Rich built-in types: lists, sets, dictionaries (hash tables), strings, ...
- Very comprehensive standard library (batteries included):
  - Text processing, networking protocols, threading, GUIs, ...
- Standard libraries for IDL/Matlab-like arrays (Numpy).
- Easy to wrap existing C, C++ and FORTRAN codes:
  - Python bindings for a vast amount of third-party libraries (scientific or not).
  - This addresses the speed issues (native python is slow).

Outline

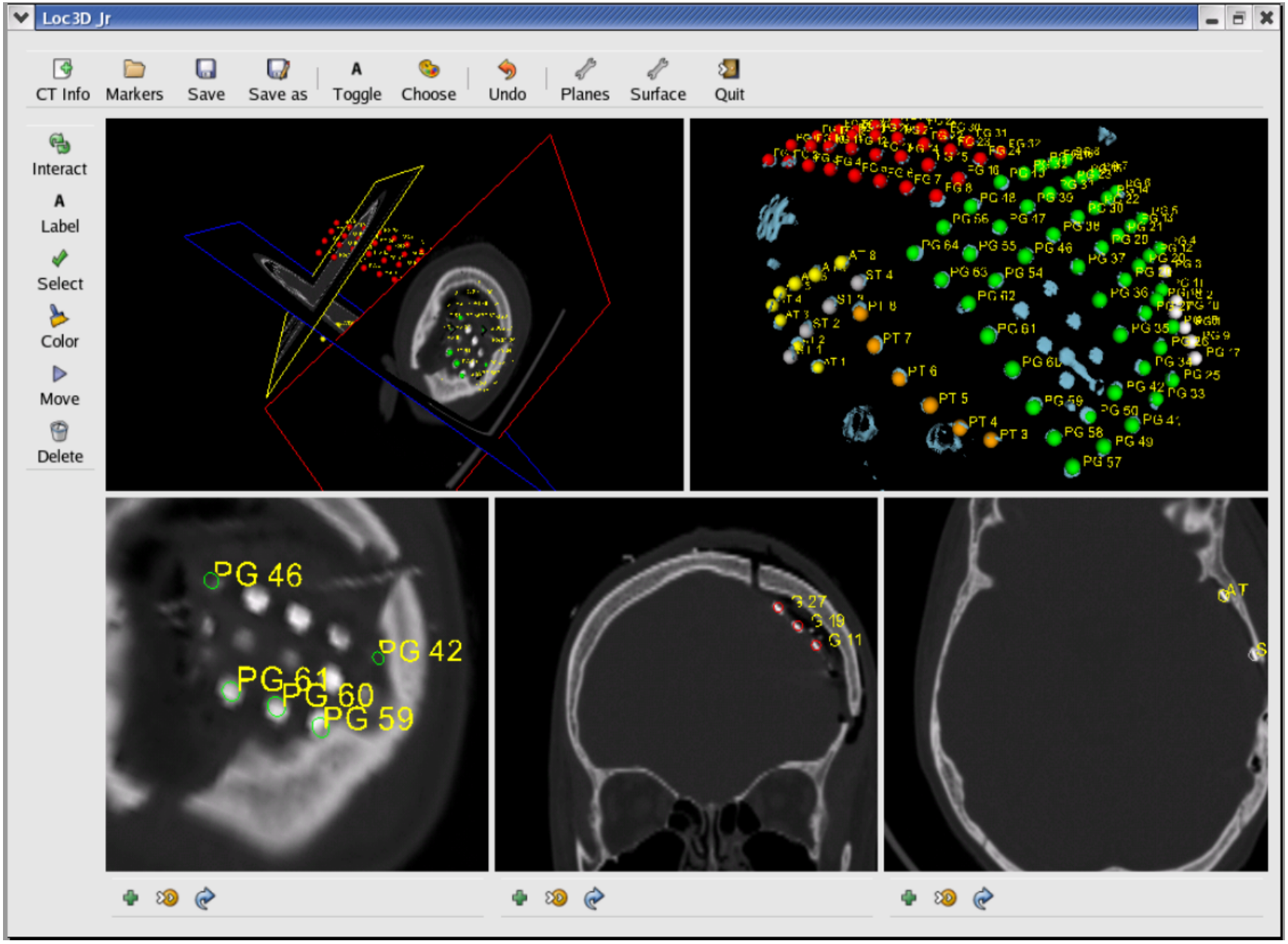
Data analysis for epilepsy surgery

Isolating the origin of drug-resistant epileptic seizures which require surgery.

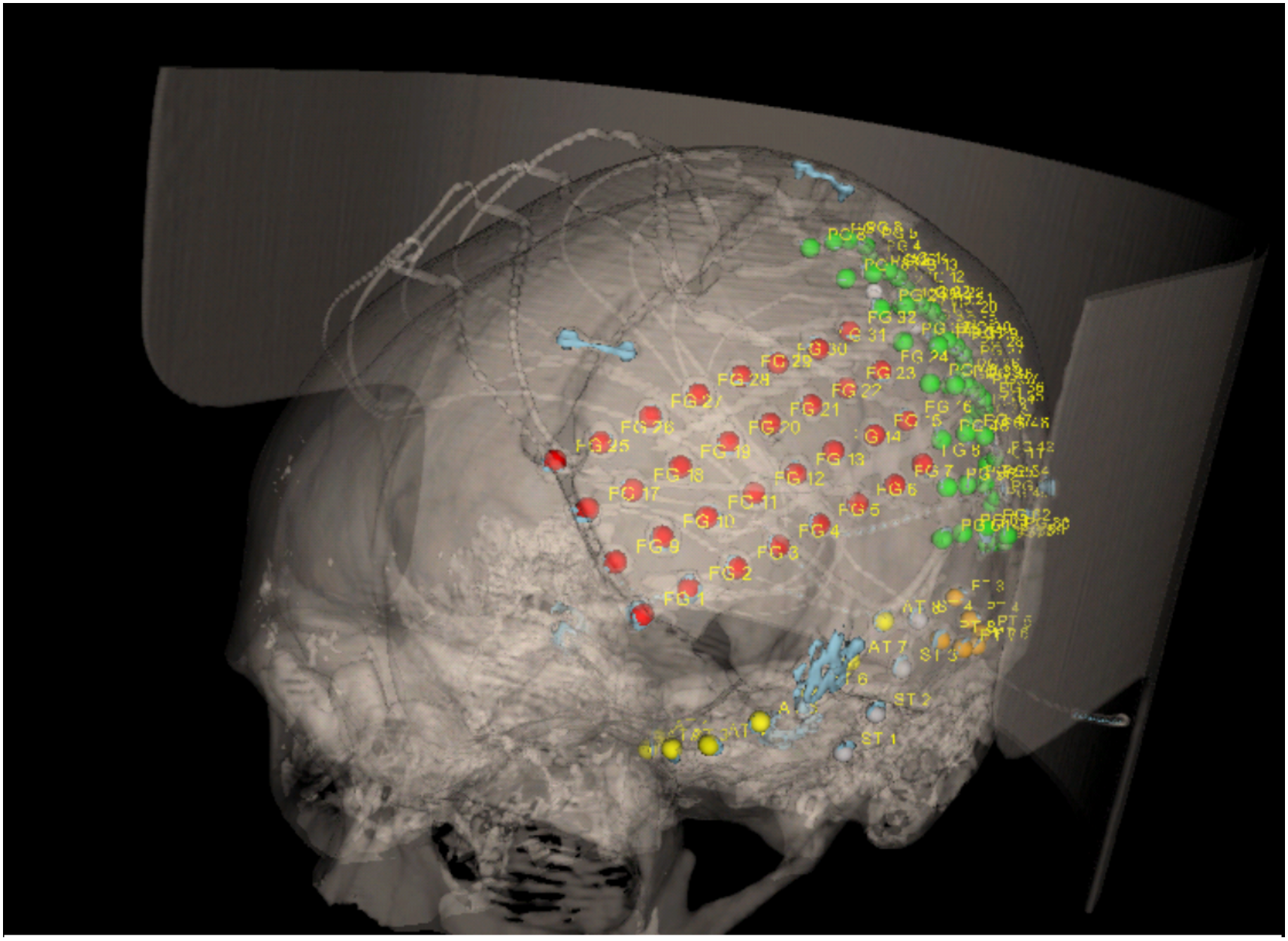
John Hunter, Department of Pediatric Neurology, University of Chicago.



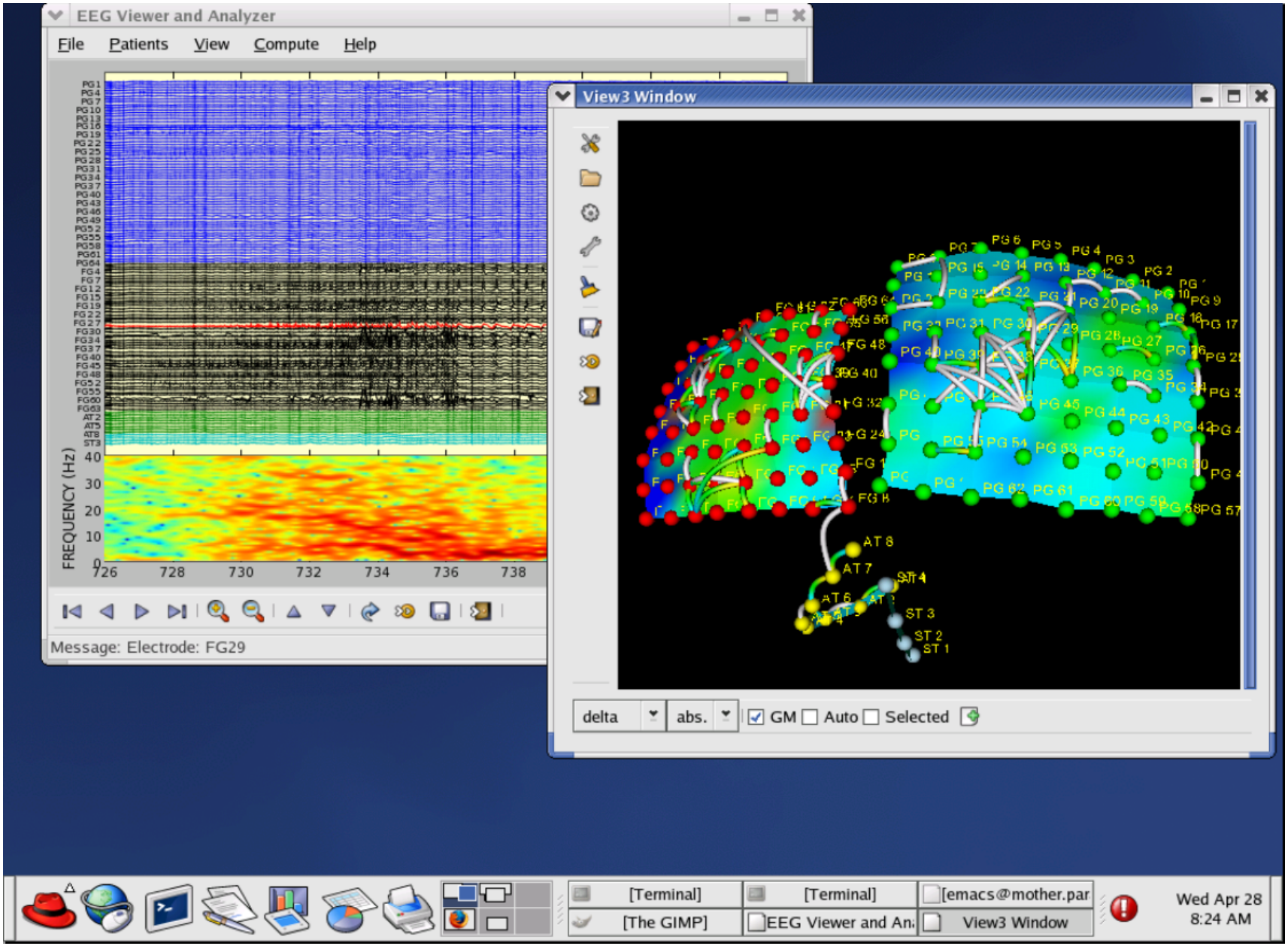
Electrode location in 3D, combined with MRI data



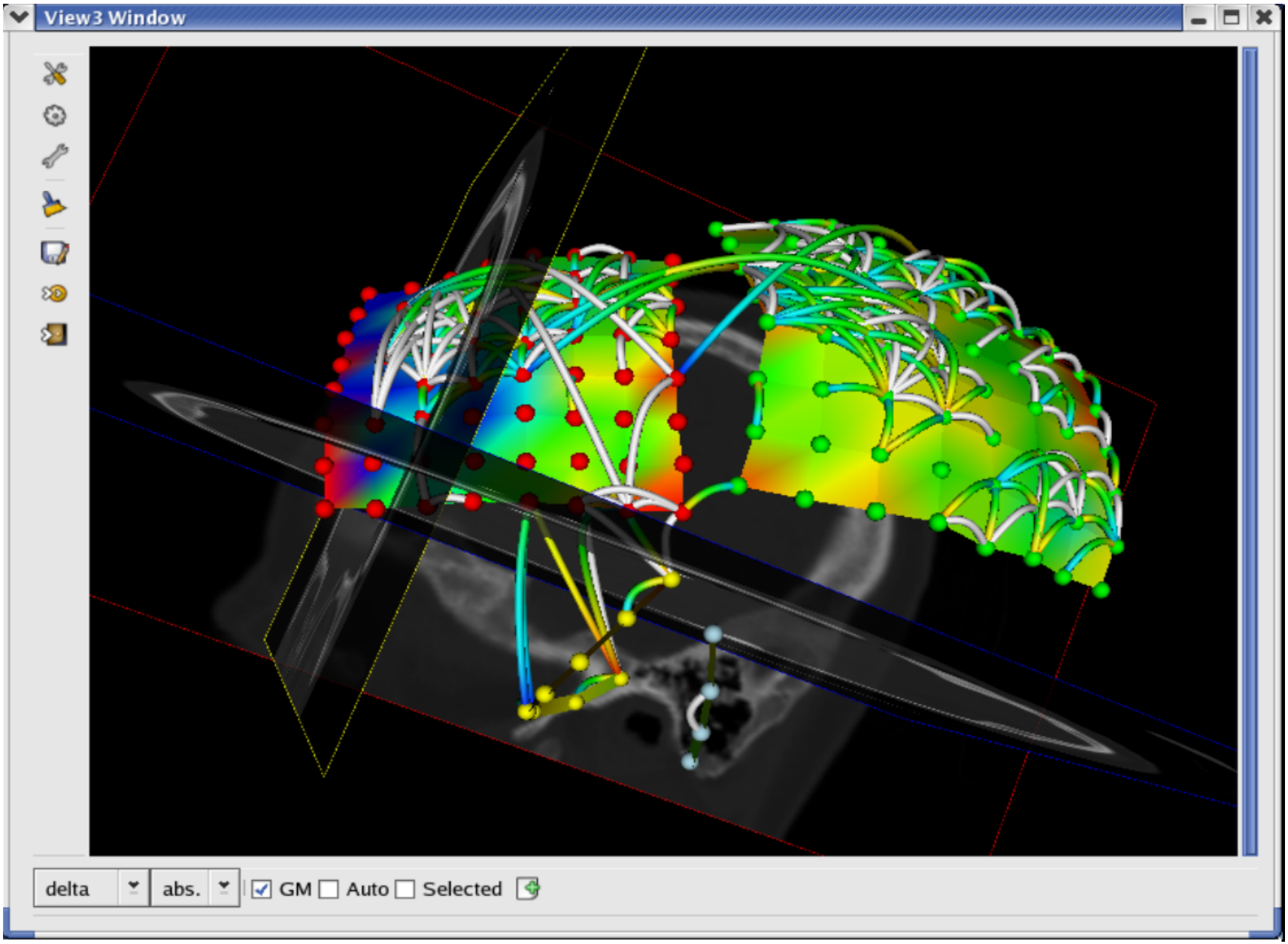
Full reconstruction of electrode location



Correlation analysis of seizure data



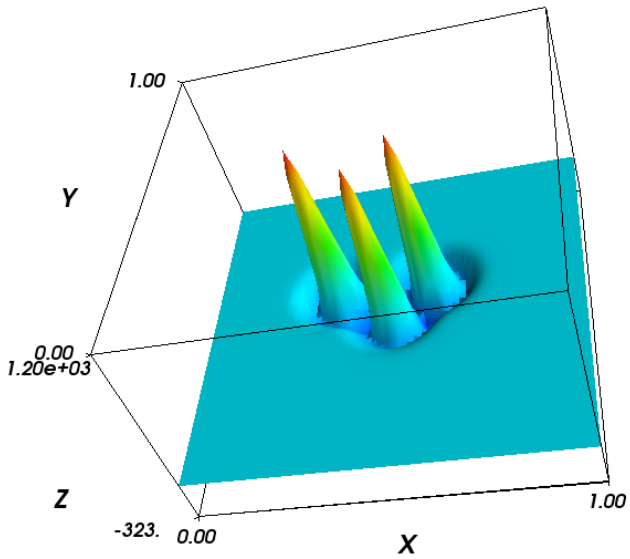
Final location of epileptic foci for surgery

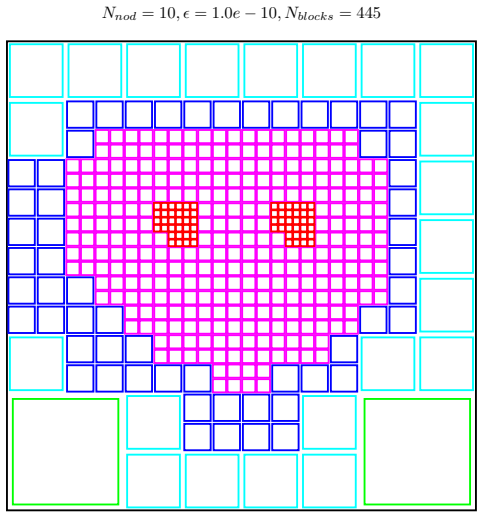


Adaptive, multiwavelet algorithms for integral operators

Gregory Beylkin, Vani Cheruvu (now NCAR), Fernando Perez. Applied Math, U. of Colorado at Boulder.

- Fast application of integral kernels. (Partial Differential Equations)
- Implementation went from 1 to 3 dimensions directly (*extremely* unusual).
- A clean Object Oriented design: the code reads like the underlying math.
- Very good performance, thanks to NumPy, F2PY and weave.





Structural Bio-Informatics

Michel F. Sanner, Molecular Biology Department, The Scripps Research Institute, La Jolla, California. http://www.scripps.edu/~sanner

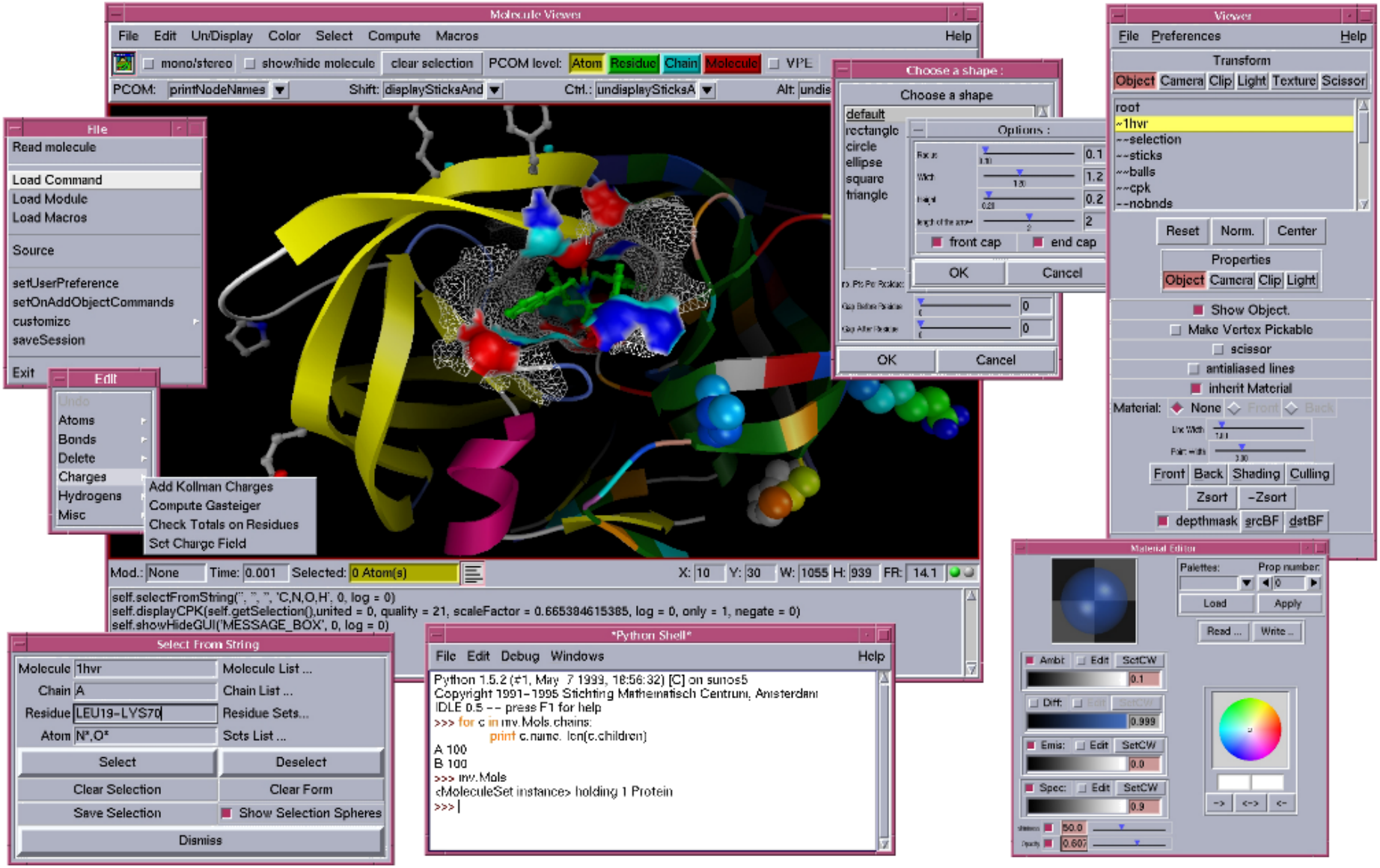
- Applications:
  - PMV: Visualization and analysis of biological molecules
  - Vision: Visual programming for Python.
  - PyARTK: Augmented Reality with Applications in Molecular Biology.
  - FlexTree: Python in the flexibility study of biological molecules
- Code
  - 11 Python packages
    - 220,000 lines of code
    - 1370 classes in Python
  - 10 C and C++ packages
    - 200 classes

Structural Bio-Informatics

Michel F. Sanner, Molecular Biology Department, The Scripps Research Institute, La Jolla, California. http://www.scripps.edu/~sanner

- Applications:
  - PMV: Visualization and analysis of biological molecules
  - Vision: Visual programming for Python.
  - PyARTK: Augmented Reality with Applications in Molecular Biology.
  - FlexTree: Python in the flexibility study of biological molecules
- Code
  - 11 Python packages
    - 220,000 lines of code
    - 1370 classes in Python
  - 10 C and C++ packages
    - 200 classes

PMV: the Python Molecule Viewer



MayaVi: sophisticated data visualization

- Free, easy to use scientific data visualizer.
- Heavy lifting of OpenGL-based rendering: VTK (a C++ library).
- A very good example of how to properly use Python:
  - MayaVi is a standalone GUI program...
  - but also a Python library callable by any other Python program.
  - Python: very easy to modify, even by adding at runtime user-defined modules which populate the GUI automatically.
  - C++: excellent rendering performance, fully hardware accelerated (OpenGL).

The punchline: fully programmable visualization, with builtin access to all kinds of numerical (and other) libraries from within the viz tool.

MayaVi: sophisticated data visualization

- Free, easy to use scientific data visualizer.
- Heavy lifting of OpenGL-based rendering: VTK (a C++ library).
- A very good example of how to properly use Python:
  - MayaVi is a standalone GUI program...
  - but also a Python library callable by any other Python program.
  - Python: very easy to modify, even by adding at runtime user-defined modules which populate the GUI automatically.
  - C++: excellent rendering performance, fully hardware accelerated (OpenGL).

The punchline: fully programmable visualization, with builtin access to all kinds of numerical (and other) libraries from within the viz tool.

MayaVi: sophisticated data visualization

- Free, easy to use scientific data visualizer.
- Heavy lifting of OpenGL-based rendering: VTK (a C++ library).
- A very good example of how to properly use Python:
  - MayaVi is a standalone GUI program...
  - but also a Python library callable by any other Python program.
  - Python: very easy to modify, even by adding at runtime user-defined modules which populate the GUI automatically.
  - C++: excellent rendering performance, fully hardware-accelerated OpenGL.

The punchline: fully programmable visualization, with builtin access to all kinds of numerical (and other) libraries from within the viz tool.

MayaVi: sophisticated data visualization

- Free, easy to use scientific data visualizer.
- Heavy lifting of OpenGL-based rendering: VTK (a C++ library).
- A very good example of how to properly use Python:
  - MayaVi is a standalone GUI program...
  - but also a Python library callable by any other Python program.
  - Python: very easy to modify, even by adding at runtime user-defined modules which populate the GUI automatically.
  - C++: excellent rendering performance, fully hardware-accelerated OpenGL.

The punchline: fully programmable visualization, with builtin access to all kinds of numerical (and other) libraries from within the viz tool.

MayaVi: sophisticated data visualization

- Free, easy to use scientific data visualizer.
- Heavy lifting of OpenGL-based rendering: VTK (a C++ library).
- A very good example of how to properly use Python:
  - MayaVi is a standalone GUI program...
  - but also a Python library callable by any other Python program.
  - Python: very easy to modify, even by adding at runtime user-defined modules which populate the GUI automatically.
  - C++: excellent rendering performance, fully hardware-accelerated OpenGL.

The punchline: fully programmable visualization, with builtin access to all kinds of numerical (and other) libraries from within the viz tool.

MayaVi: sophisticated data visualization

- Free, easy to use scientific data visualizer.
- Heavy lifting of OpenGL-based rendering: VTK (a C++ library).
- A very good example of how to properly use Python:
  - MayaVi is a standalone GUI program...
  - but also a Python library callable by any other Python program.
  - Python: very easy to modify, even by adding at runtime user-defined modules which populate the GUI automatically.
  - C++: excellent rendering performance, fully hardware-accelerated OpenGL.

The punchline: fully programmable visualization, with builtin access to all kinds of numerical (and other) libraries from within the viz tool.

MayaVi: sophisticated data visualization

- Free, easy to use scientific data visualizer.
- Heavy lifting of OpenGL-based rendering: VTK (a C++ library).
- A very good example of how to properly use Python:
  - MayaVi is a standalone GUI program...
  - but also a Python library callable by any other Python program.
  - Python: very easy to modify, even by adding at runtime user-defined modules which populate the GUI automatically.
  - C++: excellent rendering performance, fully hardware-accelerated OpenGL.

The punchline: fully programmable visualization, with builtin access to all kinds of numerical (and other) libraries from within the viz tool.

MayaVi: sophisticated data visualization

- Free, easy to use scientific data visualizer.
- Heavy lifting of OpenGL-based rendering: VTK (a C++ library).
- A very good example of how to properly use Python:
  - MayaVi is a standalone GUI program...
  - but also a Python library callable by any other Python program.
  - Python: very easy to modify, even by adding at runtime user-defined modules which populate the GUI automatically.
  - C++: excellent rendering performance, fully hardware-accelerated OpenGL.

The punchline: fully programmable visualization, with builtin access to all kinds of numerical (and other) libraries from within the viz tool.

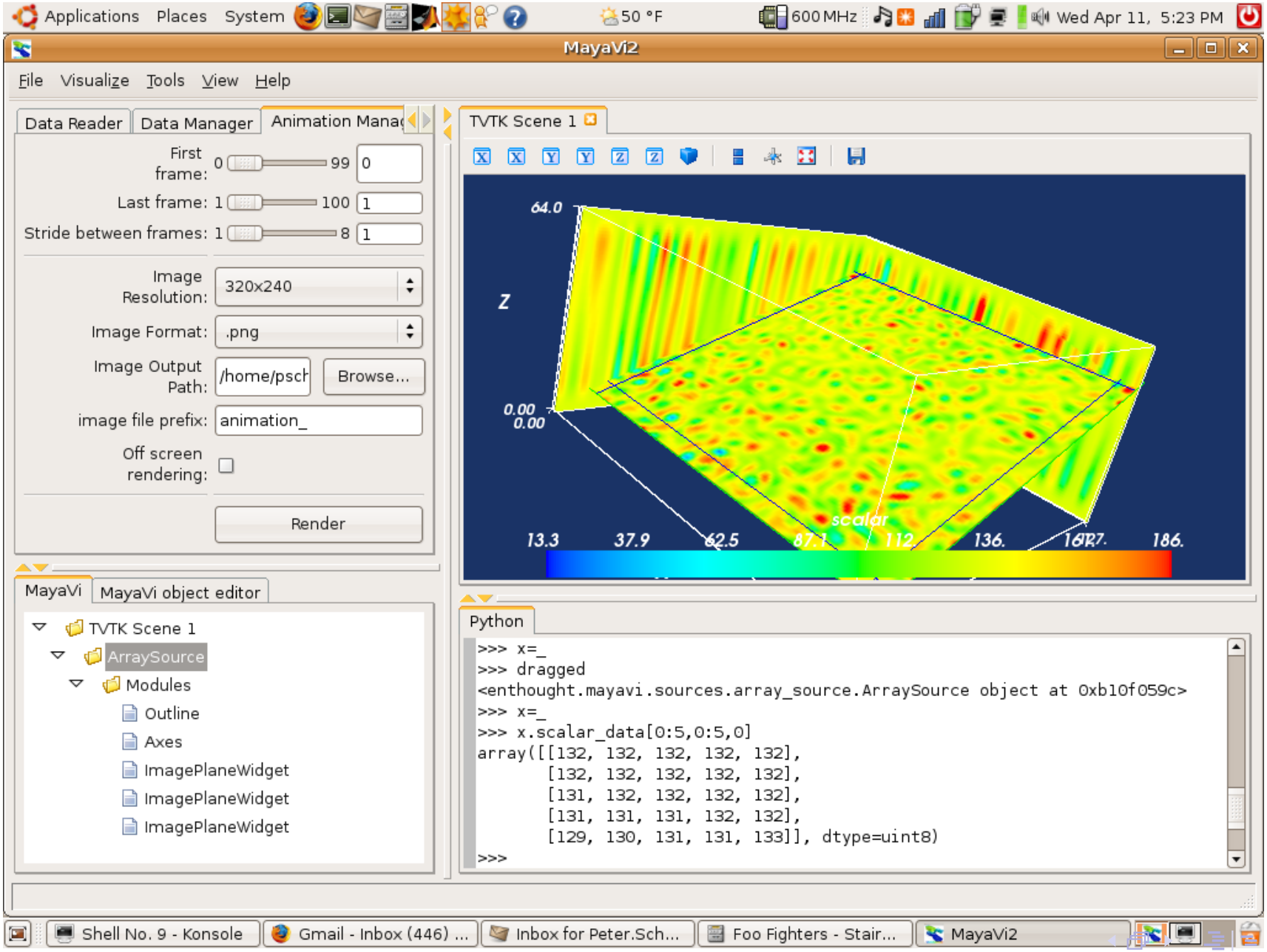
MayaVi: sophisticated data visualization

- Free, easy to use scientific data visualizer.
- Heavy lifting of OpenGL-based rendering: VTK (a C++ library).
- A very good example of how to properly use Python:
  - MayaVi is a standalone GUI program...
  - but also a Python library callable by any other Python program.
  - Python: very easy to modify, even by adding at runtime user-defined modules which populate the GUI automatically.
  - C++: excellent rendering performance, fully hardware-accelerated OpenGL.

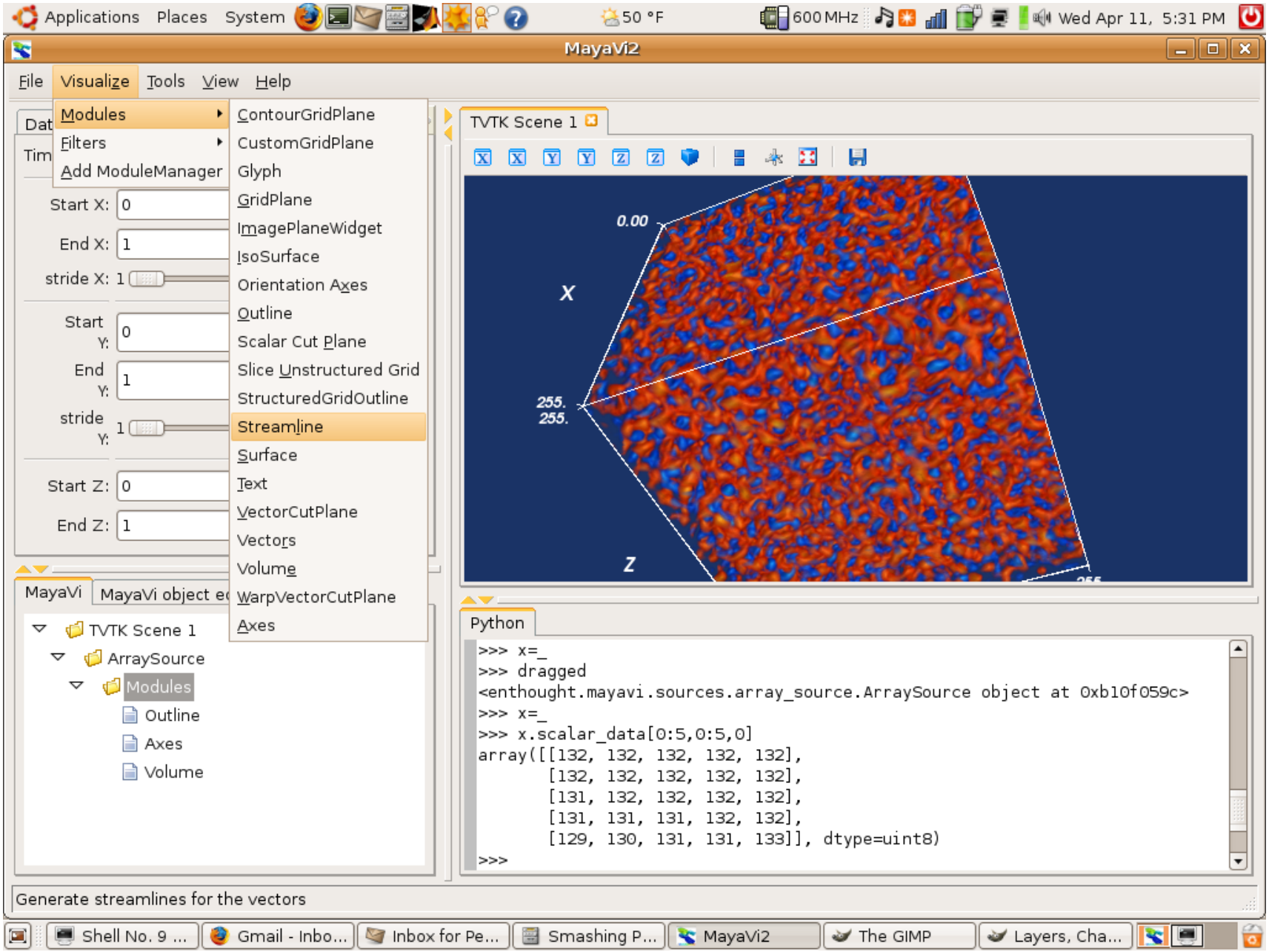
The punchline: fully programmable visualization, with builtin access to all kinds of numerical (and other) libraries from within the viz tool.

FluidLab: a MayaVi based CFD visualization tool

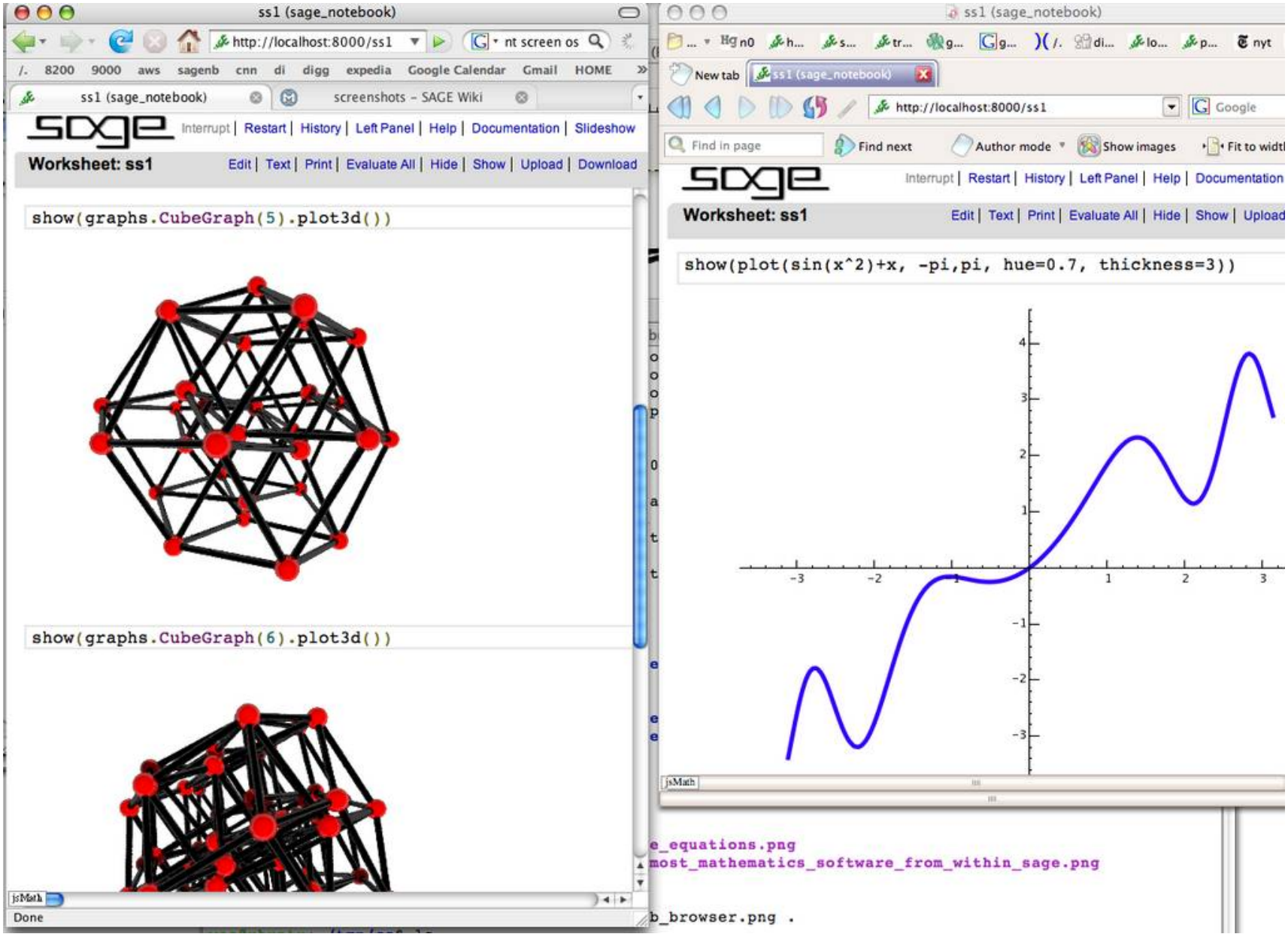
With: K. Julien, P. Schmitt (now NCAR) and B. Barrow (App. Math, CU).



Volumetric rendering with FluidLab



SAGE: open source mathematics



IPython

Extensible interactive environment with parallel computing support

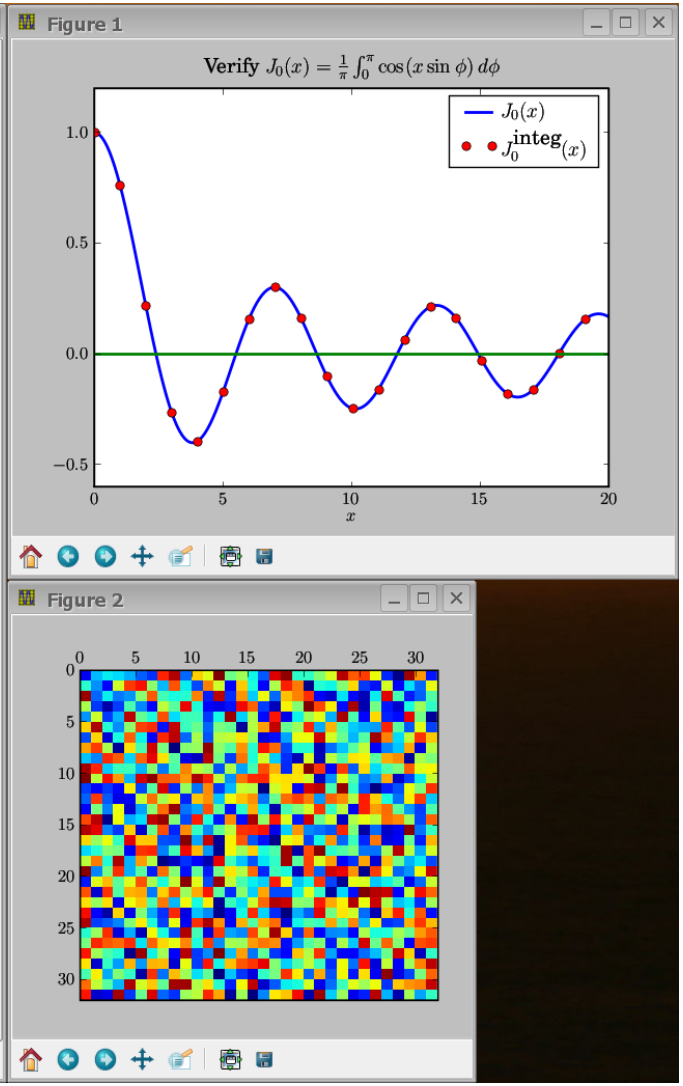
1. A better Python shell: object introspection, system access, 'magic' command system for adding functionality when working interactively, ...
2. An embeddable interpreter: useful for debugging and for mixing batch-processing with interactive work.
3. A flexible component: you can use it as the base environment for other systems with Python as the underlying language. It is very configurable in this direction.
4. A system for interactive control of distributed/parallel computing systems.
5. An interactive component we can plug into GUIs, browser-based shells, etc.

IPython: IDL-like interactive use

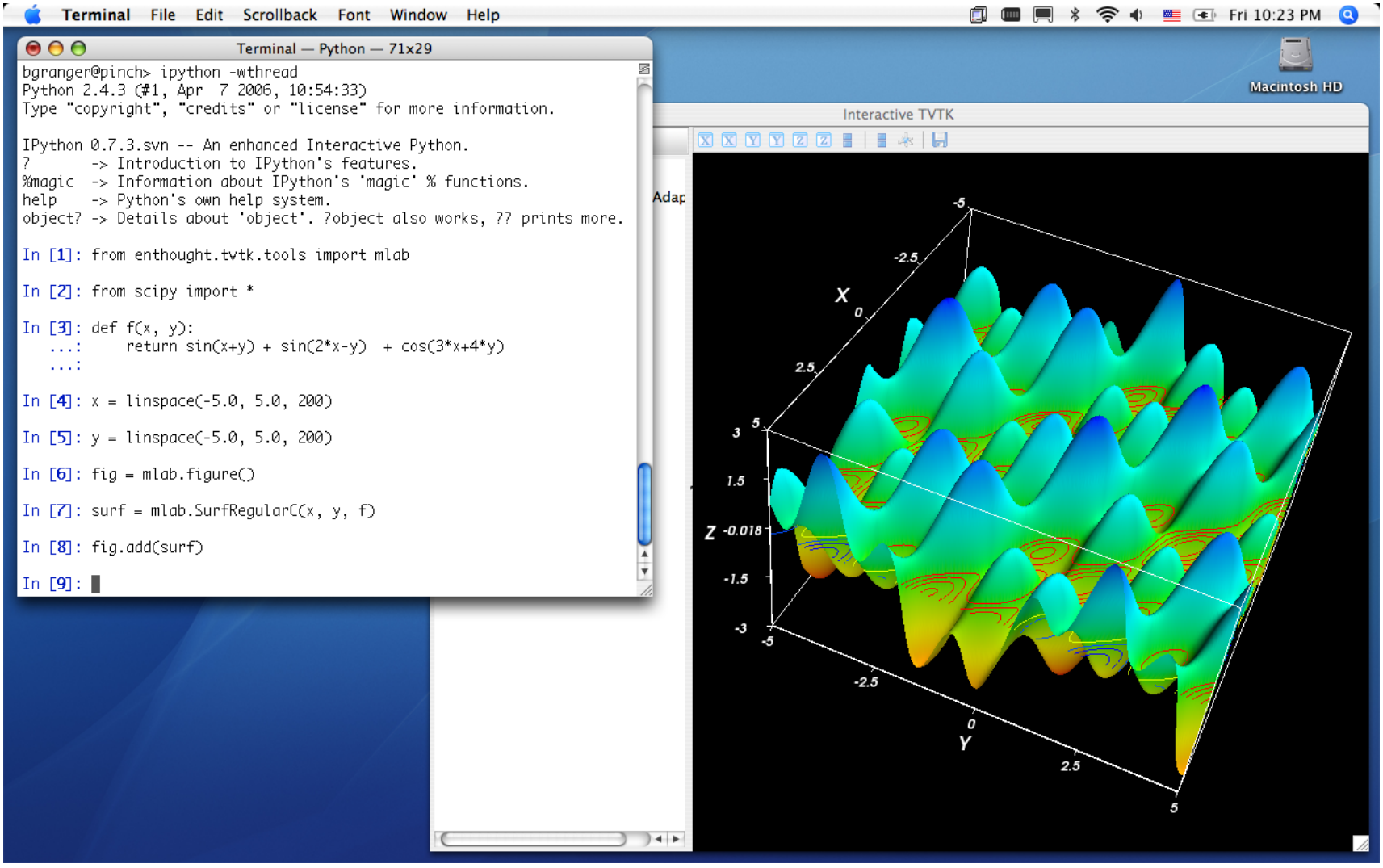
fperez@longs:/home/fperez - Shell - Konsole

```
longs[~]> ipython -pylab
Python 2.4.3 (#2, Apr 27 2006, 14:43:58)
Type "copyright", "credits" or "license" for more information.

IPython 0.7.3.svn -- An enhanced Interactive Python.
?       -> Introduction to IPython's features.
%magic  -> Information about IPython's 'magic' % functions.
help    -> Python's own help system.
object? -> Details about 'object'. ?object also works, ?? prints more.

  Welcome to pylab, a matplotlib-based Python environment.
  For more information, type 'help(pylab)'.

In [1]: import math, numpy

In [2]: from scipy.integrate import quad

In [3]: from scipy.special import j0

In [4]: def j0i(x):
   ...:     """Integral form of J_0(x)"""
   ...:     def integrand(phi):
   ...:         return math.cos(x*math.sin(phi))
   ...:     return (1.0/math.pi)*quad(integrand,0,math.pi)[0]
   ...:

In [5]: x = numpy.linspace(0,20,200) # sample grid: 200 points between 0 and 20

In [6]: y = j0(x)  # sample J0 at all values of x

In [7]: x1 = x[::10]  # subsample the original grid every 10th point

In [8]: y1 = map(j0i,x1)  # evaluate the integral form at all points in x1

In [9]: # Make a plot with these values (the ; suppresses output)

In [10]: plot(x,y,label=r'$J_0(x)$');

In [11]: plot(x1,y1,'ro',label=r'$J_0^{\textrm{integ}}(x)$');

In [12]: axhline(0,color='green',label='_nolegend_');

In [13]: title(r'Verify $J_0(x)=\frac{1}{\pi}\int_0^{\pi}\cos(x \sin\phi)\,d\phi$');

In [14]: xlabel('$x$');

In [15]: legend();

In [16]: matshow(numpy.random.random((32,32)))
Out[16]: <matplotlib.figure.Figure instance at 0x4630042c>
```

Shell



IPython: interactive control of VTK visualizations



IPython's future

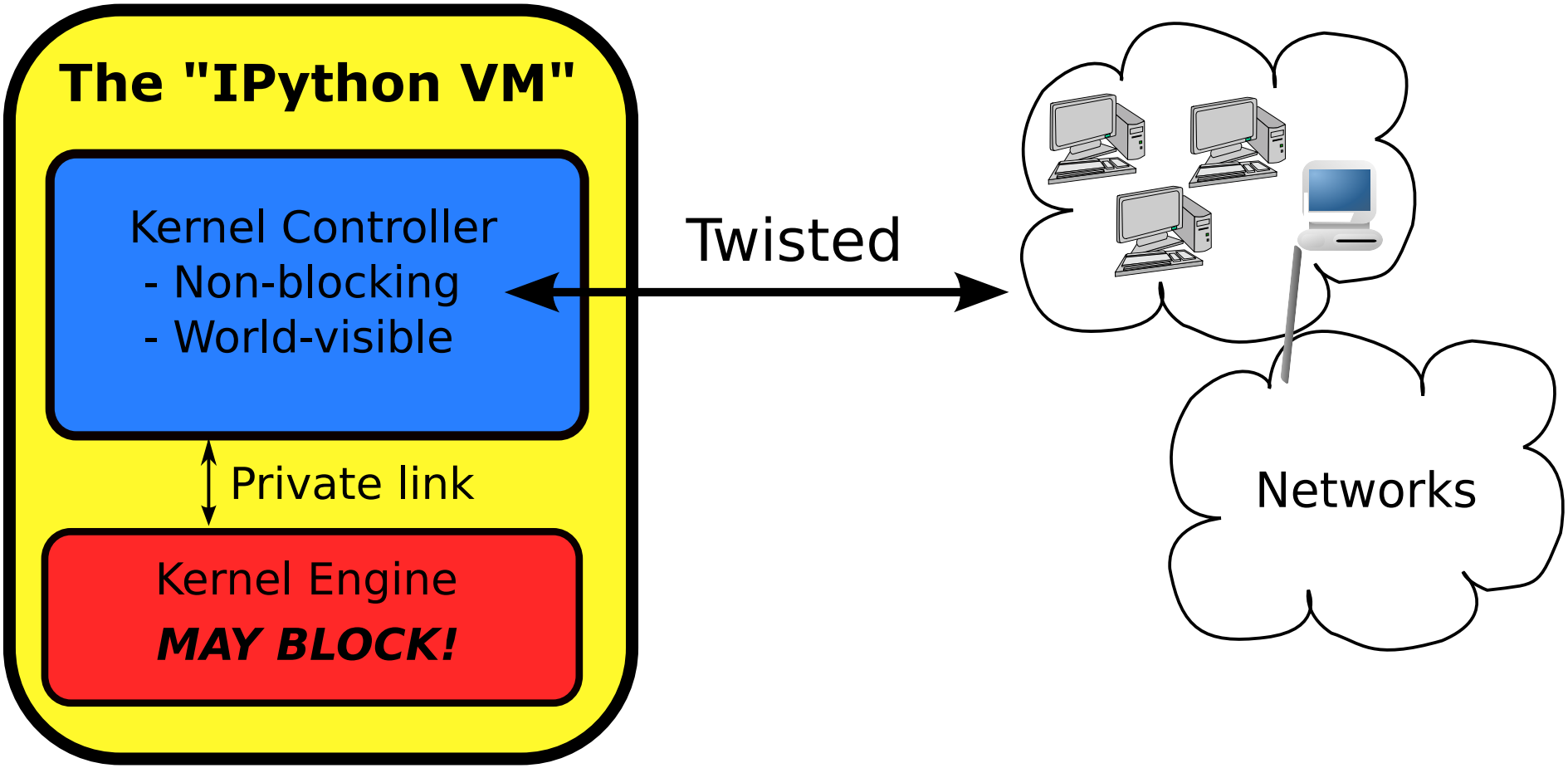
A 2-process kernel

Work with Brian Granger (Tech-X, Boulder) and Benjamin Ragan-Kelley (U. C. Berkeley physics).

Why do we need this?

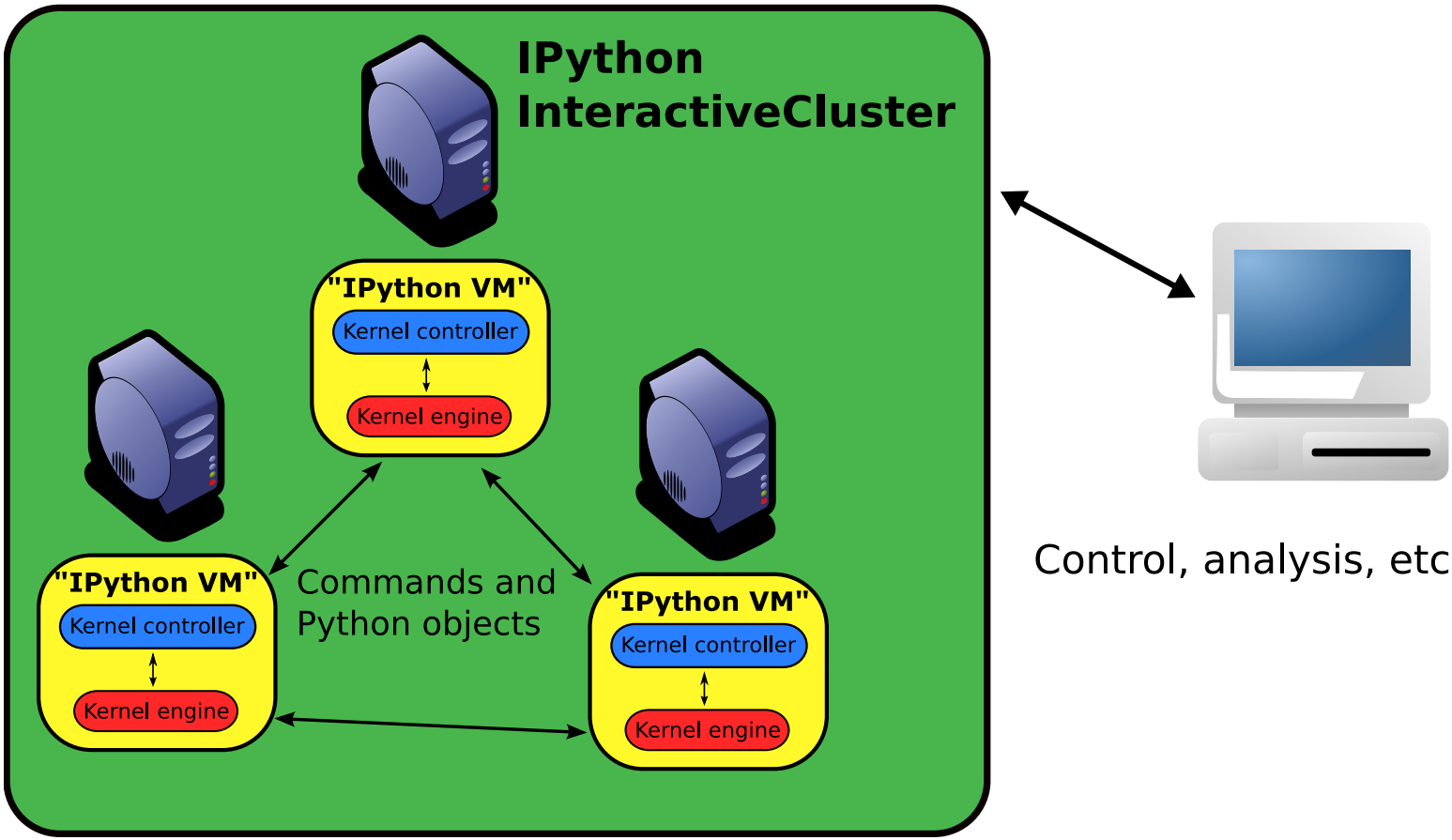
- The Python VM has a global lock (the Global Interpreter Lock – GIL).
- It protects the global state of the interpreter
  - Only one thread can execute Python code at the same time.
  - No Python variables may be modified without holding the GIL.
- Python *does* have threads: they work well for non-CPU bound tasks.
- **BUT**
  - Extensions (C, Fortran) can fully block the VM.
  - And poof goes all hope of the ability to control a cluster

A 2-process kernel on the network



Distributed/parallel computing

- Think of Python as 'the CPU'
- But these souped-up kernels let you talk to it conveniently.

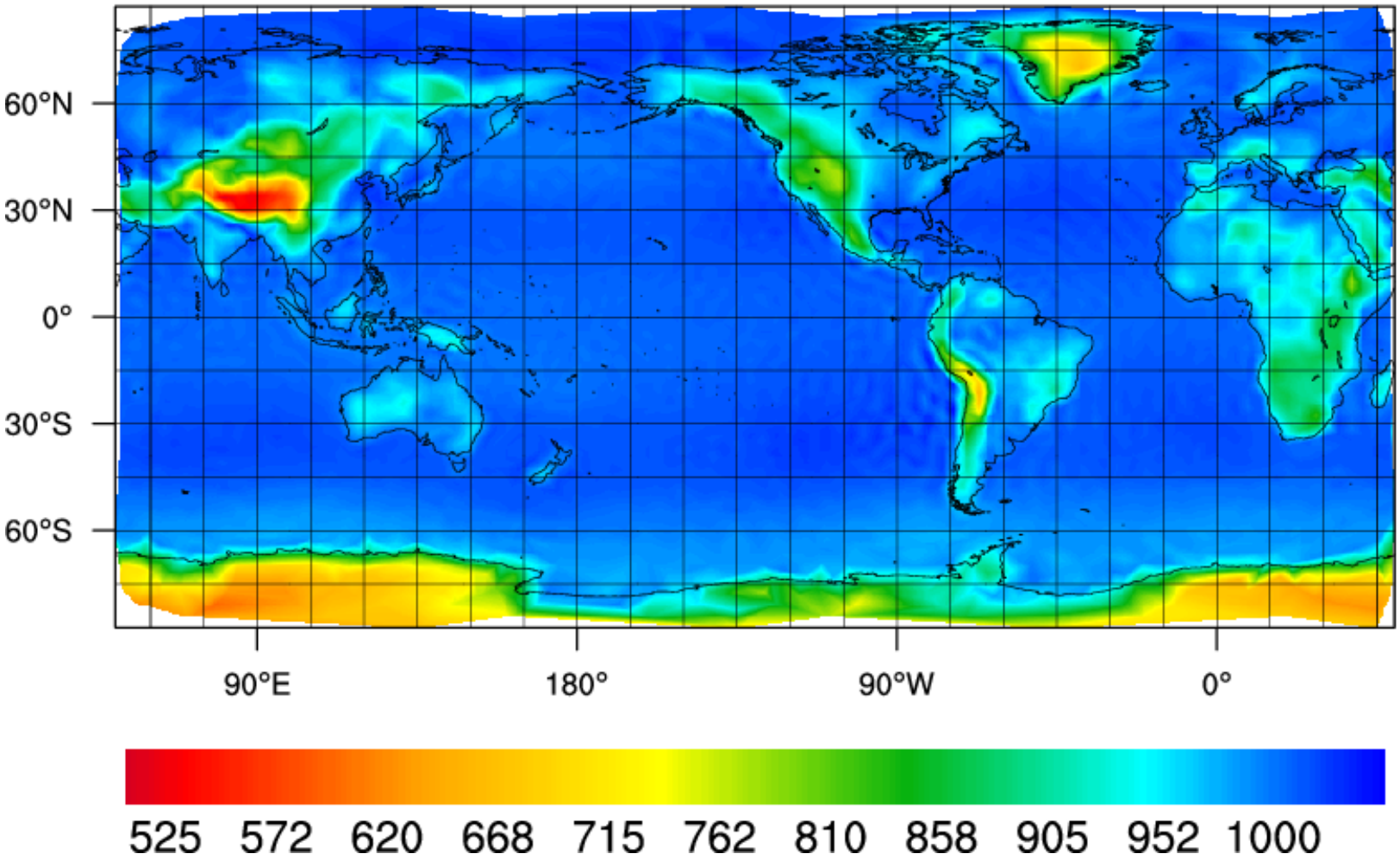


PyNGL - Python interface to NCAR's NCL

```
import numpy, Ngl, Nio
dirc = Ngl.pynglpath("data")
cfile = Nio.open_file(dirc + "/cdf/seam.nc")
lon = numpy.ravel(cfile.variables["lon2d"][:,:])
lat = numpy.ravel(cfile.variables["lat2d"][:,:])
ps = numpy.ravel(cfile.variables["ps"][0,:,:])/100.
rlist = Ngl.Resources()
rlist.wkColorMap = "BlAqGrYeOrReVi200"
wks_type = "ps"
wks = Ngl.open_wks(wks_type,"seam",rlist)
resources = Ngl.Resources()
resources.sfXArray = lon
resources.sfYArray = lat
map = Ngl.contour_map(wks,ps,resources)
Ngl.end()
```

PyNGL

HOMME grid: Surface pressure w/smoothing



A few other projects

Python is becoming very popular in many different scientific areas

- PyDAP: Python implementation of the OpenDAP protocols (client **and** server).
- PyTables: HDF-5 read-write support with excellent performance.
- PyTrilinos: Python interface to Sandia's Trilinos parallel solvers. Coupled with IPython, they can be used interactively.
- PyRAF: Hubble Space Telescope interface to IRAF, a standard in astronomical image processing.
- VPython: easy, real-time 3D programming (Carnegie Mellon, used for an introductory mechanics course).
- Galaxy: integrated access to multiple tools in genomics research. Very impressive.

Outline

Basic Python features

Meaningful indentation, self-documenting, interactive language

- Examples below: IPython (enhanced interactive environment)
- Exploratory, incremental development, with live debugging on exceptions.
- Direct access to the filesystem and OS.

```
In [8]: def hypot(a,b):
   ...:     "Return the length of the hypotenuse."
   ...:     return sqrt(a**2+b**2)
In [9]: pdoc hypot
Return the length of the hypotenuse.
In [10]: pdef hypot
hypot(a, b)
In [12]: cd talks/figs/
/home/fperez/talks/figs
In [13]: ls
cise_cover.jpg kernel2p.eps ...
```

Python Basics (2)

Dictionaries (C-implemented, well optimized hash tables)

- Perfect for building complex, sparse data structures

```
In [21]: dct={'k1':'v1',(3,4):cos,'nest':{1:2}}
In [22]: dct.keys(), dct.values()
Out[22]: (['k1', 'nested', (3, 4)], ['v1', {1:  2}, <built-in
function cos>])
In [23]: dct[3,4](pi)
Out[23]: -1.0
```

Easy access to C/C++ (via SciPy's `weave.inline`) and FORTRAN (`f2py`)

```
In [26]: code='std::cout << "a is:  " << a << std::endl;'
In [27]: a='Hello world'
In [28]: inline(code,['a'])
a is:  Hello world
In [29]: a=99
In [30]: inline(code,['a'])
a is:  99
```

Python Basics (2)

Dictionaries (C-implemented, well optimized hash tables)

- Perfect for building complex, sparse data structures

```
In [21]: dct={'k1':'v1',(3,4):cos,'nest':{1:2}}
In [22]: dct.keys(), dct.values()
Out[22]: (['k1', 'nested', (3, 4)], ['v1', {1:  2}, <built-in
function cos>])
In [23]: dct[3,4](pi)
Out[23]: -1.0
```

Easy access to C/C++ (via SciPy's `weave.inline`) and FORTRAN (`f2py`)

```
In [26]: code='std::cout << "a is:  " << a << std::endl;'
In [27]: a='Hello world'
In [28]: inline(code,['a'])
a is:  Hello world
In [29]: a=99
In [30]: inline(code,['a'])
a is:  99
```

Python compared to IDL, Matlab, etc.

Pros

A general programming language: **this is a feature**!

- Free, open source, extremely portable: from the OLPC or a cellphone (Nokia) to a supercomputer.
- Networking, text processing, XML parsing, database access, etc. . .
- Integrated support for testing (`unittest`, `doctest`)
- Automatic API documentation tools are standard (`doxygen`).
- Supports all major GUI toolkits.
- Extremely expressive for complex algorithms.

There are still rough edges!

- Installation, deployment: harder than needed (but improving rapidly).
- No good, single-point of entry integrated help system.
- Lots of good documentation, but scattered all over.
- Funding agency support for infrastructure work is difficult to get.

Python compared to IDL, Matlab, etc.

Pros

A general programming language: **this is a feature**!

- Free, open source, extremely portable: from the OLPC or a cellphone (Nokia) to a supercomputer.
- Networking, text processing, XML parsing, database access, etc. . .
- Integrated support for testing (`unittest`, `doctest`)
- Automatic API documentation tools are standard (`doxygen`).
- Supports all major GUI toolkits.
- Extremely expressive for complex algorithms.

There are still rough edges!

- Installation, deployment: harder than needed (but improving rapidly).
- No good, single-point of entry integrated help system.
- Lots of good documentation, but scattered all over.
- Funding agency support for infrastructure work is difficult to get.

A different model of development

Global optimization is the root of all evil

- Never write '`main()`' in C anymore: you are optimizing globally!
  - Prototype the code in Python.
  - Wrap existing libraries for Python access and reuse them (Numeric, LAPACK, VTK, ...)
  - Identify remaining hot spots via profiling.
  - Rewrite only the code for those hot spots in C/C++/FORTRAN.
- The resulting code will be production-ready: no throw-away codes.
  - Make your code available as a library for interactive use.
  - Integrate plotting, visualization, logging, ..., into your objects.
- Apply this to existing codes
  - Break them into a library core and control layers.
  - Wrap the libraries and expose them to Python.
  - Write all new control as quick, light Python scripts.

A different model of development

Global optimization is the root of all evil

- Never write '`main()`' in C anymore: you are optimizing globally!
  - Prototype the code in Python.
  - Wrap existing libraries for Python access and reuse them (Numeric, LAPACK, VTK, ...)
  - Identify remaining hot spots via profiling.
  - Rewrite only the code for those hot spots in C/C++/FORTRAN.
- The resulting code will be production-ready: no throw-away codes.
  - Make your code available as a library for interactive use.
  - Integrate plotting, visualization, logging, ..., into your objects.
- Apply this to existing codes
  - Break them into a library core and control layers.
  - Wrap the libraries and expose them to Python.
  - Write all new control as quick, light Python scripts.

A different model of development

Global optimization is the root of all evil

- Never write '`main()`' in C anymore: you are optimizing globally!
  - Prototype the code in Python.
  - Wrap existing libraries for Python access and reuse them (Numeric, LAPACK, VTK, ...)
  - Identify remaining hot spots via profiling.
  - Rewrite only the code for those hot spots in C/C++/FORTRAN.
- The resulting code will be production-ready: no throw-away codes.
  - Make your code available as a library for interactive use.
  - Integrate plotting, visualization, logging, ..., into your objects.
- Apply this to existing codes
  - Break them into a library core and control layers.
  - Wrap the libraries and expose them to Python.
  - Write all new control as quick, light Python scripts.

Summary

- Python
  - An excellent language for scientific computing development.
  - Scales from interactive exploration to full-blown production codes.
  - Accessible to scientists who are not professional programmers.
- Outlook
  - NumPy, SciPy, matplotlib, IPython, MayaVi, . . . are moving forward and improving.
  - Major DOE, NSF, NiH projects are adopting it as a core technology.
  - Yearly conference at Caltech (just finished) growing.
  - These projects are all Open Source: if you find a flaw, a bug, or a missing feature, *jump on board!*
  - There are still many rough edges to which various projects can contribute.

Summary

- Python
  - An excellent language for scientific computing development.
  - Scales from interactive exploration to full-blown production codes.
  - Accessible to scientists who are not professional programmers.

- Outlook
  - NumPy, SciPy, matplotlib, IPython, MayaVi, . . . are moving forward and improving.
  - Major DOE, NSF, NiH projects are adopting it as a core technology.
  - Yearly conference at Caltech (just finished) growing.
  - These projects are all Open Source: if you find a flaw, a bug, or a missing feature, *jump on board!*
  - There are still many rough edges to which various projects can contribute.

An overview of Python's features

A readable, eclectic collection of the best features from many languages.

Data types

- Arbitrary length integers

```
In [1]: 2**64
Out[1]: 18446744073709551616L
```

- Floats (standard C doubles) and complex numbers

```
In [4]: 1j**2
Out[4]: (-1+0j)
```

- Strings

```
In [6]: 'hello world'.upper()
Out[6]: 'HELLO WORLD'
```

- Lists (arbitrarily nested, variable length)

```
In [9]: [99,'hello',1j,['sublist'],99].count(99)
Out[9]: 2
```

Python Basics (2)

Data types (cont)

- Dictionaries (C-implemented, well optimized hash tables)

```
In [21]: dct={'k1':'v1',2:'v2',(3,4):math.cos,'nest':{1:2}}
In [22]: dct.keys()
Out[22]: ['k1', 'nest', 2, (3, 4)]
In [23]: dct.values()
Out[23]: ['v1', {1:  2}, 'v2', <built-in function cos>]
In [24]: dct[3,4](math.pi)
Out[24]: -1.0
```

Strongly, but dynamically typed

- One of its major strengths: extreme flexibility.
- Slow: everything is checked at runtime.

```
for x in range(10):
    print x**2 # x and ** are checked every time!
```

Python Basics (2)

Data types (cont)

- Dictionaries (C-implemented, well optimized hash tables)

```
In [21]: dct={'k1':'v1',2:'v2',(3,4):math.cos,'nest':{1:2}}
In [22]: dct.keys()
Out[22]: ['k1', 'nest', 2, (3, 4)]
In [23]: dct.values()
Out[23]: ['v1', {1:  2}, 'v2', <built-in function cos>]
In [24]: dct[3,4](math.pi)
Out[24]: -1.0
```

Strongly, but dynamically typed

- One of its major strengths: extreme flexibility.
- Slow: everything is checked at runtime.

```
for x in range(10):
    print x**2  # x and ** are checked every time!
```

Python Basics (3)

Interactive

- Efficient for exploratory, incremental development.
- Live debugging on exceptions.
- Direct access to the filesystem and OS.

Clean object system

With multiple inheritance and operator overloading:

```
In [12]: class simple:
   ....:     def __add__(self,other):
   ....:         print 'Me plus something else:',other
In [13]: a = simple()
In [14]: a + 34
Me plus something else: 34
```

Python Basics (3)

Interactive

- Efficient for exploratory, incremental development.
- Live debugging on exceptions.
- Direct access to the filesystem and OS.

Clean object system

With multiple inheritance and operator overloading:

```
In [12]: class simple:
   ....:     def __add__(self,other):
   ....:         print 'Me plus something else:',other
In [13]: a = simple()
In [14]: a + 34
Me plus something else: 34
```

Python Basics (4)

Functions are first class objects

```
def compose(f,g):
    return lambda x:  f(g(x))
```

Easy access to C/C++ (with SciPy's weave.inline module)

```
In [26]: code='std::cout << "a is:  " << a << std::endl;'
In [27]: a='Hello world'
In [28]: inline(code,['a'])
a is:  Hello world
In [29]: a=99
In [30]: inline(code,['a'])
a is:  99
```

Elegant, simple and expressive: quicksort in 3 lines (Nathan Gray)

```
def qsort(L):
    if len(L) <= 1:  return L
    return qsort( [ lt for lt in L[1:]  if lt < L[0] ] ) + \
        [L[0]] + qsort([ge for ge in L[1:]  if ge >= L[0]])
```

Python Basics (4)

Functions are first class objects

```
def compose(f,g):
    return lambda x:  f(g(x))
```

Easy access to C/C++ (with SciPy's weave.inline module)

```
In [26]: code='std::cout << "a is:  " << a << std::endl;'
In [27]: a='Hello world'
In [28]: inline(code,['a'])
a is:  Hello world
In [29]: a=99
In [30]: inline(code,['a'])
a is:  99
```

Elegant, simple and expressive: quicksort in 3 lines (Nathan Gray)

```
def qsort(L):
    if len(L) <= 1:  return L
    return qsort( [ lt for lt in L[1:]  if lt < L[0] ] ) + \
        [L[0]] + qsort([ge for ge in L[1:]  if ge >= L[0]])
```

Python Basics (4)

Functions are first class objects

```
def compose(f,g):
    return lambda x:  f(g(x))
```

Easy access to C/C++ (with SciPy's weave.inline module)

```
In [26]: code='std::cout << "a is:  " << a << std::endl;'
In [27]: a='Hello world'
In [28]: inline(code,['a'])
a is:  Hello world
In [29]: a=99
In [30]: inline(code,['a'])
a is:  99
```

Elegant, simple and expressive: quicksort in 3 lines (Nathan Gray)

```
def qsort(L):
    if len(L) <= 1:  return L
    return qsort( [ lt for lt in L[1:]  if lt < L[0] ] ) + \
        [L[0]] + qsort([ge for ge in L[1:]  if ge >= L[0]])
```