Python in Scientific Computing
An Illustration with Multiwavelets for PDEs

Fernando Pérez
<Fernando.Perez@colorado.edu>

Department of Applied Mathematics
University of Colorado
Boulder, Colorado

Colorado School of Mines
November 19th, 2004
Outline

1 Scientific Computing
   - Traditional approaches
   - Python?

2 Interlude: Python in the real world
   - PMV: molecular structures
   - EEG analysis for epilepsy
   - MayaVi: data visualization

3 Python and Scientific Computing
   - Basic features
   - A better development model

4 PDEs, Green’s Functions and Multiwavelets
   - The ‘curse of dimensionality’
   - Multiresolution analysis
   - Poisson’s equation
1. Scientific Computing
   - Traditional approaches
   - Python?

2. Interlude: Python in the real world
   - PMV: molecular structures
   - EEG analysis for epilepsy
   - MayaVi: data visualization

3. Python and Scientific Computing
   - Basic features
   - A better development model

4. PDEs, Green’s Functions and Multiwavelets
   - The ‘curse of dimensionality’
   - Multiresolution analysis
   - Poisson’s equation
Outline

1. Scientific Computing
   - Traditional approaches
   - Python?

2. Interlude: Python in the real world
   - PMV: molecular structures
   - EEG analysis for epilepsy
   - MayaVi: data visualization

3. Python and Scientific Computing
   - Basic features
   - A better development model

4. PDEs, Green’s Functions and Multiwavelets
   - The ‘curse of dimensionality’
   - Multiresolution analysis
   - Poisson’s equation
Outline

1. Scientific Computing
   - Traditional approaches
   - Python?

2. Interlude: Python in the real world
   - PMV: molecular structures
   - EEG analysis for epilepsy
   - MayaVi: data visualization

3. Python and Scientific Computing
   - Basic features
   - A better development model

4. PDEs, Green’s Functions and Multiwavelets
   - The ‘curse of dimensionality’
   - Multiresolution analysis
   - Poisson’s equation
Outline

1. Scientific Computing
   - Traditional approaches
   - Python?

2. Interlude: Python in the real world
   - PMV: molecular structures
   - EEG analysis for epilepsy
   - MayaVi: data visualization

3. Python and Scientific Computing
   - Basic features
   - A better development model

4. PDEs, Green’s Functions and Multiwavelets
   - The ‘curse of dimensionality’
   - Multiresolution analysis
   - Poisson’s equation
**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, ...).
  - 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 FORTRAN & C, clumsy and subtle in C++.

**However:**

- They deliver excellent performance.
- Millions of LOC in existing scientific libraries (LAPACK, BLAS, ...).
**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, ...).
  - 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 FORTRAN & C, clumsy 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, ...).
  - 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 FORTRAN & C, clumsy and subtle in C++.

However!

- They deliver excellent performance.
- Millions of LOC in existing scientific libraries (LAPACK, BLAS, ...).
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, ...).
  - 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 FORTRAN & C, clumsy and subtle in C++.

However!

- They deliver excellent performance.
- Millions of LOC in existing scientific libraries (LAPACK, BLAS, ...).
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, ...).
  - 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 FORTRAN & C, clumsy and subtle in C++.

However!

- They deliver excellent performance.
- Millions of LOC in existing scientific libraries (LAPACK, BLAS, ...).
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, . . .).
  - 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 FORTRAN & C, clumsy and subtle in C++.

However!

- They deliver excellent performance.
- Millions of LOC in existing scientific libraries (LAPACK, BLAS, . . .)
**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, ...).
  - 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 FORTRAN & C, clumsy 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, ...).
  - 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 FORTRAN & C, clumsy 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: a unique niche I won’t address today.
- 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: a unique niche I won’t address today.
- 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: a unique niche I won’t address today.
- 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: a unique niche I won’t address today.
- 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: a unique niche I won’t address today.
- 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: a unique niche I won’t address today.
- 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: a unique niche I won’t address today.

- 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: a unique niche I won’t address today.
- 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 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 (Numeric/numarray).
- 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 (Numeric/numarray).
- 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 (Numeric/numarray).
- 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 (Numeric/numarray).
- 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 (Numeric/numarray).
- 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 (Numeric/numarray).
- 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 (Numeric/numarray).
- 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 (Numeric/numarray).
- 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 (Numeric/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 (Numeric/numarray).
- 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

1. Scientific Computing
   - Traditional approaches
   - Python?

2. Interlude: Python in the real world
   - PMV: molecular structures
   - EEG analysis for epilepsy
   - MayaVi: data visualization

3. Python and Scientific Computing
   - Basic features
   - A better development model

4. PDEs, Green's Functions and Multiwavelets
   - The ‘curse of dimensionality’
   - Multiresolution analysis
   - Poisson’s equation
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
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
MayaVi: sophisticated data visualization

- MayaVi: [http://mayavi.sourceforge.net](http://mayavi.sourceforge.net)
- 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: sophisticated data visualization

- **MayaVi**: [http://mayavi.sourceforge.net](http://mayavi.sourceforge.net)
- 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.

[DEMO]

MayaVi: sophisticated data visualization

- **MayaVi**: [http://mayavi.sourceforge.net](http://mayavi.sourceforge.net)
- 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.
MayaVi: sophisticated data visualization

- **MayaVi**: [http://mayavi.sourceforge.net](http://mayavi.sourceforge.net)
- 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.
MayaVi: sophisticated data visualization

- **MayaVi**: [http://mayavi.sourceforge.net](http://mayavi.sourceforge.net)
- 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.

[DEMO]
MayaVi: sophisticated data visualization

- **MayaVi**: [http://mayavi.sourceforge.net](http://mayavi.sourceforge.net)
- 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.
MayaVi: sophisticated data visualization

- **MayaVi**: [http://mayavi.sourceforge.net](http://mayavi.sourceforge.net)
- 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.

[DEMO]
MayaVi: sophisticated data visualization

- **MayaVi**: [http://mayavi.sourceforge.net](http://mayavi.sourceforge.net)
- 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.
MayaVi: sophisticated data visualization

- **MayaVi**: [http://mayavi.sourceforge.net](http://mayavi.sourceforge.net)
- 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.

[DEMO]
MayaVi: sophisticated data visualization

- **MayaVi**: [http://mayavi.sourceforge.net](http://mayavi.sourceforge.net)
- 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.

[DEMO]
A few other projects (URLs in Appendix)

Python is becoming very popular in many different scientific areas

- **VPython**: easy, real-time 3D programming (Carnegie Mellon, used for an introductory mechanics course).
- **UCSF Chimera**: extensible, interactive molecular graphics program ($$$: NIH).
- **PyRAF**: Hubble Space Telescope interface to IRAF, a standard in astronomical image processing. The HST develops Numarray, the next generation Python array library.
- **BioPython**: tools for computational molecular biology.
- **VisionEgg**: real-time stimuli for vision research experiments (OpenGL).
- **Natural Language Toolkit**: symbolic and statistical natural language processing.
- **Neural Integrator**: visual programming for neural networks.
- **Orange**: component-based data mining software.
Outline

1 Scientific Computing
   • Traditional approaches
   • Python?

2 Interlude: Python in the real world
   • PMV: molecular structures
   • EEG analysis for epilepsy
   • MayaVi: data visualization

3 Python and Scientific Computing
   • Basic features
   • A better development model

4 PDEs, Green’s Functions and Multiwavelets
   • The ‘curse of dimensionality’
   • Multiresolution analysis
   • Poisson’s equation
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/0411_mines/
/home/fperez/talks/0411_mines
In [13]: ls
0411_python_scicomp.lyx doublependulum.py* ...
```
**Dictionaries (C-implemented, well optimized hash tables)**

- **Perfect for building complex, sparse data structures**
  ```python
  In [21]: mydict={'k1':'v1',(3,4):cos,'nest':{1:2}}
  In [22]: mydict.keys(), mydict.values()
  Out[22]: (['k1', 'nested', (3, 4)], ['v1', {1: 2}, <built-in function cos>])
  In [23]: mydict[(3,4)](pi)
  Out[23]: -1.0
  ```

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

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

- Perfect for building complex, sparse data structures

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

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

```python
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
```
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.
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.
Outline

1. Scientific Computing
   - Traditional approaches
   - Python?

2. Interlude: Python in the real world
   - PMV: molecular structures
   - EEG analysis for epilepsy
   - MayaVi: data visualization

3. Python and Scientific Computing
   - Basic features
   - A better development model

4. PDEs, Green’s Functions and Multiwavelets
   - The ‘curse of dimensionality’
   - Multiresolution analysis
   - Poisson’s equation
Motivation: the ‘curse of dimensionality’

A simple observation: numerical algorithms ($C = AB$, $y = Ax$) in $d$ physical dimensions scale like $O(k^d)$. Not good.

A typical example: Poisson’s equation (electromagnetics, gravity, . . . ):

$$\nabla^2 \phi(r) = \rho(r)$$

A Green’s function solution (free space, $d = 3$, ignore constants):

$$\phi(r) = \int G(r - r')\rho(r')d^3r' = \int \frac{1}{|r - r'|}\rho(r')d^3r'.$$

If we discretize using a global basis, this becomes:

$$\phi_{ijk} = \sum_{i'j'k'=1}^N G_{i'i',jj',kk'} \rho_{i'j'k'} \rightarrow O(N^6)$$

Applying an integral kernel is a matrix-vector multiplication.
Motivation: the ‘curse of dimensionality’

**A simple observation**: numerical algorithms \((C = AB, y = Ax)\) in \(d\) physical dimensions scale like \(O(k^d)\). Not good.

**A typical example**: Poisson’s equation (electromagnetics, gravity, . . .):

\[
\nabla^2 \phi(r) = \rho(r)
\]

A Green’s function solution (free space, \(d = 3\), ignore constants):

\[
\phi(r) = \int G(r - r') \rho(r') d^3r' = \int \frac{1}{|r - r'|} \rho(r') d^3r'.
\]

If we discretize using a global basis, this becomes:

\[
\phi_{ijk} = \sum_{i',j',k'=1}^{N} G_{ii',jj',kk'} \rho_{i',j',k'} \rightarrow O(N^6)
\]

Applying an integral kernel is a matrix-vector multiplication.
Motivation: the ‘curse of dimensionality’

A simple observation: numerical algorithms \((C = AB, y = Ax)\) in \(d\) physical dimensions scale like \(O(k^d)\). Not good.

A typical example: Poisson’s equation (electromagnetics, gravity, . . .):

\[
\nabla^2 \phi(r) = \rho(r)
\]

A Green’s function solution (free space, \(d = 3\), ignore constants):

\[
\phi(r) = \int G(r - r') \rho(r') d^3r' = \int \frac{1}{|r - r'|} \rho(r') d^3r'.
\]

If we discretize using a global basis, this becomes:

\[
\phi_{ijk} = \sum_{i'j'k'}^N G_{ii',jj',kk'} \rho_{i'j'k'} \rightarrow O(N^6)
\]

Applying an integral kernel is a matrix-vector multiplication.
Can we do this efficiently for $d > 1$?

- Adaptive grid & a local basis of order $p$: $O(N_{blk}p^6)$ [may be better than $O(N^6)$]

- What if we could write:

$$G_{i'i',jj',kk'} = \sum_{m=1}^{M} w_m F_{i'i'}^m F_{jj'}^m F_{kk'}^m.$$

- We could separate the different dimensions:

$$\phi_{ijk} = \sum_{m=1}^{M} w_m \sum_{i'} F_{i'i'}^m \sum_{j'} F_{jj'}^m \sum_{k'} F_{kk'}^m \rho_{i'j'k'} \rightarrow O(N_{blk}p^4)$$

The problem partially factorizes.
Can we do this efficiently for $d > 1$?

- Adaptive grid & a local basis of order $p$: $O(N_{blk}p^6)$ [may be better than $O(N^6)$]
- What if we could write:

$$G_{ii',jj',kk'} = \sum_{m=1}^{M} w_m F_{ii'}^m F_{jj'}^m F_{kk'}^m.$$ 

- We could separate the different dimensions:

$$\phi_{ijk} = \sum_{m=1}^{M} w_m \sum_{i'} F_{ii'}^m \sum_{j'} F_{jj'}^m \sum_{k'} F_{kk'}^m \rho_{i''j''k''} \rightarrow O(N_{blk}p^4)$$

The problem partially factorizes.
Can we do this efficiently for \( d > 1 \)?

- Adaptive grid & a local basis of order \( p \): \( O(N_{\text{blk}} p^6) \) [may be better than \( O(N^6) \)]

- What if we could write:

\[
G_{i'i',jj',kk'} = \sum_{m=1}^{M} w_m F_{i'i'}^m F_{jj'}^m F_{kk'}^m.
\]

- We could separate the different dimensions:

\[
\phi_{ijk} = \sum_{m=1}^{M} w_m \sum_{i'} F_{i'i'}^m \sum_{j'} F_{jj'}^m \sum_{k'} F_{kk'}^m \rho_{i'j'k'} \rightarrow O(N_{\text{blk}} p^4)
\]

The problem partially factorizes.
Key mathematical ideas:

1. Multiresolution analysis (wavelets): sparse matrix representations for a large class of kernels.
2. Separated representations: reduction of dimensionality cost.

Python implementation: ideal language for these algorithms.

- Dictionaries for sparse data structures.
- Object orientation: code which naturally follows the mathematics.
- Good numerical performance (Numeric, C, C++ and FORTRAN).

Group effort:

1. Gregory Beylkin, Lucas Monzón - CU Boulder
2. Martin Mohlenkamp - Ohio University
3. Robert Harrison, George Fann, Takeshi Yanai, Zhengting Gan - ORNL
4. Vani Cheruvu - NCAR
5. Robert Cramer - Raytheon
We can!

- **Key mathematical ideas:**
  1. **Multiresolution analysis (wavelets):** sparse matrix representations for a large class of kernels.
  2. **Separated representations:** reduction of dimensionality cost.

- **Python implementation:** ideal language for these algorithms.
  - Dictionaries for sparse data structures.
  - Object orientation: code which naturally follows the mathematics.
  - Good numerical performance (Numeric, C, C++ and FORTRAN).

- **Group effort:**
  1. Gregory Beylkin, Lucas Monzón - CU Boulder
  2. Martin Mohlenkamp - Ohio University
  3. Robert Harrison, George Fann, Takeshi Yanai, Zhengting Gan - ORNL
  4. Vani Cheruvu - NCAR
  5. Robert Cramer - Raytheon
We can!

**Key mathematical ideas:**

1. **Multiresolution analysis (wavelets):** sparse matrix representations for a large class of kernels.
2. **Separated representations:** reduction of dimensionality cost.

**Python implementation:** ideal language for these algorithms.

- Dictionaries for sparse data structures.
- Object orientation: code which naturally follows the mathematics.
- Good numerical performance (Numeric, C, C++ and FORTRAN).

**Group effort:**

1. Gregory Beylkin, Lucas Monzón - CU Boulder
2. Martin Mohlenkamp - Ohio University
3. Robert Harrison, George Fann, Takeshi Yanai, Zhengting Gan - ORNL
4. Vani Cheruvu - NCAR
5. Robert Cramer - Raytheon
Multiresolution analysis, the bare basics

Imagine a simple signal $f(t)$ you want to study:

![Graph of a signal $f(t)$](image-url)
Multiresolution analysis, the bare basics

Imagine a simple signal $f(t)$ you want to study:

At each scale $n$, divide the unit interval $[0, 1]$ into $2^n$ binary subintervals:
And compute:

- **Average** \((s)\) values of function at level \(n\): space \(V_n\).
- **Differences** \((d)\) between successive levels: space \(W_n = V_{n+1} - V_n\).
And compute:

- **Average** \( s \) values of function at level \( n \) : space \( V_n \).
- **Differences** \( d \) between successive levels: space \( W_n = V_{n+1} - V_n \).

\( f(t) \) can be studied (compressed, denoised, ...) from \( \{ V_0, W_0, W_1, \ldots \} \):

The \( d \) coefficients are **small** and **localized** around changes.
We’ll use **multiwavelets**: \( p = N_{nod} \) coefficients per subinterval.
Adaptive subdivision of the unit interval in $\mathbb{R}^d$

Simple recursive subdivision produces a d-binary tree on the unit interval, with $N_{nod}^d$ coefficient blocks on the leaves:
Decomposing functions with prescribed accuracy

A simple 1-d example, using $N_{\text{nod}} = 8$, $\varepsilon = 10^{-4}$ for

$$f(x) = \sin(16\pi x^6)$$
Resolving sharp discontinuities (a 2d example)

Consider a discontinuity along a circle (typical charge density for a 2d electrostatics problem).

\( N_{\text{nod}} = 8, \epsilon = 1.0e-02, N_{\text{blocks}} = 1276 \)
Operators (1): sparse representations

Again, project the operator on each scale and use differences:

\[
T^n = T^0 + (T^1 - T^0) + (T^2 - T^1) + \ldots = T^0 + \sum_{j=1}^{n} D^j.
\]
We can approximate a wide class of functions as sums of Gaussians:

\[
\frac{1}{\| r - r' \|} \approx \sum_{m=1}^{M} w_m e^{-\tau_m \| r - r' \|_2^2},
\]

with controlled accuracy \( \varepsilon \) over a wide dynamic range \([M \approx O(-\log \varepsilon)]\):

This gives us the factorization we wanted for our kernel:

\[
G_{ii',jj',kk'} = \sum_{m=1}^{M} w_m F_{ii'}^m F_{jj'}^m F_{kk'}^m.
\]
The 2-scale differences cancel most terms

Norm estimates as a function of the separation index, for various offsets:

We can then apply only a part of the separation series:

\[ G_{ii'', jj'', kk''} = \sum_{m=1}^{M} w_{m} F_{ii}^{m} F_{jj}^{m} F_{kk}^{m} \]  
\[ \underset{\text{In 2-scale rel.}}{\longrightarrow} \sum_{m=m_1}^{m_2} w_{m} F_{ii}^{m} F_{jj}^{m} F_{kk}^{m}. \]
Poisson’s equation: an example

Let’s solve Poisson’s equation for a simple case with a known solution, a Gaussian:

$$\rho(r; \alpha) = (6\alpha - 4\alpha^2 r^2) e^{-\alpha r^2} \implies \phi(r; \alpha) = e^{-\alpha r^2}.$$  

[DEMO]

Some comments:

- More optimizations coming down the pipe (structural)
- We’ll probably rewrite one routine in C (~25 lines of Python in 3d)
- Compares well with multigrid codes for high accuracy.
- We think it compares well with FMM codes (detailed tests coming).
- Very important:
  - This method is not kernel-specific.
  - The code only knows about the Gaussian expansion for the kernel.
- R. Harrison et al. (ORNL): Schrödinger’s equation (Lippman-Schwinger formulation).
Let’s solve Poisson’s equation for a simple case with a known solution, a Gaussian:

\[ \rho(r; \alpha) = (6\alpha - 4\alpha^2 r^2) e^{-\alpha r^2} \implies \phi(r; \alpha) = e^{-\alpha r^2}. \]

Some comments:

- More optimizations coming down the pipe (structural)
- We’ll probably rewrite one routine in C (\(\sim 25\) lines of Python in 3d)
- Compares well with multigrid codes for high accuracy.
- We think it compares well with FMM codes (detailed tests coming).
- Very important:
  - This method is not kernel-specific.
  - The code only knows about the Gaussian expansion for the kernel.
- R. Harrison et. al. (ORNL): Schrödinger’s equation (Lippman-Schwinger formulation).
Poisson’s equation: an example

Let’s solve Poisson’s equation for a simple case with a known solution, a Gaussian:

\[ \rho(r; \alpha) = (6\alpha - 4\alpha^2 r^2) e^{-\alpha r^2} \implies \phi(r; \alpha) = e^{-\alpha r^2}. \]

[DEMO]

Some comments:

- More optimizations coming down the pipe (structural)
- We’ll probably rewrite one routine in C (~25 lines of Python in 3d)
- Compares well with multigrid codes for high accuracy.
- We think it compares well with FMM codes (detailed tests coming).
- Very important:
  - This method is not kernel-specific.
  - The code only knows about the Gaussian expansion for the kernel.
- R. Harrison et. al. (ORNL): Schrödinger’s equation (Lippman-Schwinger formulation).
Poisson’s equation: an example

Let’s solve Poisson’s equation for a simple case with a known solution, a Gaussian:

\[ \rho(r; \alpha) = (6\alpha - 4\alpha^2 r^2)e^{-\alpha r^2} \implies \phi(r; \alpha) = e^{-\alpha r^2}. \]

[DEMO]

Some comments:

- More optimizations coming down the pipe (structural)
- We’ll probably rewrite one routine in C (\(\sim\)25 lines of Python in 3d)
- Compares well with multigrid codes for high accuracy.
- We think it compares well with FMM codes (detailed tests coming).
- Very important:
  - This method is not kernel-specific.
  - The code only knows about the Gaussian expansion for the kernel.
- R. Harrison et. al. (ORNL): Schrödinger’s equation (Lippman-Schwinger formulation).
Poisson’s equation: an example

Let’s solve Poisson’s equation for a simple case with a known solution, a Gaussian:

\[ \rho(r; \alpha) = (6\alpha - 4\alpha^2 r^2)e^{-\alpha r^2} \implies \phi(r; \alpha) = e^{-\alpha r^2}. \]

[DEMO]

Some comments:

- More optimizations coming down the pipe (structural)
- We’ll probably rewrite one routine in C (\(\sim\)25 lines of Python in 3d)
- Compares well with multigrid codes for high accuracy.
- We think it compares well with FMM codes (detailed tests coming).

Very important:

- This method is not kernel-specific.
- The code only knows about the Gaussian expansion for the kernel.

R. Harrison et. al. (ORNL): Schrödinger’s equation (Lippman-Schwinger formulation).
Poisson’s equation: an example

Let’s solve Poisson’s equation for a simple case with a known solution, a Gaussian:

\[ \rho(r; \alpha) = (6\alpha - 4\alpha^2 r^2) e^{-\alpha r^2} \Rightarrow \phi(r; \alpha) = e^{-\alpha r^2}. \]

[DEMO]

Some comments:

- More optimizations coming down the pipe (structural)
- We’ll probably rewrite one routine in C (~25 lines of Python in 3d)
- Compares well with multigrid codes for high accuracy.
- We think it compares well with FMM codes (detailed tests coming).
- Very important:
  - This method is not kernel-specific.
  - The code only knows about the Gaussian expansion for the kernel.

  R. Harrison et. al. (ORNL): Schrödinger’s equation (Lippman-Schwinger formulation).
Let’s solve Poisson’s equation for a simple case with a known solution, a Gaussian:

\[ \rho(r; \alpha) = (6\alpha - 4\alpha^2 r^2) e^{-\alpha r^2} \implies \phi(r; \alpha) = e^{-\alpha r^2}. \]

Some comments:

- More optimizations coming down the pipe (structural)
- We’ll probably rewrite one routine in C (≈25 lines of Python in 3d)
- Compares well with multigrid codes for high accuracy.
- We think it compares well with FMM codes (detailed tests coming).
- Very important:
  - This method is not kernel-specific.
  - The code only knows about the Gaussian expansion for the kernel.
- R. Harrison et. al. (ORNL): Schrödinger’s equation (Lippman-Schwinger formulation).
Python’s benefits

- Rich, flexible, well implemented native types: Dicts, Numeric arrays, Sets, Lists/Tuples.
- F2PY, weave.inline(): painless access to Fortran and C/C++.
- Graphics tools:
  - Gnuplot/Matplotlib
  - MayaVi
  - PyX
- Interactive work: IPython.
- Object oriented (fits the problem domain like a glove).
Rich, flexible, well implemented native types: Dicts, Numeric arrays, Sets, Lists/Tuples.

F2PY, `weave.inline()`: painless access to Fortran and C/C++.

Graphics tools:
- Gnuplot/Matplotlib
- MayaVi
- PyX

Interactive work: IPython.

Object oriented (fits the problem domain like a glove).
Python’s benefits

- Rich, flexible, well implemented native types: Dicts, Numeric arrays, Sets, Lists/Tuples.
- F2PY, weave.inline(): painless access to Fortran and C/C++.
- Graphics tools:
  - Gnuplot/Matplotlib
  - MayaVi
  - PyX
- Interactive work: IPython.
- Object oriented (fits the problem domain like a glove).
Python’s benefits

- Rich, flexible, well implemented native types: Dicts, Numeric arrays, Sets, Lists/Tuples.
- F2PY, weave.inline(): painless access to Fortran and C/C++.
- Graphics tools:
  - Gnuplot/Matplotlib
  - MayaVi
  - PyX
- Interactive work: IPython.
- Object oriented (fits the problem domain like a glove).
Python’s benefits

- Rich, flexible, well implemented native types: Dicts, Numeric arrays, Sets, Lists/Tuples.
- F2PY, weave.inline(): painless access to Fortran and C/C++.
- Graphics tools:
  - Gnuplot/Matplotlib
  - MayaVi
  - PyX
- Interactive work: IPython.
- Object oriented (fits the problem domain like a glove).
Function decomposition algorithm:

- 1-6 d in a single class (very little dimension-dependent code)
- Trivial implementation
- Performed better than the Fortran (3d only) it replaced: some algorithmic optimizations were much easier to see/implement in Python

Operator code:

- 1d first, then jumped to 3d (skipped 2d).
- We had to backtrack to 2d to compare with FMM literature.
Function decomposition algorithm:

- 1-6 d in a single class (very little dimension-dependent code)
- Trivial implementation
- Performed better than the Fortran (3d only) it replaced: some algorithmic optimizations were much easier to see/implement in Python

Operator code:

- 1d first, then jumped to 3d (skipped 2d).
- We had to backtrack to 2d to compare with FMM literature.
Code Structure

OO library:

- Implementing a new Operator requires overriding a single method, the kernel.
- An OO design naturally fits the mathematical structure of the problem.
- “Smart” objects:
  - `.info*()`, `.plot*()` methods.
  - Tailored for interactive work: “live” debugging (IPython’s `%run` and judicious use of `reload()`)
Implementing a new Operator requires overriding a single method, the kernel.

An OO design naturally fits the mathematical structure of the problem.

“Smart” objects:
- `.info*()`, `.plot*()` methods.
- Tailored for interactive work: “live” debugging (IPython’s `%run` and judicious use of `reload()`).
Code Structure

OO library:

- Implementing a new Operator requires overriding a single method, the kernel.
- An OO design naturally fits the mathematical structure of the problem.
- “Smart” objects:
  - `.info*()`, `.plot*()` methods.
  - Tailored for interactive work: “live” debugging (IPython’s `%run` and judicious use of `reload()`)

![Diagram of OperatorBase and its subclasses]

- `OperatorBase`
- `Operator_1d` with subclasses `Cotangent`, `Identity`, `Poisson`
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.

PDEs via Green’s functions

- A new fast, adaptive algorithm for applying integral kernels via multiwavelets with guaranteed precision.
- Implementation exists for $d = 1, 2, 3$.
- Core algorithm is kernel-independent, already implemented for several physically important kernels.

Outlook

- Python: the SciPy project is moving these ideas forward.
- PDEs: algorithmic improvements, boundary conditions, applications, ...
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.

**PDEs via Green’s functions**

- A new fast, adaptive algorithm for applying integral kernels via multiwavelets with guaranteed precision.
- Implementation exists for $d = 1, 2, 3$.
- Core algorithm is kernel-independent, already implemented for several physically important kernels.

**Outlook**

- **Python**: the SciPy project is moving these ideas forward.
- **PDEs**: algorithmic improvements, boundary conditions, applications, ...
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.

- **PDEs via Green’s functions**
  - A new fast, adaptive algorithm for applying integral kernels via multiwavelets with guaranteed precision.
  - Implementation exists for $d = 1, 2, 3$.
  - Core algorithm is kernel-independent, already implemented for several physically important kernels.

- **Outlook**
  - **Python**: the SciPy project is moving these ideas forward.
  - **PDEs**: algorithmic improvements, boundary conditions, applications, ...
Some useful URLs

- A collection of links on Python and Scientific Computing:
  http://amath.colorado.edu/faculty/fperez/python/scicomp

- SciPy: http://www.scipy.org

- IPython, an improved interactive shell:
  http://ipython.scipy.org

- Matplotlib, 2d plotting with Matlab syntax:
  http://matplotlib.sourceforge.net

- MayaVi, 3d data visualization:
  http://mayavi.sourceforge.net

- PyX, programmatic PostScript generation (with \LaTeX support):
  http://pyx.sourceforge.net


- SWIG (C/C++ for Python): http://swig.org
URLs for projects mentioned earlier

- VPython, real-time 3D: http://vpython.org
- UCSF Chimera - interactive molecular graphics: http://www.cgl.ucsf.edu/chimera
- PyRAF - Hubble Space Telescope interface to IRAF: http://www.stsci.edu/resources/software_hardware/pyraf
- BioPython - http://biopython.org
- VisionEgg - vision research experiments (OpenGL): http://www.visionegg.org
- Natural Language Toolkit: http://nltk.sourceforge.net
- Orange - component-based data mining: http://www.ailab.si/orange
For Further Reading

Beylkin, Coifman, Rokhlin

Alpert

Alpert, Beylkin, Gines, Vozovoi
J. Comp. Phys. 182, 149-190 (2002)

Beylkin and Mohlenkamp

Beylkin and Monzón
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
  ```
Data types (cont)

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

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

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

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

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

```python
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
```
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
```
Functions are first class objects

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

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

```python
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)

```python
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]])
```
Functions are first class objects

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

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

```python
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)

```python
def qsort(L):
    if len(L) <= 1: return L
    return qsort([lt for lt in L[1:] if lt < L[0]]) + \n    [L[0]] + qsort([ge for ge in L[1:] if ge >= L[0]])
```
Functions are first class objects

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

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

```python
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)

```python
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]])
```