IPython: A System for Interactive Scientific Computing

Python offers basic facilities for interactive work and a comprehensive library on top of which more sophisticated systems can be built. The IPython project provides an enhanced interactive environment that includes, among other features, support for data visualization and facilities for distributed and parallel computation.

The backbone of scientific computing is mostly a collection of high-performance code written in Fortran, C, and C++ that typically runs in batch mode on large systems, clusters, and supercomputers. However, over the past decade, high-level environments that integrate easy-to-use interpreted languages, comprehensive numerical libraries, and visualization facilities have become extremely popular in this field. As hardware becomes faster, the critical bottleneck in scientific computing isn't always the computer's processing time; the scientist's time is also a consideration. For this reason, systems that allow rapid algorithmic exploration, data analysis, and visualization have become a staple of daily scientific work. The Interactive Data Language (IDL) and Matlab (for numerical work), and Mathematica and Maple (for work that includes symbolic manipulation) are well-known commercial environments of this kind. GNU Data Language, Octave, Maxima and Sage provide their open source counterparts.

All these systems offer an interactive command line in which code can be run immediately, without having to go through the traditional edit/compile/execute cycle. This flexible style matches well the spirit of computing in a scientific context, in which determining what computations must be performed next often requires significant work. An interactive environment lets scientists look at data, test new ideas, combine algorithmic approaches, and evaluate their outcomes directly. This process might lead to a final result, or it might clarify how to build a more static, large-scale production code.

As this article shows, Python (www.python.org) is an excellent tool for such a workflow.[1] The IPython project (http://ipython.scipy.org) aims to not only provide a greatly enhanced Python shell but also facilities for interactive distributed and parallel computing, as well as a comprehensive set of tools for building special-purpose interactive environments for scientific computing.

Python: An Open and General-Purpose Environment

The fragment in Figure 1 shows the default interactive Python shell, including a computation with long integers (whose size is limited only by the available memory) and one using the built-in complex numbers, where the literal `1j` represents $i = \sqrt{-1}$.

1521-9615/07/$25.00 © 2007 IEEE
Copublished by the IEEE CS and the AIP

Fernando Pérez
University of Colorado at Boulder
Brian E. Granger
Tech-X Corporation

```
$ python # $ represents the system prompt
Python 2.4.3 (Apr 27 2006, 14:43:58)
[GCC 4.0.3 (Ubuntu 4.0.3-1ubuntu5)] on linux2
Type "help", "copyright", "credits" or "license"
for more information.
>>> print "This is the Python shell."
This is the Python shell.
>>> 2**45+1 # long integers are built-in
35184372088833L
>>> import cmath # default complex math library
>>> cmath.exp(-1j*cmath.pi)
(-1-1.2246063538223773e-16j)
```

Figure 1. Default interactive Python shell. In the two computations shown—one with long integers and one using the built-in complex numbers—the literal `1j` represents $i = \sqrt{-1}$.

This shell allows for some customization and access to help and documentation, but overall it's a fairly basic environment.

However, what Python lacks in the sophistication of its default shell, it makes up for by being a general-purpose programming language with access to a large set of libraries with additional capabilities. Python's standard library includes modules for regular expression processing, low-level networking, XML parsing, Web services, object serialization, and more. In addition, hundreds of third-party Python modules let users do everything from work with Hierarchical Data Format 5 (HDF5) files to write graphical applications. These diverse libraries make it possible to build sophisticated interactive environments in Python without having to implement everything from scratch.

IPython

Since late 2001, the IPython project has provided tools to extend Python's interactive capabilities beyond those shipped by default with the language, and it continues to be developed as a base layer for new interactive environments. IPython is freely available under the terms of the BSD license and runs under Linux and other Unix-type operating systems, Apple OS X, and Microsoft Windows.

We won't discuss IPython's features in detail here—it ships with a comprehensive user manual (also accessible on its Web site). Instead, we highlight some of the basic ideas behind its design and how they enable efficient interactive scientific computing. We encourage interested readers to visit the Web site and participate on the project's mailing lists.

One of us (Fernando Pérez) started IPython as a merger of some personal enhancements to the basic interactive Python shell with two existing open source projects (both now defunct and subsumed into IPython):

- LazyPython, developed by Nathan Gray at Caltech, and
- Interactive Python Prompt (IPP) by Janko Hauser at the University of Kiel's Institute of Marine Research.

After an initial development period as a mostly single-author project, IPython has attracted a growing group of contributors. Today, Ville Vainio and other collaborators maintain the stable official branch, while we're developing a next-generation system.

Since IPython's beginning, we've tried to provide the best possible interactive environment for everyday computing tasks, whether the actual work was scientific or not. With this goal in mind, we've freely mixed new ideas with existing ones from Unix system shells and environments such as Mathematica and IDL.

Features of a Good Interactive Computing Environment

In addition to providing direct access to the underlying language (in our case, Python), we consider a few basic principles to be the minimum requirements for a productive interactive computing system.

Access to all session state. When working interactively, scientists commonly perform hundreds of computations in sequence and often might need to reuse a previous result. The standard Python shell remembers the very last output and stores it into a variable named "_" (a single underscore), but each new result overwrites this variable. IPython stores a session's inputs and outputs into a pair of numbered tables called `In` and `Out`. All outputs are also accessible as `_N`, where `N` is the number of results (you can also save a session's inputs and outputs to a log file). Figure 2 shows the use of previous results in an IPython session. Because keeping a very large set of previous results can potentially lead to memory exhaustion, IPython lets users limit how many results are kept. Users can also manually delete individual references using the standard Python `del` keyword.

A control system. It's important to have a secondary control mechanism that is reasonably orthogonal

to the underlying language being executed (and independent of any variables or keywords in the language). Even programming languages as compact as Python have a syntax that requires parentheses, brackets, and so on, and thus aren't the most convenient for interactive control systems.

IPython offers a set of control commands (or *magic commands*, as inherited from IPP) designed to improve Python's usability in an interactive context. The traditional Unix shell largely inspires the syntax for these magic commands, with white space used as a separator and dashes indicating options. This system is accessible to the user, who can extend it with new commands as desired.

The fragment in Figure 3 shows how to activate IPython's logging system to save the session to a named file, requesting that the output is logged and every entry is time stamped. IPython automatically interprets the `logstart` name as a call to a magic command because no Python variable with that name currently exists. If there were such a variable, typing `%logstart` would disambiguate the names.

Operating system access. Many computing tasks involve working with the underlying operating system (reading data, looking for code to execute, loading other programs, and so on). IPython lets users create their own aliases for common system tasks, navigate the file system with familiar commands such as `cd` and `ls`, and prefix any command with `!` for direct execution by the underlying OS. Although these are fairly simple features, in practice they help maintain a fluid work experience—for example, they let users type standard Python code for programming tasks and perform common OS-level actions with a familiar Unix-like syntax. IPython goes beyond this, letting users call system commands with values computed from Python variables. These features have led some users (especially under Windows, a platform with a very primitive system shell) to use IPython as their default shell for everyday work.

Figure 4 shows how to perform the simple task of normalizing the names of a few files to a different convention.

Dynamic introspection and help. One benefit of working interactively is being able to directly manipulate code and objects as they exist in the runtime environment. Python offers an interactive help system and exposes a wide array of introspective capabilities as a standard module (`inspect.py`) that provides functions for exploring various types of objects in the language.

```
$ ipython
Python 2.4.3 (Apr 27 2006, 14:43:58)
Type "copyright", "credits" or "license" for more
information.

IPython 0.7.3 -- An enhanced Interactive Python.
?         -> Introduction to IPython features.
%magic    -> Information about IPython magic %
functions.
Help      -> Python help system.
object?   -> Details about object. ?object also
works, ?? prints more.
In [1]:2**45+1
Out[1]:35184372088833L
In [2]:import cmath
In [3]:cmath.exp(-1j*cmath.pi)
Out[3]:(-1-1.2246063538223773e-16j)
# The last result is always stored as '_'
In [4]:_ ** 2
Out[4]:(1+2.4492127076447545e-16j)
# And all results are stored as N, where _N is
their number:
In [5]:_3+_4
Out[5]:1.2246063538223773e-16j
```

Figure 2. The use of previous results in an IPython session. In IPython, all outputs are also accessible as `_N`, where `N` is the number of results.

```
In [2]: logstart -o -t ipsession.log
Activating auto-logging. Current session state
plus future input saved.
Filename               : ipsession.log
Mode                   : backup
Output logging         : True
Raw input log          : False
Timestamping           : True
State                  : active
```

Figure 3. Activating IPython's logging system to save the session to a named file. IPython interprets the `logstart` name as a call to a control command (or *magic* command).

IPython offers access to Python's help system, the ability to complete any object's names and attributes with the Tab key, and a system to query an object for internal details, including source code,

```
In [36]: ls
tt0.dat tt1.DAT tt2.dat tt3.DAT
# 'var = !cmd' captures a system command into a
Python variable:
In [37]: files = !ls
==
['tt0.dat', 'tt1.DAT', 'tt2.dat', 'tt3.DAT']
# Rename the files, using uniform case and 3-digit
numbers:
In [38]: for i, name in enumerate(files):
 ....:    newname = 'time%03d.dat' % i
 ....:    !mv $name $newname
 ....:
In [39]: ls
time000.dat time001.dat time002.dat time003.dat
```

Figure 4. Normalizing file names to a different convention. These code fragments show how IPython allows users to combine normal Python syntax with direct system calls (prefixed with the "!" character). In such calls, Python variables can be expanded by prefixing them with "$."

```
In [1]:from universe import DeepThought
In [2]:DeepThought. # Hit the Tab key here
       DeepThought._doc_ DeepThought.answer
       DeepThought.question
       DeepThought._module_ DeepThought.name
In [2]:DeepThought??
Type:          classobj
String Form:   universe.DeepThought
Namespace:     Interactive
File:          /tmp/universe.py
Source:
class DeepThought:
       name = "Deep Thought"
       question = None
       def answer(self):
         """Return the Answer to The Ultimate
Question Of Life, the Universe and Everything"""
         return 42
```

Figure 5. Information returned by IPython after querying an object called `DeepThought` from a module called `universe`. When the user hits the Tab key (line 2), IPython lists all attributes defined for `DeepThought`. For the sequence `DeepThought??`, IPython finds as much information about the object as it can, including its source code.

by typing the object's name and one or two "?" (two for extra details). These features are useful when developing code, exploring a problem, or using an unfamiliar library because direct experimentation with the system can help produce working code that the user can then copy into an editor as part of a larger program.

Figure 5 shows the information returned by IPython after querying an object called `DeepThought` from a module called `universe`. In line 2, we've hit the Tab key, so IPython completes a list of all the attributes defined for `DeepThought`. Then, for the sequence `DeepThought??`, IPython tries to find as much information about the object as it can, including its entire source code.

Access to program execution. Although typing code interactively is convenient, large programs are written in text editors for significant computations. IPython's `%run` magic command lets users run any Python file within the IPython session as if they had typed it interactively. Upon completion, the program results update the interactive session, so the user can further explore any quantity computed by the program, plot it, and so on. The `%run` command has several options to assist in debugging, profiling, and more. It's probably the most commonly used magic function in a typical workflow: you use a text editor for significant editing while code is executed (using `run`) in the IPython session for debugging and results analysis. Typing `run?` provides full details about the `run` command.

Figure 6 compares IPython to the default Python shell when running a program that contains errors. IPython provides detailed exception tracebacks with information about variable values, and can activate a debugger (indicated by the `ipdb>` prompt), from which a user can perform postmortem analysis of the crashed code from its in-memory state, walk up the call stack, print variables, and so on. This mechanism saves time during development, because the user doesn't need to reload libraries used by a program for each new test. It also lets the user perform expensive initialization steps only once, keeping them in memory while the user explores other parts of a problem by making changes to code and running it repeatedly.

A Base Layer for Interactive Environments

In addition to these minimal requirements, IPython exposes its major components to the user for modification and customization, making it a

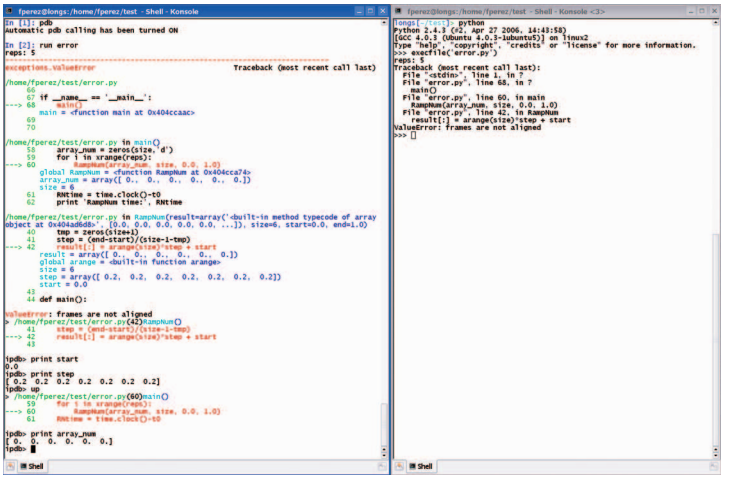
flexible and open platform. Other scientific computing projects have used IPython's features to build custom interactive environments. A user can declare these customizations in a plaintext file—an *IPython profile*—and load them using the `-profile` flag at startup time.

Input syntax processing. Underlying IPython is a running Python interpreter, so ultimately all code executed by IPython must be valid Python code. However, in some situations the user might want to allow other forms of input that aren't necessarily Python. Such uses can range from simple transformations for input convenience to supporting a legacy system with its own syntax within the IPython-based environment.

As a simple example, IPython ships with a physics profile, which preloads physical unit support from the ScientificPython library (http://sourcesup.cru.fr/projects/scientific-py), and installs a special input filter. This filter recognizes text sequences that appear to be quantities with units and generates the underlying Python code to define an object with units, without the user having to type out the more verbose syntax, as Figure 7 shows.

IPython exposes the input filtering system, which users can customize to define arbitrary input transformations that might suit their problem domains. For example, the Software for Algebra and Geometry Experimentation (Sage)[2] project uses an input filter to transform numerical quantities into exact integers, rationals, and arbitrary precision floats instead of Python's normal numerical types. (See the "Projects Using IPython" sidebar for a description of this and other examples.)

Error handling. A common need in interactive environments is to process certain errors in a special manner. IPython offers three exception handlers that treat errors uniformly, differing only in the amount of detail they provide. A custom environment might want to handle internal errors, or errors related to certain special objects, differently from other normal Python errors. IPython lets users register exception handlers that will fire when an exception of their registered type is raised. Python's uniform and object-oriented approach to errors greatly facilitates this feature's implementation: because all exceptions are classes, users can register handlers based on a point in the class hierarchy that will handle any exception that inherits from the registered class. The PyRAF interactive environment at the Space Telescope Science Institute has used this capability to handle its own internal errors separately from errors that are meaningful to the user.



(a) (b)

Figure 6. Comparison of IPython to the default Python shell. (a) IPython provides detailed error information and can automatically activate an interactive debugger to inspect the crashed code's status, print variables, navigate the stack, and so on. (b) The same error displayed in the default Python shell.

```
In [1]:   mass = 3 kg
In [2]:   g = 9.8 m/s^2
In [3]:   weight=mass*g
In [4]:   weight
Out[4]:   29.4 m*kg/s^2
# We can see the actual Python code generated by
IPython:
In [5]:   %history # %history is an IPython "magic"
command
1: mass = PhysicalQuantityInteractive(3, 'kg')
2: g = PhysicalQuantityInteractive(9.8, 'm/s**2')
3: weight=mass*g
4: weight
```

Figure 7. Code using IPython's physics profile and input filter. The filter recognizes text sequences that appear to be quantities with units and generates the underlying Python code to define an object with units.

Tab completion. Tab completion is a simple but useful feature in an interactive environment because the system completes not only on Python variables but also on keywords, aliases, magic commands, files, and directories. IPython lets users register new completers to explore certain objects.

Projects Using IPython

Several scientific projects have exploited IPython as a platform rather than as an end-user application. Although the vast majority of IPython users do little customization beyond setting a few personal options, these projects show that there is a real use case for open, customizable interactive environments in scientific computing:

- Sage (http://modular.math.washington.edu/sage), a system for mathematical research and teaching with a focus on algebra, geometry, and number theory, uses IPython for its interactive terminal-based interface.
- The Space Telescope Science Institute's PyRAF environment (www.stsci.edu/resources/software_hardware/pyraf) uses IPython for astronomical image analysis. PyRAF provides an IPython-based shell for interactive work with several special-purpose customizations. We made numerous enhancements to IPython based on requests and suggestions from the PyRAF team.
- The National Radio Astronomy Observatory's Common Astronomy Software Applications (CASA, http://casa.nrao.edu) uses IPython in its interactive shell.
- The Ganga system (http://ganga.web.cern.ch/ganga/), developed at the European Center for Nuclear Research (CERN) for grid job control for the large hadron collider beauty experiment (LHCb) and Atlas experiments, uses IPython for its command-line interface (CLIP).
- The PyMAD project (http://ipython.scipy.org/moin/PyMAD) uses IPython to control a neutron spectrometer at CEA-Grenoble and the Institute Laue Langevin in France.
- The Pymerase project (http://pymerase.sourceforge.net) for microarray gene expression databases exposes an IPython shell in its interactive iPymerase mode.

Based on the lessons learned from this usage, we're currently restructuring IPython to allow interactive parallel and distributed computing, to build better user interfaces, and to provide more flexible and powerful components for other projects to build on. We hope that if more projects are developed on top of such a common platform, all users will benefit from the familiarity of having a well-known base layer on top of which their specific projects add custom behavior.

The PyMAD project at the neutron scattering facility of the Institute Laue Langevin in Grenoble, France, uses this feature for interactive control of experimental devices. The IPython console runs on a system that connects to the neutron spectrometer over a network, but users interact with the remote system as if it were local, and Tab completion operates over the network to fetch information about remote objects for display in the user's local console.

Graphical Interface Toolkits and Plotting

Python provides excellent support for GUI toolkits. It ships by default with bindings for Tk, and third-party bindings are available for GTK, WxWidgets, Qt, and Cocoa (under Apple OS X). You can use essentially every major toolkit to write graphical applications from Python. Although few scientists look forward to GUI design, they increasingly have to write small- to medium-sized graphical applications to interface with scientific code, drive instruments, or collect data. Python lets scientists choose the toolkit that best fits their needs.

However, graphical applications are notoriously difficult to test and control from an interactive command line. In the default Python shell, if a user instantiates a Qt application, for example, the command line stops responding as soon as the Qt window appears. IPython addresses this problem by offering special startup flags that let users choose which toolkit they want to control interactively in a nonblocking manner.

This feature is necessary for one of scientists' most common tasks: interactive data plotting and visualization. Many traditional plotting libraries and programs have Python bindings or process-based interfaces, but most have various limitations for interactive use. The matplotlib project (http://matplotlib.sourceforge.net) is a sophisticated plotting library capable of producing publication-quality graphics in a variety of formats, and with full LaTeX support.[3] Matplotlib renders its plots to several back ends, the components responsible for generating the actual figure. Some back ends (such as for PostScript, PDF, and Scalable Vector Graphics) are purely disk-based and meant to generate files; others are meant for display in a window. Matplotlib supports all these toolkits, letting users choose which to use via a configuration file setting. (The Scientific Programming department on p. 90 explores matplotlib in more detail.)

IPython and matplotlib developers have collaborated to enable automatic coordination between the two systems. If given the special `-pylab` startup flag, for example, IPython detects the user's matplotlib settings and automatically configures itself to enable nonblocking interactive plotting. This

provides an environment in which users can perform interactive plotting in a manner similar to Matlab or IDL but with complete flexibility in the GUI toolkit used (these programs provide their own GUI support and can't be integrated in the same process with other toolkits).

In the example in Figure 8, plots are generated from an interactive session using matplotlib. We use the special function and numerical integration routines provided by the SciPy package[4] to verify, at a few points, the standard relation for the first Bessel function

$$\mathcal{J}_0(x) = \frac{1}{\pi}\int_0^{\pi} \cos\left(x \sin\phi\right) d\phi \,.$$

The last line shows matplotlib's capabilities for array plotting with a simple 32×32 set of random numbers.

Although matplotlib's main focus is 2D plotting, several packages exist for 3D plotting and visualization in Python. The Visualization Toolkit (VTK) is a mature and sophisticated visualization library written in C++ that ships with Python bindings. Recently, developers have introduced a new set of bindings called Traits-enabled VTK (TVTK),[5] which provides seamless integration with the NumPy array objects and libraries as well as a higher-level API for application development. Figure 9 shows how to use TVTK interactively from within an IPython session. Because matplotlib has WXPython support, you can use both TVTK and matplotlib concurrently from within IPython.

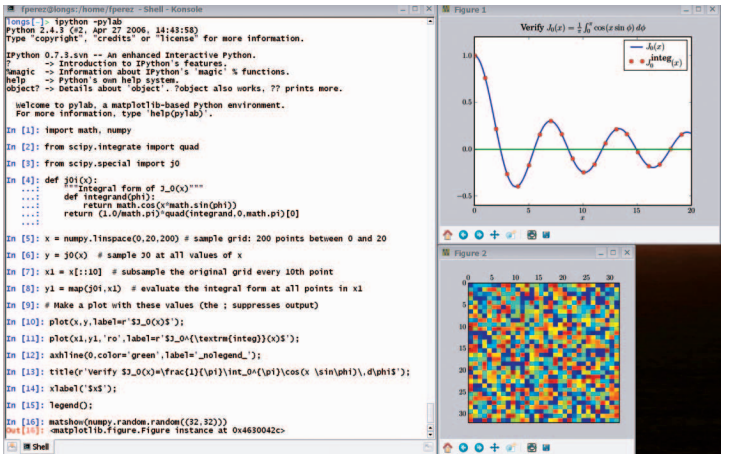


Figure 8. IPython using the `-pylab` flag to enable interactive use of the matplotlib library. Plot windows can open without blocking the interactive terminal, using any of the GUI toolkits supported by matplotlib (Tk, WxWidgets, GTK, or Qt).

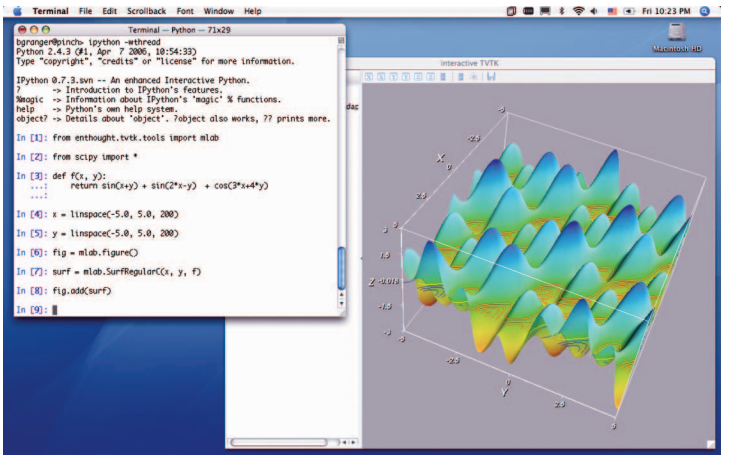


Figure 9. An IPython session showing a 3D plot done with TVTK. The GUI toolkit used is WXPython, so IPython is started with the `-wthread` flag.

Interactive Parallel and Distributed Computing

Although interactive computing environments can be extremely productive, they've traditionally had one weakness: they haven't been able to take advantage of parallel computing hardware such as multicore CPUs, clusters, and supercomputers. Thus, although scientists often begin projects using an interactive computing environment, at some point they switch to using languages such as C, C++, and Fortran when performance becomes critical and their projects call for parallelization. In recent years, several vendors have begun offering distributed computing capabilities for the major commercial technical computing systems (see the "Distributed Computing Toolkits for Commercial Systems" sidebar for some examples). These provide various levels of integration between the computational back ends and interactive front ends. An early precursor to these systems, whose model was one of full interactive access to the computational nodes, is ParGAP (www.ccs.neu.edu/home/gene/pargap.html), a parallel-enabled version of the open source package Groups, Algorithms, and Programming (GAP) for computational group theory.

In the Python world, several projects also exist that seek to add support for distributed computing. The Python-community-maintained wiki keeps a list of such efforts (http://wiki.python.org/moin/ParallelProcessing). Of particular interest to scientific users, Python has been used in parallel computing contexts both with the message-passing interface (MPI, http://sourceforge.net/projects/pympi; http://mpi4py.scipy.org)[6,7] and the Bulk Synchronous Parallel[8] models.

Distributed Computing Toolkits for Commercial Systems

Some vendors offer distributed computing capabilities for the major commercial technical computing systems:

- Matlab Distributed Computing Toolbox, www.mathworks.com/products/distribtb.
- FastDL, www.txcorp.com/products/FastDL.
- Mathematica Parallel Computing Toolkit, http://documents.wolfram.com/applications/parallel.
- Mathematica Personal Grid Edition, www.wolfram.com/products/personalgrid.
- Grid Mathematica, www.wolfram.com/products/gridmathematica.
- HPC-Grid, www.maplesoft.com/products/toolboxes/HPCgrid.
- Star-P, www.interactivesupercomputing.com.

Other projects seek to support distributed computing using Python (see http://wiki.python.org/moin/ParallelProcessing).

Building on Python's and IPython's strengths as an interactive computing system, we've begun a significant effort to add interactive parallel and distributed capabilities to IPython. More specifically, our goal is to enable users to develop, test, debug, execute, and monitor parallel and distributed applications interactively using IPython. To make this possible, we've refactored IPython to support these new features. We've deliberately built a system whose basic components make no specific assumptions about communications models, data distribution, or network protocols. The redesigned IPython consists of

- the IPython *core*, which exposes IPython's core functionality, abstracted as a Python library rather than as a terminal-based application;
- the IPython *engine*, which exposes the IPython core's functionality to other processes (either local to the same machine or remote) over a standard network connection; and
- the IPython *controller*, which is a process that exposes a clean asynchronous interface for working with a set of IPython engines.

With these basic components, specific models of distributed and parallel computing can be implemented as user-visible systems. Currently, we support two models out of the box: a load-balancing and fault-tolerant task-farming interface for coarse-grained parallelism, and a lower-level interface that gives users direct interactive access to a set of running engines. This second interface is useful for both medium- and fine-grained parallelism that uses MPI for communications between engines. Most importantly, advanced users and developers can use these components to build customized interactive parallel/distributed applications in Python. End users work with the system interactively by connecting to a controller using a Web browser, an IPython- or Python-based front end, or a traditional GUI.

Specific constraints that are relevant in scientific computing guided this design:

- It should support many different styles of parallelism, such as message passing using MPI, task farming, and shared memory.
- It should run on everything from multicore laptops to supercomputers.
- It should integrate well with existing parallel code and libraries written using C, C++, or Fortran, and MPI for communications.
- All network communications, events, and error handling should be fully asynchronous and nonblocking.
- It should support all of IPython's existing features in parallel contexts.

The architectural requirements for running IPython in a distributed manner are similar to those required for decoupling a user front end from a computational back end. Therefore, this restructuring effort also lets IPython offer new types of user interfaces for remote and distributed work, such as a Web browser-based IPython GUI and collaborative interfaces that enable multiple remote users to simultaneously access and share running computational resources and data.

The first public release of these new components was in late 2006. While it should still be considered under heavy development and subject to changes, we've already been contacted by several projects that have begun using it as a tool in production codes. Details about this work are available on the IPython Web site. CiSE

Acknowledgments

IPython wouldn't be where it is today if it weren't for its user community's contributions. Over the years, users have sent bug reports, ideas, and often major portions of new code. Some of the more prolific contributors have become codevelopers. As a Free

Software project, it is only because of such a community that it continues to improve. We thank Ville Vainio for maintaining the stable branch of the project, and Benjamin Ragan-Kelley for his continued work as a key developer of IPython's distributed and parallel computing infrastructure.

This research was partially supported by US Department of Energy grant DE-FG02-03ER25583 and DOE/Oak Ridge National Laboratory grant 4000038129 (F. Pérez) and by Tech-X Corporation (B. Granger). We thank Enthought for the hosting and infrastructure support it has provided to IPython over the years.

References

1. T.-Y.B. Yang, G. Furnish, and P.F. Dubois, "Steering Object-Oriented Scientific Computations," *Proc. Technology of Object-Oriented Languages and Systems* (TOOLS), IEEE CS Press, 1998, pp. 112–119.
2. W. Stein and D. Joyner, "SAGE: System for Algebra and Geometry Experimentation," *Comm. Computer Algebra*, vol. 39, 2005, pp. 61–64.
3. P. Barrett, J. Hunter, and P. Greenfield, "Matplotlib: A Portable Python Plotting Package," *Astronomical Data Analysis Software & Systems*, vol. 14, 2004.
4. E. Jones, T. Oliphant, and P. Peterson, "SciPy: Open Source Scientific Tools for Python," 2001; www.scipy.org.
5. P. Ramachandran, "TVTK: A Pythonic VTK," *Proc. EuroPython Conf.*, EuroPython, 2005; http://svn.enthought.com/enthought/attachment/wiki/TVTK/tvtk-paper-epc2005.pdf.
6. D.M. Beazley and P.S. Lomdahl, "Extensible Message Passing Application Development and Debugging with Python," *Proc. 11th Int'l Parallel Processing Symp.*, IEEE CS Press, 1997, pp. 650–655.
7. P. Miller, "Parallel, Distributed Scripting with Python," *Third Linux Clusters Inst. Int'l Conf. Linux Clusters: The HPC Revolution*, Lawrence Livermore Nat'l Laboratory, 2002; www.llnl.gov/tid/lof/documents/pdf/240425.pdf.
8. K. Hinsen, "High-Level Parallel Software Development with Python and BSP," *Parallel Processing Letters*, vol. 13, s2003, pp. 473–484.

Fernando Pérez *is a research associate in the Department of Applied Mathematics at the University of Colorado at Boulder. His research interests include new algorithms for solving PDEs in multiple dimensions with a focus on problems in atomic and molecular structure, the use of high-level languages for scientific computing, and new approaches to distributed and parallel problems. Pérez has a PhD in physics from the University of Colorado. Contact him at Fernando.Perez@colorado.edu.*

Brian E. Granger *is a research scientist at Tech-X. He has a background in scattering and many-body theory in the context of atomic, molecular, and optical physics. His research interests include interactive parallel and distributed computing, remote visualization, and Web-based interfaces in scientific computing. Granger has a PhD in theoretical physics from the University of Colorado. Contact him at bgranger@txcorp.com.*