Editor: Paul F. Dubois, pauldubois@home.com

EXTENDING PYTHON WITH FORTRAN

By Paul F. Dubois and T.-Y. Yang

PYTHON IS A GREAT SCRIPTING LANGUAGE. IT IS PORTABLE, FREE, AND HAS A POWERFUL NUMERICAL FACILITY, OBJECT-ORIENTED FEATURES, AND A LIBRARY OF MODULES THAT ENABLE A HUGE VARIETY OF APPLICATIONS: CRYPTOGRAPHY, IMAGE processing, special effects for movies, Web programming Web site search engines, and so on.[1–4]

Powerful tools make it easy to extend Python with C or C++.[5] Given a library written in C, and some written in C++, David Beazley's Software Wrapper Interface Generator[6] (SWIG) makes it easy to automatically connect that library's routines to Python. Python thus becomes a very attractive tool for both direct computation and computational steering, in which the compiled routines do all the real work and Python serves as a superstructure.

Python also has modules for user interface programming. Tkinter, the most popular such module, lets a Python programmer use the Tk library normally associated with the Tcl scripting language.[7–9]

Although SWIG can be used to create interfaces to Fortran modules, the procedure required is not suitable for use by ordinary scientists. We wanted to create a tool that took as its input something as close as possible to the Fortran 95 interface block. Not only is such a syntax already familiar to the Fortran 95 programmer, it is close to the normal syntax used by the Fortran 77 programmer.

We have created a tool, Pyfort, for connecting Fortran routines to Python. To use Pyfort, you create an input file that describes the Fortran functions and subroutines you wish to access from Python. This file uses a syntax that is close to a subset of the Fortran 95 interface syntax. Once the input file is prepared, you execute the Pyfort tool. The tool produces one or more Python extension modules, which you then compile and load into Python, either statically or dynamically, as desired.

The current release does not yet support explicit-interface Fortran 95 routines. However, the tool was designed with this in mind for a future release. Also, connection to common block and module variables might be added in the future.

Pyfort was created using the Earley-algorithm parsing tool by John Aycock, University of British Columbia.[10] One of us (Paul Dubois) implemented Pyfort by adding a new front end to part of a tool written by the other (Brian Yang). Paul has substantially modified Brian's wrapper generator to fit a more general application audience.

Creating the Python extension

Download pyfort.tgz from ftp://ftp-icf.llnl.gov/pub/python. After extracting the files from the archive, view the file pyfort.pdf; it is an Adobe Acrobat 3 file containing the latest documentation. The documentation describes the installation procedure and changes to the software that might have occurred since we wrote this installment of "Scientific Programming."

First, you prepare one or more input files containing a description of the Fortran routines you wish to link with Python. To execute Pyfort, you type

```
pyfort [-c compiler] file(s)
```

For each module *mymodule* described in the input file(s), two files are produced:

- File `mymodulemodule.c` is the C extension file that will connect Python to your Fortran routines.
- File `mymodule.txt` contains documentation for the Python calling sequence to call each function.

The optional *compiler* name specifies one of the compiler configurations built into Pyfort. You must pick a compiler that matches the Fortran compiler you are using with respect to

1. how the compiler maps Fortran names to link names,
2. how the compiler passes variables of type character, and

Café Dubois

Java jive

Scott Adams' strip *Dilbert* is a documentary, not a cartoon. I continue to hear Dilbert-like experiences connected to Java. As they used to say on Dragnet, the names of the victims have been changed to protect the innocent:

Frank's boss comes in and orders him to convert one of their scientific applications to Java, so that the boss can tell his boss that "we're using Java."

Paul goes to a demo of a product add-on that the company announces is written in Java so that it will be portable. The company's main product is written in C and is one of the most portable products Paul uses. The demo crashes. The demonstrator comments that it doesn't have a certain feature because he had to use the old version of Java since the new one is so unreliable.

Phil goes to Elaine's boss and says Elaine is making a terrible mistake, that she should be using Java rather than Python for a project. Elaine explains that she needs an interactive scripting language and that Java is not a scripting language. Phil insists that he is right because "everyone is switching to Java."

Doug is a consultant to some big companies. He says the big companies have decided to rewrite all their big CPU-intensive scientific simulations in Java. They have heard rumors that Java Grande will eliminate the performance problems. They haven't written any Java yet.

Winston is putting together a panel on using Java for scientific computing. When he tries to recruit panelists, nobody will be on the panel because they all believe this is a bad idea right now but are afraid to say so in public.

Whatever the future of Java, it deserves to be the subject of many PhD theses—in sociology. Write to me at pauldubois@home.com and tell me your story.

Déjà vu

Are you old enough to remember The Software Crisis? The Software Crisis occurred after a period of explosive growth in the use of software in business. At that point, the businesses really depended on computers and their business software, but were beginning to see some astounding costs in writing and maintaining software. New methodologies, principally the object-oriented revolution, came along to try to deal with the problem. We learned to use version control, unit testing, encapsulation, and so forth.

Now it is déjà vu, but this time in the Web business, says Susan Dart, Dart Technology Strategies Inc. (http://www.susandart.com). Susan gave an interesting talk at the Perforce User's Conference in June (http://www.perforce.com). She points out that there are businesses with million-page Web sites. Many Web site creators are not software professionals and they are learning the same lessons again the hard way, with the additional complexities of dynamic content and critical timing on releases.

I've been struggling with this problem on a smaller scale.

My employer requires a review of any Web page before you make it available to the public, so we have to have an internal site that the reviewers can review as well as a public site. Naturally, it would be good if we had some sort of backup or records short of the nightly tapes. So I was a rapt listener as Laura Wingerd showed how to use a version-control system such as Perforce to manage Web sites. A simple idea that fits my situation is to use your VCS tool with two branches, one representing each site. You make nearly all the changes on the internal site branch and then integrate them over to the "public" branch. You combine this with some cron-job scripts on workspaces dedicated to each Web and you have an easy-to-manage system.

CodeWarrior

CodeWarrior is a name I've always identified with the Mac, where it has a solid reputation as a good C++ development environment. See http://www.metrowerks.com/desktop/pro. I was pleased to learn it is available for the PC and will soon be for Linux. Someone whose opinion I respect, but whose employer forbids endorsements, says,

"CodeWarrior has been out for the PC for more than a year. However, they started having a real C++ compiler (that is, one that can understand templates) with CWP 4. I've been using a beta version of CWP 5 and it is really solid. The full CodeWarrior package for Linux will be out in the fourth quarter."

Object-oriented numerics

Todd Veldhuizen has moved the Object-Oriented Numerics page to http://oonumerics.org. The oon-list is now oon-list@oonumerics.org. The oonstd list is now oonstd@oonumerics.org.

—Paul F. Dubois, pauldubois@home.com

3. how Fortran types map to C (and therefore Python) types.

The default *compiler* is f77, which defines a compiler that links Fortran names as-is, and which uses the standard "address + extra argument for length" convention for passing Fortran character variables. The g77 compiler follows the GNU Fortran convention for linking. The compiler "solaris" adds an underscore to form the link name. There documentation contains a complete list and instructions on how to add new compilers.

Having created the extension module, we now need to compile and link it into Python. Python lets you

- create a statically linked Python interpreter containing the new module, or
- create a module that will be dynamically loaded on demand.

Usually, a dynamically loadable module is created. Python can also be embedded in your own application.

Currently, the Python community is developing tools to let users make these modules easily and portably. Rather than include instructions here, we will include the currently best available instructions in the Pyfort distribution.

Having created the extension module, we now need to compile and link it into Python.

Input. In the following description, *italics* denote names that the user will supply as desired, while square brackets [...] indicate optional input. Input is free-format but for compatibility with Fortran 95, an ampersand (&) at the end of a line can indicate a continuation line.

The input file consists of one or more module statements:

```
module modulename
...one or more descriptions of routines...
end [module [modulename]]
```

If the name of the module is repeated in the end statement, the name must match. This applies to all similar "end tags" we will describe: the structure name and particular name, if given, must match the name given at the beginning of the block.

To describe a function or subroutine, you simply enter the part of the function or subroutine that describes the input and return values, using a syntax that is similar to Fortran 95.

Fortran 95 has a new form of declaration for variables:

```
typename [, attribute-list ::] variable [,
variable…]
```

where, as before, the variables might include dimensioning information, for example, `x(10), y(25, 25).` The salient new feature here is the optional attribute list, followed by a double colon. The attribute list is a comma-delimited list of properties of the variables. The one that especially interests us here is the intent attribute. This takes the form

```
intent(intention)
```

where *intention* is one of the keywords *in*, *out*, or *inout*. The intent attribute can only be specified in the declaration of a formal parameter to a subprogram. The keywords *in*, *out*, and *inout* respectively declare that the argument is to be used as input, output, or both input and output. In Pyfort, if you do not declare an intent, it is taken to be intent(in).

Pyfort produces a Python module containing a method whose name is the same as your routine's name and that takes the input arguments as its argument list and produces the output arguments, including a function value if any, as its result. Thus, as called from Python, the routine will have a possibly different number of arguments. The documentation file that is produced by Pyfort shows the Python calling sequence for easy reference.

A special intent *temporary*, not corresponding to any Fortran intent, can be declared for an array that is used only by the called routine as a workspace but whose contents is not wanted as output. Such workspace arrays are common in older Fortran routines since Fortran lacked dynamic memory capabilities.

Array arguments, both input and output, should have their specified dimensions described in terms of expressions involving other (integer) arguments, integer constants, and the usual arithmetic operators.

Example 1: A simple function. Suppose we have a Fortran function sum that adds up the elements of an array argument x which is of length n:

Tser-Yuan Yang is a physicist at the Lawrence Livermore National Laboratory. His interests are in the areas of integrated modeling of physical systems with multiple space and time scales. He received a BS in electrical engineering from National Taiwan University, Taipei, and a PhD in physics from the Massachusetts Institute of Technology. He is a member of the American Physical Society. Contact him at tbyang@llnl.gov.

```
      function sum (n, x)
      integer n
      real x(n), sum
      integer i
      sum = 0.0
      do 100 i = 1, n
         sum = sum + x(i)
100   continue
      return
      end
```

Suppose we are going to write more routines like this one and we plan to make them available in Python as a module named arrayut. We create a file named, say, `arrayut.pyf` (the name of the file has no effect on the process). The file `arrayut.pyf` contains

```
module arrayut
      function sum (n, x)
      ! sum (n, x) = sum x(1..n)
            integer n
            real x (n), sum
      end function sum
end module arrayut
```

As you can see, this is essentially the text of the function with the body of it stripped out. The spaces between identifiers and keywords are meaningful, but not ones such as the space between a name and its dimension; either `x(n)` or `x (n)` would be accepted. The line that begins with an exclamation point is a comment that documents the function for the user, as explained below.

Next, we run the Pyfort tool:

```
pyfort –c compiler arrayut.pyf
```

This creates the desired module `arrayutmodule.c` and the documentation file `arrayut.txt`. The names of these files and of the Python module is determined by the name used in the module statement in `arrayut.pyf`. More than one module can be described in a single input file, and in that case a pair of output files is produced for each.

In Python, then, we can access the function sum:

```
import arrayut
x = [1.0, 2.0, 2.4, 3.6, 5.5]
print arrayut.sum (len (x), x)
```

Note that Pyfort will handle any needed type conversion of input values:

```
y = [1, 3, 5, 7, 9]
print arrayut.sum (len (y), y)
```

Of course, Python lets us directly insert the name of our extension routine, sum, into our current namespace if desired. Then we can omit the arrayut prefix from the call:

```
from arrayut import sum
x = [1.0, 2.0, 2.4, 3.6, 5.5]
print sum (len (x), x)
```

In these examples, we passed a Python list as an argument. The Pyfort-created extension will also accept anything that is or can be converted into an array of the correct Fortran type. In most cases, of course, we will be dealing with arrays created using Python's Numeric extension.

```
import Numeric, arrayut
x = Numeric.arange(1000) / 2.0
print arrayut.sum (len (x), x)
```

In our input file, we included a line beginning with an exclamation point right after the line declaring the function sum. This exclamation point line is a Fortran 95-style comment. This optional comment will be used as a Python "doc string" for sum. The user can see the doc string by typing

```
print sum.__doc__
```

Example 2: array output. Assume `g (command, x, y, n)` is a Fortran routine whose first argument is a string that should either be "norm" or "none," a real input array `x (n)`, and a real output array `y (n)`, where n is the integer length of these two arrays. The specification for this function can be added to `arrayut.pyf` by adding the following just below the end of the sum function:

```
    subroutine g (command, x, y, n)
        character*(*) command
        integer n
        real x (n)
        real, intent(out):: y(n)
    end
```

The resulting Python method is called like this:

```
y = g ("norm", x, len (x))
```

Note that while the Fortran g has four arguments, in the Python call there are only three, because the output array y has become a function result rather than an input. A Fortran 95 programmer who wants to return more than one result, or a Fortran 77 programmer who wants to return an array result, must use a formal parameter of intent(out) to hold such results. The user of such a routine must explicitly create variables to hold the answers and pass them into the routine as argument.

Python, on the other hand, can return many results in a tuple object. For example, this Python routine returns both the sum and difference of two numbers:

```
def sumdiff (x, y):
        return x + y, x – y
```

We can then assign the returned values to a list of results:

```
a, b = sumdiff (3, 9)
```

The Python signature is determined from the Fortran signature. As the preceding example illustrates, any Fortran argument declared as having intent *out* is not present in the Python calling signature; rather, it becomes an output. If the routine is a function, the function value is also an output. If there is only one output, it becomes the return value of the python function. Otherwise, if there is more than one output, a tuple of the values is returned, with the Fortran function value if any first, followed in order by the other outputs.

An argument of intent *inout* is present in the Python function's calling signature but is not one of the outputs. The user is restricted to passing an array of the exact type and size required.

An argument of intent *temporary* is present in neither the input nor output of the Python function. Instead, it is created on the fly before the call to the Fortran routine and disposed of immediately afterwards. This saves the Python program from needing to create a *temporary* to pass in.

Python's native floating-point type is promised to be the same as a C "double."

Interlanguage communication issues

When interfacing any two languages, there is a problem caused by differing representations. Standards committees being what they are, the representation for a particular type is usually compiler-dependent. Fortran has a notorious problem with type "logical," for example; attempts to mix routines compiled with different compilers might encounter a problem because the two compilers do not agree on what constitutes ".true.".

Python's native floating-point type is promised to be the same as a C "double." On many machines, this corresponds to a Fortran "doubleprecision." The Numeric package includes the ability to explicitly create arrays of 32-bit or 64-bit floating-point numbers. This is done by adding a second argument to the array constructor, for example,

```
x = array(arange(1000), Float32)
```

This is generally not necessary with Pyfort, as Pyfort will do the conversions required. However, if you wish to minimize the space requirements of a particular algorithm, you might wish to specify the correct type at the Python level.

One-dimensional array issues. Given that we are going to pass a Python array, list, or tuple to a Fortran routine, we have different semantics for Fortran and Python: Python arrays have a length; Fortran array arguments (except for explicit-interface F95 arrays) are passed only by address, and it is assumed that one of the arguments or a common block variable is available to provide the length. Currently, Pyfort requires that the length of the array be derivable from one of the integer input arguments using ordinary arithmetic operations. So, for example, we might have

```
function h (n, m, x, y, z)
        integer n, m
        real x (n), y(n + m –1),
real z ((n+1) * m, n)
end
```

On input, the actual sizes of the arrays `x, y`, and `z` will be compared to the values of `n` and `m` and an exception will be thrown if all is not as expected.

Fortran arrays are usually indexed from 1, but this is changeable on a per-array basis. Python arrays are always indexed from zero. It is possible to design a new "Fortran array" object for Python, but the indexing of such arrays would have a unique interpretation and that would mean users would have to be very conscious of which kind of array they were dealing with. For example, in Python, `x[-1]` denotes the last element of the array; if we had a Python object with arbitrary lower index, we would have to change that interpretation. We have instead simply chosen to consider the Python indices as zero-based counters into the array extent.

Multiple-dimension array issues. There are two issues with multiple-dimensioned arrays that do not occur for 1D arrays.

1. C and Python use row-major order but Fortran uses column-major order. This means that if x is 2D, `x [0, 0]` is the first element in memory and `x [0, 1]` is the second element in memory. In Fortran, `x (1,1)` is the first element and `x (2,1)` is the second.
2. Should we check that the number and individual extents of each dimension are correct, or simply that the total length is correct?

Using this approach, we hope to be able to increase Pyfort's ability to deal with explicit interfaces and user-defined types.

The storage-order problem presents an insoluble dilemma for a tool such as Pyfort. An argument passed to Fortran by Pyfort might have come from a native C or Python source, in which case the data, but not the shape, needs to be transposed in order to have Fortran perceive it correctly. Or, it might have come from an extension object and be in the right order already.

For multiple-dimensional arrays, Pyfort leaves it to you to put the array in the correct storage order for Fortran. To do otherwise might be a convenience for some cases but reduce the range of applicability of Pyfort. The following Python routine can be used for passing a Numeric array to Fortran:

```
import Numeric
def row_major(x):
  "Same shape but row-major order for data."
        return Numeric.reshape\
            Numeric.transpose(x), x.shape)
```

When Fortran returns an array, Pyfort creates the returned object as an array object whose data area is still in column-major order but which Python considers noncontiguous. This "lazy transpose" means the returned array will behave correctly in Python and leaves open the possibility that the actual movement of the data into row-major order might never have to be carried out.

Another aspect of this question is whether we should design for minimum data movement or for maximum safety and convenience. The current design takes the latter approach, but we continue to investigate how both goals could be achieved.

What kind of checking should we do of array sizes? Here again there is no obvious answer. We could check total size, total size and rank, or rank plus specific dimension extents. Pyfort checks ranks and extents to be consistent with the spirit of Fortran 95. However, the final dimension can be declared of size 1 or *, in which case this dimension's size is not checked.

Supporting explicitly interfaced routines

This first release of Pyfort has a number of limitations. Most seriously, it cannot call routines with "explicit" interfaces, which are a Fortran 95 concept. A subprogram has an explicit interface if it is contained in a module, another subprogram, or is declared in an interface block in some module. Thus, older Fortran programs have no routines with explicit interfaces. Only routines with explicit interfaces can use some of Fortran 95's powerful features.

Fortran 95 has no standard for the descriptors that will be used to pass an array to an explicit-interface routine. Because the argument must include more information than just the address, some sort of structure is necessary, but the standard leaves exactly what structure as an "implementation detail." Thus, a tool like Pyfort will have to deal with this on a compiler-specific basis. For the moment, it is possible to interface to an explicitly interfaced routine by writing a simple Fortran wrapper that itself does not have an explicit interface, and then connect Python to that wrapper.

Second, it would be nice to support the ability to define

```
module IMSL
        subroutine corvc (ido, nrow, nvar, x, ldx, ifrq, iwt, mopt, &
                icopt, xmean, cov, ldcov, incd, ldincd, nobs, nmiss, sumwt)
        integer, intent(in):: ido, nrow, nvar, ldx, ifrq, iwt, mopt, icopt
        integer, intent(in):: ldcov, ldincd
        integer, intent(out):: incd(ldincd, nvar), nmiss, nobs
        real, intent(in):: x(ldx, nvar)
        real, intent(out):: cov(ldcov, nvar)
        real, intent(out):: xmean(nvar), sumwt
        end subroutine corvc

        subroutine princ (ndf, nvar, cov, ldcov, icov, eval, pct, std, &
                evec, ldevec, a, lda)
        integer ndf, nvar, ldcov, icov, ldevec, lda
        real, intent(in):: cov(nvar,nvar)
        real, intent(out):: eval(nvar), pct(nvar), std(nvar), evec(nvar,nvar)
        real, intent(out):: a(nvar,nvar)
        end subroutine princ
end module IMSL
```

The documentation file produced by Pyfort shows the resulting calling sequences:

```
xmean, cov, incd, nobs, nmiss, sumwt =
        corvc (ido, nrow, nvar, x, ldx, ifrq, iwt, mopt, icopt, ldcov, ldincd)

eval, pct, std, evec, a = princ (ndf, nvar, cov, ldcov, icov, ldevec, lda)
```

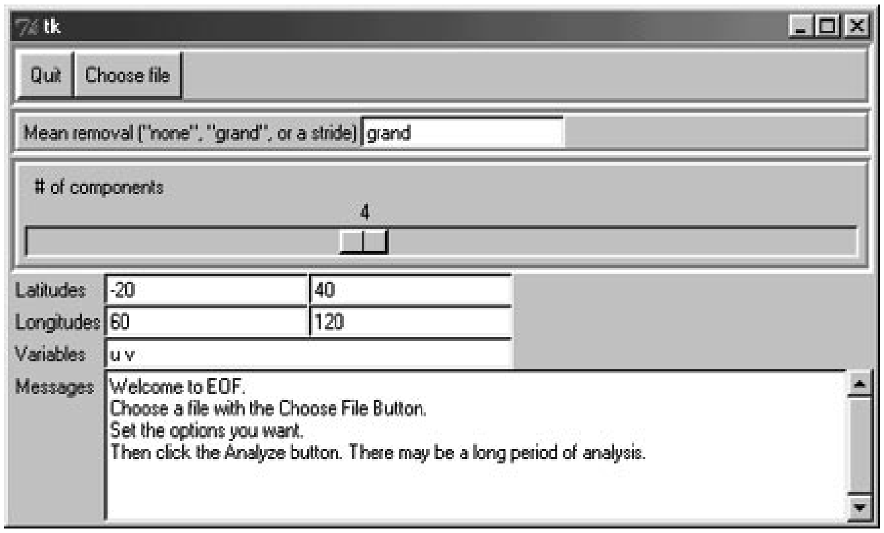
Then we wrote some Python code to package up the pair of calls we needed in a more convenient form, hiding issues such as the row-major question inside our own coding:

```
class Analyzer:
  "Interface to IMSL routines corvc and princ."

        def __init__ (self, x):
                self.x = asarray (row_major (x), Float32)

        def calculate_princ (self, icov = 0):
                "Call IMSL routine princ with given icov option."
                self.calculate_cov (0)
                nvar = self.cov.shape[0]
                self.eval, self.pct, self.std, self.evec, self.a = \
                        IMSL.princ (nvar, nvar, self.cov, nvar, icov, nvar, nvar)

        def calculate_cov (self, icopt = 3):
                "Call IMSL routine corvc with given icopt."
                ido = 0
                ifrq = 0
                iwt = 0
                mopt = 0
                nrow = self.x.shape[0]
                nvar = self.x.shape[1]
                ldx = nrow
                ldcov = nvar
                ldincd = 1
                self.means, self.cov, self.incd, self.nobs, self.nmiss, self.sumwt = \
                        IMSL.corvc (ido, nrow, nvar, self.x, ldx,
                                ifrq, iwt, mopt, icopt, ldcov, ldincd)
```

which we could then call in our Python application. Here we have gotten some data into a matrix ***x*** and we find the principal components, *pc*:

```
a = Analyzer (x)
a.calculate_princ ()
pc = matrixmultiply (x, a.evs)
```

Figure 1. Python/Pyfort packaging of two IMSL routines.



grand
-20 40
60 120
u v

```
Welcome to EOF.
Choose a file with the Choose File Button.
Set the options you want.
Then click the Analyze button. There may be a long period of analysis.
```

Figure 2. Small GUI interface created using Tk.

structured types. There is no standard for how the pieces of such a type will be arranged in memory unless a `SEQUENCE` specifier is given, but such a specifier might not be desirable for every user.

While the details are too complex for this report, in Pyfort the Fortran compiler is represented by an object and it performs certain parts of the code generation through its methods. New compilers can be added to Pyfort by inheritance and redefinition of these code-generation methods. Using this approach we hope to be able to increase Pyfort's ability to deal with explicit interfaces and user-defined types.

Tying it all together

Python is excellent for tying together resources. In a recent project, we had Python modules that would open and extract data from climate model runs and from observational data. Using the Numeric facility, we could do most of the required algorithm in Python already. But there were two routines from the IMSL statistical library[11] that we wanted to use, `corvc` and `princ`. As Figure 1 shows, we first described these routines in a Pyfort input file. We changed an *inout* array in one case to *in* because we had no need for the output in our application.

Finally, we added a small Tk GUI interface using less than 200 lines of Python. The GUI allowed the user to choose the file and some details about the arrays to be analyzed and to see a summary of the output. The GUI enabled other scientists to use our software without a big learning curve. As you can see from the opening screen shown in Figure 2, despite the small amount of source code required, the GUI contains features such as buttons, sliders, entry boxes, and scrollable text.

This sort of layering is very typical of effective Python development. Pyfort adds one more piece to the already powerful Python approach. The vast bulk of the time we spent on this project was spent in the right place: deciding what mathematical operations were appropriate. We reused software for database access, mathematical calculations, and creating the GUI. This let us create a powerful application very quickly.

Acknowledgment

This work was produced at the University of California, Lawrence Livermore National Laboratory (UC LLNL) under contract no. W-7405-ENG-48 (Contract 48) between the US Department of Energy (DoE) and the Regents of the University of California for the operation of UC LLNL. The views and opinions of the authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

References

1. G. van Rossum, Corp. for Nat'l Research Initiatives, Reston, Va.; http://www.python.org.
2. D. Ascher et al., *Numerical Python*, UCRL-MA-128569, Lawrence Livermore Nat'l Lab, Livermore, Calif.; http://xfiles.llnl.gov.
3. *Python Imaging Library*, Secret Labs AB; http://www.pythonware.com.
4. *Zope,* Digital Creations; http://www.digicool.com.
5. P.F. Dubois, "A Facility for Creating Python Extensions in C++," *Proc. Seventh Int'l Python Conf.*, Foretec Seminars, Boston, 1998, pp. 61–68.
6. D.M. Beazley, "SWIG and Automated C/C++ Scripting Extensions," *Dr. Dobb's J.,* Vol. 282, Feb. 1998, pp. 30–36.
7. M. Conway, *A Tkinter Life Preserver;* http://www.python.org.
8. E.F. Johnson, *Graphical Applications with Tcl & Tk,* M&T Books, New York, 1996.
9. F. Lundh, *An Introduction to Tkinter;* http://www.pythonware.com.
10. J. Aycock, "Compiling Little Languages in Python," *Proc. Seventh Int'l Python Conf.*, Foretec Seminars, Boston, 1998, pp. 69–78.
11. *IMSL Library,* Visual Numerics, Inc.; http://www.visualnumerics.com.