Editors: Paul F. Dubois, paul@pfdubois.com
David Beazley, beazley@cs.uchicago.edu

EXTENDING A SCIENTIFIC APPLICATION WITH SCRIPTING CAPABILITIES

By Fahri Basegmez

LIKE MANY OF US, YOUR APPLICATION PROBABLY STARTED OUT SMALL. YOU HAD SEVERAL SUBROUTINES YOU USED DAILY, AND OVER TIME, OTHER PEOPLE STARTED USING THEM AS WELL. THE MORE PEOPLE USED YOUR SUBROUTINES, HOWEVER, the more changes and additions became inevitable.

Your coworkers eventually started asking for a better alternative to the modify-recompile-run routine, so you created an input file-based interface to your application. Suddenly, what used to be a single subroutine now has thousands of lines of code. Some users then decided they wanted a graphical user interface, so you created it using a different language than the one in which you wrote your computational subroutines. To your amazement, your user base quadrupled in three months. You also quickly discovered that making two different languages talk to each other takes more time than successfully replicating cold-fusion experiments.

It pleased you to see that more people used your application, but, obviously, more users mean more demands. Now they want to perform optimizations, implement design of experiments, automate frequently performed operations, create 2D and 3D data plots, generate reports automatically, interact with databases, and perform many other tasks that you could not even have dreamed about. To satisfy all these demands, you decide that adding scripting capability to your application was the answer, but what's the best approach?

What's out there

In my own investigations for a similar situation, I discovered that several tools and intermediary languages existed for different platforms that could help different languages talk to each other. From these options, I picked COM (the Component Object Model) to implement scripting capability in Shvib for Windows. Shvib is a shock and vibration analysis program based on Tom Derby's Fortran subroutines that I have been developing at Barry Controls.

Freely and commercially available scripting languages abound out there, but after investigating some of them, I decided to use Python as a scripting language for Shvib. Shvib has a Visual Basic 6 GUI and several Fortran subroutines compiled as Windows DLLs to perform computations (see Figure 1). Because Shvib is a Windows application, COM was the natural choice to glue the Visual Basic GUI, the Fortran DLLs, and Python scripting together.

Why Python?

Python is a free, dynamically typed, object-based interpreted programming language that Guido van Rossum invented in 1989. It lets you create structural, object-oriented, and, to some extent, functional programs (or a mixture of them). Clean syntax, platform independence, built-in advanced data structures, a rich set of standard and nonstandard libraries, an interactive shell, and a mature user community are among Python's strengths. Plus, it's easy to learn, which might be the most desirable feature for any scripting language.

Python's freely available third-party visualization libraries address the visualization functionality that many scientific applications require. I was able to add simple 3D visualization to Shvib with zooming and rotating capabilities in one day by using David Scherer's Visual Python. VPython enables simple creation and manipulation of 3D primitives such as spheres, cylinders, boxes, arrows, springs, and so forth without having to deal with low-level graphics programming details.[1] For more advanced 3D graphics, the Visualization Tool Kit interfaces nicely with Python for both Unix and Windows platforms. I also used the Biggles 2D plotting package to generate GIF pictures and EPS files of 2D plots.

An application with scripting capabilities lets users create their own scripts from scratch, modify the parameters in existing scripts, and rerun them or just use the application without any programming. Users can also extend the application functionality via scripting using third-party or custom-built data-plotting and visualization packages. They can also create mechanisms that let them modify or customize the GUI via scripting. Experienced programmers tend to forget that

1521-9615/02/$17.00 © 2002 IEEE

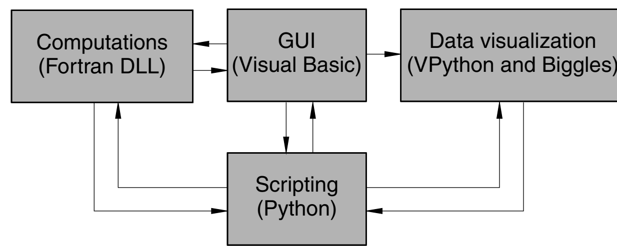

Figure 1. Shvib's communications schematic.

learning to program in a new language can be a painful process. Thus, letting users go through some simple programming exercises before exposing them to application-specific examples is crucial. Users should feel comfortable with language basics before starting to script the application.

COM in a nutshell

COM is Microsoft's object interface protocol for Windows. We can define a COM interface as the definition and organization of the methods and properties an object implements. COM standardizes this definition format so that objects created with different languages can communicate with each other without having to know implementation details. Every COM object implements the IUnknown interface—the absolute minimum for a COM object to have. The IUnknown interface implements `QueryInterface`, `AddRef`, and `Release` functions. These functions are used for communication with other objects, memory management, reference counting, and garbage collection. Every other interface must come from the IUnknown interface.

A COM object exposes a table of pointers—the virtual function table, or *vtable*—to functions through its interface. All COM objects communicate through function calls; even a COM object's properties are accessed through function calls (specifically, the `Get` and `Set` functions). Although Visual Basic and Python programmers do not have to know much about vtables, they must know that changing the list of methods and properties a COM object exports will alter the vtable and make the COM object incompatible with older versions. Once you've established and distributed a COM interface, you can't change it. However, this limitation does not apply to the COM object's implementation details. You can modify the implementation without causing any incompatibilities, given that the existing COM object's interface is preserved.

Globally unique identifiers

Globally unique identifiers help differentiate COM objects from each other. GUIDs are 128-bit numbers and are usually generated automatically to prevent two different COM objects from using the same identification number. Visual Basic creates them without user intervention; PythonCOM uses the `pythoncom.CreateGuid()` function to create them. For example, an output of the `CreateGuid` function on my PC is `{BA218881-AF07-11D6-8B65-B005E906D85F}`. The Windows registry stores these numbers to let other programs call COM objects via their IDs.

The algorithm that generates the GUIDs makes sure that the probability of two different developers generating the same number is practically zero (at least for a while). GUIDs can help identify interfaces (IID, or interface identification), classes (CLSID, or class identification), and libraries (LIBID, or library identification). Although machines can work with these large numbers comfortably, they are too long and cryptic to be practical for humans. To overcome this problem, user-created program identifications (ProgID) can name COM objects. For example, I used `Shvib.FORTRANInterface` to name one of the COM servers in Shvib. Table 1 summarizes the basic implementation details of a COM object in Visual Basic and Python.

IDispatch interface

Because we can use scripting languages without compiling, the language might not know a COM object's interface at design time. So, it must use ProgIDs and method and property names at runtime. IDispatch is the mechanism a

Table 1. COM basics in Visual Basic and Python.

	Visual Basic	PythonCOM
Class identification (CLSID)	Automatically assigned	`_reg_clsid_ = str(pythoncom.CreateGuid())`
Program identification (ProgID)	Automatically assigned by Visual Basic as `(Project name) + "." + (Class name)`	User typically defines a string as `(Company name) + "." + (Class name)`
Public methods	All *public subs* and *public functions* in the COM object	User exposes the list of public methods as `_public_methods_ = [ method1, method2, ...]`
Public properties	All *public properties* in the COM object	User exposes the list of public properties as `_public_attrs_ = [ property1, property2, ...]`

Café Dubois

Python steering C++

I am changing assignments at Lawrence Livermore; I now belong to the Advanced Software Technology Group in the Center for Applied Scientific Computing (www.llnl.gov/CASC). In my new assignment, I'm back to a Python-steered C++ code. Just in time, because there are two really great developments.

David Beazley's SWIG (http://swig.org) has a new release with significant improvements in the C++ support area, and the Boost Python Library by David Abrahams has many new facilities for wrapping C++ in version 2 (www.boost.org/libs/python/doc). These tools differ radically in approach, but they produce similar results.

Either of these tools would justify a full article just to scratch the surface of what they are capable of doing. Yet, both are much simpler to use than you might imagine. Just to give you an idea of how these tools work, consider this class definition, and suppose that we want to create and manipulate instances of it from Python:

```
#include <iostream>
class Vector2d {
public:
   Vector2d(const double x0=0.0, const double
         y0=0.0)
     :x(x0),y(y0) {}
     ~Vector2d(){};
     void SetX(const double x0) {x=x0;}
     void SetY(const double y0) {y=y0;}
     double GetX() const {return x;}
     double GetY() const {return y;}
     void printV() {std::cout<<"("<<x<<",
        "<<y<<")"<<std::endl;}
     double a, b;

protected:
     double x, y;
};
```

SWIG

With SWIG, the class definition itself is parsed and used to generate wrappers. Special directives can be added to more finely control the wrapping process. In this case we make a small SWIG interface specification file:

```
// attributeTest.i
%module attributeTest
%{
#include "vector2d.h"
%}
%immutable Vector2d::b;
%include "vector2d.h"
```

Using this file, you simply run SWIG on it to create an extra file of wrapper code. For example,

```
% swig -c++ -python attributeTest.i
```

You could also just add the interface code at the top of vector2d.h inside an "#ifdef SWIG ...#endif" and run SWIG on that.

Boost Python

Boost Python uses C++ itself to create wrappers. Here is the same example using Boost:

```
// boost_wrapper.cc
#include "vector2d.h"
#include <boost/python/class.hpp>
#include <boost/python/module_init.hpp>
using namespace boost::python;

BOOST_PYTHON_MODULE_INIT(attributeTest)
{
   class_<Vector2d>("Vector2d")
       .def_init(args<double>())
       .def_init(args<double, double>())
       .def("SetX", &Vector2d::SetX)
       .def("SetY", &Vector2d::SetY)
       .def("GetX", &Vector2d::GetX)
       .def("GetY", &Vector2d::GetY)
       .def("printV", &Vector2d::printV)
       .def_readwrite("a", &Vector2d::a)
       .def_readonly("b", &Vector2d::b)
     ;
}
```

To minimize typing, this example uses a syntactic sugar that might not be clear to less experienced C++ programmers. The result of each of the "def" family of methods returns the "self" on which it was called, so that you can apply the next "def" to that—this is equivalent to creating the object x = class_<Vector2d> ("Vector2d") and then having a whole series of statements x.def(...). Note the lonely semicolon at the bottom that terminates this one long statement.

Note also that the call to add the Python method setX as a wrapper of Vector2d's setX method does not explicitly need to state setX's signature. That is an important contributor to ease of maintenance. You can do many more things, such as creating properties and managing memory related to arguments.

Using the extensions from Python

For either tool, the Python session is exactly the same:

```
>>> from attributeTest import Vector2d
```

```
>>> v = Vector2d(5.5, 9.5)
>>> v.GetX(),v.GetY()
(5.5, 9.5)
>>> v.a=5.0
>>> v.setB(7.0)
>>> v.a, v.b
(5.0, 7.0)
>>> v.a = 10.0
# but v.b = 10.0 would throw an exception
```

This example doesn't begin to show the sophisticated capabilities of either tool (such as dealing with templates), but I hope that seeing how easy they are will encourage you to investigate them further.

Use value versus sale value

Say that a group writes some really great software for scientific computing. The management of their company, laboratory, or university gets the idea that maybe this software can be sold and become a cash cow for the organization, so they decide they won't release it as open source. In fact, though, there is almost no market for it whatsoever. Consequently, they are merely doing themselves a disservice because if they released it as open source, the rest of the community would help them fix it, expand it, and make it more portable. The resulting fame could also help with recruiting and retaining good people and with acquiring grant money and projects.

Eric Raymond has explained the source of such confusion in his essay "The Magic Cauldron" (available as Chapter 5 of *The Cathedral and the Bazaar*, O'Reilly, 2001, www.oreilly.com/catalog/cathbazpaper). Raymond has done some informal surveys of programmers and estimates that more than 90 percent of them are writing software that is not for sale. When told of this result, most programmers are very surprised, as I was. Raymond explains in his essay that most software has a "use value" but not a "sale value," and explores the reasons that this is so. I think management people might be able to let go of their illusion that the software their scientists produce might have sale potential if they could understand this point. My personal opinion is that very, very few programs of the type that we write could ever be sold. Can you imagine buying a shrink-wrapped parallel conjugate gradient solver in Comp USA? Or a framework for writing particle codes?

You can help authors by letting their management know that you would enjoy working with and helping to improve their software if it were open source, and that this would lead to recognition of their organization as a leader. That kind of reputation is something an organization can take to the bank.

My literary summer

Dateline, London: "A theatre company has dropped the word 'hunchback' from its stage adaptation of the classic novel *The Hunchback of Notre Dame* to avoid offending disabled people. Oddsocks Productions has renamed its touring production *The Bellringer of Notre Dame* after discussions with a disability adviser raised the possibility of offending people with spina bifida or the disfiguring scoliosis of the spine."

I hope that the great bull market bubble in Political Correctness popped with this announcement. What really had me laughing was that Victor Hugo would not recognize either title. I have attempted for some years to read *Notre Dame de Paris* and have repeatedly had to back off to something more at my level, such as Raymond Queneau's *Zazie dans le Metro* (turned into a classic film by Louis Malle in 1960) or a Georges Simenon *Maigret* mystery. But I keep trying to improve. I watch a French TV channel to practice, and I believed my accent was improving. Until Paris, when my accent market crashed.

Arriving in Paris this June, I proceeded to my favorite spot, the Jardin du Luxembourg, and approached the bandstand. I planned to find out when the next band would play, purchase a double espresso, assume the classic Dubois position (see cartoon), and enjoy the trees, music, children with their toy boats, and lovers kissing on the benches. As I arrived at the place that posted the list of bands, I noticed an elderly French woman standing there reading it. With my suavest of French accents, I gave a slight bow and said, "Bonjour, Madame." She turned to me and said helpfully, in the most beautiful French since Catherine Deneuve's, "The Americans are playing tomorrow."

Crushed, I proceeded to my holiday at The Dickens Universe at the University of California, Santa Cruz (http://humwww.ucsc.edu/dickens). I outraged the faculty by mentioning that Tiny Tim is a cripple. One paper analyzed *Dombey and Son*, using William Gibson's cyberspace classic, *Neuromancer*. But don't let that put you off—*Dombey* is interesting.

The Pulitzer Prize-winning novelist and recent Dickens biographer Jane Smiley appeared and gave an excellent talk. She likes *Our Mutual Friend*, and I do, too. After the first chapter you'll be hooked. Or if the "South Park" cartoon version of *Great Expectations* was your last exposure to Dickens, maybe you should give the book a try. To avoid disappointment, you should know that Miss Havisham's army of robot monkeys is not in the book.

Quiz: compare and contrast the Hunchback and Tiny Tim. You could become an English professor if you can learn to instinctively ask questions like, "And why is Tiny Tim tiny? Is Dickens attempting to confuse his gender?"

Fortran DLL

```
C Compile the following Fortran code as a DLL
C Place the DLL in the system32 subdirectory
      SUBROUTINE TESTDLL(AIN,AOUT)
        !DEC$ATTRIBUTES DLLEXPORT :: TESTDLL
        REAL** AIN(2), AOUT(2)
        DO 1 I=1,2
1         AOUT(I)=2. *AIN(I)
        RETURN
      END
```

Visual Basic ActiveX DLL

```
'Create a new ActiveX DLL project called COMTest
'Insert the following code into a module
Declare Sub TESTDLL Lib "testdll.dll"_
      (AIN As Double, AOUT As Double)

'Create a class Module called FORTRANCall
'Insert the following code into the class module
Public Function CallFortran(data() As Double) As Variant
    Dim FortranResults(0 To 1) As Double
    Call TESTDLL(data(0), FortranResults(0))
    CallFortran = FortranResults
End Function
'Name the project as COMTest in VB Project Explorer
'Under the File menu click on the Make COMTest.dll
```

Calling Fortran from Python via ActiveX DLL

```
# Execute the following code from Python
import win32com.client
test = win32com.client.Dispatch("COMTest.FORTRANCall")
data = [1,2.5]
print test.CallFortran(data)
```

Figure 2. The Fortran DLL is compiled as a regular DLL with Digital Visual Fortran Version 5 (at the DOS prompt, enter df/dll testdll.for to compile it as a dll; different Fortran compilers might require different syntax). The resultant "testdll.dll" is copied into the c:\windows\system32 directory. This function receives a size two array of floating numbers and returns another array with doubled values.

COM object uses to advertise its interface during runtime. IDispatch, like any other COM interface, is derived from the IUnknown interface. IDispatch implements `GetTypeInfoCount`, `GetTypeInfo`, `GetIDsOfNames`, and `Invoke` functions. It enables another object to call the COM object's public methods and properties by their names via the `GetIDsOfNames` function. In the COM example in Figure 2, the Visual Basic ActiveX DLL has only one method (`CallFortran`), so the vtable for the COM server has eight functions in the following order: `QueryInterface`, `AddRef`, `Release`, `GetTypeInfoCount`, `GetTypeInfo`, `GetIDsOfNames`, `Invoke`, and `CallFortran`.

When you create a COM object with Visual Basic, the IUnknown and IDispatch interfaces are created for you automatically. Think of the IDispatch interface as a just-in-time communication mechanism between the COM server and the client. The act of using this process is usually called *late binding*; applications or DLLs supporting late binding are called *automation servers*. Visual Basic, Delphi, Python (with Mark Hammond's win32com extensions),[2] or any language that supports COM can call any automation server. Figure 3 illustrates a simple example of calling Excel from Python to create a 2D plot and placing it into a PowerPoint presentation via the IDispatch interface. If you are not familiar with Excel's or PowerPoint's object model, an easy way to automate your tasks is by studying the recorded macro routines. Simply go to Tools, Macro, and Record New Macro in any Microsoft Office application and then perform the task you are trying to automate. Then, open the Visual Basic editor by going to Tools, Macro, Visual Basic Editor, and then study the macro you just created. For a more detailed study, in the Visual Basic editor, go to View and then Object Browser (or just press F2) to see the available objects' attributes and methods. (You might have to install additional help options for Microsoft Office to see the Object Browser.) The Immediate panel in the Visual Basic editor (press Ctrl + G if you do not have it open) lets you test some simple commands and expressions interactively.

Early versus late binding

You can bind COM objects before runtime via early binding. This process is established through type libraries, which contain all the necessary information to access a COM object's methods and properties.[3] When an interface is created in Visual Basic, a type library defining it is automatically created for the user. When you create a client in Visual Basic, you access the type library through the Project menu and then the

References submenu in the Visual Basic editor. In Python, you can access a type library from the Tools menu or the COM MakePy utility button in PythonWin. Even if you run the MakePy utility prior to running a program, you can force late binding with the `win32com.client.dynamic.Dispatch` method. Table 2 illustrates examples of early and late binding in both Visual Basic and Python. After creating a COM server, you can examine its type library information by using Microsoft Visual Studio's OLE View tool.

Once early binding is established, the communication efficiency between client and server grows significantly higher than achievable with late binding. Although you only have to establish early binding once in a client machine, the process can be very time and space consuming. In general, type checking is safer with early binding. However, in some cases, early binding might be the only possible way to establish a COM connection. Although early binding is case sensitive, late binding might not be. Enumerated COM constants are not available with late binding, so the user must use integer literals instead of named constants.

```
# Using COM from Python.
# For early binding:
# Run "COM Makepy Utility" for
# Excel and PowerPoint object libraries
# For late binding:
# Use win32com.client.dynamic.Dispatch
# Comment out the third line (# vbcons = win32com…)
# Use the following integer literals instead of the
# enumerated constants
# xlXYScatterLines = 74
# xlColumns = 2
# xlLocationAsObject = 2
# vbcons.ppLayoutBlank = 12
import win32com.client

xl = win32com.client.Dispatch("Excel.Application")
vbcons = win32com.client.constants
xl.Visible = 1
xl.Workbooks.Add()
data_range = xl.ActiveSheet.Range("A1:A24")
# generate some random numbers
data_range.Formula = "=Rand()"
xl.ActiveWorkbook.Charts.Add() # add a chart
xl.ActiveChart.ChartType = vbcons.xlXYScatterLines
xl.ActiveChart.SetSourceData(Source = data_range,
                             PlotBy = vbcons.xlColumns)
xl.ActiveChart.Location(Where = vbcons.xlLocationAsObject,
                        Name = xl.ActiveSheet.Name)
xl.ActiveChart.ChartArea.Copy()   # copy the chart
pp = win32com.client.Dispatch("Powerpoint.Application")
pp.Visible = 1
pp.Presentations.Add()
pp.ActivePresentation.Slides.Add(1, vbcons.ppLayoutBlank)
pp.ActiveWindow.View.Paste()   # paste the copied chart
```

Figure 3. A simple automation demonstration that uses Microsoft Excel and PowerPoint, both of which must be installed on your computer to run this example. If you have Excel but not PowerPoint, you can comment out the lines involving PowerPoint objects (insert # at the beginning of a line to comment it out).

ActiveX DLL versus ActiveX EXEs

In Visual Basic, there are three different ways to create a COM server, ActiveX EXE, ActiveX DLL, and ActiveX

Table 2. Early versus late binding.

	Early binding	Late binding

Visual Basic	`Dim TestObj as COMTest`
	`Set TestObj = New COMTest`
		`Dim TestObj as New COMTest`
		`Or`
		`Set TestObj = CreateObject("COMTest.Interface")`

PythonCOM	# Run MakePy for the COM object in question
	# prior to running the code
	`from win32com.client import Dispatch`
	`TestObj=Dispatch("ComTest.Interface")`
		`from win32com.client import Dispatch`
		`TestOBJ = Dispatch("COMTest.Interface")`

OCX. ActiveX OCXs are custom-made Visual Basic controls. They function similarly to ActiveX DLLs, but you can use them inside other Visual Basic controls like forms. An ActiveX DLL (or COM DLL) must implement the following four functions: `DllRegisterServer`, `DllUnregisterServer`, `DllGetClassObject`, and `DllCanUnloadNow` (see http://msdn.microsoft.com/library).

Several differences exist between ActiveX DLLs and ActiveX EXEs. Once you call an ActiveX DLL from your application, ActiveX DLL functionality is loaded into the same namespace as the calling application; ActiveX EXE files are executed in their own separate namespaces. Consequently, if an ActiveX DLL crashes, it can take your application down with it, whereas a crashing ActiveX EXE file could generate an error message in your application but won't modify the memory space allocated by your application. ActiveX DLLs perform faster than ActiveX EXEs. If you call an ActiveX EXE twice, it will create two different namespaces; if you pass global variables to these ActiveX EXEs, the variables end up being nonglobal variables and might produce obscure bugs. Because ActiveX DLLs use the same namespace as the calling application, creating multiple versions of the same global variable with ActiveX DLLs is impossible. Threaded applications can have similar complications as well; for these reasons, it is not a good idea to pass global variables around.

Distributing COM servers

As I mentioned earlier, when you compile your COM server, the type library is written into the Windows registry. This is how other programs know which DLL or EXE file to call—by knowing only the COM server's ProgID or CLSID. Now your machine knows about the COM object you just created, but what about the other machines that will use this COM server? Several options exist to register a COM server, including adding COM server information into the Windows registry manually, writing a program to register the COM object, letting an installation program (such as the Package and Deployment Wizard that comes with Visual Studio) register it automatically, or distributing the COM object across a network from a server by using DCOM (Distributed COM). Naturally, backing up the registry before attempting to modify it is a good idea.

Calling Fortran from Python

In Unix or Linux, PyFort or F2PY (the Fortran-to-Python interface generator) is commonly used to call Fortran from Python. There is a third possibility (at least on paper) for calling Fortran from Python through C: using SWIG (the simplified wrapper and interface generator). In Windows, in addition to PyFort and F2PY, COM can call a Fortran DLL from Python (see Figure 4). In Shvib, I created a COM server to call Fortran subroutines compiled as regular DLLs. Visual Basic or Python scripts can call these Fortran subroutines through the COM server—the main application does not have to run in order to run Python scripts.

Using Visual Basic for the GUI, implementing the program logic and computations in Python, and using COM to make Python and Visual Basic communicate with each other is also possible. In some cases, you might need to know the state of your application while running scripts. In this case, however, Visual Basic and Python can pass their objects to each other via a COM interface and modify each other's properties with callback mechanisms. Passing a Visual Basic object to Python lets Python call this object's methods or modify its properties. You can pass a

Useful URLs

Biggles: http://biggles.sourceforge.net
COM: www.microsoft.com/com/tech/com.asp
F2PY: http://cens.ioc.ee/projects/f2py2e
Mayavi: http://mayavi.sourceforge.net/screenshots/index.html
Numerical Python: http://pfdubois.com/numpy
PyFort: http://sourceforge.net/projects/pyfortran
PyNaC: http://sourceforge.net/projects/pynac
Python: www.python.org
Python COM server: http://starship.python.net/crew/pirx/spam7
PythonWin: www.python.org/windows/pythonwin
Scientific Python: http://starship.python.net/crew/hinsen/scientific.html
SciPy: http://scipy.org
SWIG: www.swig.org
VB-Python: www.hps1.demon.co.uk/users/andy/pyvb/index.htm
Visual Python: www.vpython.org
Visualization Tool Kit: www.kitware.com/vtk

built-in Visual Basic object, such as a button or form, or a custom-created object. In this scheme, Visual Basic has to pass the object in question before Python can call any of its methods. Once this object is passed, Python can use its public methods and properties.

Users never fail to surprise developers with their creativity. Predicting what your potential users' future usage patterns will be is a formidable—if not impossible—task. However, when the developer himself is a user, the development process tends to go a bit smoother. With scripting, the need to guess future requirements diminishes, and application development becomes a continuous process.

At the end of the day, we can deem an application to be successful when the scripting does not intimidate novice users but also lets advanced users perform any task with ease. A good scripting language not only increases productivity but also enables users to glue new tools and components to the main application. For scientific applications, availability of scientific routines and tools is as important as the language selection itself. Python proved to be the perfect tool for the job at Barry Controls.

Python COM server to call Python from Visual Basic

```
# A simple COM server.

import  pythoncom
import  win32com.server.register

class   COMDemo:
    _reg_progid_ = "VB_Python.Demo"
    _reg_clsid_ = str(pythoncom.CreateGuid())
    _public_methods_ = ['ReverseArray']
    def ReverseArray(self, Sequence):
        aList = list(Sequence)
        aList.reverse ()
        return aList

if _name_ == '_main_':
    try:
        register.UseCommandLine (COMDemo)
    except:
        print 'Error! COM server is not registered!'
```

Calling Python from Visual Basic

```
'Create a new button in a form and name it btnReverse
'Place the following code in the form's module
'Make sure to run the preceding Python code first
Private Sub btnReverse_Click()
    Dim VB_Python As Object
    Dim varArray(1 To 3) As Variant
    Dim varResp As Variant
    Set VB_Python = CreateObject ("VB_Python.Demo")
    varArray(1) = 0.75
    varArray(2) = "Hello"
    varArray(3) = 2
    varResp = VB_Python.ReverseArray(varArray)
    For Each item In varResp
        MsgBox item
    Next
    Set VB_Python = Nothing
End Sub
```

Figure 4. Calling the `COMTest.FORTRANCall` COM server from Python. If you used different names for your Visual Basic project and the class in this project, you must use the appropriate name for the COM server ProgID.

References

1. D. Scherer, P. Dubois, and B. Sherwood, "VPython: 3D Interactive Scientific Graphics for Students," *Computing in Science & Eng.*, vol. 2, no. 5, Sept./Oct. 2000, pp. 56–62.
2. M. Hammond and A. Robinson, *Python Programming on Win32*, O'Reilly, Sebastopol, Calif., 2000.
3. J. Mojica, *COM+ Programming with Visual Basic*, O'Reilly, 2001; http://msdn.microsoft.com/library.

Fahri Basegmez is a senior finite-element analyst at Barry Controls in Brighton, Massachusetts. His research interests include shape optimization, hyperelastic material models, finite-element meshing techniques, and computer–user interfaces. He has a BS in mechanical engineering from Yildiz Teknik Universitesi, Turkey, and an MS in manufacturing engineering from Worcester Polytechnic Institute, Massachusetts. Contact him at Barry Controls, 40 Guest St., Brighton, MA 02135; fbasegmez@bcdi.com.