Lessons Learned in Developing PERP

(Python Environment for Radar Processing)

Joseph VanAndel

National Center for Atmospheric Research[*]

P.O. Box 3000, Boulder, CO, 80307, vanandel@ucar.edu

Abstract

Our work at the National Center for Atmospheric Research includes processing data from research meteorological radars. We evaluate and tune signal-processing algorithms that process as much as 1500 megabytes of data per hour in real-time. In the past year, we have designed and implemented the Python Environment for Radar Processing (PERP) using Numeric Python to aid in our research. Because Python is an interactive, interpreted language that does not require the compile and link steps needed by compiled languages, Numeric Python is a very productive programming environment. We were pleased to find that Numeric Python was also quite efficient in executing our algorithms. We found both SWIG and CXX quite valuable in writing our Numeric Python extensions. Although we did encounter some challenges in learning Numeric Python and debugging our extensions, Python allowed us to build a flexible, yet efficient, system to process our large data-sets and test our algorithms.

1. Introduction

To support our work in developing and testing new signal processing techniques, we needed a processing environment to perform IIR filtering and pulse-pair processing. In addition, we wanted to implement a fuzzy logic algorithm to automatically recognize specific types of radar targets. We also needed the ability to graph subsets of radar data for quality control purposes.

Although we could have written a conventional C/C++ program consisting of compiled subroutines all linked together, we wanted the flexibility that an interpreted language would provide, without sacrificing performance. We wanted to build a set of modules that could easily be invoked with different sets of data and parameters. We needed a "production" environment that could easily process gigabytes of data, yet we wanted the ease of programming that an interpreted language would provide. When we discovered that Numeric Python[1] could meet our requirements, we designed and implemented the Python Environment for Radar Processing (PERP). PERP implements IIR filtering and pulse-pair processing as well as a fuzzy logic based recognizer.

The next sections will explain what is Numeric Python, why we chose Numeric Python and how we built the Python Environment for Radar Processing (PERP). We then describe the capabilities of PERP, the challenges we overcame in using Numeric Python and some additional benefits of Numeric Python.

2. What is Numeric Python?

Numeric Python (NumPy) was written by Jim Hugunin at MIT, based on earlier work by Jim Fulton. It is now being maintained by Lawrence Livermore National Laboratory. NumPy was inspired by the array processing features of IDL, MATLAB and Mathematica. NumPy adds efficient operations on N-dimensional numeric arrays to the Python language. NumPy has its own SIG (Special Interest Group) on the www.python.org web site.

3. Why Numeric Python?

We chose Numeric Python to build our data analysis package for a variety of reasons, including its free availability, efficiency, open source, and general purpose features.

3.1 Numeric Python is free

We wanted to be able to freely share our package with our collaborators, without forcing our colleagues to buy a particular commercial package, like MATLAB or IDL. Since our algorithms may run as part of a transportable research weather radar, it is easier to use a free package, rather than having to buy a separate license for our "field" computer. Python and Numeric Python are freely available, and do not require any licensing fees or license managers.

3.2 Numeric Python is very efficient

Our past experience with some commercial packages like MATLAB or IDL shows that although developing an algorithm in these languages may be straight-

[*] NCAR is supported by the National Science Foundation.

forward, the interpreted implementation may be quite inefficient when processing large amounts (gigabytes) of data. Our experiments with Numeric Python showed that Numeric Python's overhead was small compared to the I/O and computations the application would perform.

3.3 Numeric Python is Open Source

Since Numeric Python is Open Source, it has significant advantages over proprietary solutions. We saw several advantages in using an Open Source solution:

- Debugging is much easier
- Performance tuning is easier.
- Adding new features is easier.
- Support is great!

When using a computation environment like Numeric Python, MATLAB, or IDL to build a complex signal processing or data analysis system, we must write our own compiled extensions to the environment to efficiently perform particular computations. All of these computation environments support user written subroutines in C/C++ to extend their functionality. However, we've found it can be quite difficult to debug programs that are linked to vendor provided proprietary object code. Our experience indicates that having the full source code for the data analysis program is extremely valuable, if not essential. For example, to debug certain memory usage problems (dangling pointers or memory leaks), the **entire** application must be re-compiled using Insure++[2], Purify[3] or some other memory validation tool. Without source code for the entire application, a programmer is dependent on the vendor to find problems in his/her source code or to help trace problems in his/her code. Clearly, if the vendor source code is not available, a programmer can only debug or fix his/her own code, not the vendor's code. But without **all** the source code, a programmer will find it difficult to trace problems that originate in his/her own code, particularly if his/her code triggers a bus error within the vendor provided analysis package.

When developing a data analysis package, it is frequently necessary to tune the system to improve performance and avoid excessive memory use. For example, we wrote some signal processing code in MATLAB using MATLAB's high-level graphics package. This code required enormous amounts of virtual memory to display our output (i.e., processing 20 megabytes of data required 700 megabytes of virtual memory.) Without the source code, we have been unable to determine the reason for this problem. With the source code, it would be possible to determine the reason for this problem.

As has been noted in *The Cathedral and the Bazaar*[4], Open Source software benefits from having many talented programmers scrutinizing the source, and contributing bug fixes and enhancements. Therefore, Open Source data analysis packages may actually be more reliable than commercial, proprietary solutions.

When using an analysis package, we sometimes need to add a new feature. For example, when we needed the Numeric Python "interp()" routine to accept multi-dimensional arrays, we simply modified the Numeric Python source code. This source code simply would not have been available if we were using MATLAB or IDL. As a result, either we would have had to write the routine "from scratch", or live without it.

On occasion, we have needed support to solve problems with our data analysis package. On the rare occasions we've had any problems with Python or the Numeric extensions, we have gotten very quick support (sometimes in less than 8 hours time) from the developers or from other users, and of course we could examine the source code ourselves. In contrast, when using commercial software, we have found it difficult to get a timely response from some of our vendors.

3.4 Python is a general purpose language

Commercial products like MATLAB and IDL provide powerful array processing data analysis environments. In fact, they provide a richer set of basic signal processing libraries than currently available for Numeric Python. However, MATLAB and IDL are **not** general-purpose programming languages. (For example, we would not write a web-site link consistency checker or a database interface in MATLAB!) In contrast, in addition to Numeric Python's very powerful array processing features, Python can be used as a scripting language and a systems programming language. Python is a full-featured programming language and provides an extensive set of runtime libraries for general-purpose program development. Python makes it considerably easier to interface the PERP package to other software or to build a GUI to control the PERP package.

4. How is PERP built?

When writing PERP, we wanted to concentrate on our problem domain, and not on the details of interfacing Python to C++. The standard C API for Python is quite low level, and requires writing lots of "boilerplate" code to access Python data types (lists, tuples, dictionaries), perform parameter checking, and handle

Python's reference counting. Fortunately, we could use CXX and SWIG to simplify our extension writing.

4.1 SWIG

SWIG[5] is the Simplified Wrapper Interface Generator, written by David Beazley. SWIG uses an interface description file derived from an existing C/C++ include file. SWIG processes the interface file and generates C or C++ code that interfaces Python to C/C++ extensions. Python programs can invoke C/C++ functions or methods, and can reference elements of C/C++ structures or objects. SWIG has a fairly complete manual, and an active mailing list that can provide answers to questions about SWIG or problems with using SWIG. We found SWIG very helpful in building our interface functions. However, when writing C++ extension code that manipulates Python objects, we wanted a higher level interface than is available from either Python or SWIG.

4.2 CXX

Paul Dubois wrote a package called CXX[6] that is included with the Livermore Labs Numerical Python distribution. CXX is specifically designed to simplify manipulating Python objects from C++. Since we represent our radar data as arrays, tuples and dictionaries contained in a dictionary (see section 5.1), CXX was quite helpful in building and accessing these data structures in our C++ extensions. The following is an excerpt from one of our C++ extensions that uses CXX.

```
PyObject *array = PyArray_FromDims(2, dims,
PyArray_FLOAT);

// Store all per-variable info in a dictionary

Dict d;

d["scale"] = Float(scale);

d["bias"] = Float(bias);

d["badValueInt"] = Int(badValue);

d["badValueFloat"] = Float((badValue-
bias)/scale);

d["array"] = Object(array);

// this array is only referenced by this dictionary.

Py_DECREF(array);
```

Note the ease with which we created a dictionary, and added floating point, integer, and array objects to the dictionary. However, also note the use of Py_DECREF(), to maintain the correct reference count on the newly created array. CXX only supports multi-dimensional NumPy arrays by using the Blitz[7] scientific computation package. Since we were not ready to use Blitz (which at that time was only available as an alpha release), we used the standard C Numeric Python interface, which requires the programmer to explicitly handle reference counts.

Because PERP uses CXX, PERP requires a modern C++ compiler that has comprehensive template and exception handling support. Fortunately, as we were designing PERP, the EGCS version of the GNU C compiler was just adding these features. Now that EGCS is the official GNU C compiler, Gnu C++ (2.95 or better) provides the necessary C++ features. Although some of the CXX template-based C++ routines are noticeably slower to compile than ordinary C++ code, the compilation speed is quite acceptable on a modern, 400Mhz, Linux PC. Although PERP does require a modern C++ compiler, it is portable to other architectures. We've also built PERP on Sun/Solaris using gcc 2.95. Apparently, Kuck & Associates' KAI C++[8] compiler will also compile CXX-based code.

5. What capabilities does PERP provide?

5.1 Input/Output of data.

Clearly, to be useful, PERP needed to read and write our radar data. For time-series data, we use netCDF[9] to store our data. NetCDF stores binary data in a machine independent form, such that we can read the same data files on either PCs or Sun SPARC machines, without having to code our I/O routines to swap bytes. In addition netCDF data files are self-describing, so an application can determine the dimensions of all arrays, and what sort of data is stored in each array. PERP uses the netCDF Python package written by Konrad Hinsen[10] to read and write radar time-series data. We've written our own translator programs in C to translate "raw" time-series data into netCDF, so that PERP can process it.

Once PERP has processed our time-series data into reflectivity and velocity fields, we store it in NCAR's DORADE[11] format, since this format is used by an existing radar display program and by other analysis programs. As a result, we wrote our own input/output routines to translate Numeric Python arrays to/from the DORADE format. We used SWIG's ability to map C structures to Python to allow our Python scripts to read and write individual members of DORADE "C" structures that describe the radar data.

At first, we had some difficulty deciding how to represent our collection of numeric arrays and their associated attributes in Python (Attributes include the

integer value used for a "bad data flag" and the scale factor used when the data is stored in a file). We experimented with storing the array's attributes in a tuple, but we finally decided to use dictionaries to organize our data, since the key/value pairs in the dictionary provided "self-documentation" for our data. The entire dataset was stored in a dictionary that contains dictionaries, as shown below:

```
{'DZ': { 'array': array((20, 1040),f), 'scale':100.0,
'bias': 0.0, 'badValueFloat': -327.68 },

'VE' : {'array': array((20, 1040),f), 'scale': 100.0,
'bias': 0.0, 'badValueFloat': -327.68} }
```

Each radar variable's dictionary can be located by the variable's name. Once the dictionary for a particular radar variable is located, the attributes and the actual data can also be located by name. Using nested dictionaries in Python provided some of the same functionality that an array of structures does in C++.

5.2 Radar Signal Processing

PERP allows the user to filter radar data using a collection of Infinite Impulse Response (IIR) filters. I converted a colleague's filter code to C++, and then wrapped the resulting IIR_Filter class with SWIG. An example of using the IIR_Filter class in Python is shown below:

```
coeff = FloatTuple(len(coeff_list), coeff_list))
f = IIR_Filter(numPoles, const_gain, coeff)
iFiltered, qFiltered = f.filter(iRaw, qRaw)
```

Note that FloatTuple is a simple class that converts a variable length tuple of floating point values to a C++ object used by IIR_Filter. If we had used CXX for the IIR_Filter class, we would have not needed the FloatTuple class.

PERP also contains a complex auto-correlation algorithm (pulse-pair) algorithm that processes time-series data into radar reflectivity, velocity and width. Again, I converted a colleague's "C" routine to C++, and wrapped the PulsePair class with SWIG. Below is an example of calling this class from Python:

```
p = PulsePair(scale,offset, prt, radarConst,
    gateSpacing, firstGate)
# I,Q,P,Z,V,W are Numeric arrays
p.compute(I,Q, P,Z,V,W)
```

5.3 Fuzzy Logic Implementation

In addition to our signal processing work, we have been exploring algorithms that automatically identify radar data that has been contaminated by "anomalous propagation ground clutter" (AP). AP occurs when atmospheric conditions refract (bend) the radar beam causing it to be reflected from the ground, rather than continuing to sample the atmosphere. This phenomenon is comparable to observing what appears to be water on the road on a hot, dry day (an optical mirage.) If AP is not properly identified, meteorologists or automatic algorithms may interpret AP as an intense storm, which can cause false alarms for flooding, or can cause aircraft to be routed around a non-existent "storm".

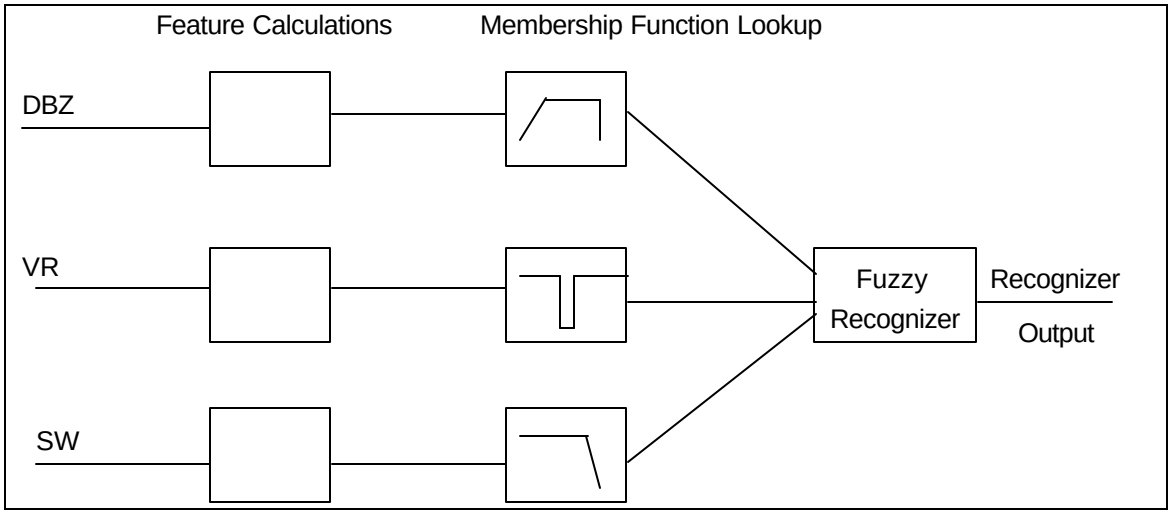

Figure 1 - Fuzzy Logic Algorithm

Our previous research had shown that a "Fuzzy Logic" algorithm could be used to identify AP. In brief, we compute "features" or "interest fields" such as mean or standard deviation for each spatial position in our radar data. A membership function (related to a probability density function) is then applied to each of these features, yielding a set of values in the range [0-1]. We then computed a weighted sum of the membership functions for each spatial position. If the weighted sum exceeds a specified threshold, the fuzzy logic algorithm has identified AP at this location. Figure 1 shows a block diagram of a fuzzy logic computation.

We had already implemented a "Fuzzy Logic" algorithm as a stand-alone C++ program, but wanted to build a more flexible, higher performance implementation. We wanted an environment which would allow us manipulate entire arrays of data, as with MATLAB or IDL, that would allow us to dynamically choose what calculations were performed. To add fuzzy logic calculations to PERP, we re-implemented the feature computation algorithms from our previous C++ code and used SWIG to build a dynamically loaded Numeric Python extension. The membership function lookup tables were implemented using the interp() function in arrayfnmodule.c. The final weighted sum calculation is performed directly in Numeric Python.

5.4 Graphs of radar data

We have found it quite helpful to graph our radar data from inside Numeric Python. We ended up installing "Yorick" [12] and using the "gist" module from the Numeric Python distribution for scientific plotting. Using "gist", we've displayed scattergrams of our radar data. Also, we produced graphs of received power versus antenna position that helped us verify the performance of our radar antenna. For example, Figure 2 shows a contour plot of received power versus antenna position. Each contour indicates a given amount of received power. (Ideal antenna performance would be indicated by uniformly circular contours.) To produce this plot, we wrote a simple Python *Contour* class. The *Contour* class "grids" values of power for a given azimuth and elevation into a 2D array and plots the result with the "gist" contour plotting routine. We were very pleased how easily we could plot our data using Numeric Python.

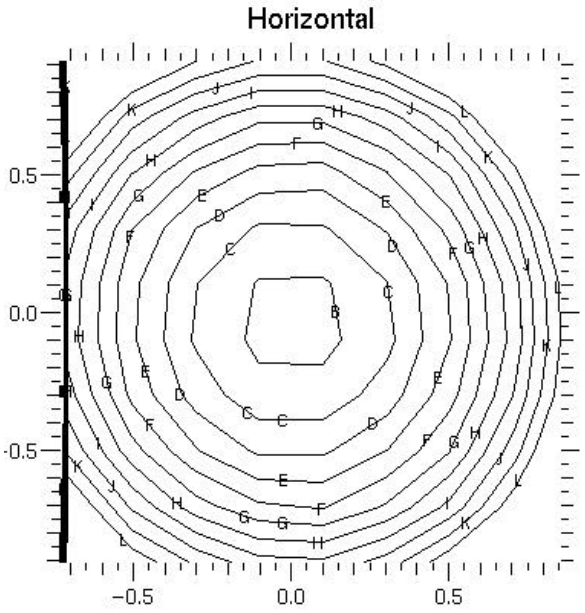

Figure 2 -Contour Plot of Received Power

6. Challenges in using Numeric Python

Although we were strongly motivated to use Numeric Python for this project, we did have to overcome several significant challenges.

6.1 Documentation needs improvement.

Until I attended David Ascher's tutorial - "An Overview of the Numeric Extensions to Python" at the 7th International Python Conference, it was difficult to learn Numeric Python. The available documentation has been improving, but learning Numeric Python is still not easy. The scarcity of Numeric Python documentation has made it harder to train other potential users. In particular, there is very little documentation on writing Numeric Python extensions. We had to carefully study existing code to understand extension programming. More Numeric Python "How-To" documents are needed.

6.2 Unwanted type promotion

The current implementation of Numeric Python has the undesired side effect of automatically producing double precision arrays when performing calculations on single precision variables. For example:

```
>>> x= ones((10,),'f')
>>> x.typecode(), x.itemsize()
('f',4)
```

```
>>> y = x * 2
>>> y.typecode(), y.itemsize()
('d',8)
```

Note that 'y' is a double precision array, even though 'x' was a single precision array. Since we simply could not afford to double the memory usage of our program when manipulating multiple 20-megabyte data-sets, we had to consistently use the 'SameSizeAs()" procedure recommended by David Ascher.

```
def SameSizeAs(input, ref):
    return array(input, ref.typecode())

>>> y = x * SameSizeAs(2,x)
>>> y.typecode(), y.itemsize()
('f',4)
```

Using the SameSizeAs() routine eliminates the "up-casting" problem.

Numeric Python programs would be easier to write and understand if each programmer did not have to use such "helper" functions to avoid unwanted up-casting. These helper routines tend to obscure the algorithm. Besides using these "helper" functions, I wrote my own version of the interp() function in arrayfnmodule.c that returns a single-precision array, rather than the default double precision arrays.

6.3 CXX Parameter Type Checking Can Yield Confusing Diagnostics

Using CXX allows automatic type checking of parameters using C++ exceptions to signal the Python runtime that an error has occurred. However, if a set of parameters is checked by a single C++ "exception handler", this technique can easily produce obscure runtime error messages. Since only a single generic error message is returned, the caller of the extension may not know which parameter is incorrect. We found it difficult to find parameter errors in our own Python code calling our own CXX extensions, given these vague error messages. The obvious solution is to use an exception handler for each parameter, even though it does tend to clutter up the C++ code. It would be helpful if the CXX documentation had better examples of such error checking.

6.4 Reference counting can still be tricky.

Although CXX handles most of the reference counting details, we still encountered reference count issues when using the Numerical Array API with CXX (see section 4.2). We found using the Python function *sys.getrefcount()* invaluable in finding reference count errors. For example, when one of our C++ extension functions returned a dictionary, we invoked *sys.getrefcount()* for each key in the dictionary, and verified the reference count was correct. (Otherwise, each call to an array processing routine can "leak" memory, causing the application's memory usage to grow at an enormous rate!). When debugging memory usage in PERP, we have found it helpful to use Insure++[2]. We also found it valuable to link Python or our "embedded" Python application with the "*Electric Fence*"[13] library to verify our use of memory. *Electric Fence* uses the computer's virtual memory hardware to stop a program at the moment it references memory outside valid regions.

6.5 Debugging an extension can be tricky

Python supports dynamically loaded extensions. If possible, extensions should be built as a dynamic extension, so that any application can use them. However, when debugging a dynamically loaded extension in the standard Python interpreter, it is necessary to set a breakpoint in the *PyImport_LoadDynamicModule* routine and repeatedly stop at this breakpoint until the extension has been loaded. Once the extension is loaded, breakpoints can be set. We found it much easier to debug our extensions by building a test program that statically links our extension with the Python interpreter. In this "embedded Python" environment, we could immediately set breakpoints in any of our routines. Fortunately, SWIG makes it fairly simple to build both a shared library and a custom application with an embedded Python interpreter. (See my website for a sample makefile[14]).

6.6 Packaging is painful

One of the strengths of Python and Numeric Python is the availability of so many optional packages – e.g., for netCDF file access, plotting, and GUIs. However, sometimes the "do-it-yourself" nature of building an application using (Numeric) Python is frustrating. For example, how is the average programmer supposed to determine which of the six or more available Numeric Python plotting packages is best? In contrast, the plotting functions provided by commercial packages like MATLAB or IDL are highly polished and fully integrated with the product.

In addition, when trying to install PERP on another machine, we've found it somewhat painful to gather up the source code for all the separate packages to build on the new machine. We end up having to manually maintain an (ever-growing) list of packages needed by

PERP. To illustrate the problem, here's the current dependency tree for PERP:

```
SWIG
Python 1.5.2
Numeric Python
  netCDF package
      netCDF library
  yorick
wxPython
  wxWindows
    gtk+
      glib
    glcanvas
      Mesa
```

In contrast, if we were using a commercial package, we would simply install the vendor provided CD-ROM or tar archive, and install a tar file of our scripts and extensions.

7. More Benefits of Numeric Python

Numeric Python and its extensions greatly simplify writing data analysis algorithms. In fact, some algorithms became trivial – we implemented the core of the fuzzy logic routines in 80 lines of Python – no C or C++ was required[15]. Since Numeric Python encourages "array-at-a-time" computations, we found our applications are much easier to read, since the code does not contain the details of indexing arrays in 2 (or more) dimensions.

We also found it easy to synthesize test data using standard numeric python routines. This simplifies testing of user written routines. For example, we tested our pulse-pair radar processing code by generating a synthetic sinusoidal input. Having invoked a user-written extension, Numeric Python's ability to interactively access subsets of arrays and structure members is a great debugging aid. Rather than having to compile in debug print statements in our code, we simply used Python's "print" command. At first, when using Python's print commands, we were frustrated that Python would always try to print entire arrays, even if the array contained megabytes of data. However, even this default behavior could be modified, using routines in NumUtil.py[16]. With this utility code, the user can specify the maximum number of array elements that are printed. Arrays that exceed this size are printed in the form:

```
array((20,1000), f)
```

8. Conclusion

We found both CXX and SWIG extremely valuable in generating the Python specific portions of our C++ code. After learning these tools, we could concentrate on our extension's code, rather than generating our own Python<->C++ interfaces using Python's native C interface. CXX greatly simplifies access to Python's list, tuple, and dictionary data structures. SWIG greatly simplifies the code required to access C++ structures and classes from Python, and allowed us to easily interface our C++ routines to Python.

We were quite pleased with our use of Numeric Python to build our application. Even with the flexibility provided by Numeric Python, our application is quite efficient. The time our application spends in the Python interpreter is small (<10%) compared to the time required for data I/O and data manipulation in our C/C++ extensions.

9. Future Plans

We want to try one of the "packaging" options for Python, so that we can build a single executable for "production" use. Currently, it is fairly tedious to find and install all the Python packages we need to run the PERP package. We are considering using "squeezeTool.py"[17] to package our python scripts. However, we may still need to build an RPM file to install all the extension scripts and shared libraries needed for PERP.

Also, we are starting to write a radar data display package using wxPython[18] and Mesa[19]. This will allow us to view radar data from inside PERP with out having to write out the data to a file, and run a separate standalone program to view our radar data.

10. Acknowledgments

This research is sponsored by the National Oceanic and Atmospheric Administration/National Weather Service's Operational Support Facility (OSF). Thanks to Guido, all the Numeric Python Developers, David Beazley for SWIG, and Paul Dubois for CXX. I particularly want to thank the reviewers of this paper for their helpful suggestions.

11. References

(All URLs current as of November 1999.)

[1] Lawrence Livermore National Laboratory, *Numerical Python*, http://xfiles.llnl.gov/NumDoc4.html

[2] Parasoft Corporation, *Insure++*, http://www.parasoft.com/

[3] Rational Software, *Purify*. http://www.rational.com/

[4] Eric S. Raymond, *The Cathedral and the Bazaar.* http://www.tuxedo.org/~esr/writings/cathedral-bazaar

[5] David Beazley, *SWIG*. http://www.swig.org/

[6] Paul F. Dubois. *A Facility for Creating Python Extensions in C++*, Seventh International Python Conference, p. 61-68. available online from http://xfiles.llnl.gov/CXX_Objects/cxx.htm

[7] Todd Veldhuizen, *BLITZ*, http://oonumerics.org/blitz

[8] Kuck and Associates. *KAI C++*. http://www.kai.com

[9] Unidata, Unidata, *netCDF*, http://www.unidata.ucar.edu/packages/netcdf

[10] Konrad Hinsen, *Scientific Python*, http://starship.python.net/crew/hinsen/scientific.html

[11] NCAR Atmospheric Technology Division, *Doppler Radar Data Exchange Format*, http://www.atd.ucar.edu/rsf/dorade_software/dorade.ps

[12] Lawrence Livermore National Laboratory, *Yorick*, ftp://ftp-icf.llnl.gov:/pub//Yorick

[13] Bruce Perens. *Electric Fence*, http://perens.com/FreeSoftware

[14] Joseph VanAndel. *Sample Makefile using SWIG to build Python extensions and embedding Python in an application*. http://starship.python.net/crew/vanandel/ipc8/Makefile.txt

[15] Joseph VanAndel. *Fuzzy Logic Implementation in Python*. http://starship.python.net/crew/vanandel/ipc8/fuzzy.py

[16] Joseph VanAndel. *Numeric Python Utilities*. http://starship.python.net/crew/vanandel/ipc8/NumUtil.py

[17] Fredrik Lundh, *SqueezeTool*. http://www.pythonware.com/downloads.htm

[18] Robin Dunn, *wxPython*. http://alldunn.com/wxPython

[19] Brian Paul et al., *The Mesa 3D Graphics Library*, http://www.mesa3d.org/