DSC 2001 Proceedings of the 2nd International Workshop on Distributed Statistical Computing
March 15-17, Vienna, Austria
http://www.ci.tuwien.ac.at/Conferences/DSC-2001
K. Hornik & F. Leisch (eds.) *ISSN 1609-395X*

Octave: Past, Present, and Future

John W. Eaton*
Department of Chemical Engineering
University of Wisconsin-Madison
Madison, WI USA 53706

Abstract

This paper outlines the history and development of GNU Octave, an interpreter for a high-level matrix-based language for numerical computations. A number of undesirable features of the current implementation are examined, and proposals for future development are presented.

1 Introduction

GNU Octave is a high-level matrix-based language, primarily intended for numerical computations. It provides a convenient command line interface for solving linear and nonlinear problems numerically, and for performing other numerical experiments using a language that is mostly compatible with MATLAB.[1] It may also be used as a batch-oriented language.

Octave includes a collection of tools for solving common numerical linear algebra problems, finding the roots of nonlinear equations, integrating ordinary functions, manipulating polynomials, and integrating ordinary differential and differential-algebraic equations. It is easily extensible and customizable via user-defined functions written in Octave's own language, or using dynamically-loaded modules written in C++, C, Fortran, or other languages.

Octave is free software distributed under the terms of the GNU General Public License (GPL) as published by the Free Software Foundation (FSF). Sources,

*jwe@bevo.che.wisc.edu
[1] MATLAB is a registered trademark of the MathWorks, Inc.

mailing list archives, and other information are available on the web at http://www.octave.org.

John W. Eaton is the primary author of Octave, though many others have contributed significant amounts of code, documentation, ideas, and (of course) bug reports. Among the larger contributions are the statistics functions written by Kurt Hornik and his students at Technische Universität Wien and the linear control systems functions written by Scottedward Hodel and his students at Auburn University. Much of the basic numerical capabilities are based on well-known Fortran libraries such as ODEPACK [6], DASSL [11], MINPACK [9], LAPACK [1], and the BLAS [7, 3, 2]. Octave also relies on Bison and Flex (the GNU parser generator tools), the GNU Readline library (for command-line editing and history), and the Kpathsearch library (for quickly finding files in large directory trees).

Octave was originally conceived (in about 1988) to be companion software for an undergraduate-level textbook on chemical reactor design being written by James B. Rawlings of the University of Wisconsin-Madison and John G. Ekerdt of the University of Texas. We originally envisioned some specialized tools for the solution of chemical reactor design problems, probably in the form of a subroutine library in Fortran. However, we had often seen students struggle with Fortran programming assignments and spend far too much time trying to understand why their Fortran code failed and not enough time learning about chemical engineering. These experiences motivated us to attempt to build a much more flexible tool. We believed that with an interactive environment like Octave, most students would be able to pick up the basics quickly, and begin using it confidently in just a few hours.

We were somewhat familiar with MATLAB, having seen the original Fortran version around the same time that we were beginning to think about software tools for chemical reactor design. Additionally, a growing number of our colleagues were beginning to use the new commercial version of MATLAB as their primary computational tool. We eventually decided to implement something that would be mostly MATLAB-compatible so that our colleagues could switch to using Octave without having to learn a completely new language.

Although we chose to implement a nearly MATLAB-compatible language, we did not want to limit ourselves strictly to that language. From the beginning of the project, we planned to add various additional features to the language and hoped to be able to fix those things that we believed were misfeatures of the language or just simply bugs. Now it seems that the goal of "compatibility at a reasonable cost" has caused many problems for the project, as we will see in detail in Section 4.

Full-time development began in the Spring of 1992. The first alpha release was January 4, 1993, and version 1.0 was released February 17, 1994. Since then, Octave has been through several major revisions, is included with Debian, SuSE, and Red Hat Linux distributions. It has also been reviewed in the Linux Journal [10] and the Journal of Applied Econometrics [4]. The next section includes a more complete history of Octave's development.

Clearly, Octave is now much more than just another "courseware" package with limited utility beyond the classroom. Although our initial goals were somewhat vague, we knew that we wanted to create something that would enable students to solve realistic problems, and that they could use for many things other than

chemical reactor design problems. Today, thousands of people worldwide are using Octave in teaching, research, and commercial applications.

Because Octave may be freely distributed, it is difficult to estimate the total number of people who use it. There are at least four mirrors of the Octave FTP site, and the Octave sources are also available from the GNU FTP site and all of its mirrors (more than 75 advertised sites at last count). The FTP logs from our system alone showed more than 19,000 transfers of the Octave source in the first 950 days that Octave 2.0.x was available. Many Linux users receive Octave in binary form as well. No one really knows how many people using Linux are also using Octave, but some recent estimates place the number of Linux users at more than 20 million.

Just about everyone thinks that the name Octave has something to do with music, but it is actually the name of one of the author's former professors who wrote a famous textbook [8] on chemical reaction engineering, and who was also well known for his ability to do quick "back of the envelope" calculations.

2 Development History

I began working on the interpreter for Octave on February 20, 1992 while I was still finishing my Ph.D. thesis at the University of Texas. I had been experimenting with some C++ classes for numerical problems for about a year before that, but Octave would be my first real project in C++.

The early entries in the `ChangeLog` file show that progress was rapid, and by early July, Octave was able to solve some simple differential equations. By September, several other members of my research group began trying to use it (and finding lots of bugs). On January 4, 1993, I made Octave available for anonymous FTP and posted an announcement to several Usenet groups. By this time, Octave included about 100 built-in functions and 30 function files, and could handle most of the operators available in the then current version of MATLAB.

During this first year, I was the only person who had done any programming on the project. Today, it is hard to imagine a free software project intended for widespread use that would be developed in such isolation.

After the release, I began to receive feedback almost immediately, and I was encouraged by the level of interest. I continued to put new test releases on the FTP site for about another year, but made no more announcements outside of the Octave mailing lists until version 1.0 was released on February 17, 1994.

During the next year, many people reported bugs and requested features, and some sent patches or new functions. I periodically made snapshots available, but the next public release (version 1.1.0) was not made until January 12, 1995.

Development was interrupted in mid-1995 as our research group moved from Texas to Wisconsin. Much of my work on Octave over the next year focused on cleaning up the internals (plugging memory leaks, trying to use C++ more effectively, etc.), adding new functions, and making Octave ready for a new release. Octave had gained significant capability by the time version 2.0 was released on December 10, 1996.

For the next several months, I worked mostly to fix bugs, releasing versions 2.0.1 through 2.0.5 in fairly quick succession. I had hoped to quickly reach a mostly bug-free state so that I could begin concentrating on version 2.1 and adding significant new features, but that never seemed to happen. Eventually, I decided to follow the lead of the Linux kernel developers who had opted for stable and unstable development branches, and I split the 2.1.x sources sometime in mid-1997 and began making periodic releases from both the development and stable branches. The idea was to speed up the incorporation of new ideas in the unstable branch while maintaining the other branch for bug fixes only, but I found it difficult to deny useful new features to the users of the stable version, so I often spent extra time patching the stable sources to include new features. To avoid wasting time like this in the future, I would only attempt to have two actively maintained development trees if another individual (or group) were available to maintain the stable branch while I work with the development sources (I've tried to find volunteers for this task, but so far, no one has stepped forward).

In May of 1997, Richard Stallman and I agreed to make Octave an official part of the GNU project (the only real change was that I began calling it "GNU Octave" rather than just "Octave").

By early 1999, many free software projects began to be available by anonymous CVS, and in response to some discussion on the Octave mailing lists, I made a read-only copy of the Octave CVS archive available. The idea was that this would help to speed up Octave development by allowing people to see their changes in the source tree quickly, without having to wait for a new snapshot. I believe that Eric Raymond's paper, *The Cathedral and the Bazaar* [12] convinced many people that an open development model was the key to rapid development, but I don't believe it is that simple. Wide-open availability of source code doesn't mean much without a dedicated group of qualified hackers who are generally aligned toward a common goal.

At the time of this writing, Octave is a useful and widely used tool for numerical computing. As with any sizeable piece of software, however, there are many areas for improvement. I believe there are problems with the language that should be fixed (a number of these are examined in the next section). Octave also lacks a few important data types such as sparse matrices and multidimensional matrices. Some users would benefit from a version of Octave that could easily support parallel processing. Others would like to see more advanced graphics and graphical user interface (GUI) features. And, of course, many users request improved compatibility with MATLAB (we will examine this topic in more detail in Section 4).

On December 7, 2000, I announced to the Octave mailing list that I am planning to take a break from Octave, review the progress we have made, and decide what is the best course for the future. That may involve some derivative of Octave, or it may be something that is only loosely related. If it is a system based on Octave, then I would like for it to avoid many of the problems described in the next section.

3 Design and Implementation

In the following sections, various features of Octave are examined. Some are worth noting because they are valuable features that should probably be kept in any future Octave derivatives, and others are mentioned because they are the trouble spots that should be removed. In some cases, the troublesome features are quite popular with former MATLAB users, so I have tried to present clear arguments for why these features are really not so good after all. Although I would like to be able to say that all of the bad features of Octave were inherited from MATLAB, I must admit that quite a few of them are my own invention.

3.1 Free software

One of Octave's most important features is that it is free software. All of the source code is available, so users are able to examine exactly how each computation is performed. There are no "black-boxes" that cannot be opened. Users are also free to modify any part of Octave and distribute their changes (provided that they follow the terms of the GPL). I believe that it is important for everyone, especially researchers and academics, to have access to the details of all computations they perform, so that it is possible for the results to be reviewed and reproduced.

3.2 C++ as implementation language

Octave is written in a mixture of C++, C, Fortran, and Octave's own extension language. Only a few new functions have been implemented in C or Fortran. Most of the interpreter is written in C++, and I am still not sure that this was the best choice. When I began work on Octave in early 1992, a draft ANSI/ISO standard for C++ was in the works (the Annotated C++ Reference Manual [5] had been published in 1990), but the language was still evolving. Because vendor C++ compilers varied widely in their capabilities, I decided that it would be too difficult to write code that could be compiled by very many of them. Instead, I chose to try to make Octave work only on those systems that had working ports of the GNU C++ compiler. This choice made portability somewhat easier, but a significant amount of effort was still required to keep up to date as new language features were added and some old ones removed, invalidating previously working code. Each new release of the GNU C++ compiler seemed both a blessing and a curse.

I believe that using C instead would not have been much better, however, as I much prefer the features of C++ classes over what is available in C, and they have made memory management easier. The string class of the standard library has been quite helpful, and templates have also been useful in defining Octave's low-level array classes.

3.3 Application layers

Octave has a fairly clean separation between the interpreter and the libraries that implement the basic numerical functions. There are three main layers:

- Parser and interpreter, mostly written in C++. This is built in the form of a library (`liboctinterp`) plus one small file (about 500 lines) containing the main Octave program.
- The `liboctave` library, a C++ interface to numerical code and to Unix library functions and system calls. This library provides the matrix and array classes for Octave and presents a nice interface that hides details such as work arrays and many other memory management and portability problems from the interpreter.
- The `libcruft` library, which is a collection of mostly Fortran code written by others that provides most of the base numerical capability.

I believe that this division is reasonable, but some refinement would not hurt. For example, the `liboctave` library should probably be divided into at least two parts, one for numerical tasks and one for the operating system interface, so that users who only want the numerical capabilities would not also have to link code for command line editing, file searching, and Posix functionality. The `liboctave` library should also be examined and revised where needed to present a consistent interface for programmers wishing to access Octave's functionality at this level. It might also be useful to define a public interface to the interpreter library `liboctinterp`, so that the Octave interpreter could more easily be embedded in other applications.

3.4 Interpreter implementation

The design of the interpreter was originally patterned after the GNU shell Bash, primarily because I had the source code for Bash and found it reasonably easy to understand. The parser converts source code to an internal tree structure. Functions are stored as lists of statements along with information about input and output parameters and local symbols. The interpreter evaluates code by looping over the elements in a list of statements, and by recursively evaluating nodes in expression trees.

The lexer and parser are written using Flex and Bison, which make them relatively easy to modify and extend. There is also a C++ class interface to the parse tree that provides a clean interface for walking the tree. This is currently used for pretty-printing functions and for setting breakpoints for Octave's interactive debugger.[2]

Using a simple tree-based interpreter seemed easy to implement and a good way for me to make it reliable, though I suspect that it is one reason that Octave's interpreter is slow.

Paul Kienzle, a Octave user and contributor, recently compared Octave's performance on a loop of the form

```
x = [1:100000];
total = 0;
```

[2]This feature is currently under development.

System	CPU Time (seconds)	Method
Octave	24.	script
Octave	6.9	C++ using operations on `octave_value`
Octave	0.032	C++ using operations on `Matrix`
Guile	1.3	script
clisp	3.3	script
clisp	0.70	byte-compiled on load
Python	0.73	script
OCAML	0.17	script
OCAML	0.034	compiled script
GCC	0.20	unoptimized
GCC	0.0060	optimized

Table 1: Performance of Octave compared to other interpreters and compilers.

```
for i = 1:100000
  total = total + x(i);
end
```

and obtained the results shown in Table 1. Clearly, there is a lot of room for improvement in the implementation of loops and array indexing in Octave.

At this point, I believe that it may make more sense to modify Octave's parser to emit Scheme or some other general-purpose language for which a fast interpreter exists, and then execute the code within that context instead of trying to improve the performance of Octave's interpreter. The motivation would be to improve the speed and reliability of Octave's interpreter while at the same time reducing the effort required to maintain it. Presumably, the maintainers of the general-purpose scripting language would be experts in interpreter design, which would leave us more time to contribute numerical functionality instead.

3.5 Dynamically-linked functions

Although Octave is extensible using the Octave language, it also allows extensions to be written in C++ (or other compiled languages that can be called from C++). Users are generally discouraged from writing extensions as dynamically-linked functions unless acceptable performance cannot be achieved using Octave's own extension language, or there is a need to link to existing compiled code. Nevertheless, this is an important feature for cases when performance is critical, or when code already exists in some other language.

3.6 Extensible data types

Octave also allows users to define new data types and load them at run time using a dynamically-linked function. Currently, the new data type must be defined by a C++ class, but it would be useful to provide a way of doing this within Octave's

language. The MATLAB language now allows the creation of new data types, but the method for doing so does not seem to be particularly well designed, and probably should not be copied.

3.7 Mixed-type operations

Binary operations in Octave include mixed-type operations for many data types. For example, there are special functions for operations on complex and real matrices. Saving memory and avoiding unnecessary copying of large amounts of data are the motivations for providing these extra functions rather than simply promoting the operands to a common type. In a scalar-based language, one can afford to define operations on similar types only, because the cost of converting an operand to the wider type is small. But when working with large arrays the cost of converting an operand can be quite large.

One problem with this approach is that as new data types are added, there is an explosion in the number of new functions for mixed-type operations that need to be defined. It may be possible to get around this by some clever use of templates, but I still see this problem as unsolved.

3.8 Data type conversion

Octave can automatically convert objects of one data type to another when it seems reasonable to do so.

One example of the type of conversion that Octave can perform is a narrowing conversion that may happen at object creation. For example, given the complex numbers `x = 1+i` and `y = 1-i`, the type of the result of the expression `x + y` will be real, not complex. The conversion happens when the new object that holds the result is created. The result is originally complex, but the constructor for the complex scalar object checks to see if the imaginary part is zero, and if so, converts the result to a real value. Similar conversions can happen for complex matrices. There is some overhead for these narrowing conversions, but there is also a potential benefit if subsequent operations can be performed using less expensive operations on the narrower type.

Octave also performs all arithmetic in double precision (there are no integer values with the integer arithmetic rules used in languages like C or Fortran) so integer values are converted to double precision values when they are parsed. As a consequence, the expression `1/2` produces the value 0.5, not 0. This type of conversion makes some sense, because the value of 1/2 is one half, not zero. It would be equally valid for the type of the result to be rational rather than a double precision. It is not clear whether the lack of integer arithmetic is a serious limitation. It does not seem to be much of a practical problem for the types of programs that are typically written using Octave.

Currently there are no integer matrices in Octave, but if they are added, some design decisions must be carefully considered. For example, should results for all binary operations be converted to doubles? If the results are all integer values and no overflow occurs, it would be more memory efficient to return another integer matrix.

If result of the operations on some of the elements could produce non-integer values or overflow, then should the results follow the rules of integer arithmetic used by other languages (this would be inconsistent with the behavior for scalars, unless those rules are also changed) or should the results be double precision values?

Perhaps the simplest approach would be to disable all automatic conversions, and implement integer arithmetic. Unfortunately, this may not be the most useful design for the typical user of a system like Octave.

3.9 Parser problems

There are a number of difficulties in parsing the MATLAB language. Because Octave tries to be compatible in many ways, Octave's parser is significantly more complex that it really ought to be. By dropping some of the more inconsistent language features, Octave's parser could be much simpler and easier to understand and maintain.

Transpose operator and character strings. The single quote character (`'`) is used as the transpose operator and to delimit strings, so the parser must do a lot of extra work to keep track of when the single quote should be recognized as a transpose operator or when it should be recognized as a string delimiter.

The language would be more consistent if the single quote character were used as a transpose operator only. This would allow Octave's parser to be significantly simpler and also make parsing easier for humans.

Whitespace in matrix lists. Another example of unnecessary parser complexity is the feature that allows matrices to be written

```
[ 1 2 3 ]
```

which is interpreted as a vector of three elements. For simple cases like the one above, this seems fine, but for more complex cases, it is ambiguous. For example, it is not immediately clear whether

```
[ f (x) -3 ]
```

should be equivalent to

```
[ f, (x), -3 ]
```

(three distinct elements) or

```
[ f(x), -3 ]
```

(two elements—a function call or indexed array followed by a scalar) or

```
[ f, (x)-3 ]
```

(two elements—a variable (or function call, since MATLAB does not require function calls with no arguments to be written as `f()`) followed by a binary expression) or

```
[ f(x)-3 ]
```

(one element, the result of subtracting a scalar from the result of the function call or indexed array). I also see no reason for the meaning of the expression

```
f (x) -3
```

to change when it is surrounded by square brackets. For any future project, I would have the parser ignore spaces inside matrix lists and require separators between elements.

Command-style function call syntax. MATLAB allows functions to be called using a command-style syntax. For example, the function `save` can be called using the syntax

```
save -ascii var
```

or

```
save ("-ascii", "var")
```

Both forms are equivalent. Users may also define functions in the MATLAB language that may be called this way. No special function declaration is needed, the only requirement is that the function be able to accept all arguments passed as strings. Although this seems like a nice feature, it leads to an inconsistency that would be better avoided. The problem is that, given a function called `fcn`, MATLAB will parse both of the following statements

```
fcn-1
fcn- 1
```

as a subtraction of 1 from the value returned from the function `fcn`, but will parse

```
fcn -1
```

as a call to the function `fcn` with the string argument `"-1"`, and it will parse

```
fcn - 1
```

as a call to the function `fcn` with the string arguments `"-"` and `"1"`. Even worse, MATLAB won't allow you to bypass the confusion by writing

```
f() - 1
```

because it doesn't allow this notation for calling a function with no arguments (the MATLAB parser emits an error for the empty pair of parentheses).

I believe that this kind of inconsistency is unacceptable in a programming language. For this reason, Octave does not currently allow users to call their own functions using a command-style syntax, though it does have some special built-in functions that may be called this way (e.g., `load`, `save`, `ls`, etc.).

The command-style syntax is useful, however, and many users find it awkward to have to type things like

```
ls ("-l" "/foo/bar")
```

when they are used to typing just

```
ls -l /foo/bar
```

at a shell prompt. Because the language has some commands that can be called with a command-style syntax, it seems inconsistent to forbid users from defining their own functions that can be called using a similar syntax. For any future Octave-like project, I propose requiring that function call syntax (i.e., `foo ("-arg")`) be used in scripts or function files, and then requiring a prefix of some sort that can be used on the command line to introduce command-like syntax. For example, something like

```
octave:13> M-x ls -l /foo/bar
```

similar to the way Emacs uses the prefix *M-x* (or *ESC* `x`) for calling functions interactively. This solution would completely avoid the conflict, and would not be much harder to type. I don't believe that forcing the use of function call syntax (i.e., `ls ("-l", "/foo/bar")`) in scripts and function files would pose much of a problem (it does not seem to be a problem for users of Emacs and Emacs Lisp).

3.10 Built-in variables to control semantics

The Octave language has a number of built-in variables that may be used to control various features of the interpreter and even the language. In some cases these variables are relatively harmless, but in others, they can cause significant trouble.

Among the least harmful variables are those like `history_size`, which specifies the number of command lines to save in the command-line history buffer, or `PAGER`, which specifies the program to use as a filter for output that is going to the screen in interactive sessions. Variables such as these only affect user interaction and can be safely ignored when writing new functions or scripts.

Another set of variables affects the behavior of the parser. For example, the variable `whitespace_in_literal_matrix`, is used to control the way that Octave parses matrix lists. Continuing the example from the previous section, consider the matrix list

```
[ f (x) -3 ]
```

If the value of `whitespace_in_literal_matrix` is `"ignore"`, then commas or semicolons are required to separate elements in the list, and all whitespace is ignored. If the value is `"traditional"`, then Octave parses the command as MATLAB would, which would produce

```
[ f, (x), -3 ]
```

(three distinct elements). Otherwise, Octave will not treat spaces between an identifier and an open parenthesis character as an element separator (this is the way Octave originally behaved, before someone noticed that it was not compatible with MATLAB).

The variable `whitespace_in_literal_matrix` was added so that users could choose whether they wanted Octave to work as it always had, or to behave in a MATLAB-compatible way, or to require users to explicitly separate elements in matrix lists. Unfortunately, this means that to write scripts that can be interpreted correctly regardless of the value of this variable, one must avoid spaces between identifiers and open parenthesis characters (or surround the expression in another set of parenthesis, like this: `(f (x))`) and use commas and semicolons to explicitly separate elements.

The most problematic set of variables affects run-time behavior. Most of these were introduced to allow compatibility with what I believed to be misfeatures of MATLAB while also allowing Octave to behave in a more consistent way. Here are two examples of this type of variable.

The function `zeros` may be used to create a matrix filled with zeros. It can be called in several different ways. The simplest is

```
zeros (n)
```

which will create a square matrix with dimension `n` that is filled with zeros. If the argument is a real number but not an integer, both MATLAB and Octave silently round to the nearest integer. If the value of the argument is complex, MATLAB silently ignores the imaginary part, and if it is negative, MATLAB treats it as zero.

After I had already implemented the zeros function in Octave with rather strict requirements for the argument to be an integer, I noticed these other odd behaviors and decided to allow them. In the case of non-integer values, I assumed (incorrectly?) that it might be fairly common for the dimensions to be computed by some method that would produce a non-integer value, so I simply allowed Octave to behave as MATLAB does. But for the cases of complex and negative argument values, I decided to make compatible behavior optional through the variables. `ok_to_lose_imaginary_part` and `treat_neg_dim_as_zero`.

I now believe that this is an example of compatibility gone too far. It seems to me that simply rejecting any non-integer argument is reasonable. Anyone that needs to pass a complex (or any non-integer) value to `zeros` (or similar functions) could do so by explicitly converting the argument to an integer. My guess is that this would not cause trouble for much existing code.

3.11 Overloading of ()

In MATLAB and Octave, parentheses are used to delimit argument lists for function calls and index lists for array indexing. In the expression

```
x (y, z)
```

it is not immediately clear whether the symbol `x` is a function that is called with two arguments, or an array that is indexed by two values. Although this is virtually impossible to change without breaking nearly all the code written for Octave up until now, it would be nice to be able to distinguish function calls from array indexing by syntax alone.

3.12 Syntax problems for matrix lists

The square brackets used to introduce matrix lists are limited to two dimensional matrices. Originally, MATLAB only had two-dimensional matrices, and arrays of three or more dimensions were not allowed. Even in the current version of MATLAB there is apparently no list-like syntax for introducing a matrix of more than two dimensions. Similar limitations appear to exist for MATLAB's "Cell" data type as well. Because the matrix list notation occurs in much of the code written for Octave, it would be quite difficult to remove this restriction without breaking many existing programs.

3.13 Flat namespace

Octave has no good way to control namespaces. All functions share the same namespace, and conflicts are likely. MATLAB does allow private functions which are only visible as subfunctions in a single function file, but that seems too limited. I believe that proper support of namespaces would be useful, particularly as the number of functions increases, though it may not be absolutely necessary if some care is exercised by people writing functions for others to use. Emacs, which has many thousands of functions written for it that are shared by many people, has no provision for namespaces other than some suggested naming conventions for functions and global variables.

3.14 Gnuplot plot command syntax

Octave has a tightly coupled interface to gnuplot that is implemented by duplicating much of the gnuplot command syntax directly in Octave's parser. At first, I was not sure it would even be possible to implement this feature at all because it seemed to create too many conflicts in the parser, but I was eventually able to make it work. I believe that Octave would be better off today if I had failed and given up, but after some discussions with others, I was convinced that the problems could be solved with enough code. Unfortunately, I lacked the experience to see how much trouble it would really cause.

Obviously, the gnuplot-style commands that Octave understands are fairly simple to convert into commands that can be passed to gnuplot. It would be much more difficult to convert them into commands or subroutine calls to another plotting package, however, so replacing gnuplot by another plotting packing is problematic, because it implies a large effort or breaking backward compatibility. Probably it would be best to drop this functionality in any future project.

4 MATLAB Compatibility Issues

I originally believed that compatibility with MATLAB was an important and reasonable goal (but certainly not the most important one). I believed that it would help people begin using Octave by making it easier to learn and by making it possible for them to use code they had developed for MATLAB without having to make

substantial changes. Unfortunately, attempting to provide good MATLAB compatibility has also caused many problems and diverted resources away from improving the capabilities of Octave in other more important areas.

Because of its close compatibility with MATLAB, Octave attracted many MATLAB users. Some found features that were not supported, or behaviors that were inconsistent, and reported them as bugs. These problem reports were often given a relatively high priority, particularly when the problems were either minor or seemed gratuitous. Although I did not intend to, I believe that by fixing these problems and making Octave even more compatible with MATLAB, I encouraged people to think that a primary goal was to create an exact replica of MATLAB.

I no longer believe that compatibility is a reasonable goal. It stifles innovation and places the project in the position of never being a leader, always being a follower, and quite far behind almost all the time.

If compatibility is a goal, almost no innovation is allowed, because there is the potential for any new feature to be incompatible with future versions of MATLAB. (This is not just an imagined problem—it has happened in the past, and I expect it will happen again.)

When incompatibilities like this arise, there are several choices. One is to add the new MATLAB-compatible feature and still support the old way too. Even if this is possible (it may be that the features simply cannot coexist), it adds additional maintenance costs because one must now support two different ways of doing essentially the same thing. Another choice is to adopt the new way and drop the old, but breaking backward compatibility is usually seen as undesirable. Ignoring the new feature is also possible, but tends to result in bug reports asking why there is an incompatibility (typically the assumption seems to be that whoever implemented the Octave feature somehow got it wrong).

Deciding that it would be best to avoid innovation and focus on complete compatibility is also a difficult path to choose. Because there is no standard for it, the definition of the MATLAB language is whatever behavior the current version of MATLAB happens to exhibit. Attempting compatibility requires following a moving target, and because The MathWorks almost never openly discuss their development plans in advance of new releases, the compatibility effort will be years behind at each new release of the target system.

There are also some non-technical problems associated with striving for compatibility with a system like MATLAB. I believe that many people have started using Octave because they see it as a "MATLAB clone" available free of charge rather than a freely available system for doing useful work. Often, these users have assumed the role of "customers" who would ask for help, a bug fix, or a new feature, then wait for a solution instead of contributing. I don't believe that this type of behavior is the norm among successful free software projects. Indeed, I believe that for a free software project to be successful, it must have many competent users who also function in the role of developers. Although there have been some significant contributions, Octave has never developed a strong core of dedicated and competent developers.

5 Future

Immediate plans for Octave include the release of a new "stable" version that I will maintain to fix serious bugs only. If people are interested in having some form of Octave continue along its previous development path, they will need to organize that effort—my focus is now on future directions.

Although I will not be working much with Octave as it is currently implemented, the future may involve a derivative of Octave. If the problems described in Section 3 are fixed in the ways that I have proposed, I believe Octave will provide a much-improved language. It will remain similar to MATLAB, but incompatible. It will be easy for users of one language to understand the other, but, in most cases, difficult to share code between the two systems. If these changes are made, I can see significant advantages and disadvantages. On the positive side, we will have a more consistent language and an interpreter that is easier to maintain, and there will be less effort wasted on the implementation of poorly designed features just to have compatibility. Some possible disadvantages are that people will generally have to choose one system or the other, we lose some users who see compatibility as an absolute must, and we invent yet another scripting language.

Another possibility for the future is to leave the Octave language alone and focusing on additional features (sparse matrices, parallelism, etc.) while fending off the requests for additional MATLAB compatibility. But as long as Octave is generally compatible, users are likely to continue to want more, and I would prefer a clean break from this problem.

If it is undesirable to invent yet another scripting language, then perhaps it would be best to simply choose another general purpose scripting language that already exists and add the numerical features from Octave to it. I was asked about a merger of this type for Octave and Python, but as I examined Python more closely, I decided that it did not seem to be well-suited for numerical computing (though many seem to be using it for that purpose anyway). Other popular scripting languages have similar problems.

At this point, I'm leaning more toward a streamlined Octave built on top of some faster general-purpose scripting language such as Guile, the GNU project's embeddable Scheme interpreter. This would preserve the syntax of Octave (which seems to be reasonably good for expressing numerical algorithms) and allow access to the capabilities of the more general purpose language. Reasons for choosing Guile are that it is intended to be used as an embedded interpreter, and it should not be too difficult to translate Octave to Scheme.

6 Final Comments

Some comments in this paper (particularly those related to the problems with MATLAB compatibility) may seem quite pessimistic, but I don't think that the situation is hopeless. Those of us who care about free software tools like Octave should seek to avoid these problems by not making compatibility with proprietary software a primary goal, particularly if doing so comes at a very large cost. We must be-

come leaders rather than followers and develop a vision beyond reimplementation of existing proprietary tools.

To be successful, a project like Octave needs to have many contributors aligned with a common goal. I would like to see that goal be a freely available, high quality system for numerical (and possibly symbolic) computations that inspires people to contribute innovative ideas and code.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenny, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philidelphia, 1992.

[2] Jack J. Dongarra, Jeremey Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[3] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, March 1988.

[4] Dirk Eddelbuettel. Econometrics with Octave. *Journal of Applied Econometrics*, 15:531–542, 2000.

[5] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, 1990.

[6] Alan C. Hindmarsh. ODEPACK, a systematized collection of ODE solvers. In R. S. Stepleman, editor, *Scientific Computing*, pages 55–64, Amsterdam, 1983. North-Holland.

[7] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, September 1979.

[8] Octave Levenspiel. *Chemical Reaction Engineering*. John Wiley and Sons, New York, 2nd edition, 1972.

[9] Jorge J. More, Burton S. Garbow, and Kenneth E. Hillstrom. User guide for minpack-1. Technical Report ANL–80–74, Argonne National Laboratory, March 1980.

[10] Malcolm Murphy. Octave: A free, high-level langauge for mathematics. *Linux Journal*, July 1997.

[11] L. R. Petzold. A description of DASSL: A differential-algebraic system solver. In R. S. Stepleman, editor, *Scientific Computing*, pages 65–68, Amsterdam, 1983. North-Holland.

[12] Eric S. Raymond. The cathedral and the bazaar. Web document, available from http://www.tuxedo.org/~esr/writings/cathedral-bazaar, 1997.