



---

# *Journal of Statistical Software*

MMMMMM YYYY, Volume VV, Issue II.

<http://www.jstatsoft.org/>

---

## 19 Ways of Looking at Stat-Ware\*

**Micah Altman**

Harvard University

**Simon Jackman**

Stanford University

---

### **Abstract**

In this paper we reflect on on the crucial contributions innovations in statistical software have made to political methodology. And wee identify principles for writing statistical software with maximum benefit to the scholarly community.

(\* With apologies to Wang Wei and Eliot Weinberger.)

*Keywords:* resampling , statistical computation, programming methods .

---

## 1. Introduction

People who read journals like this are continually creating snippets of code. We can hardly avoid it. Anyone who performs statistical analysis on a regular basis naturally encounters repetitive tasks that beg for automation, data that needs to be prepared in new ways for analysis, and models that cannot be estimated well with canned statistical packages. So we write code – to automate tasks, manipulate data, extend existing methods of analyses, and in to create new ones.

Most of this code is never seen by anyone else. Much of it evaporates soon after its task is completed. Without doubt, a good portion of code deserve this fate. However the rest is useful, and in general continues to persist for a time, while gradually ossifying or mutating until eventually, either lifeless or monstrous, it is buried in an unmarked grave. This is a lost opportunity – with a small additional effort this code could be shared, and lead long healthy lives in service to the community.

We write this article to identify some principles for writing statistical code that benefits the community. This is based on our own experience writing statistical code professionally, and having closely observed many others creating it. We identify three groups of principles and conclude by reflecting on the contributions that statistical software makes to the study of politics.

## 2. Five Motivations for Writing Statistical Software

*Solve a problem.* Start from a real problem that you need to solve. It doesn't have to be a big problem, but it should be one for which a good solution does not already exist. Before you write code do some research: Check books, documentation, and software archives that could contain solutions to your problem. Software development tends to involve hidden complexities, so avoid building a solution from scratch if an adequate solution exists, which can be used or improved. When existing software fails to do what you want, or is too inaccurate, slow, tedious to use, or awkward to integrate into your larger research work-flow, it is then time to build.

*Do good.* Code that solves your problem could often be useful to others. This is especially likely whenever you implement a statistical analysis that is not available in canned statistical packages. Think about the problem that your code solves – is it unique, or are there other problems like it that real people are actively trying to solve? Can your code solve these problems too? Can it be extended easily? Change your code if you can easily solve more real problems for real people by doing so.

*Build incrementally, based on use.* Document your code so that others can use it. Then make it available through a code archive, and pay close attention to whether people use it, what they do

with it, and where they encounter problems. Pay attention to suggestions from users and other developers.

*Understand your problem.* When you solve a problem by writing software you test both your knowledge of both the problem and the adequacy of your proposed method of solving it. As Don Knuth famously <sup>1</sup> wrote: “It has been often said that a person does not really understand something until he teaches it to someone else. Actually a person does not really understand something until he can teach it to a computer, i.e., express it as an algorithm.”

*Get credit.* For many of us, credit is money. Making your code useful to and available to others is an excellent way of making it possible for people to try a method that you have developed, and to aid in the replication and extension of work with which you are involved. And work that is replicable is more likely to be used and cited more. [Gleditsch (et. al 2003)] For example, King, et. al’s (2000) tremendously influential article, which eloquently advocated the use of simulation to improve the interpretation and presentation of statistical analyses, comprised simulation methods already well-known by methodologists. It had a striking impact in large part because the authors simultaneously developed, discussed, demonstrated, and distributed the “CLARIFY” package, which made it easy for other members of the discipline to apply these techniques. <sup>2</sup>

Citing software, particularly when using more complex models, is a best practice, and is essential to ensuring the replicability of research results [Altman, et. al 2003]. Encourage users to cite your software directly by supplying a clear citation for your work. ( When programming in R, you should provide a CITATION file that clearly documents how your module should be cited. )

*Make money.* Greed is good. Most of us, however, will not get rich selling statistical software. Statistical programming can contribute to your economic well-being in other ways: It can allow you more easily to fulfill consulting requests which involve applying a method, or extend your software to encompass new methods. And there is always the remote possibility of that eight figure movie deal coming through ...

### 3. Eight Ways to Make Your Code More Useful

Applied in moderation, all of the following techniques will aid in the reuse and value of your code, by preventing ossification, confusion, and uncontrolled mutation. These techniques will make your code easier to use, lend more confidence to the results, and enable others to add to it without disruption. And even if you never share it with anyone outside of your office – your code will still

---

<sup>1</sup>For some definition of ‘fame’.

<sup>2</sup>The software itself was later published separately as Tomz, et. al (2003).

benefit from becoming more reliable, easier to understand, to maintain and to extend.

*Document algorithms, implementation, and use.* Insufficiently documented code and algorithms are indistinguishable from magic <sup>3</sup> (which is no longer considered a sound basis for inference).<sup>4</sup> It also goes without saying (although we'll say it anyway) that if you expect others (or yourself, after you have moved on to future projects, and, inevitably, forgotten the details of the current one) to use your software, you should document how and why to use it. What is often overlooked, is documentation that systematically and explains when *not* to use the software. Documentation should include known limitations of the output and implementation, such as degradation in accuracy outside a certain range of inputs. All algorithms and implementations are limited, and experienced (or hard-bitten) users are justly suspicious of programs that do not admit to their own limitations. [See Altman, et. al 2003 for dramatic examples of what can go wrong when these considerations are omitted.]

*Choose algorithms appropriately.* To make correct inferences requires the convergence of relevant statistical theory, informative data, well-chosen algorithms, and correct encoding of those algorithms using a programming language. The most sophisticated algorithm need not be used to solve the problem, as long as it is not horribly inefficient, and is demonstrably accurate enough to produce an answer to the required tolerances. However, avoid algorithms with known performance and numerical problems, or with unknown accuracy. And, again, always document the algorithms you have chosen.

*Design programs for use and reuse.* A number of programming techniques are well-understood to lead to more durable, reusable code: Modular organization of files, clean separation of functionality into components and classes, interface encapsulation, naming & style conventions, and consistency applied to all levels of design and implementation. (See McConnell 2004 for a detailed summary of these methods and the results of applying them.) Furthermore, you will benefit by becoming familiar with the conventions and idioms that have been developed within the communities that will be using your work. A little time spent in preparation for writing code, skimming previous discussions on the developers' and users' mailing lists, and examining the code from other projects, will save a great deal of time later in debugging, rewriting, and documentation.

*Program defensively.* Users (and other programmers) are imperfect. They don't always read documentation, and when they do they don't always understand it. At some point, your program is going to be asked to do the impossible. When the impossible happens, your program should not explode, or worse, produce plausible nonsense, but should instead complain noticeably and informatively.

---

<sup>3</sup>Apologies to Arthur C. Clark.

<sup>4</sup>For an opposing view, see [Abramson 1997].

In particular, you should adopt at least the following defensive programming techniques:

- **Check inputs and outputs:** Check all inputs values received by every external interface (this includes any public library procedure or object method you supply). Provide reasonable default values for inputs where possible. Document the valid range of inputs for your code, and check these. Explicitly check that the output your program or function is producing is itself valid. (Check outputs even if you know this should be true by construction. Sometimes limits in theory, algorithm, or implementation causes the ‘impossible’ to occur.).
- **Report any problems the code encounters:** For non-fatal problems, warn the user through the system’s standard warning facility. For fatal problems, throw an exception (and document which exceptions your code will throw).
- **Avoid coding for a single system:** Although outside of large-scale commercial development, it probably not possible to make sure your program runs on all other platforms and system configurations (such as the Estonian version of Windows XP Home Edition, service pack 3a) one should avoid inasmuch as possible the assumption that system on which your program runs is identical to your own. Avoid using system-dependent values, and make full use of system-independent interfaces. For example, in R use the `file.*` family of functions to manipulate files and paths, rather than using hard-coded paths or system-level commands.

Not only will following these methods prevent problems from being falsely attributed to you and your code, they will protect you from many of your own errors: Research on software quality and defect removal clearly shows that these techniques lead to the production of code with drastically fewer errors [McConnell, 2004].

*Use version control.* Inevitably, at some point in your project you are going to discover that because of some change that was made to your software, something important that used to work no longer does. Version control provides a safety net, since you can use it to restore any previous version – particularly those that worked. Version control also provides a diagnostic tool, since you can use it to see exactly what was changed between the working and non-working versions. Finally, version control supports replication – since it ensures that any version of the code used to produced a published analysis remains available.

*Write tests early.* You should provide tests that verifies correct output based on known input. This makes it possible for others to use and extend the code with confidence. A number of modern software engineering techniques go so far as to advocate that complete sets of tests should be developed prior to any coding, then used as an indicator of the completeness of the

implementations. There are many frameworks available for automating tests. For example, in R writing can be as easy as specifying a set of code to run, and the providing a copy of the expected output, and putting these in the appropriate module directory.

*Measure accuracy.*

Unlike most software, which simply fails to produce results when broken or used improperly, statistical code almost always produces output with some gloss of plausibility. Thus, stable algorithms, conservative implementation, and full documentation are vital to producing trustworthy statistical software. Formal benchmarks, testing in extended precision environments, and sensitivity analyses are vital to verifying that trust. Trust, but verify.

*Provide an open source license.* To put it simply, give people *permission* to use your software. Code that you write is automatically copyrighted, and without a license others cannot have confidence that they can reuse the code, or even share the results that it produces. Using a standard open source license (such as the GPL, or other OSS approved license) allows others to use, reuse, and extend the code itself. This maximizes the usefulness and influence of your work. (And, after doing all of the work we describe above, you want people to use your code, don't you? )

## 4. Three Places to Share Statistical Software

*Post the code to your web site.* This is the simplest way to share your code, and has the virtue of making preliminary work quickly available.

*Deposit your program in a well-known software repository.* Programs distributed through ad-hoc (individual, or institutional) web sites are analogous to self-circulated drafts of manuscripts. Well-known software repositories, while not equivalent to peer review, provides a number of features that together establish a higher level of reliability and authority. Code repositories frequently require some minimal degree of standardization in packaging and documentation; prevent the software from being withdrawn arbitrarily once deposited; require that the source code be supplied whenever a binary is distributed; and require that a software license be clearly indicated (and often that that license be in conformance with open source standards). In addition, code archives generally provide a common infrastructure for versioning code and accessing previous versions, for reporting bugs, and for recording and making available the comments of other user's on the software.

*Publish your software in a peer-reviewed outlet.* In addition to *The Journal of Statistical Software*, journals such as *Computational Statistics and Data Analysis*, *Journal of Statistical Computation and*

*Simulation*, *ACM Transactions on Mathematical Software* and more than a dozen others regularly publish statistical software or algorithms, and related articles. (See [http://www.hmdc.harvard.edu/micah\\_altman/numal/resources/](http://www.hmdc.harvard.edu/micah_altman/numal/resources/) for a regularly maintained list of publication outlets.)

## 5. Three Ways to Contributes to the Study of Politics

*It contributes to methodology* - Models are not fully useful until there exists a way for non-methodologists in the relevant discipline to apply them. Often, seeing how a theoretical statistical model can be applied to real data, and how that statistical model can be implemented, is neither obvious nor easy. The developers of statistical software make unique contributions by extending algorithms to new problems.

*It contributes to teaching* - Open software does more than enable application of methods, it also opens a window into the black box of computing. Students are given an opportunity to more deeply understand what is involved in analysis.

*It contributes to the study of politics.* Models of politics increasingly reflect the natural complexity of human interactions. Advances in political methodology have increasingly been recognized as a necessity for the scientific investigation of these interactions. And public, usable software is vital to the widespread application of those methods.

Political methodology has grown from an almost unacknowledged niche to a robust sub-field. This is attested to by the existence of a top-ranked journal, *Political Analysis*; a high-quality annual conference; tremendous growth in the number of teaching and research positions devoted to it; and continued support from a large group of APSA members. [Box-Steffensmeier & Sokhey 2007]

Statistical software seems poised to grow in a similar way. Commercial software has often fallen far short of keeping up with advances in political methodology (or with statistical methods in general). Commercial software vendors can be slow to add methods that while perhaps statistically more appropriate, are not in current demand, or that trade performance for robustness and accuracy. [Stromberg 2004] In contrast, communities of methodologies, developing open code, (sometimes in partnership with commercial companies, such as *Stata*) have begun to fill the gap.

Writers of statistical software for political analysis contribute to a number of different communities. Other political scientists benefit from having sophisticated statistical models available for their own research. Students benefit from the opportunity to see examine *precisely* how results are generated. And the applied statistics community benefits from the development of new algorithms for statistical analysis. These can be large contributions, and a movement is well underway to recognize these

contributions with the academic incentives of citation and peer review. We created this issue of the journal of statistical software to recognize a sample of the contributions that have recently been made in this area.

## 6. References

- Abramson, Paul R., 1997. "Probing Well Beyond the Bounds of Conventional Wisdom", *American Journal of Political Science* 41(2).
- Altman, M. , Gill, J. and M.P. McDonald (2003). *Numerical Issues in Statistical Computing for the Social Scientist*. John Wiley & Sons, New York.
- Box-Steffensmeier, J.M., and Sokhey, A. 2007 "A Dynamic Labor Market: How Political Science is Opening Up to Methodologists, and How Methodologists are Opening Up Political Science", *PS: Political Science and Politics* 15(1):125-9
- Gleditsch, N.P, C. Metelits, H. Strand. 2003. "Posting Your Data: Will You be Scooped or will You be Famous?" *International Studies Perspectives* 4(1),
- Knuth, Donald E. (1974). "Computer Science and its Relation to Mathematics". *American Mathematical Monthly* 81,4.
- McConnell, Stephen. (2004) *Code Complete, (2nd Ed.)*, Microsoft Press.
- R development Core Team (2005). "R: A language and environment for statistical computing." *R Foundation for Statistical Computing, Vienna, Austria*. ISBN 3-900051-07-0.
- King, G., Tom, M., and Wittenberg, J. 2000. "Making the Most of Statistical Analyses: Improving Interpretation and Presentation", *American Journal of Political Science* 44: 241–55.
- Stromberg AJ (2004). "Why write statistical software? The case of robust statistical methods", *Journal of Statistical Software*, **10**(5).
- Tomz, M., Wittenberg, J., and King, G.. 2003. "CLARIFY: Software for Interpreting and Presenting Statistical Results.", *Journal of Statistical Software* 8(1).
- Venables, W.N., and B.D. Ripley. (2000). *S Programming*, Springer Verlag .

**Affiliation:**

Micah Altman  
Institute for Quantitative Social Science  
Harvard University  
1737 Cambridge Street, Room 325  
Cambridge, MA, 02138.  
United States of America  
E-mail: [micah\\_altman@harvard.edu](mailto:micah_altman@harvard.edu)  
URL: [http://www.hmdc.harvard.edu/micah\\_altman/](http://www.hmdc.harvard.edu/micah_altman/)

Simon Jackman  
Department of Political Science  
Encina Hall, Stanford University  
Stanford, California 94305-6044, USA  
United States of America  
[jackman@stanford.edu](mailto:jackman@stanford.edu)  
<http://jackman.stanford.edu>