# How to Write Good Documentation

The IDEAS Scientific Software Productivity Project
*ideas-productivity.org/resources/howtos/*

Determining the right amount of documentation is the first step to producing good documentation. The second step is to prioritize the types and extent of each type that is optimal for the team. And the third and final step is to provide the right set of incentives to the team. The following steps outline one possible approach. Note that other approaches may work for different teams; this outline is intended as an example rather than as a prescriptive solution.

**Gathering requirements:** Consider the size of the team, the code, and the reach of the code. Make an educated guess about the effort involved in documentation. Consider the priorities of the team in terms of all its deliverables. Also consider the technical debt (see [1]) of not developing a specific type of documentation. Determine the minimum documentation necessary to maintain the code robustly and the maximum documentation that the team can afford to produce.

**Selecting documentation types and extent:** From the exercise of gathering requirements, select all documentation types that are necessary for the team to meet its goals. Involving team members in making that choice is valuable in ensuring their buy-in. For each selected documentation type determine the extent of writing and the form for the documentation. For example, if a model and its algorithm are written up in a paper, the duplicate effort of producing a users guide may be unnecessary. A reference to the paper and a description of how to use the model and algorithm are better ways of spending the team's effort. If no such paper exists, however, it is important to include everything in the documentation.

For developers, a *design document* and *developers guide* are most useful. The design document should describe the software architecture, the infrastructure design choices, and the reasons behind those choices. The developers guide should include requirements of any code added to the software and also the coding standards. Both documents should be written for any software that has multiple components and more than a couple of developers. If the software is publicly distributed, it should also provide a *users guide*, and a *reference guide* if possible. Tutorials, examples, and FAQs are particularly helpful to users. Keeping the users manual up to date is often difficult, but it should be done when changes are made to the source code. Tools such as Emacs Etags and VIM Tags can be useful because they allow developers to access the users manual source code while developing the code.

For a large team with a transient developer population and a long-lived software product, *process documentation* is essential. Such documentation includes practices and policies; and licensing and release documentation should highlight those practices and policies that pertain to the release of publicly distributed software. In particular, the distribution policies should be clearly articulated if any restrictions exist.

**Useful rules of thumb:**

(1) Inline comments in less descriptive languages such as Fortran are always useful.

(2) Coding standards should emphasize efforts to make variables names as self-descriptive as possible.

(3) Simple and well explained constructs (as opposed to complex composite constructs) ease code maintainability.

(4) Inline documentation should be updated whenever the code is updated. Same applies to embedded user level documentation.

(5) If the code is for internal use in a small team focus, there is no need to spend effort on user level documentation. However, inline code documentation and model and

This material is based upon work supported by the U.S. Department of Energy Office of Science, Advanced Scientific Computing Research and Biological and Environmental Research programs.

algorithm specifications are still important.

(6) If the code is composable and targets wide and diverse user base, it must have a good user's guide and supplementary online documentation, such as howtos.

(7) Every algorithm should specify its range of validity and interoperability.

(8) If the code has a distributed developers community, then coding standards, code reviews, and a well documented software process are indispensable.

(9) For large distributed teams and users, it is also necessary to document verification benchmarks and validation process.

**Producing documentation:** Use of literate programming is a really good way to produce user level documentation. This approach allows text for manual pages be embedded directly into the source code files and not in separate locations, and workflow for documentation can be integrated into the workflow for development and use. The same is true of inlined informative documentation about the implementation choices. The options for embedded user's documentation include Doxygen, NDoc, javadoc, EiffelStudio, Sandcastle, ROBODoc, POD, and *TwinTex*. A particular challenge exists for writers of Fortran codes, because there isn't really a good option for automatic generation of manual pages. ROBODoc comes the closest.

**Examples of documentation in scientific software projects:**

*https://bitbucket.org/pflotran/pflotran-dev/wiki/Home*

*http://yt-project.org/doc/*

*https://www.earthsystemcog.org/projects/esmf/dev_docs/*

*http://flash.uchicago.edu/site/flashcode/user_support/*

*http://cactuscode.org/documentation/*

[1] *http://martinfowler.com/bliki/TechnicalDebt.html*

This document was prepared by Anshu Dubey with key contributions from Roscoe A. Bartlett, Barry Smith, and Jeffrey Johnson.

This material is based upon work supported by the U.S. Department of Energy Office of Science, Advanced Scientific Computing Research and Biological and Environmental Research programs.