

ETSF Coding Standards

European Theoretical Spectroscopy Facility

Carl-Olof Almbladh, Andrea Cucca, Xavier Gonze, Miguel Marques, Yann Pouillon

Version 1.3.1, last updated February 5, 2009

Contact author: Yann Pouillon <yann.pouillon@ehu.es>

The ETSF Coding Standards, last updated February 5, 2009.

Copyright © 2004, 2005, 2006, 2007, 2008, 2009 European Theoretical Spectroscopy Facility.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, with no Front-Cover Texts, and with no Back-Cover Texts.

Important note: this document was formerly known under the title *Guidelines for code development and code documentation* and used as a reference within the Nanoquanta Network of Excellence, the main objective was the creation of the ETSF. The content of the current document is a typo-fixed release of the version 3 of these guidelines, published on February 2nd 2008 by the Nanoquanta Integration Team 9 “Integration of theory and code developments”.

1 Context of this document

The software development effort in the Nanoquanta network and the European Theoretical Spectroscopy Facility (ETSF) is a long-term effort. The pieces of code that are and will be developed in the network and the facility, will likely be examined and/or reused by other developers than the one who originally wrote it. This might happen because cases not originally taken into account have to be developed, because the codes are ported to new platforms, because the initial testing was not careful enough and left some bugs, etc... It is well known that the time spent to modify a complex software (often referred to as the “maintenance” of a software) can be larger than the initial time needed to develop it.

By proposing guidelines for code development inside the ETSF, we aim at facilitating the future maintenance of a piece of software, with a minimal increase of the initial development time, mostly through the application of “good practice” rules. We have chosen a set of guidelines that are quite universal (e.g. they are independent of the programming language), widely accepted, and independent of the platform. We also propose some advice to ease the initial development period.

2 Guidelines

2.1 Documentation

The time needed to understand a piece of code can vary dramatically with the documentation available for this piece of code. Moreover, it is a large fraction of the time needed to modify a code. Thus, the initial developers should try to ease this process of understanding. They should:

- choose carefully the names of their variables, subroutines, etc ... so that they express as adequately as possible the underlying quantities, concepts or operations. The question of choosing good, self-explanatory names is important primarily for global entities (functions/subroutines, modules, datatypes ...);
- write comments (in good English), related to each logical section of their code, typically, one line of comment to explain 10-20 lines of code (more is even better);
- describe the purpose, options and arguments of the routines and functions, and refer — when possible — to existing published literature, e.g. cite a paper and the (exact) number of the equation that is coded, supply similar comments for members of modules, structures ...;
- keep notes of the formulas that are implemented, if they are not (yet) published, and ensure others might get it with the code;
- not try to produce “clever” coding: unless it is very well documented, “clever” coding might be much harder to understand than “dull” coding; if it is not critical for speed, the best code is simple, and easy-to-read.

2.2 Standards

For optimal portability of code (thus, sparing the time of somebody else), the developers should stick to standards. They should NOT use particular features of a compiler only available on some specific platforms. For instance, new Fortran code should adhere to the Fortran 90 or Fortran 95 standards unless there are strong reasons for doing otherwise. C code should be ANSI C with every function properly prototyped. New C++ should adhere to the C++ standard. If there is strong need for some system- or platform-dependent part, it should be isolated into separate, replaceable units.

2.3 Code reuse and libraries

The best way to develop rapidly a code that works is to reuse a piece of code that has already been written and tested! However, there is often a psychological barrier to reuse existing code: one likes to have the complete understanding of the code, and this cannot be obtained in this case. However, this is often a large waste of time: the own estimation of the time needed to develop a piece of code is very often restricted to the writing part, while there exist a large additional time, usually overlooked, needed to test and debug it. Developing always takes more time than expected, especially if high-quality coding is aimed at.

Libraries are (large) collections of reusable, carefully tested code. We strongly advice the developers to use the following libraries:

- BLAS and LAPACK for basic linear algebra operations (including vector-vector, matrix-vector, matrix-matrix operations, solution of linear systems of equations, eigenvalue problems ...);
- NETCDF (or HDF) for I/O operations (allowing portable files, more flexible formatting, etc);

Try to use libraries which are open-source as much as possible, in order to avoid copyright issues for the ETSF.

Note that Numerical Recipes routines are copyrighted.

2.4 Modularity

Aim for a modular structure with as few cross-dependencies as possible. Let the different parts of your code know little as possible about the rest of the program.

Try to dissociate tasks that fulfill different goals, e.g. write separate subroutines for the numerical treatment of data, and for their printing. Try that each routine or function performs a logical task of a well-defined character: data handling, or numerical treatment, or driving different routines of equivalent “level”. Try to think in terms of re-usability: will this subroutine easily reused by somebody else?

2.5 Use a versioning tool

This is not a “good practice” rule, but proves very useful to developers. There are excellent free software to manage the different versions that always appear in the development process : e.g. Bazaar, or Subversion, or Git. Use such tools ! Their use involves a very small overhead, but they can save your day. This is nearly mandatory for group development, but is also quite interesting in case of an isolated person.

The point is that such software keeps track of all the changes that your code is undergoing. This helps considerably in the debugging process. As an example, when you develop a new feature, it often happens that there are “ripple” effects, that cause new problems with some functionality of our code, not immediately linked with the functionality your are implementing. Such problems might go unnoticed during some time. When the faulty behavior is seen, the debugging might be very difficult. With the ability to come back step-by-step to the point where the code was working, it is usually easier to isolate the bug.

Additional references

“The mythical man-month” F.P. Brooks, Addison Wesley, London (1995) ISBN 0201835959

“Software maintenance. Concept and practice” P. Grubb and A.A. Takang, World Scientific, London, 2nd edition (2003) ISBN 981-238-426-X