An overview of ABINIT development

INFO
This page provides an overview of the strategy, tools and procedures followed for the development of the ABINIT package through collaboration of different groups of persons, based in different places in the world.
Any comment or suggestion to improve these procedures will be welcome !

Introduction

The ABINIT package is used by many researchers worldwide, without mandatory control of their work by the main contributors to ABINIT (thanks to its licensing policy.) In the same way, the ABINIT development project is fundamentally open to the worldwide contributions of people. The set of people contributing to ABINIT is refered to as the “ABINIT developers”.

People using the code might consider managing their personal subroutines, without trying to make them part of the official ABINIT package. However, this has two drawbacks for them : in subsequent versions, their modifications will not be incorporated, so that they might have to check and modify the interface for each new version ; moreover, their contribution is not tested by other users of the code, so that bugs might remain unnoticed. It is also nicer to share the result of their coding efforts with other users of the code.

Of course, a collaborative effort has also some drawbacks. In particular, the work of distant developers might generate orthogonal modifications of the same piece of code by two different people at the same time, generating “negative progress”, i.e. a large waste of time when synchronization is to be done. Also, it is required to use a well-defined coding style, to provide test case files, and to comment the modifications and additions as much as possible, with the aim to facilitate the maintenance, and the future modifications.

This document aims at providing a broad introduction to the techniques to be followed by the ABINIT developers. The analogy with the procedures to be used for the parallelization of a code is obvious. The aim is that each external 'node' does not waste its time, that communications are kept at the lowest level possible, and that the final result is correct ! The coding style has to be specified, as well as the development environment...

Basic philosophy for developers

One can distinguish between **“Debugging or fixes” contributions** and **“Development” contributions**. “Debugging or fixes” contributions are typically modifications of a few lines, in one or relatively few routines, needed for the code to work properly, or to be properly documented. Sometimes, they are related to comments within routines, or corrections to documents. Such modification might be sent directly to the coordinator (X. Gonze) via email, in the form of a patch. However, most developers use the Version Control System of ABINIT, called “git”. Both “Debugging” and “Development” contributions are easier in this model.

“Development” contributions usually involve the addition of new capabilities to the code. Despite the use of git, synchronization with the **other developers** is needed : one has to make sure nobody else is already working on a similar project. The development contributions might be quite localized (basically adding one routine, called by a few lines from an existing routine), or, on the contrary, involve modifications of many existing routines. Even for the localized type of modifications, discussion with the other developers is mandatory. Git deals with conflicts only at a formal level, so that semantic conflicts must be avoided from the very beginning. However, git is complemented with a Web application, gitlab, that is centralized, and allows the developers to quickly and easily communicate with each others (combining capabilities like commenting upon code modifications, managing the merge of branches, etc). For other contributors to ABINIT, a forum is available.

It will be the responsibility of the developer to make enough checks of the correctness of their modifications or additions. The developer should provide adequate documentation (basically the description of the input variables and output data, as well as a possible update of the bibliography). They should also provide one (maybe two) test(s) to be added to the standard suite of tests. This is needed to ensure that the transfer to the official version of the code has been done properly, and also that the new capabilities will be preserved by the subsequent modifications of the code. Finally, they have to show that their modifications have not suppressed existing capabilities of the codes, by running the set of already existing automatic tests.

It will be the responsibility of the coordinator to transfer the result of the development effort of each external developer (the content of their “zone”) to an official version, in such a way that the test is reproduced in a satisfactory way. The subsequent maintenance will be automatically done by checking that the corresponding test is still working despite modifications.

Developer environment

In order to implement this basic philosophy of development within ABINIT, different languages/tools are used, and mastered at different levels by different people (depending the kind of development contribution) : the Autotools (autoconf, automake, …), buildbot, git, gitlab, CPP, Fortran90, HTML, MPI, Python, ROBODOC. These, with the developer's Web pages of the ABINIT web site, constitute the ABINIT developer environment. For some of these languages/tools, the developers Web pages provide documentation: external links (like the language reference, repository, tutorials), some internal links (additional tutorials), and the particulars of the use of such language/tools within ABINIT.

These languages/tools do not have to be known or used at the same level. For some of them, only a rudimentary understanding of such languages/tools is sufficient. For others, the knowledge must be deeper.

- The autotools are at the core of the “build system”, but their use should be transparent to the developers. Unlike the maintainers of the package, developers are not requested to understand how to learn and use the autotools. By contrast, the developers need to have the autotools installed on their development machine.
- Buildbot is the engine that runs automatically the full suite of ABINIT tests every night. Developers will not have to install this tool, neither to understand how it works. But the ability to understand the results of the buildbot runs, i.e. the ABINIT test suite, is mandatory.
- git is the Version Control System of ABINIT. Except for minor debugging contributions, the developers need to have git installed on their development machine, and moreover, developers need to know how to use git.
- CPP is the preprocessor for the Fortran 90 sources. Thanks to build system directives passed to

CPP, the sequential or parallel version of ABINIT are generated, different external libraries may or may not be incorporated, etc …. A CPP version should come by default with every UNIX/Linux-like operating system. The developers need to know the effect of CPP directives (which is quite simple).

- Fortran 90 (or F90) is the language in which the vast majority of ABINIT routines is written. The developer need to have a good knowledge of Fortran 90 to develop within ABINIT. They should have a F90 compiler available on their machine.
- HTML is the language in which most of the ABINIT documentation is written, either within the package (e.g. description of input variables), or on the Web site (the present text is written in HTML). HTML files are text files, that can be edited using simple editors like vi or emacs, or more advanced editors. This language is interpreted by Web browser. ABINIT developers should have at least a basic knowledge of HTML to be able to document properly their work in ABINIT.
- MPI is the Message-Passing Interface library, used for the parallelism within ABINIT. In order to generate parallel code, the user and developer need to have the MPI library compiled on their machine. In order to develop in ABINIT, a basic knowledge of MPI is requested. Of course, a deep knowledge of MPI might be needed for several developments.
- Python is a scripting language, that is used for the automatic testing of ABINIT and in the build system. Its use should be transparent to the developers. Although a knowledge of Python is not requested to develop in ABINIT, it might help for the post-analysis. A suitable version of Python should come by default with every UNIX/Linux-like operating system.
- ROBODOC is a tool to create HTML (Web-browsable) file from source files (F90 for ABINIT). Developers are not required to use ROBODOC, but the ABINIT F90 must conform to the ABINIT-ROBODOC standard to be processed by ROBODOC before delivery over the Web. So, understanding the ABINIT-ROBODOC standard is requested by ABINIT developers.

The set-up of the documentation concerning these different tools (including a basic initiation), especially adapted for the ABINIT developer, is progressing. Please consult the developer's Web pages .

Here follows some more information, concerning the way the ABINIT community is working.

Code repositories

As mentioned above, managing different versions of ABINIT is done thanks to a a tool called git. For an introduction to this useful tool, you should look at the git dedicated pages.

Thanks to git, the development of a project becomes completely transparent, since all the changes in the files are registered: the latest version is of course available, but one can come backwards in time to track bugs easily or to know what anybody did. The development of different branches is also managed, as well as the subsequent merging procedure. This feature is important to allow development by many different people. The place where all the files are stored, including their history, is called a repository.

For developers, using the repository is a privileged way of managing their contributions, since the coordinator is automatically and instantly informed of their progress. For ABINIT, branches and versions have been normalized, whereas commit indexing are managed by GNU Arch.

Protocol for the developer

After installation on the development machine (e.g. */*/makemake ; configure ; make), and prior to any modification, the developer runs the suite of sequential tests on his machine (e.g. ./runtests.py command in the tests directory). The developer might check that the parallel tests are also OK. Checking against previous reference files is done automatically. All these steps are explained in the developer's Web pages . Then, the development effort can begin.

The source files are contained in subdirectories of the src directory. Supposing the developer initiate some new routines, introduce new input variables, and add some new tests :

- the new routine(s) should be placed in a subdirectory of the src directory, and the “abinit.src” file in the same subdirectory should be updated (because the new routines must be listed there), followed by the commands */*/makemake ; configure ; make ;
- the documentation about the new input variables (as well as new options of input variables) should be updated, see README.md in doc/input_variables
- the new tests should be introduced in some subdirectory of the tests directory (supposing ABINITv8, these new tests should be introduced in tests/v8), which means introducing new input and reference files in src/v8/Input and src/v8/Refs .

When one developer modifies or introduces a F90 routine, the ABINIT coding style must be followed. This will allow several automatic tool to process the routine. Also the developer should mention her/his initials in the header (copyright) of the new or modified routines. This will be useful if somebody needs information about this routine some time later.

At the end of the development effort, it is mandatory that the developer runs again the test suite (same command as for the initial checking), to be sure that the developments have not spoiled some other feature of the code. It is recommended to run this test suite more often, of course.