Exploring the source code of ABINIT

Y. Pouillon

Université Catholique de Louvain - Louvain-la-Neuve, Belgium

ABINIT Summer School
Santa Barbara, CA, USA
2005/08/27

Outline

Outline

License

The ABINIT package is Free Software

- Free for freedom, not price
- License: GNU General Public License (GPL)
  - availability of source code
  - permission to study, copy and modify the code
  - permission to redistribute modifications under the same conditions
  - non discrimination towards
    - persons or groups
    - fields of endeavour
- All developer contributions included under GPL

License

The ABINIT package is Free Software

- Free for freedom, not price
- License: GNU General Public License (GPL)
  - availability of source code
  - permission to study, copy and modify the code
  - permission to redistribute modifications under the same conditions
  - non discrimination towards
    - persons or groups
    - fields of endeavour
- All developer contributions included under GPL

License

The ABINIT package is Free Software

- Free for freedom, not price
- License: GNU General Public License (GPL)
  - availability of source code
  - permission to study, copy and modify the code
  - permission to redistribute modifications under the same conditions
  - non discrimination towards
    - persons or groups
    - fields of endeavour
- All developer contributions included under GPL

Numbering policy

3-digit version numbers: `x.y.z`, e.g. 4.5.3

- `x`: major version number (2 years)
- `y`: minor version number (4–6 months)
- `z`: debug status number (1–2 months)

  $\longrightarrow$ 0: maintainer version, unpublished
  $\longrightarrow$ 1: test version ($\approx \alpha$)
  $\longrightarrow$ 2: developers' version ($\approx \beta$)
  $\longrightarrow$ 3: production version
  $\longrightarrow$ 4: robust product version
  $\longrightarrow$ 5: very robust production version
- usually:
  - 3 last minor versions: active
  - older ones: obsolete

Numbering policy

3-digit version numbers: `x.y.z`, e.g. 4.5.3

- `x`: major version number (2 years)
- `y`: minor version number (4–6 months)
- `z`: debug status number (1–2 months)
  - $\longrightarrow$ 0: maintainer version, unpublished
  - $\longrightarrow$ 1: test version ($\approx \alpha$)
  - $\longrightarrow$ 2: developers' version ($\approx \beta$)
  - $\longrightarrow$ 3: production version
  - $\longrightarrow$ 4: robust product version
  - $\longrightarrow$ 5: very robust production version
- usually:
  - 3 last minor versions: active
  - older ones: obsolete

Numbering policy

3-digit version numbers: `x.y.z`, e.g. 4.5.3

- `x`: major version number (2 years)
- `y`: minor version number (4–6 months)
- `z`: debug status number (1–2 months)
  $\longrightarrow$ 0: maintainer version, unpublished
  $\longrightarrow$ 1: test version ($\approx \alpha$)
  $\longrightarrow$ 2: developers' version ($\approx \beta$)
  $\longrightarrow$ 3: production version
  $\longrightarrow$ 4: robust product version
  $\longrightarrow$ 5: very robust production version
- usually:
  - 3 last minor versions: active
  - older ones: obsolete

Numbering policy

3-digit version numbers: `x.y.z`, e.g. 4.5.3

- `x`: major version number (2 years)
- `y`: minor version number (4–6 months)
- `z`: debug status number (1–2 months)
  $\longrightarrow$ 0: maintainer version, unpublished
  $\longrightarrow$ 1: test version ($\approx \alpha$)
  $\longrightarrow$ 2: developers' version ($\approx \beta$)
  $\longrightarrow$ 3: production version
  $\longrightarrow$ 4: robust product version
  $\longrightarrow$ 5: very robust production version
- usually:
  - 3 last minor versions: active
  - older ones: obsolete

Numbering policy

3-digit version numbers: `x.y.z`, e.g. 4.5.3

- `x`: major version number (2 years)
- `y`: minor version number (4–6 months)
- `z`: debug status number (1–2 months)
  $\longrightarrow$ 0: maintainer version, unpublished
  $\longrightarrow$ 1: test version ($\approx \alpha$)
  $\longrightarrow$ 2: developers' version ($\approx \beta$)
  $\longrightarrow$ 3: production version
  $\longrightarrow$ 4: robust product version
  $\longrightarrow$ 5: very robust production version
- usually:
  - 3 last minor versions: active
  - older ones: obsolete

Numbering policy

3-digit version numbers: `x.y.z`, e.g. 4.5.3

- `x`: major version number (2 years)
- `y`: minor version number (4–6 months)
- `z`: debug status number (1–2 months)
  $\longrightarrow$ 0: maintainer version, unpublished
  $\longrightarrow$ 1: test version ($\approx \alpha$)
  $\longrightarrow$ 2: developers' version ($\approx \beta$)
  $\longrightarrow$ 3: production version
  $\longrightarrow$ 4: robust product version
  $\longrightarrow$ 5: very robust production version
- usually:
  - 3 last minor versions: active
  - older ones: obsolete

Numbering policy

3-digit version numbers: `x.y.z`, e.g. 4.5.3

- `x`: major version number (2 years)
- `y`: minor version number (4–6 months)
- `z`: debug status number (1–2 months)
  $\longrightarrow$ 0: maintainer version, unpublished
  $\longrightarrow$ 1: test version ($\approx \alpha$)
  $\longrightarrow$ 2: developers' version ($\approx \beta$)
  $\longrightarrow$ 3: production version
  $\longrightarrow$ 4: robust product version
  $\longrightarrow$ 5: very robust production version
- usually:
  - 3 last minor versions: active
  - older ones: obsolete

Numbering policy

3-digit version numbers: `x.y.z`, e.g. 4.5.3

- `x`: major version number (2 years)
- `y`: minor version number (4–6 months)
- `z`: debug status number (1–2 months)
  $\longrightarrow$ 0: maintainer version, unpublished
  $\longrightarrow$ 1: test version ($\approx \alpha$)
  $\longrightarrow$ 2: developers' version ($\approx \beta$)
  $\longrightarrow$ 3: production version
  $\longrightarrow$ 4: robust product version
  $\longrightarrow$ 5: very robust production version
- usually:
  - 3 last minor versions: active
  - older ones: obsolete

Numbering policy

3-digit version numbers: `x.y.z`, e.g. 4.5.3

- `x`: major version number (2 years)
- `y`: minor version number (4–6 months)
- `z`: debug status number (1–2 months)
  $\longrightarrow$ 0: maintainer version, unpublished
  $\longrightarrow$ 1: test version ($\approx \alpha$)
  $\longrightarrow$ 2: developers' version ($\approx \beta$)
  $\longrightarrow$ 3: production version
  $\longrightarrow$ 4: robust product version
  $\longrightarrow$ 5: very robust production version
- usually:
  - 3 last minor versions: active
  - older ones: obsolete

Numbering policy

3-digit version numbers: `x.y.z`, e.g. 4.5.3

- `x`: major version number (2 years)
- `y`: minor version number (4–6 months)
- `z`: debug status number (1–2 months)
  $\longrightarrow$ 0: maintainer version, unpublished
  $\longrightarrow$ 1: test version ($\approx \alpha$)
  $\longrightarrow$ 2: developers' version ($\approx \beta$)
  $\longrightarrow$ 3: production version
  $\longrightarrow$ 4: robust product version
  $\longrightarrow$ 5: very robust production version
- usually:
  - 3 last minor versions: active
  - older ones: obsolete

Coding style

All routines follow explicitely ABINIT rules (abirules)

- Special format for processing by ROBODOC (`http://www.xs4all.nl/˜rfsber/Robo/robodoc.html`)
- Documentation available inside each routine
- Many comments forced by the ABINIT style
- Input or output intent specified
- Automatic generation of “parent” and “children” lists
- Automatic enforcement of coding rules
- Sources available on-line for browsing

Self-testing

“Self-testing” software concept

- $\geq$ 400 PERL-script-driven automatic tests
- All capabilities of ABINIT covered
- Automatic comparison to reference files
- Multi-level analysis
  - stability $\longrightarrow$ how went each test
  - accuracy $\longrightarrow$ diff of floating-point values
  - diagnostics $\longrightarrow$ detailed log file with errors and warnings
- can be used as examples for beginners

Self-testing

“Self-testing” software concept

- $\geq$ 400 PERL-script-driven automatic tests
- All capabilities of ABINIT covered
- Automatic comparison to reference files
- Multi-level analysis
  - stability $\longrightarrow$ how went each test
  - accuracy $\longrightarrow$ diff of floating-point values
  - diagnostics $\longrightarrow$ detailed log file with errors and warnings
- can be used as examples for beginners

Self-testing

"Self-testing" software concept

- $\geq$ 400 PERL-script-driven automatic tests
- All capabilities of ABINIT covered
- Automatic comparison to reference files
- Multi-level analysis
  - stability $\longrightarrow$ how went each test
  - accuracy $\longrightarrow$ diff of floating-point values
  - diagnostics $\longrightarrow$ detailed log file with errors and warnings
- can be used as examples for beginners

Self-testing

"Self-testing" software concept

- $\geq$ 400 PERL-script-driven automatic tests
- All capabilities of ABINIT covered
- Automatic comparison to reference files
- Multi-level analysis
  - stability $\longrightarrow$ how went each test
  - accuracy $\longrightarrow$ diff of floating-point values
  - diagnostics $\longrightarrow$ detailed log file with errors and warnings
- can be used as examples for beginners

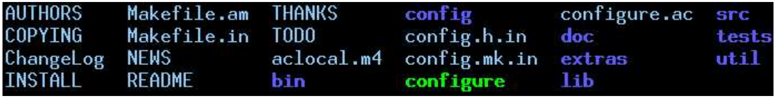
Self-testing

"Self-testing" software concept

- $\geq$ 400 PERL-script-driven automatic tests
- All capabilities of ABINIT covered
- Automatic comparison to reference files
- Multi-level analysis
  - stability $\longrightarrow$ how went each test
  - accuracy $\longrightarrow$ diff of floating-point values
  - diagnostics $\longrightarrow$ detailed log file with errors and warnings
- can be used as examples for beginners

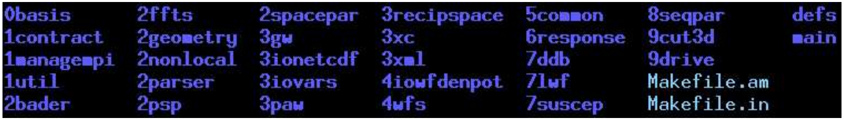
Outline

The source tree


```
AUTHORS    Makefile.am  THANKS     config        configure.ac  src
COPYING    Makefile.in  TODO       config.h.in   doc           tests
ChangeLog  NEWS         aclocal.m4 config.mk.in  extras        util
INSTALL    README       bin        configure     lib
```

- 8 different sections
  - scripts, ready for local use: `bin/`
  - configuration: `config/` + `configure` script
  - documentation: `doc/`
  - external libraries: `lib/*/`
  - core source: `src/*/`
  - test suite: `tests/`
  - maintainer scripts: `util/`
  - miscellaneous extra stuff: `extras/`

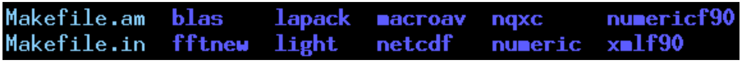
The core source



```
0basis      2ffts      2spacepar   3recipspace  5common    8seqpar      defs
1contract   2geometry  3gw         3xc          6response  9cut3d       main
1managempi  2nonlocal  3ionetcdf   3xml         7ddb       9drive
1util       2parser    3iovars     4iowfdenpot  7lwf       Makefile.am
2bader      2psp       3paw        4wfs         7suscep    Makefile.in
```

- Subdirectories of `src/`
- 12 different levels
  - defs: “underground” or “root” modules
  - 0–9: all different parts of the code (internal libraries)
  - main: main programs
  - **hierarchical substructure**
    $\longrightarrow$ each level depends only on preceeding ones
- Detailed in `doc/developers/dirs_and_files`

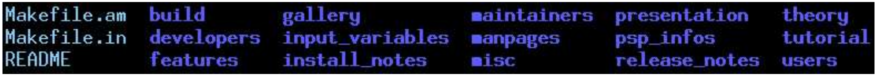
The external libraries



```
Makefile.am  blas     lapack  macroav  nqxc     numericf90
Makefile.in  fftnew   light   netcdf   numeric  xmlf90
```

- Bigger and bigger subset of BLAS / LAPACK routines
- S. Gődecker's FFT routines
- Full version of NetCDF
- Full version of Nanoquanta libXC
- Full version of XMLF90
- Miscellaneous non-abirule-compliant routines

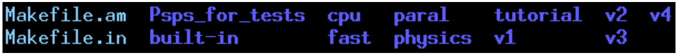
The documentation



```
Makefile.am  build       gallery          maintainers  presentation   theory
Makefile.in  developers  input_variables  manpages     psp_infos      tutorial
README       features    install_notes    misc         release_notes  users
```

- Currently being restructured
- Dispatch documents into categories:
  - build
  - users
  - developers
  - maintainers
- DFSG: one manpage per binary
- Provide at least plain-text and HTML
  $\Longrightarrow$ use markdown for now
  (`http://daringfireball.net/projects/markdown/`)

The test suite



```
Makefile.am  Psps_for_tests  cpu   paral    tutorial  v2  v4
Makefile.in  built-in        fast  physics  v1        v3
```

- Pseudopotentials
- Built-in tests
  - very basic
  - very fast
- Several test series
  - covering all aspects of ABINIT
  - may require some time (e.g. `physics`)
  - require a lot of free disk space
    $\longrightarrow \approx$ 3Gb for all tests
- Tutorial input files

Outline

ABINIT routines

- ABIRULES: 11 sections describing how to write routines
  $\longrightarrow$ see `doc/developers/rules_coding`
- Routines inside `src/`: must follow abirules
  - Fortran 90/95
  - lower-case characters
  - locality of information
    $\longrightarrow$ everything needed contained inside the routine
- Other routines: recommendations
  - should be in Fortran 90/95
  - should require as few maintenance as possible
  - minimize their number
- To create a new routine: `mkroutine <name>`

ABINIT routines

- ABIRULES: 11 sections describing how to write routines
  $\longrightarrow$ see `doc/developers/rules_coding`
- Routines inside `src/`: must follow abirules
  - Fortran 90/95
  - lower-case characters
  - locality of information
    $\longrightarrow$ everything needed contained inside the routine
- Other routines: recommendations
  - should be in Fortran 90/95
  - should require as few maintenance as possible
  - minimize their number
- To create a new routine: `mkroutine <name>`

ABINIT routines

- ABIRULES: 11 sections describing how to write routines
  $\longrightarrow$ see `doc/developers/rules_coding`
- Routines inside `src/`: must follow abirules
  - Fortran 90/95
  - lower-case characters
  - locality of information
    $\longrightarrow$ everything needed contained inside the routine
- Other routines: recommendations
  - should be in Fortran 90/95
  - should require as few maintenance as possible
  - minimize their number
- To create a new routine: `mkroutine <name>`

ABINIT routines

- ABIRULES: 11 sections describing how to write routines
  $\longrightarrow$ see `doc/developers/rules_coding`
- Routines inside `src/`: must follow abirules
  - Fortran 90/95
  - lower-case characters
  - locality of information
    $\longrightarrow$ everything needed contained inside the routine
- Other routines: recommendations
  - should be in Fortran 90/95
  - should require as few maintenance as possible
  - minimize their number
- To create a new routine: `mkroutine <name>`

What a routine looks like: header

```
!{\src2tex{textfont=tt}}
!!****f* ABINIT/abinit_subroutine
!! NAME
!! abinit_subroutine
!!
!! FUNCTION
!!
!! COPYRIGHT
!! Copyright (C) 2005 ABINIT group (the_author)
!! This file is distributed under the terms of the
!! GNU General Public License, see ~ABINIT/Infos/copyright
!! or http://www.gnu.org/copyleft/gpl.txt .
!!
!! INPUTS
!!  argin(sizein)=description
!!
!! OUTPUT
!!  argout(sizeout)=description
!!
!! SIDE EFFECTS
!!
!! NOTES
!!
!! PARENTS
!!  Will be filled automatically by the parent script
!!
!! CHILDREN
!!  Will be filled automatically by the parent script
!!
!! SOURCE
```

What a routine looks like: declarations

```
 subroutine abinit_subroutine(argin,argout,option,sizein,sizeout)

 use defs_basis
 implicit none

!Arguments ------------------------------------
 integer , intent(in)  :: option,sizein,sizeout
 integer , intent(in)  :: argin(sizein)
 integer , intent(out) :: argout(sizeout)
 real(dp), intent(out) ::                        ! to be filled, if needed

!Local variables-------------------------------
 integer ::                                      ! to be filled, if needed
 real(dp) ::                                     ! to be filled, if needed
!character(len=500) :: message                   ! to be uncommented, if needed

! *************************************************************************

!DEBUG
!write(std_out,*)' abinit_subroutine : enter '
!ENDDEBUG
```

What a routine looks like: body

```
!DEBUG                                            ! to be uncommented, if needed
! if(option/=1 .and. option/=2 )then
!  write(message,'(a,a,a,a,a,a,i6)') ch10,&
!&   ' abinit_subroutine: BUG -',ch10,&
!&   '  The argument option should be 1 or 2,',ch10,&
!&   '  however, option=',option
!  call wrtout(std_out,message,'COLL')
!  call leave_new('COLL')
! endif
! if(sizein<1)then
!  write(message,'(a,a,a,a,a,a,i6)') ch10,&
!&   ' abinit_subroutine: BUG -',ch10,&
!&   '  The argument sizein should be a positive number,',ch10,&
!&   '  however, sizein=',sizein
!  call wrtout(std_out,message,'COLL')
!  call leave_new('COLL')
! endif
!ENDDEBUG


!DEBUG
!write(std_out,*)' abinit_subroutine : exit'
!stop
!ENDDEBUG

 end subroutine abinit_subroutine
!!***
```

Embedded documentation

"Self-documentation" software concept

- For each subprogram: formatted header
  - functional description
  - copyright reminder, with list of authors
  - inputs (arguments not modified)
  - outputs (arguments initialized)
  - side effects (arguments and variables modified)
  - warnings
  - notes or todo list
  - parents & children (automatically generated)
- Translation into HTML by ROBODOC
  $\Longrightarrow$ **web-browsable source code**

Embedded documentation

"Self-documentation" software concept

- For each subprogram: formatted header
  - functional description
  - copyright reminder, with list of authors
  - inputs (arguments not modified)
  - outputs (arguments initialized)
  - side effects (arguments and variables modified)
  - warnings
  - notes or todo list
  - parents & children (automatically generated)
- Translation into HTML by ROBODOC
  $\Longrightarrow$ **web-browsable source code**

Outline

Supported architectures and compilers

- Well-supported architectures
  - x86 / Linux and Windows
  - Mac / OS X
  - DEC Alpha / OSF and Linux
  - Sun / Solaris and Linux
  - IBM / AIX and Linux
  - Cray, Fujitsu, Hitachi, HP, NEC, SGI, VAX
- Compilers
  - On x86: GNU, Intel, ABSoft, NAGWare, PathScale, Portland
  - On other architectures: native compilers
- Some configurations need workarounds
- New ones: let us know!

The traditional build trilogy

- First create a build directory, e.g.:
  "`make build && cd build`"
  $\longrightarrow$ Will preserve a clean source tree
  $\longrightarrow$ highly recommended
- Then:
  1. `../configure [options]`
  2. `make`
  3. `make install`
- Optionally: "`make check`" before "`make install`"

The traditional build trilogy

- First create a build directory, e.g.:
  "`make build && cd build`"
  $\longrightarrow$ Will preserve a clean source tree
  $\longrightarrow$ highly recommended
- Then:
  1. `../configure [options]`
  2. `make`
  3. `make install`
- Optionally: "`make check`" before "`make install`"

The traditional build trilogy

- First create a build directory, e.g.:
  "`make build && cd build`"
  $\longrightarrow$ Will preserve a clean source tree
  $\longrightarrow$ highly recommended

- Then:
  1. `../configure [options]`
  2. `make`
  3. `make install`

- Optionally: "`make check`" before "`make install`"

Configure

- By default: detection of libraries, workarounds, ...
- Without options: make a safe build / use defaults for install
- `--prefix=DIR`: install into `DIR`
- `--disable-parallel`: disable build of parallel code
- `--enable-netcdf`: add support for NetCDF
- `--enable-nqxc`: add support for Nanoquanta libXC
- `--enable-xmlf90`: add support for libXMLF90
- `--with-<lib>-prefix=DIR`: look for *<lib>* in `DIR`
  $\longrightarrow$ *<lib> = blas, lapack, netcdf, nqxc, xmlf90*
- Options can be saved in
  `${HOME}/.abinit/build/<hostname>.ac`

Make

- Without arguments: build all main binaries
- `allseq`: build all sequential binaries
- `<bin>`: build main binary *<bin>*
  $\longrightarrow$ *abinip, abinis, aim, anaddb, band2eps, conducti, cut3d, lwf, macroave, mrgddb, mrggkk, newsp, optic*
- `check`: build binaries and perform selected tests (still in development)
- `dist`: create source tarball
- `distcheck`
  - create source tarball
  - build all binaries from it
  - perform selected tests

Install

- Default install prefix: `/usr/local`
- Without arguments
  - use `${prefix}/lib/abinit/x.y/` as base directory
  - install wrapper script in `${prefix}/bin/`
  - install documentation in `${prefix}/share/doc/abinit/x.y/`
- `make install prefix=DIR`: change prefix for DIR
- `make install DESTDIR=DIR`: use DIR as DESTDIR (packages)

Performing tests

Going in the `tests/` directory

- `make`: obtain help on how to perform tests
- `make test_in`: perform built-in tests
- `make test_<series> start=#a stop=#b`
  - perform tests of *<series>*
    $\longrightarrow$ *cpu, fast, physics, tutorial, v1, v2, v3, v4*
  - start at test #a
  - stop at test #b
  - results stored in `<series>/„tmp_make_tests`
  - to perform only one test: use either *start* or *stop*
  - omitting *start* and *stop*: perform whole series (requires a lot of free disk space)

Outline

Ensuring portability

- Autoconf $\Longrightarrow$ build on many architectures
- Installation and tests can be automated
  - set-up only once for a given architecture
  - several builds sharing the same physical source tree
  - can be built on a “compile farm”
- Test suite highly portable (PERL)

Highly-distributed development

- More than 40 active developers all around the world
- Many other occasional contributors

$\Longrightarrow$ **version management by GNU Arch**

- Highly-customizable design (suits your project)
- Contributions stored by *category--branch--version--revision*
- One or more branches per developer (high flexibility)
- Clever merge system

Highly-distributed development

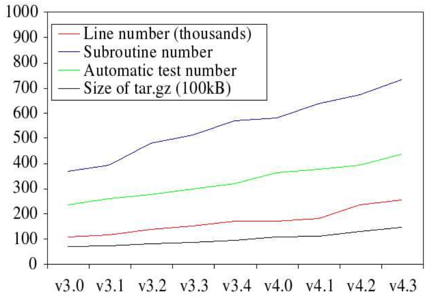
- More than 40 active developers all around the world
- Many other occasional contributors
  $\Longrightarrow$ **version management by GNU Arch**
  - Highly-customizable design (suits your project)
  - Contributions stored by *category--branch--version--revision*
  - One or more branches per developer (high flexibility)
  - Clever merge system

Highly-distributed development

- More than 40 active developers all around the world
- Many other occasional contributors
  $\Longrightarrow$ **version management by GNU Arch**
  - Highly-customizable design (suits your project)
  - Contributions stored by *category--branch--version--revision*
  - One or more branches per developer (high flexibility)
  - Clever merge system

Outline

Growing size of the code



From 2000 to 2005

- source code:
  - 105 $\longrightarrow$ 292 kLines
  - 364 $\longrightarrow$ 981 routines
  - 234 $\longrightarrow$ 418 tests
- package:
  - 2000 $\longrightarrow$ 3100 files
  - 6.7Mb $\longrightarrow$ 17.5Mb

Strategic choices

Current challenges

1. Improve the quality of the code along with its growth in size
2. Continue to provide a high-quality support
3. Enhance the integration of ABINIT with other codes

Three lines of action

- Improving conformance to programming standards
- Refining the directory structure
- Increasing modularity

Strategic choices

Current challenges

1. Improve the quality of the code along with its growth in size
2. Continue to provide a high-quality support
3. Enhance the integration of ABINIT with other codes

Three lines of action

- Improving conformance to programming standards
- Refining the directory structure
- Increasing modularity

Strategic choices

Current challenges

1. Improve the quality of the code along with its growth in size
2. Continue to provide a high-quality support
3. Enhance the integration of ABINIT with other codes

Three lines of action

- Improving conformance to programming standards
- Refining the directory structure
- Increasing modularity

Strategic choices

Current challenges

1. Improve the quality of the code along with its growth in size
2. Continue to provide a high-quality support
3. Enhance the integration of ABINIT with other codes

Three lines of action

- Improving conformance to programming standards
- Refining the directory structure
- Increasing modularity

Improving conformance to standards

- Goals:
  - conciliate quality and growth
  - take benefit from free (libre) development tools
  - install ABINIT system-wide in standard directories
  - be able to create Debian and RPM packages
- Step-by-step:
  1. uncompress in `abinit-<version>/` (4.4.3)
  2. strengthen code quality checks (4.4.3 $\longrightarrow$ 5.0.3)
  3. add support for the GNU Autotools (4.4.3 $\longrightarrow$ 5.0.3)
  4. harmonize preprocessing options (4.5.3 $\longrightarrow$ 5.0.3)
  5. improve integration of test suite (4.5.3 $\longrightarrow$ 5.0.3)
  6. restructure and enhance documentation (4.5.3 $\longrightarrow$?)

Improving conformance to standards

- Goals:
  - conciliate quality and growth
  - take benefit from free (libre) development tools
  - install ABINIT system-wide in standard directories
  - be able to create Debian and RPM packages
- Step-by-step:
  1. uncompress in `abinit-<version>/` (4.4.3)
  2. strengthen code quality checks (4.4.3 $\longrightarrow$ 5.0.3)
  3. add support for the GNU Autotools (4.4.3 $\longrightarrow$ 5.0.3)
  4. harmonize preprocessing options (4.5.3 $\longrightarrow$ 5.0.3)
  5. improve integration of test suite (4.5.3 $\longrightarrow$ 5.0.3)
  6. restructure and enhance documentation (4.5.3 $\longrightarrow$?)

Improving conformance to standards

- Goals:
  - conciliate quality and growth
  - take benefit from free (libre) development tools
  - install ABINIT system-wide in standard directories
  - be able to create Debian and RPM packages
- Step-by-step:
  1. uncompress in `abinit-<version>/` (4.4.3)
  2. strengthen code quality checks (4.4.3 $\longrightarrow$ 5.0.3)
  3. add support for the GNU Autotools (4.4.3 $\longrightarrow$ 5.0.3)
  4. harmonize preprocessing options (4.5.3 $\longrightarrow$ 5.0.3)
  5. improve integration of test suite (4.5.3 $\longrightarrow$ 5.0.3)
  6. restructure and enhance documentation (4.5.3 $\longrightarrow$?)

Improving conformance to standards

- Goals:
  - conciliate quality and growth
  - take benefit from free (libre) development tools
  - install ABINIT system-wide in standard directories
  - be able to create Debian and RPM packages
- Step-by-step:
  1. uncompress in `abinit-<version>/` (4.4.3)
  2. strengthen code quality checks (4.4.3 $\longrightarrow$ 5.0.3)
  3. add support for the GNU Autotools (4.4.3 $\longrightarrow$ 5.0.3)
  4. harmonize preprocessing options (4.5.3 $\longrightarrow$ 5.0.3)
  5. improve integration of test suite (4.5.3 $\longrightarrow$ 5.0.3)
  6. restructure and enhance documentation (4.5.3 $\longrightarrow$?)

Improving conformance to standards

- Goals:
  - conciliate quality and growth
  - take benefit from free (libre) development tools
  - install ABINIT system-wide in standard directories
  - be able to create Debian and RPM packages
- Step-by-step:
  1. uncompress in `abinit-<version>/` (4.4.3)
  2. strengthen code quality checks (4.4.3 $\longrightarrow$ 5.0.3)
  3. add support for the GNU Autotools (4.4.3 $\longrightarrow$ 5.0.3)
  4. harmonize preprocessing options (4.5.3 $\longrightarrow$ 5.0.3)
  5. improve integration of test suite (4.5.3 $\longrightarrow$ 5.0.3)
  6. restructure and enhance documentation (4.5.3 $\longrightarrow$?)

Improving conformance to standards

- Goals:
  - conciliate quality and growth
  - take benefit from free (libre) development tools
  - install ABINIT system-wide in standard directories
  - be able to create Debian and RPM packages
- Step-by-step:
  1. uncompress in `abinit-<version>/` (4.4.3)
  2. strengthen code quality checks (4.4.3 $\longrightarrow$ 5.0.3)
  3. add support for the GNU Autotools (4.4.3 $\longrightarrow$ 5.0.3)
  4. harmonize preprocessing options (4.5.3 $\longrightarrow$ 5.0.3)
  5. improve integration of test suite (4.5.3 $\longrightarrow$ 5.0.3)
  6. restructure and enhance documentation (4.5.3 $\longrightarrow$?)

Improving conformance to standards

- Goals:
  - conciliate quality and growth
  - take benefit from free (libre) development tools
  - install ABINIT system-wide in standard directories
  - be able to create Debian and RPM packages
- Step-by-step:
  1. uncompress in `abinit-<version>/` (4.4.3)
  2. strengthen code quality checks (4.4.3 $\longrightarrow$ 5.0.3)
  3. add support for the GNU Autotools (4.4.3 $\longrightarrow$ 5.0.3)
  4. harmonize preprocessing options (4.5.3 $\longrightarrow$ 5.0.3)
  5. improve integration of test suite (4.5.3 $\longrightarrow$ 5.0.3)
  6. restructure and enhance documentation (4.5.3 $\longrightarrow$?)

Refining the directory structure

- Conform to the GNU coding standards
  - mandatory plain-text files in top directory, e.g. `README`, `INSTALL`, `COPYING`
  - base documentation in plain-text format in `doc/`
  - modular directory structure
  - one `Makefile` per source directory
- Better separation between
  - source and non-source files
  - use, development, and maintenance
- Increase modularity, aka “breaking the monolith”
- Start to share responsibilities

Refining the directory structure

- Conform to the GNU coding standards
  - mandatory plain-text files in top directory, e.g. `README`, `INSTALL`, `COPYING`
  - base documentation in plain-text format in `doc/`
  - modular directory structure
  - one `Makefile` per source directory
- Better separation between
  - source and non-source files
  - use, development, and maintenance
- Increase modularity, aka “breaking the monolith”
- Start to share responsibilities

Refining the directory structure

- Conform to the GNU coding standards
  - mandatory plain-text files in top directory, e.g. `README`, `INSTALL`, `COPYING`
  - base documentation in plain-text format in `doc/`
  - modular directory structure
  - one `Makefile` per source directory
- Better separation between
  - source and non-source files
  - use, development, and maintenance
- Increase modularity, aka “breaking the monolith”
- Start to share responsibilities

Refining the directory structure

- Conform to the GNU coding standards
  - mandatory plain-text files in top directory, e.g. `README`, `INSTALL`, `COPYING`
  - base documentation in plain-text format in `doc/`
  - modular directory structure
  - one `Makefile` per source directory
- Better separation between
  - source and non-source files
  - use, development, and maintenance
- Increase modularity, aka “breaking the monolith”
- Start to share responsibilities

Increasing modularity

- Monolithic structure not efficient beyond a critical size *(already reached by ABINIT)*
  - maintenance heavier and heavier
  - dependency tracking becomes a nightmare
  - release timeline cannot be respected anymore
- More and more code re-use
  $\longrightarrow$ `blas`, `lapack`
  $\longrightarrow$ `netcdf`, `libxc`, `xmlf90`
- Future projects
  $\longrightarrow$ BigDFT (order-N methods in ABINIT)

Increasing modularity

- Monolithic structure not efficient beyond a critical size *(already reached by ABINIT)*
    - maintenance heavier and heavier
    - dependency tracking becomes a nightmare
    - release timeline cannot be respected anymore
- More and more code re-use

  $\longrightarrow$ `blas`, `lapack`

  $\longrightarrow$ `netcdf`, `libxc`, `xmlf90`
- Future projects

  $\longrightarrow$ BigDFT (order-N methods in ABINIT)

Increasing modularity

- Monolithic structure not efficient beyond a critical size *(already reached by ABINIT)*
    - maintenance heavier and heavier
    - dependency tracking becomes a nightmare
    - release timeline cannot be respected anymore
- More and more code re-use

  $\longrightarrow$ `blas`, `lapack`

  $\longrightarrow$ `netcdf`, `libxc`, `xmlf90`
- Future projects

  $\longrightarrow$ BigDFT (order-N methods in ABINIT)

Summary

- Big source code growing at a constant pace
- Freedom to use, copy, modify and redistribute (GNU GPL)
- Strict development model, enforced by scripts
- Hierarchical structure, to ease dependency tracking
- Build: *configure + make + make install* trilogy
- Current projects affect structure of source code
- “Breaking the monolith”

Acknowledgments

- All α- and β-testers of the 5.0 version
- CISM (center for high-performance computing in Louvain-la-Neuve)
- ABINIT community

Thank you for your time!

Acknowledgments

- All α- and β-testers of the 5.0 version
- CISM (center for high-performance computing in Louvain-la-Neuve)
- ABINIT community

Thank you for your time!