Sharing electronic structure and crystallographic data with ETSF_IO

D. Caliste⋆ [a,b,c] Y. Pouillon [a,d,c] M.J. Verstraete [a,e]
V. Olevano [a,f,g] X. Gonze [a,c]

[a] *European Theoretical Spectroscopy Facility (ETSF)*

[b] *CEA, INAC/SP2M/L_Sim, Grenoble (France)*

[c] *Université Catholique de Louvain, Louvain-la-Neuve (Belgium)*

[d] *Universidad del País Vasco UPV/EHU, Donostia-San Sebastián (Spain)*

[e] *University of York, York (United Kingdom)*

[f] *LSI, CNRS-CEA, École Polytechnique, Palaiseau (France)*

[g] *Institut Néel, CNRS and Université Joseph Fourier, Grenoble (France)*

Abstract

We present a library of routines whose main goal is to read and write exchangeable files (NetCDF file format) storing electronic structure and crystallographic information. It is based on the specification agreed inside the European Theoretical Spectroscopy Facility (ETSF). Accordingly, this library is nicknamed ETSF_IO. The purpose of this article is to give both an overview of the ETSF_IO library and a closer look at its usage. ETSF_IO is designed to be robust and easy to use, close to Fortran `read` and `write` routines. To facilitate its adoption, a complete documentation of the input and output arguments of the routines is available in the package, as well as six tutorials explaining in detail various possible uses of the library routines.

PACS: 70; 71.15.Mb; 78

Key words: NetCDF, ETSF, electronic structure, crystallographic data

Program summary

Program Title: ETSF_IO
Journal Reference:
Catalogue identifier:
Licensing provisions: LGPL [1]
Program summary URL: http://etsf.eu/index.php?page=tools

Email address: `damien.caliste@cea.fr` (D. Caliste⋆).

Programming language: Fortran95
Distribution format: tar.gz
Keywords: NetCDF, ETSF, electronic structure, crystallographic data
PACS: 70; 71.15.Mb; 78
Classification: 7.3 Electronic Structure, 8 Crystallography
External routines/libraries: NetCDF [2]

Nature of problem:
Store and exchange electronic structure data and crystallographic data independently of the computational platform, language and generating software.
Solution method:
Implement a library based both on NetCDF file format and an open specification [3].
References:

[1] http://www.gnu.org/licenses/lgpl.html

[2] http://www.unidata.ucar.edu/software/netcdf

[3] http://etsf.eu/index.php?page=standardization

1 Introduction

The existence of multiple software projects in a given domain makes the existence of standardization advisable — if not mandatory. This allows the different programs to interact and gives the user more choice. Different standards can be used concurrently, as, for example, MP3, Ogg Vorbis, AAC, ... , that encode sound. In such a case, the availability of open specifications makes the creation of conversion utilities much easier. These specifications also facilitate the development of independent libraries, regardless of the projects and topics they originate from.

First-principles (electronic-structure based) calculations of materials properties require to store both crystallographic and electron-related data. The specification of atomic and molecular structures have already been standardized, using the widely known XYZ format, or with file formats such as CML[1] or PDB[2]. This allows electronic structure programs to output crystallographic data that can be rendered with common viewers, or to load atomic positions using these formats.

However, the field of electronic structure calculations is in constant evolution and each software project usually develops its own type of files for electronic data, matching its capabilities at one moment in time, without paying much attention to file standardization. The exchange of data representing the results of electronic structure calculations among different groups using different codes or even the same code but on different platforms, turns out to be a difficult task. This appears retrospectively as a crucial step in the cross-validation of simulations performed by different groups working with different theories, and as a consequence, an obstacle to research progress.

Currently, output files are usually in native programming language (*e.g.* Fortran or C) binary format, where data are recorded in a given order. This has several drawbacks: (i) binary formatted files might not be portable between big-endian and little-endian platforms, or between 32-bit and 64-bit platforms; (ii) files written in Fortran cannot be easily read by C codes (and vice-versa); (iii) keeping the backward compatibility between versions may be hard while keeping in the same time a good consistency in the order of data records.

These problems are not restrained to a given scientific field, and have already been dealt with by other research communities. In particular, the meteorology community has developed the NetCDF[3,4] input/output library which is able to write and read files presenting the following characteristics:

compactness: the files are binary, allowing to save storage space;
platform-independence: the files are portable across platforms, independently of the native machine binary encoding, big or little endian, and also

of the machine precision specificity (for example the 8-byte encoding of single precision on Cray machines, instead of the usual 4 bytes);

language-independence: the NetCDF library can be called from many different programming languages, including C, Fortran, Java, Perl, Python;

easy access: the content of the files is not accessed sequentially but via alphanumeric tags, much like in XML, yielding optimal backward compatibility.

NetCDF is also recognized by other software such as Grace[5] or OpenDX[6] allowing quick and easy data treatment. Actually, NetCDF is not the only possible standard in the above-mentioned context. HDF[7] also implements a software solution that addresses the portability and extensibility problems. As NetCDF is being already used by several of the targeted software in our project, we capitalize on this format. NetCDF is also simpler to use if the data formats are flat, while HDF has definite advantages if one is dealing with hierarchical formats. Typically, we will need to describe many different multi-dimensional arrays of real or complex numbers, for which NetCDF is an adequate tool.

The idea of standardization of file formats is not new in the electronic-structure community[8] either. However, it has proven difficult to achieve without a formal organization, gathering code developers with a sufficient incentive to realise effective file exchange between software. In the EU Nanoquanta Network of Excellence (which has given birth to the “European Theoretical Spectroscopy Facility” - ETSF[9]), the standardization of file formats appeared as an explicit goal of the project. Based on a collegial agreement between ETSF members, a specification[10] has been written to describe electronic structure files encoded with NetCDF. This specification will be briefly described in the beginning of this article. To implement this specification in different software applications, the ETSF_IO library has been developed, and its full description is the subject of the latter part of this article.

2 The ETSF file format specification

The reader is advised to consult Ref.[11] for a detailed description of the specification[10]. The latter describes the structure of a valid ETSF file: it lists which data (called “variables” from now on) are mandatory, what is their shape in the file (from scalar values to n-dimensional arrays) and what are the lengths of the different array dimensions. For instance, it is specified that the atomic positions of the elements must be stored in reduced coordinates, with respect to the super-cell definition, in a double precision variable called `reduced_atom_positions` that is a 2-dimensional array of size number_of_atoms×number_of_reduced_dimensions. The required sizes are stored as NetCDF dimensions and are also described in the specification. The document introduces three normalized kinds of file contents: one for crystallographic data, one for density and potential data, and one last type for wavefunction descriptions. For each kind, lists of mandatory and optional variables are given. These three kinds are compatible and all of them may be found in one file at once. Finally, modifiers, such as physical units, are described as NetCDF attributes. In total, the specification gathers around 40 variables and 25 dimensions, a few of which have complex interactions.

3 Overview of the ETSF_IO library

The ETSF_IO library is a standard and unified software implementation of I/O routines to access ETSF format files. Besides incorporating the advantages brought by NetCDF access to files, such as backward compatibility, the ETSF_IO library is designed to keep the nice features that developers are accustomed to when using direct `read` and `write` Fortran keywords. Three points have been essential during the development of ETSF_IO: that there be no possibility of memory fault if array shapes differ between core memory and file; that there is no memory duplication when the data are accessed, whatever the shape they have in the main memory; and last but not least, that the use of NetCDF not complexify the workflow, which has been kept as close as possible to an `open-read-close` cycle.

The library is divided into three levels: the low level, the ETSF level and the utility level. The first makes some useful wrappers around NetCDF calls available, to ensure memory checking and detailed error reporting. The second, the ETSF level, corresponds to the implementation of the ETSF specifications for I/O. It provides interfaces to routines used to access one or several ETSF variables in one call, from simple read or write capabilities to more complex actions like copy, or to correctly access files split with respect to k-points, bands, or spin, as described in the specification. The third level, composed of utilities, proposes several routines for high-level data processing, such as checkers to validate against the ETSF specifications. The hierarchy of routines for the two first levels is shown on figure1.

3.1 The low-level API

The low-level routines ensure no memory faults occur, by always checking the shape of the given arguments against the dimensions declared in the NetCDF file. When given, they also check the sub-access values (see the optional ar-

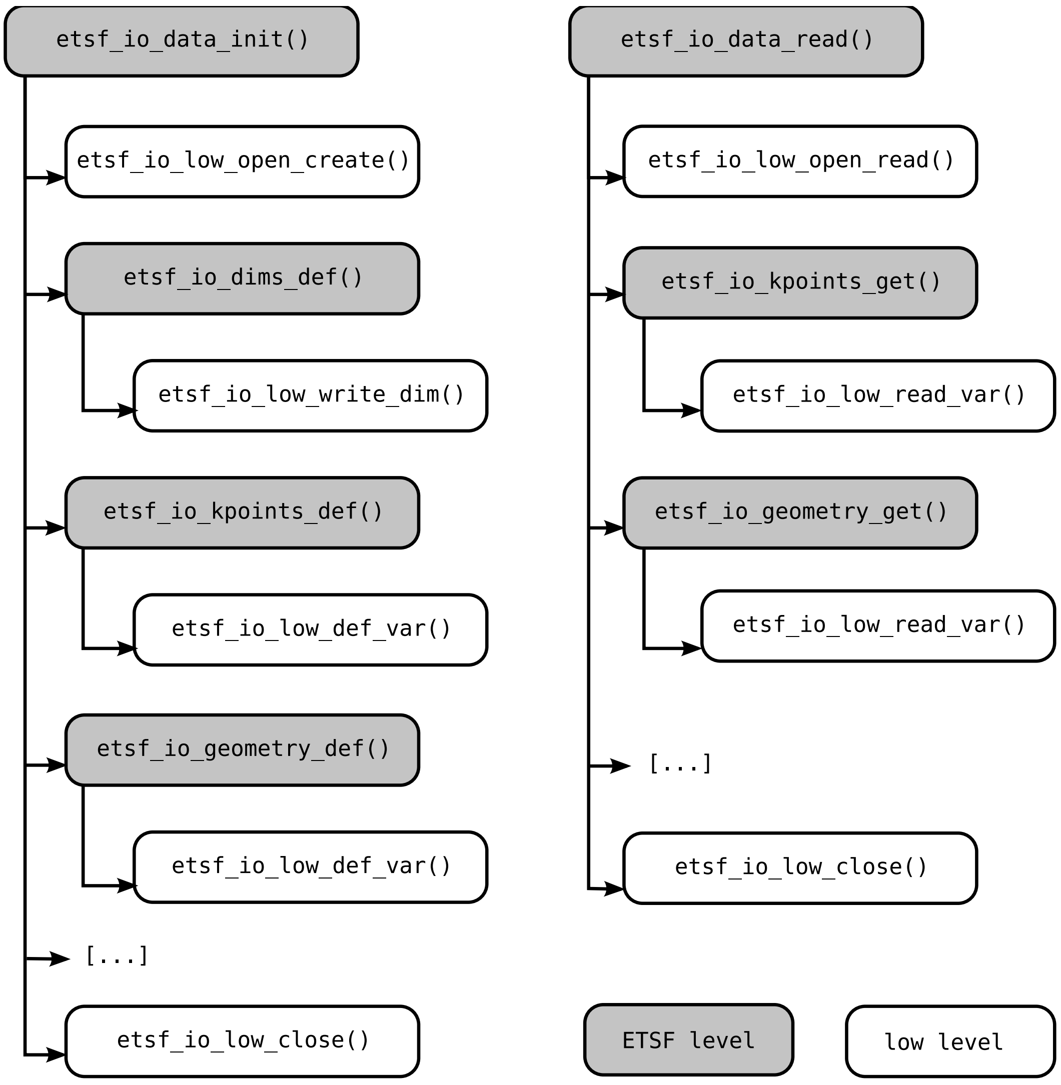

Fig. 1. Hierarchy of calls for common routines of the low and ETSF levels. The read action is shown and an equivalent scheme is valid for the write action. All the routines of this figure are public and can be called separatedly at the convenience of the user of the library if the calling program requires to split the actions in separated parts of the code.

guments `start`, `count` and `map`, as defined in the NetCDF documentation) to ensure that they will point to valid memory locations. The corresponding time overhead may be neglected in common-use cases, as shown in section 5.

Using Fortran interfaces, the API[1] is also kept minimal, types being guessed at compilation time. This keeps routine names simple and readable, *e.g.* `etsf_io_low_read_var()` or `etsf_io_low_write_dim()`.

All routines in the low-level module have an optional argument called `error_data`.

[1] Application Programming Interface

This is a Fortran type object that can store information to give rich error reporting. The action (*e.g.* inquire a NetCDF dimension id), the target of the action (*e.g.* the name of the dimension) and a string message are available to the calling routine. Moreover, the `error_data%backtrace` array can give the name of all the routines called between the error and the calling routine, helping developers to find the location of bugs. This type can be easily transformed to a string using `etsf_io_low_error_to_str()`.

The routines of the low-level API deal with several areas:

- File access, with routines to open or close them. By default, the `etsf_io_low_open_create()` routine adds the ETSF header to each new file and `etsf_io_low_open_modify()` can update the ETSF header as described in the specification. As an example, `etsf_io_low_open_read()` can check that the ETSF header is present and valid, with a minimum version value.
- Reading dimensions, variables and attributes. In addition to wrappers around the “get”-like NetCDF routines, the library has an integrated `etsf_io_low_read_flag()` to handle the “yes” or “no” attributes as required by the specification. When polling an unknown file, the routine `etsf_io_low_read_all_var_infos()` can be used to obtain the list of all variables, their names, their shapes and optionally their attribute and dimension names.
- Writing dimensions, variables and attributes. Here again, in addition to NetCDF calls there is a routine to copy at once all the attributes of a given variable or all the global attributes at once.

The low-level API extends the given NetCDF routines, making it possible to have new composite actions in one call, such as reading all variable descriptions or copying all attributes, and to ensure existing read and write actions produce no memory corruptions and provide rich error reporting.

3.2 The ETSF level

This is the core of the library and the main point of interest for developers eager to implement compatibility with the ETSF specification[10]. The main idea of this set of routines is to provide a complete access to the specification. The access to ETSF information is gathered into customized Fortran types: One for all ETSF dimensions and six others that deal with variables of interest such as the geometry description or the basis set definition. For each of these types, one can read or write all or part of its content using `etsf_io_`*`<types>`*`_[get,put]()`. There is also a routine to copy the variables related to each group from one file to another. In addition, there are all-in-one routines, to read, write or copy all variables in one call, see the three routines `etsf_io_data_[read,write,copy]()`. To be complete about the implementation, the library also defines a Fortran type called `etsf_split`, that is used to store the details of data distribution as described in the specification.

We will now take the example of the definition of the k-points mesh to illustrate the use of these routines. In this example, the k-points mesh was given using a $2 \times 2 \times 2$ Monkhorst-Pack[12] grid, which results in 4 irreducible k-points. The file has been opened earlier in the program using `etsf_io_low_open_create()` and it is referred to by the variable `ncid`. The process will follow the NetCDF one, with two parts: one at stratup to define the variables and then one to actually write them in the file. We begin thus with the definition of the variables that will be used. This must be done after having declared the dimensions using `etsf_io_dims_def()`, since the "define" routine for variables will fetch the required dimensions in the file.

```
call etsf_io_kpoints_def(ncid, lstat, error, flags = &
                         & etsf_kpoints_mp_folding + &
                         & etsf_kpoints_red_coord_kpt + &
                         & etsf_kpoints_kpoint_weights)
```

The `lstat` variable is a mandatory logical argument of all ETSF_IO routines, and is `.true.` when the routine completes without error. If `lstat` is `.false.`, the argument `error` will contain the details of what happened. In the present case the optional argument `flags` is used, since we don't want to use all variables of the kpoints group (see table 1 for the comprehensive list of variables in the kpoints group). Defining only the variables we want to use, with the `flags` argument, helps to save disk space since in NetCDF, defining a variable to a file actually create a file using the required disk space.

Later in the program, we will want to write our k-points definitions. Let us suppose that in the main program, the Monkhorst-Pack folding is stored in the array `mkfolding(3)`, the reduced coordinates in `kpt(3,4)` and the k-point weights in `wkpt(4)`, then we will call:

```
type(etsf_kpoints) :: kpoints

! The main program has already computed
! the values in kpt and wkpt.
kpoints\%monkhorst_pack_folding => mkfolding
kpoints\%reduced_coordinates_of_kpoints => kpt
kpoints\%kpoint_weights => wkpt
call etsf_io_kpoints_put(ncid, kpoints, lstat, error)
```

The attributes in the `kpoints` type are all pointers to be associated with the arrays of the main program. Therefore, there is no duplication of memory, which is especially important for larger arrays such as the coefficients of wavefunctions. The names of the elements of each type are the names of the

```
! Data type for electrons
type etsf_electrons
  integer, pointer :: number_of_electrons => null()
  double precision, pointer :: fermi_energy => null()
  [...]

  ! Attributes
  ! Units attributes for variable fermi_energy
  character(len = etsf_charlen) :: &
    & fermi_energy__units = "atomic units"
  double precision :: &
    & fermi_energy__scale_to_atomic_units = 1.0d0
  [...]
end type etsf_electrons
```

Fig. 2. Part of the customized Fortran type storing information about electrons. All elements related to ETSF variables are pointers, to avoid memory duplication between the main program and the I/O part. When a variable has an ETSF attribute, such as *units*, this attribute is present in the customized type of the variable.

variables as defined in the specification. Finally, only the associated pointers of the `kpoints` argument will be processed, giving a simple way to write one or several variables in one call. A full detailed program is given in Appendix A, and is an example of how to use the all-in-one routine. It is important to notice that variables refered by associated pointers must have already been defined as shown in the previous paragraph. Otherwise, an error will be raised.

When variables have a *unit* attribute as defined in the specifications, a flag element that indicates whether the variable uses atomic units or not, and a value for the conversion factor can be found in the customized Fortran type the variable is a member of. It is the same for other attributes such as *k_dependent*. See figure 2 for an example of a customized Fortran type with attributes.

For variables that have a dimension beginning with `max_` (*e.g.* `max_number_of_states`), the user can provide less data than declared in the dimension. To implement this simply, the Fortran type has an effective element (*e.g.* `wfs_coeff__number_of_states`) which can be set to the actual data size. Then the library routine will take care of the NetCDF calls to read or write the data at the right position in the file. See figure 3 for an example of reading in the context of a system with a different number coefficients at different k-points.

Finally, there is a simple mechanism, for the appropriate ETSF variables, to write them element by element. This can be done with respect to k-points, spin or electronic states. For instance, one may want to read the coefficients of the wavefunctions only for the second k-point (this variable is in the `etsf_main` Fortran type). To do so, one has to set the value of the element `wfs_coeff__kpoint_access` of the main type to 2 and provide the amount of

```
type(etsf_main) :: main
double precision, target :: cg(1108)

! We only want to read the values of the second k-point
! There are 1108 of them among 1135.
main\%wfs_coeff__number_of_states = 1108
main\%wfs_coeff__kpoint_access = 2
main\%coefficients_of_wavefunctions => cg
call etsf_io_main_get(ncid, main, lstat, error)
```

Fig. 3. Example of k-point selective read access, in the case of an ETSF dimension beginning with `max_`. In this example, the ETSF variable `coefficients_of_wavefunctions` is declared with the following dimensions: `max_number_of_states` and `number_of_kpoints` being respectively 1135 and 3. But the second k-point has only 1108 relevant values, so we use the sub-access mechanism of the library, providing values to the ad-hoc integer element of the `etsf_main` Fortran type.

data for just one state. The library will handle the NetCDF calls to read or write the data at the right position in the file. See figure 3 for an example of reading in the context of a system with 3 k-points.

Based on the low-level API, the ETSF-level thus provides routines that can safely access from just one variable to a complete set of variables. Available actions are the usual "read" and "write", plus "define", and the library also provides a "copy" action. For complex variables, the library provides a simple way to access part of the data. The complete list of associations between ETSF variables and customized Fortran types can be found in table 1.

4 Accessing valid ETSF files

The library also provides high-level routines for optional actions, such as validity checking, or specialized functions, such as element name retrieval. They are separated into two Fortran modules, one for file handling called `etsf_io_file` and one for convenient routines called `etsf_io_tools`.

For parallel jobs lacking parallel IO, it is possible for each process to create its own ETSF file. This possibility is called 'split' in the ETSF file format specifications. Since the specification describes how to organise several split files, the library provides a merge routine to be able to create a unique file out of several. The merge routine will check the compatibility of the files and create a new ETSF file with gathered data. The specifications also define several kinds of ETSF files, such as a crystallographic file, or a wavefunction file. The identification of the file type is done on the availability of a given set

etsf_basisdata contains:	*etsf_main* contains:
`basis_set`	`density`
`kinetic_energy_cutoff`	`exchange_potential`
`number_of_coefficients`	`correlation_potential`
`reduced_coordinates_of_plane_waves`	`exchange_correlation_potential`
	`coefficients_of_wavefunctions`
	`real_space_wavefunctions`
etsf_geometry contains:	*etsf_electrons* contains:
`space_group`	`number_of_electrons`
`primitive_vectors`	`exchange_functional`
`reduced_symmetry_matrices`	`correlation_functional`
`reduced_symmetry_translations`	`fermi_energy`
`atom_species`	`smearing_scheme`
`reduced_atom_positions`	`smearing_width`
`valence_charges`	`number_of_states`
`atomic_numbers`	`eigenvalues`
`atom_species_names`	`occupations`
`chemical_symbols`	
`pseudopotential_types`	
etsf_kpoints contains:	*etsf_gwdata* contains:
`kpoint_grid_shift`	`gw_corrections`
`kpoint_grid_vectors`	`kb_formfactor_sign`
`monkhorst_pack_folding`	`kb_formfactors`
`reduced_coordinates_of_kpoints`	`kb_formfactor_derivative`
`kpoint_weights`	

Table 1
Association between customized Fortran types and ETSF variables.

of variables. The library then provides routines to check that a file contains the right variables with the right dimension definitions. These two actions, merge and check, are available both as routines and as a separate executable.

Finally, the *tools* module provides a convenient routine to get the names of elements present in a file. In the specification, atom names can be read from these three variables: `atomic_numbers`, `atom_species_names` and `chemical_symbols`.

The first listed variable is preferred, and the routine tries to provide both atomic numbers and string names when the data are present in the input file.

5 Library engineering and performance considerations

The library is shipped with several tutorials and a complete documentation of the API, detailing all input and output arguments. The tutorials contain highly documented source codes of stand-alone programs illustrating various aspects of the library. There are currently six tutorials:

- The first tutorial details the steps required to create a density file, using high-level routines (*i.e.* `etsf_io_data_<action>()`). It shows how to use the pointers, especially the so-called "unformatted" ones. The latter are types introduced in ETSF_IO and are used to map differently shaped arrays between the ETSF definition and the main program memory. This first tutorial is provided in appendix A.
- The second tutorial introduces the group-level routines and explains how to access only sub-parts of arrays. This sub-access is possible when one array has a dimension on spin, k-points or states. In this tutorial a wavefunction file is created and the coefficients of wavefunctions are written for one k-point at a time.
- The third tutorial shows how to use the high-level modules `etsf_io_file` and `etsf_io_tools` to check the conformance of an input ETSF file to crystallographic specifications, and then to read atomic coordinates and names to create a simple XYZ file.
- The fourth tutorial shows how to use the split definitions in the specification to handle MPI-parallelized computations. This tutorial creates a density file with a parallelization on z planes. Each process computes a Gaussian in its own z planes and creates an ETSF file with a split on `number_of_grid_points_vector3`.
- The fifth tutorial shows how to write an ETSF file with additional non-ETSF variables. This tutorial is not focused on the low-level API but uses it in several areas. The additional variables are defined and written directly by using the low-level API.
- The sixth tutorial introduces the read actions in a simple case, when we know that the file conforms to the specification.

The library is labelled 1.0 and we guarantee that the API will not change during all versions 1.x. There will be additions and also new optional arguments to existing routines but currently available ones will be kept, to ensure a stable interface to developers. To reinforce this, a complete set of unitary tests are provided with the library, performing up to 600 tests on routine behaviours, and checking their stability among other things.

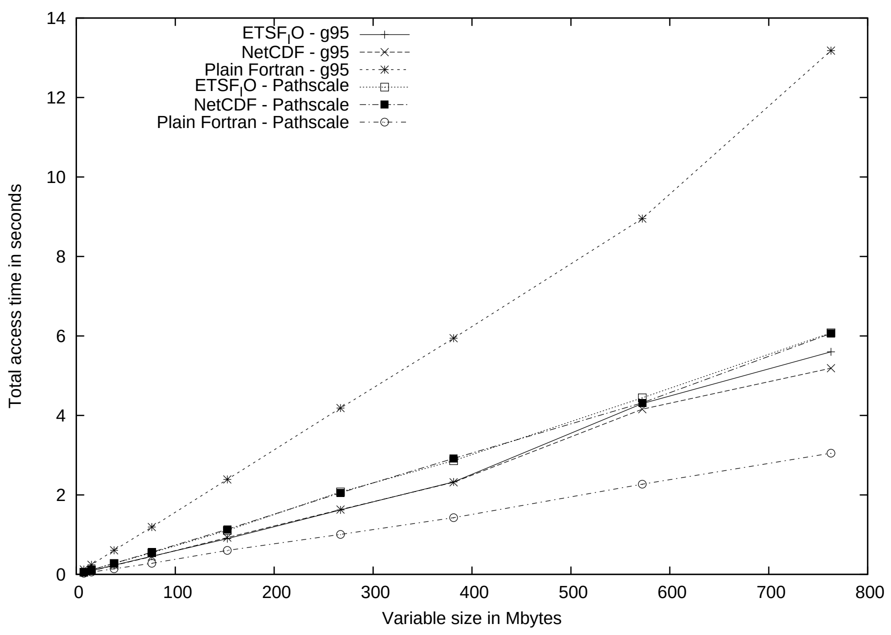

Fig. 4. Measurement of times for writing using plain `write()` Fortran, direct NetCDF calls or routines provided by ETSF_IO. The test result is the time taken to write ten accesses of the amount of data printed on x axis.

The performance of the ETSF_IO library is mainly due to that of the NetCDF library. The dimension-checking only brings an negligible overhead, as shown on figure 4.

6 Current implementations

The library is included in the ABINIT[13] Density Functional Theory[14,15] (DFT) code since version 5.3, and can produce restart files with the wave-function information and the density. It will also be used as a starting point for excited-state calculations in time-dependent DFT or the GW approximation[16] from version 5.5.

The GWST code [17] implements a converter from the ETSF format (since v0.8 of the library) to its specific binary format. As an example of the working implementation, we show below the results of two GW calculations with GWST based on ground-state wavefunctions calculated with two different codes. ABINIT and SFHINGX[18] were used, using identical silicon pseudopotentials made with the fhi98pp[19] code, to create two ground state wavefunction files, one in ETSF and the other in SFHINGX's own NetCDF format. The

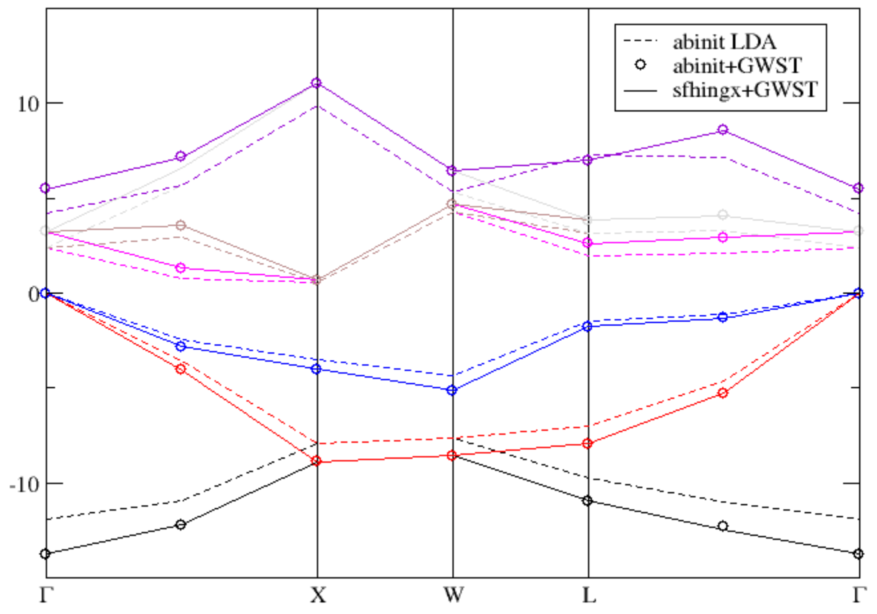

Fig. 5. Comparison of quasi-particle corrections (in eV) for Si, using the GWST code and two input wavefunction files, generated respectively by ABINIT and SFHINGX.

two files were used as inputs to GWST, and Fig. 5 shows the excellent agreement obtained. The absolute numbers should not be taken literally, as the k-point grid and number of unoccupied states used are not fully converged, but clearly the inputs from the completely independent codes can be used interchangeably.

Still in the post-ground-state calculation area, support for reading ETSF format files by the SELF[20], DP[21], and EXC[22] is under development.

As far as visualization is concerned, basic reading of crystallographic data and densities is available in V_Sim[23] with an independent implementation of the file format specification. The forthcoming C implementation of the ETSF_IO library will be useable in V_Sim and other commonly used visualization software very simply.

7 Conclusions and Outlook

We have presented a new Fortran library focused on electronic structure codes. It provides a unified and code-independent way to read and write restart and output files. Based on NetCDF, it is designed for backward-compatibility and platform-independence. The layer over NetCDF provided by ETSF_IO ensures robust I/O access, simple programming with integrated routines to handle from one to all variables at one time, without performance loss compared to NetCDF.

The stable 1.0 release can be retrieved on the ETSF web site[24]. The ongoing development of the library is now focused on the implementation of the API in C.

8 Acknowledgments

We acknowledge the help, interest and encouragements of several Nanoquanta members including R.W. Godby, L. Reining, A. Cucca, C. Hogan, A. Rubio, T. Patman, G. Bruant, M. Marques, F. Sottile. This work was carried out in the Nanoquanta EU Network of Excellence (NMP4-CT-2004-500198). M.J.V. acknowledges support from Marie Curie fellowship MANET (MEIF-CT-2005-024152). X.G. acknowledges support from the Communaute Francaise de Belgique through the ARC 07/12-003, and the "Pole d'attraction interuniversitaire PAI/UIAPP phase VI".

A Create a density file

```
program create_a_crystal_den_file

  use etsf_io

  integer :: i

!!  All routines from the group level requires two output arguments:
!!   * lstat which is a logical. When .false. something goes wrong
!!     in the routine and the action is aborted. No actions are
!!     atomic, which means that if lstat is .false., the status of
!!     the NetCDF file (what have been done) is not guarantee.
!!   * error_data which a of type #etsf_io_low_error. It contains
!!     many information about the error if lstat is .false..
!!     One can use etsf_io_low_error_to_str() to get a
!!     character(len = etsf_io_low_error_len) describing the error,
!!     or one can implement one itself since the type is public and
!!     documented.
  logical                 :: lstat
  type(etsf_io_low_error) :: error_data
```

```
!!  To create a NetCDF, we need to give at creation time all the
!!  dimensions that define the variables. The file will then be
!!  allocated on disk and may be write with values later. All
!!  dimensions declared in the ETSF specifications are stored
!!  in a type called etsf_dims. Some of these dimensions are fixed
!!  by the specification such as character_string_length and will
!!  be set by the etsf_io_data_init() routine itself. Other values
!!  are free to be chosen.
  type(etsf_dims)         :: dims

!!  To write values in one call into an already defined ETSF file,
!!  the type etsf_groups is used as a container for several groups.
!!  Here our container will have associated pointers on an
!!  etsf_geometry and an etsf_main types. So we declare them. All
!!  the structures used in this library are only containers and do
!!  not have the allocated memory. This is done to avoid memory
!!  duplication when using the library with a code with its own
!!  variables. So we also need some variables (in a real case, they
!!  are declared in the main program) to stored our density and
!!  geometric information.
  ! Specific variables required by the library
  type(etsf_groups_flags)     :: flags
  type(etsf_groups)           :: groups
  type(etsf_geometry), target :: geometry
  type(etsf_main), target     :: main
  ! Variables that are declared in the main program in a real case
  double precision, allocatable, target :: density(:)
  integer, target                       :: space_group
  double precision, target              :: primitive_vector(3, 3)
  double precision, allocatable, target :: red_atom_pos(:,:)
  integer, allocatable, target          :: atom_species(:)
  character(len=2), allocatable, target :: chemical_symbols(:)
  integer, allocatable, target          :: red_sym_mat(:,:,:)
  double precision, allocatable, target :: red_sym_trans(:,:)

!!  We will create for example a file for the density
!!  of the silane molecule, without spin nor spin-orbit, 1 k-point.
!!  We imagine that the molecule no symmetry except identity.
  dims\%max_number_of_coefficients = 1400
  dims\%max_number_of_states = 6
  dims\%number_of_atoms = 5
  dims\%number_of_atom_species = 2
  dims\%number_of_components = 1
  dims\%number_of_grid_points_vector1 = 36
  dims\%number_of_grid_points_vector2 = 36
  dims\%number_of_grid_points_vector3 = 36
  dims\%number_of_kpoints = 1
  dims\%number_of_spinor_components = 1
  dims\%number_of_spins = 1
  dims\%number_of_symmetry_operations = 1

!!  Now that dimensions have been stored in the appropriated
!!  structure, we can call the etsf_io_data_init() routine itself.
!!  The 'groups' argument is very important. It will tell which
!!  variables will we allocated on disk. All variables are gathered
!!  by groups and one can choose one or several groups to be
!!  defined. To do it, use the flags from #FLAGS_VARIABLES, in a
!!  summation for each group in the etsf_groups_flags structure. By
!!  default no group will be defined, to add the geometry group, we
!!  will use the value etsf_geometry_all (from #FLAGS_VARIABLES) ;
!!  and to add the density variable (from the main group), and only
!!  this one, we will use etsf_main_density.
!!
!!  Other arguments of the routine are quite easy to understand.
!!  The optional k_dependent argument is here to handle the case of
!!  reduced_coordinates_of_plane_waves which shape depends on the
!!  value of this attribute. If k_dependent is given .false.
!!  (default is .true.), then all variables with this attribute will
!!  be labelled "no" and the variable
!!  reduced_coordinates_of_plane_waves will be a two dimensional
!!  array.
  flags\%geometry = etsf_geometry_all
  flags\%main     = etsf_main_density
  call etsf_io_data_init("create_a_crystal_den_file.nc", flags, &
                      & dims, "Tutorial, create a density file", &
                      & "Created by the tutorial example", &
                      & lstat, error_data)
!!  The required variables for a density file are in etsf_geometry
!!  and in etsf_main, that's why the groups argument is the sum of
!!  the two flags.
!!
!!  We can now, handle the error, if one occured. The method
!!  etsf_io_low_error_handle() is used to print the contains of an
!!  error type on the standard output. If one is interested on
!!  printing the error on something different than the standard
!!  output, one should convert the error into a
!!  character(len = etsf_io_low_error_len) with
!!  etsf_io_low_error_to_str() before.
  if (.not. lstat) then
    call etsf_io_low_error_handle(error_data)
    stop
  end if
```

```
!!  At this time of the example, the disk space to store the density
!!  and the geometric information has been reserved. In a real case,
!!  we let the main program computing the density and setting up the
!!  geometric information.
  ! The main program allocate memory for its computation.
  allocate(density(36 * 36 * 36))
  allocate(red_atom_pos(3,5))
  allocate(atom_species(5))
  allocate(chemical_symbols(2))
  allocate(red_sym_mat(3, 3, 1))
  allocate(red_sym_trans(3, 1))

  ! The main program compute all symmetries and set up the
  ! positions...
  space_group = 1
  primitive_vector = reshape( (/ 10,  0,  0, &
                              &  0, 10,  0, &
                              &  0,  0, 10 /), (/ 3, 3 /))
  red_sym_mat = reshape( (/ 1, 0, 0, &
                         & 0, 1, 0, &
                         & 0, 0, 1 /), (/ 3, 3, 1 /))
  red_sym_trans = reshape( (/ 0, 0, 0 /), (/ 3, 1 /))
  red_atom_pos = reshape( (/ 0.5d0, 0.5d0, 0.5d0, &
                          & 0.6d0, 0.6d0, 0.6d0, &
                          & 0.6d0, 0.4d0, 0.4d0, &
                          & 0.4d0, 0.4d0, 0.6d0, &
                          & 0.4d0, 0.6d0, 0.4d0 /), (/ 3, 5 /))
  atom_species = (/ 2, 1, 1, 1, 1 /)
  chemical_symbols = (/ "H ", "Si" /)

  ! We compute the density with a powerful algorithm.
  density = (/ (0.d0 + i, i = 1, 36 * 36 * 36) /)

!!  Before calling the etsf_io_data_write() routine, we associate
!!  the pointers of our group types to the main program memory data.
!!  Only associated pointers will be written. All other defined
!!  variables will be let untouched. Some variable are defined with
!!  a type called etsf_io_low_var_double or etsf_io_low_var_integer.
!!  These variables are arrays which could have a different shape in
!!  the main program and in the specifications. For instance, our
!!  density is 1D only whereas in the specification the density is
!!  5D. So we use the attribute \%data1D of the structure
!!  etsf_io_low_var_double for the density. This will work because
!!  data in the main program memory has the same number of elements
!!  than the space defined in the ETSF file AND data are ordered in
!!  the same way (elements along X axis are varying quicker than
!!  along Y or Z).
```

```
  ! We associate the geometry
  geometry\%space_group => space_group
  geometry\%primitive_vectors => primitive_vector
  geometry\%reduced_symmetry_matrices => red_sym_mat
  geometry\%reduced_symmetry_translations => red_sym_trans
  geometry\%atom_species => atom_species
  geometry\%reduced_atom_positions => red_atom_pos
  geometry\%chemical_symbols => chemical_symbols
  ! We associate the main data
  ! We don't want to duplicate the density data even if ours is 1D
  ! and ETSF is 5D, so we use the unformatted pointer in the
  ! etsf_main structure.
  main\%density\%data1D => density
  ! We associate our two group in the container.
  groups\%geometry => geometry
  groups\%main => main

  ! We write.
  call etsf_io_data_write("create_a_crystal_den_file.nc", &
                        & groups, lstat, error_data)
  ! We handle the error
  if (.not. lstat) then
    call etsf_io_low_error_handle(error_data)
    stop
  end if

  ! The main program will deallocate its own memory.
  deallocate(density)
  deallocate(reduced_atom_positions)
  deallocate(atom_species)
  deallocate(chemical_symbols)
  deallocate(reduced_symmetry_matrices)
  deallocate(reduced_symmetry_translations)
end program create_a_crystal_den_file
```

References

[1] Murray-Rust, P. and Rzepa, H., J. Chem. Inf. Comp. Sci. **39** (1999) 928.

[2] url: http://http://www.wwpdb.org/docs.html.

[3] Rew, R. and Davis, G., IEEE Computer Graphics and Applications **10** (1990) 76.

[4] url: http://www.unidata.ucar.edu/software/netcdf.

[5] Turner, P. J. and *et al.*, url: http://plasma-gate.weizmann.ac.il/Grace/.

[6] Abram, G., Kirchner, P., Thompson, D., Tignor, M., and *et al.*, url: http://www.opendx.org/.

[7] url: http://hdf.ncsa.uiuc.edu.

[8] Gonze, X., Zerah, G., Jakobsen, K., and Hinsen, K., Psi-k Newsletter **55** (2003) 129.

[9] url: http://www.etsf.eu.

[10] Gonze, X. et al., url: http://etsf.eu/index.php?page=standardization.

[11] Gonze, X. et al., Specification of an extensible and portable file format for electronic structure and crystallographic data., accepted for publication in Computational Materials Science, available on arXiv at http://arxiv.org/abs/0805.0192.

[12] Monkhorst, H. and Pack, J., Phys. Rev. B **13** (1976) 5188.

[13] Gonze, X. et al., Zeit. Kristall. **220** (2005) 558.

[14] Hohenberg, P. and Kohn, W., Phys. Rev. **136** (1964) 864.

[15] Kohn, W. and Sham, L., Phys. Rev. **140** (1965) A 1133 .

[16] Hedin, L., Phys. Rev. **139** (1965) A796.

[17] Rojas, H., Godby, R., and Needs, R., Phys. Rev. Lett. **74** (1995) 1827.

[18] Boeck, S. and *et al.*, url: http://www.sphinxlib.de.

[19] Fuchs, M. and Scheffler, M., Comp. Phys. Comm. **119** (1999).

[20] Marini, A. and *et al.*, url: http://statistics.roma2.infn.it/ self/index.html.

[21] Olevano, V. and *et al.*, url: http://www.dp-code.org/.

[22] Sottile, F. and *et al.*, url: http://www.bethe-salpeter.org/.

[23] Caliste, D., Billard, L., and D'Hastier, O., url: http://www-drfmc.cea.fr/sp2m/L_Sim/V_Sim/.

[24] Caliste, D., Pouillon, Y., Verstraete, M., and Olevano, V., url: http://etsf.eu/index.php?page=tools.