D1.4

Second release of MAX software: Report on first common APIs, data structures and domain-specific libraries

Stefano Baroni, Augustin Degomme, Pietro Delugas, Stefano de Gironcoli, Andrea Marini, Davide Sangalli, Daniele Varsano, Fabrizio Ferrari Ruffino, Andrea Ferretti, Alberto Garcia, Luigi Genovese, Paolo Giannozzi, Anton Kozhevnikov, Ivan Marri, Nicola Spallanzani, and Daniel Wortmann

Due date of deliverable	30/11/2020 (**month 24**)
Actual submission date	30/11/2020
Final version	30/11/2020
Lead beneficiary	SISSA (participant number 2)
Dissemination level	PU - Public

Document information

Project acronym	MAX
Project full title	Materials Design at the Exascale
Research Action Project type	European Centre of Excellence in materials modeling, simulations and design
EC Grant agreement no.	824143
Project starting/end date	01/12/2018 (month 1) / 30/11/2021 (month 36)
Website	www.max-centre.eu
Deliverable no.	D1.4

Authors	Stefano Baroni, Augustin Degomme, Pietro Delugas, Stefano de Gironcoli, Andrea Marini, Davide Sangalli, Daniele Varsano, Fabrizio Ferrari Ruffino, Andrea Ferretti, Alberto Garcia, Luigi Genovese, Paolo Giannozzi, Anton Kozhevnikov, Ivan Marri, Nicola Spallanzani, and Daniel Wortmann.

To be cited as	Baroni et al. (2020): Second release of MAX software: Report on first common APIs, data structures and domain-specific libraries. Deliverable D1.4 of the H2020 CoE MaX (final version as of 30/11/2020). EC grant agreement no: 824143, SISSA, Trieste, Italy.

Disclaimer

Contents

Executive Summary

This report describes the M24 releases of the MAX flagship codes and focuses mostly on their reorganisation as well as on the libraries extracted, developed, adopted within them.

The current structure of the codes and of the SIRIUS platform is described, together with other general software improvements, in code-specific subsections. These subsections also provide a quantification, when applicable, of the work done so far in terms of new libraries and submodules.

A full update on the status of the development of MAX libraries was already given in the D1.3 deliverable at month 18. Most of the libraries have by then already reached beta or production stages and most of the further performed activity consists in testing and re-usage. The update on libraries status will thus mention only those cases for which important changes in the structure and interfaces of the APIs have occurred in the last six months.

Many of the actions reported in this document are directly related and provide input to performance portability (WP2) and algorithmic improvement (WP3) activities. Results reached on these sides are only reported in D2.2 and D3.3 documents, that have been prepared together with the present one by the same code developers.

1 Introduction

MAX focuses on selected flagship codes: widely-used applications, based on rather diverse models, mainly oriented to structural, electronic, magnetic properties and to spectroscopies of materials from first principles, encompassing Density Functional Theory (DFT) and Many Body Perturbations Theory (MBPT) methods. A description of the main features of the flagship codes can be found on the MAX website at http://www.max-centre.eu/codes-max.

The MAX flagship codes at M24 releases are strongly influenced by the actions taken during these two years of the project. After the development and testing of the first year, many of the results obtained by the MAX WPs are starting to appear in the production versions.

- The modularisation and the creation on new libraries and interfaces planned in WP1, have significantly improved the development sustainability and the flexibility of the codes.
- For each code, the regular MPI/OpenMP version and the one for heterogenous architectures are converging towards a common code base, if not already in place. In particular, the high-level parts of the codes have been made more portable and architecture agnostic with the introduction of new general interfaces for domain-specific computational kernels. This part of the work proceeds as planned in WP2 and has been done in tight collaboration with the HPC experts from WP4 (exascale & codesign).
- The adoption of the algorithmic improvements developed by WP3 either directly or via libraries.
- Improvement of functionalities, interoperability and I/O to meet the requests coming from WP5 (data) and WP6 (demonstration).

We will present the new releases in Sec. 2 describing the main improvements and new features introduced in each code. We will focus on the structure and organisation of the codes, the release of new applications, and the adoption of new libraries and interfaces. Progresses and achievements on the performance portability and algorithmic improvement sides are described in detail in the associated reports of WP2 (D2.2) and WP3 (D3.3). We will thus mention such aspects only briefly.

As concerns libraries, a recent update was given at month 18 [1]. At that time a significant fraction of the libraries had already reached a stable point (production or beta stage) and most of the activity on them was dedicated to re-usage and testing. In Sec. 3 we thus provide an update only for those libraries that have undergone important changes during this period. We also describe some new libraries that have been introduced in the development plan only after M18.

In Sec. 4 we conclude the report summarising the progress and status of WP1 and discussing the most important issues to be dealt with in the last year of the project, which also constitute requirements or inputs for other WPs.

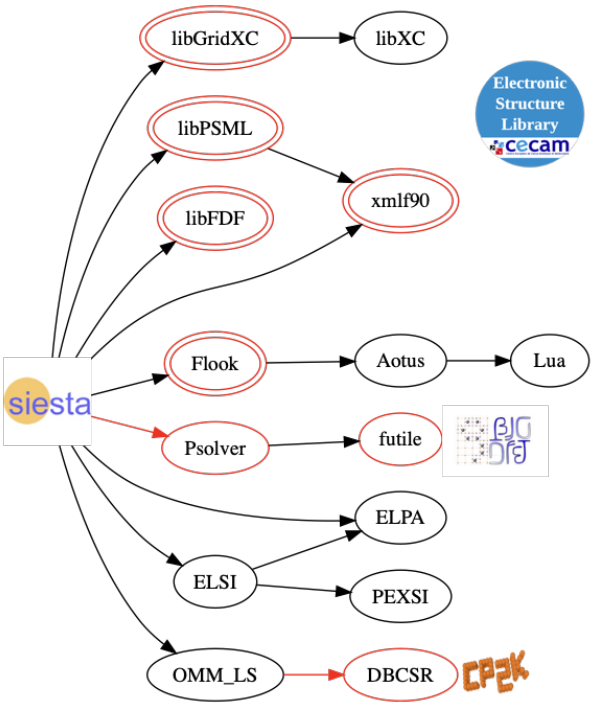

Figure 1: Sketch of the modularisation of the Siesta program. Each bubble represents a library. Dependencies are indicated by arrows, and logos at the end of a path show the MAX code in which the libraries originate. Bubbles with double contours mark those libraries originating in Siesta, and made available also through the Electronic Structure Library

2 Work on the different codes

2.1 Siesta

In the past year, the Siesta program has seen substantial enhancements. We report in D2.2 the major milestone of GPU acceleration of the diagonalization solver, which was achieved, following the spirit of separation of concerns, via the use of appropriate libraries. Work on modularisation, and on interface refactoring to enable further modularisation, has continued since M12. We can summarise the enhancements in the following list:

- Incorporation of the PSolver library. This provides the important capability of performing simulations without imposing periodic boundary conditions.
- Extension of the interfaces for creation of parallel distributions for the real-space grid. This was needed to simplify the interface with PSolver, and also opens the door to the use of alternative FFT libraries in Siesta.
- Refactoring of the OMM (Orbital Minimisation Method) linear-scaling capability, using the sparse matrix-matrix multiplication library DBCSR and the general matrix handling layer library `MatrixSwitch`. This is one of the actions featured in WP3, and can have significant impact on the demonstrators of WP6 that deal with systems with an energy gap in the electronic structure.

- Improvements in solvers:
  - GPU-acceleration of the diagonalization solver. Further improvements from work in the underlying libraries are automatically available to the code as the appropriate interfaces are in place.
  - Enabled parallel-over-k calculations for non-collinear spin.
- Performance increase for tall+skinny matrices in `TranSiesta`
- Further enhancements to the resilience of the code in massively parallel environments.

These developments have taken the modularisation of Siesta to a very significant level, as can be seen in Fig. 1. The now very large number of potential dependencies of the code calls for enhancing the flexibility and robustness of the building and deployment system. This is a very demanding endeavour, but we are making progress by decoupling the tasks of configuration management, the settings of options for compilation, and the compilation and deployment, and handing them over to different tools. In particular, we are using `git submodules` for the first task, and leverage the power of a pre-processor for the second one.

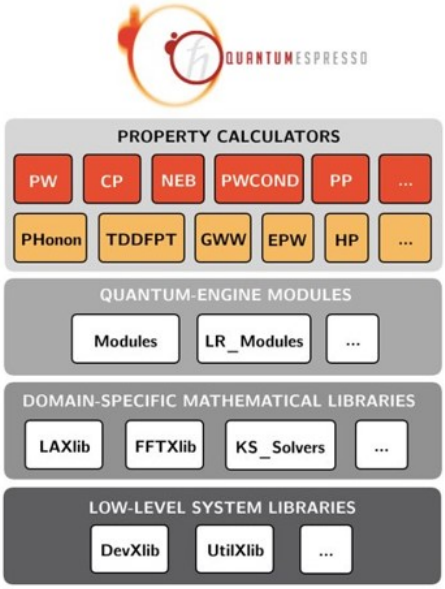

Figure 2: Code layers in QUANTUM ESPRESSO. The **high-level code** is constituted by property calculators and quantum engines. The **quantum-engine modules** provide all those electronic structure specific functionalities frequently used in the applications. The **mathematical libraries** provide access to compute-intensive functionalities. **Low-level utility libraries** provide APIs for the basic management of data, parallelism, timing and errors.

2.2 QUANTUM ESPRESSO

Since the `qe-6.5` release at month 12 (M12), two further QUANTUM ESPRESSO versions have been released: `qe-6.6` in July 2020 and in this M24 MAX release, tagged as `qe-6.7`. In both cases a corresponding GPU-enabled CUDA Fortran version has been made available at the same time. The GPU-enabled version has also undergone two more minor updates in the same 12-month period. The synchronisation process of the accelerated version is now well-integrated in the development process and the two versions are quickly converging towards a unified code base. In particular, it is now possible to compile and use the GPU-enabled version also for usage on CPU's of non-accelerated platforms.

Code Organisation. The code has been progressively reorganised into different layers, separating the high-level code of the applications, quantum-engines or property calculators, from the code parts that implement lower-level functionalities. Currently we have 4 different layers in the code as depicted in Fig. 2.

The application layer is constituted by partially independent "property calculators":

- `PW`, containing the `pw.x` quantum engine for self consistency, plus a set of related property calculators;
- `CPV`, containing the `cp.x` quantum engine for performing Car-Parrinello ab initio molecular dynamics;
- `PP`, containing a wide range of post-processing applications;

- `atomic`, providing an all-electron solver for atomic problems, plus utilities for generating and testing pseudopotentials and PAW datasets;
- `PHonon`, providing applications for vibrational and dielectric properties of solids using DFPT;
- `TDDFPT`, for computation of electronic excitation spectra using TDDFPT;
- `HP`, computing on-site and inter-site Hubbard correction terms to DFT using DFPT.
- `EPW`, computing electron-phonon interactions coefficients and related properties.

The application layer depends upon lower-level libraries and interfaces layers:

- A few domain-specific algorithmic modules:
  - `Modules`, a general interface module containing many common functionality calls;
  - `LR_modules`, containing common functionalities for linear-response codes;
  - `UPFlib`, described above.
- Computation-intensive mathematical Kernels `KS_solvers`, `FFTXlib`, `LAXlib`.
- `UtilXlib`: a low-level general utility library for environment initialization, timing, logging and error handling.

In total we have 8 application modules; 3 domain-specific algorithmic modules, 3 computation-intensive mathematical kernels that can be separately compiled and are part of the MAX library bundle; one low-level utility library, also belonging to the MAX library bundle. Soon after the current release the development version will also start using the `DevXlib` library making the management of offloaded data transparent and architecture agnostic. Similar abstraction will be also done for the computations of the exchange-correlation functionals on 3D grids, which will be usage of the recently developed `XC_lib`.

Build system. `autoconf` has been updated and simplified. An alternative build system based on `CMake` has been added. It streamlines the experimentation of MAX on external libraries and provides a better integration with the `spack` tool and the exploration of alternative toolchains. External libraries used by QUANTUM ESPRESSO are being downloaded and collected into a single directory, `external`, and their version control is done using `git` submodules.

Modularization. Most of the work has been done in libraries and interfaces with the mathematical kernels, integrating the work done in WP2 for performance portability.

- `LAXlib`: it is no longer needed to include a fortran module, with related portability problems, to call routines from LAXlib. Instead of a module, a file 'laxlib.fh' is included, containing fortran interfaces. Descriptors have been simplified and are no longer structures but simple integers.

- `FFTXlib`: the build and testing systems have been improved. It is now possible to choose at build time between the "pencil" (1D-1D-1D) and "slab" (1D-2D) decomposition of 3D FFT grids.
- `KS_Solvers`: new iterative diagonalization algorithms ParO, PPCG, RMM-DIIS, developed in the frame of WP3 actions, have been made available.

In addition, the first kernel of a new portable library of pseudopotential-related code, `UPFlib`, has been released. A further `XC_lib` library of exchange-correlation related code, fully supporting `libxc`, is close to be released.

New developments. are mostly in the field of Hubbard-corrected functionals and in Density-Functional perturbation theory (DFPT), for which QUANTUM ESPRESSO provides an effective development base. Concerning the former, Hubbard inter-site V parameters (DFT+U+V) and multiple Hubbard U parameters per atom have been implemented into several codes; the calculation of forces and stress has been extended to orthogonalised Hubbard manifolds. Concerning DFPT, we mention in particular the implementation of the Sternheimer algorithm in the turboEELS code, the extension of the PHonon code to the non-collinear case with magnetization, and the many improvements and extensions to EPW.

2.3 FLEUR

Work on the FLEUR code has centred around some more advanced refactoring with the aim of improving the usability of the code, of separating property calculators from the more computational intensive kernels and of incorporating new functionality. While most of the effort focusing on performance optimisation and improvement or on new functionality was done in collaboration with WPs 2 and 3, respectively, the refactoring and modularisation effort is reported here.

In detail the new FLEUR release includes:

- **A refactored property calculator framework.**
  The evaluation of spectral properties is one of the most frequently used tasks performed after achieving self-consistency in a DFT calculation. Such calculations include the output of bandstructure data, of integrated quantities like the density of states as well as of more complex properties like MCD spectra or projections of the data on particular regions of space. The corresponding code has undergone significant changes with the aim of creating a more homogeneous users experience, and o f encapsulating the functionality for easier extensions and maintenance. A hierarchy of corresponding programming objects has been created with a well-defined API and corresponding IO.

- **Restructuring the input generator of FLEUR.**
  As a simulation using the FLEUR code requires a large amount of simulation parameters to be specified, a dedicated input-generator is provided to create default and recommended values for most of these parameters from simple structural input. This pre-processing step was redesigned to allow for a clearer separation of the different steps performed in setting up a FLEUR calculation. In particular, tasks typically for an initial setup are moved to the pre-processing step to allow for better automatisation of the calculations. This work was largely triggered by the requirements identified in WPs 5 and 6.

- **Modularisation of wavefunction operations.**
  One of the potentially most time-consuming operations the FLEUR code performs is the evaluation of matrix elements of the exchange operator as needed for example by hybrid functional calculations. One of the main objectives during the course of this project is the implementation of an efficient set of routines for this code. The required refactoring is close to being finished and the key routines of the LAPWlib are being tested.

In addition, we continued our work to encapsulate the functionality as outlined above into new and existing modules and libraries and also worked on including these libraries into FLEUR. We are most strongly involved in the use and development of the LAXlib, the juDFT and the LAPWlib, but also follow the developments of the DevXLib as well as the various FFT libraries within MAX .

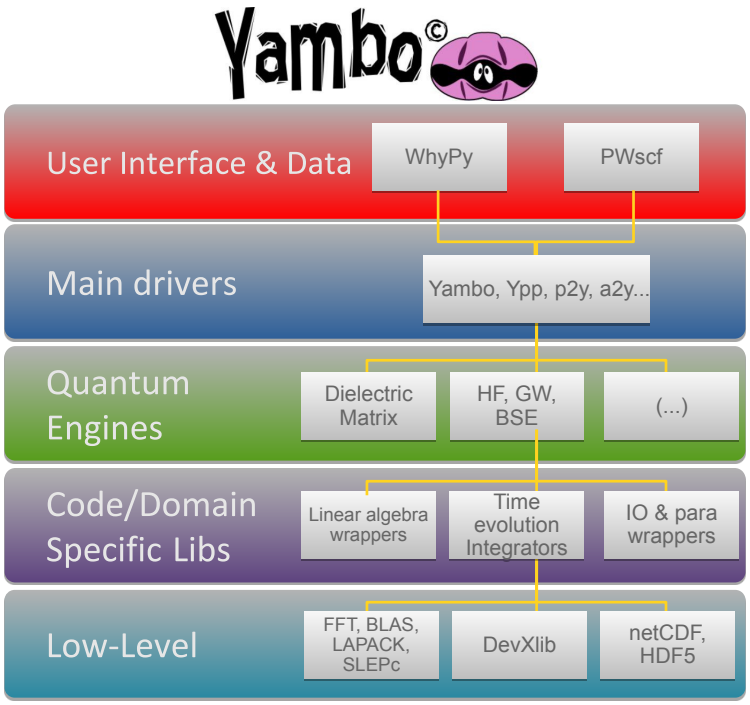

Figure 3: Code layers in YAMBO. The code can be driven directly or by using the `whypy` python layer. The high-level code is constituted by main drives that use the `Ydriver` library. These drivers call the second–level drivers that are quantum engines that perform property calculations. These interact via the modular structure described later with the domani specific and low–level libraries.

2.4 YAMBO

In the following we list the main actions and achievements accomplished during the M12-M24 perdio (Dic 2019 - Nov 2020).

New Libraries. Stable, well-tested and portable sections of the code have been extracted and turned into standalone libraries. The process has been fully terminated for the driver library (`Ydriver`) that is now hosted on a dedicated git repository. `Ydriver` can now be self compiled and linked to a test program released with the library itself. A similar process is undergoing for the I/O library (`Yio`).

Modularization

- **FFT**. In versions 4.x, YAMBO was able to accommodate only one FFT grid at a time. This had some important consequences: it was difficult to manipulate and impossible to adapt the grid size to the different code sections. Moreover any addition would have impacted on the already large infrastructure of data and variables connected to the FFT operations. A dedicated module has been introduced to avoid these limitations:

```
type FFT_t
  character(schlen) :: name
```

```
   integer            :: NG_impose=0
(...)
#if defined _SP_FFT
   integer            :: XY_size  =0
#endif
(...)
   integer, allocatable :: R_inv_rotation(:)
   integer, allocatable :: R_rotation(:,:)
#if defined _FFTQE
   integer                    :: plan = 0
   complex(DP), allocatable :: c(:)
#endif
#if defined _FFTW || defined _SP_FFT
   integer(8)                 :: plan= 0
   complex(DP), allocatable :: c(:)
#endif
#ifdef _CUDA
   integer, allocatable    ,device :: G_map_d(:,:)
   integer, allocatable    ,device :: R_inv_rotation_d(:)
   integer, allocatable    ,device :: R_rotation_d(:,:)
   integer                         :: plan_d = 0
   complex(DP), allocatable,device :: c_d(:)
#endif
 end type
```

This module hosts different library–specific sections and can be loaded more than once. This allows YAMBO to use simultaneously different FFT setups. More importantly the FFT can now be passed as subroutine/function argument making specific sections of the code FFT–structure agnostic.

The new FFT module of YAMBO is gradually incorporating more libraries: FFTQE, FFTW, SpFFT and it also has the CUDA support.

- **Quasiparticles, real–time components and more**. More and more structures of the code have been enclosed in dedicated modules. In addition the inner structure of selected module sources have been changed in order to include in a single module file the following aspects: the module definition, the module allocation, and the module deallocation procedures. In this way the use of the a given module is greatly simplified. A clear example is given below:

```
module RT_occupations
 type RT_occupation
   character(schlen)        :: KIND
(...)
 end type RT_occupation
 type(RT_occupation) :: RT_el_occ,RT_ho_occ,RT_ph_occ,RT_life_occ
 contains
   subroutine RT_occupation_clean(OCC)
(...)
   end subroutine
   subroutine RT_occupation_alloc(KIND,OCC,D1,D2)
(...)
   end subroutine
   subroutine RT_occupation_free(OCC)
(...)
   end subroutine
end module
```

- **Descriptors.** YAMBO adopts an extensive use of output text files and intermediate databases. Those are created and re-read during the simulation process in order

to save calculated quantities for post-processing and more. An enormous difficulty, that also leads to an increasing complexity, is the storing of variables in the databases in order to link them to specific calculation flows. The `Yio` library used by YAMBO includes specific modules to write databases. In addition a new feature has been coded: **descriptors**, objects handled via a dedicated module that encapsulate all database variables in a compact way. Descriptors are written to databases and can be dumped to human-readable output files using a single call to a specific routine. In this way the duplication of coding lines is greatly reduced.

- **Simplified output module.** The creation of output file has been greatly simplified by coding a new modular layer to the output engine of YAMBO. This layer is based on the `OUTPUT_simple` module and on the `OUTPUT_add_column routine`. In practice the routine can handle the entire opening, writing, closing of the output file. An example of an output file creation and filling is given in the following:

```
call OUTPUT_add_column("ph",action="open")
call OUTPUT_add_column("ph",TITLES=(/"Q-point","Branch "/),I_VALUES=(/iq,
    il/))
call OUTPUT_add_column("ph",TITLES=(/"Energy"/),R_VALUES=(/ph_freq/),UNIT
    ="meV")
call OUTPUT_add_column("ph",action="write")
call OUTPUT_add_column("ph",action="close")
```

  In this example and output file, `o.ph` is opened and filled with data distributed in three columns. The third column, moreover, is written converting data units to meV.

- **Double-Grid support.** The Double–Grid feature of YAMBO is a special algorithmic approach that allows one to improve the convergence with respect to the BZ sampling by approximating the dependence on the number of **k**–points only in the electronic energies, with the wavefunctions assumed to be smooth enough to be taken out of the integrals. This feature is largely used in the code but, similarly, to the `FFT` case it was hard coded and not modularised. In the latest YAMBO release also this part of the code has been deeply rewritten and modularised.

- **Parallel Structure**. The parallel structure of YAMBO is extended and organised in the form of multi-level CPU assignment. In each level of this pyramid, several `MPI` communicators, indexes and variables are created. Taking in consideration that some calculation flows involve up to 5 levels of MPI parallelism, it is clear that the number of components of the parallel infrastructure of YAMBO can be very large (close to one hundred). In order to simplify this structure we have introduced a novel data structure that incorporates all components of a single parallel level. We named this object a `Parallel Scheme`. Its structure is reported below:

```
! PARALLEL SCHEME ...
!=======================
type PAR_scheme
  type(MPI_comm)       :: COMM_i
  type(MPI_comm)       :: COMM_a2a
  type(PP_indexes)     :: IND
  integer              :: ID
  integer              :: D(2)
```

```
  integer              :: N_ser
  integer              :: N_par
  logical              :: consecutive
  integer,allocatable  :: table(:)
end type
...
```

The components of this object are enough to fully define the parallelism of a given environment. We have indeed a `MPI` communicator for inter-/intra-chain communications and a `MPI` index to distribute the workload. These schemes are compact and can be passed as arguments of routine in such a way to fully define the parallel environment of the routine. They allow for a flexible parallel coding and efficient resource distribution.

Memory tracking. Memory tracking has been further extended and extensively used within the code base; separate tracking of CPU and GPU memory has been implemented. Moreover we have implemented a macro YAMBO_PAR_ALLOC_CHECK which keeps track of the amount of memory required by all MPI task belonging to the same node and checks if this fits the overall available memory on each node before each MPI task performs the allocation. Below we show an example of the associated message in a successful run

```
[MEMORY] Parallel distribution of BS_MAT on HOST wnode01 with size 4507.93 [Mb]
```

This is particularly useful because the user gets an error message directly from the code in case not enough memory is available.

CUDA-aware Yambo. GPU-related implementation (already merged into the main development branch) was cleaned up to make source files easier to read and maintain. Source duplication was largely removed. Overall this was quite a global change to the code base. See Deliverable D2.2 for more details about the actual strategies put in place.

Interfaces with other codes: p2y. The p2y interface has been further developed to keep in sync with QUANTUM ESPRESSO (QE). Moreover, we added the ability to read the QE xml file `atomic_proj.xml` where the projections of the wavefunctions onto atomic orbitals are stored. This made possible the implementation of the projected density of states (PDOS) at the GW level in the `ypp` utility.

External libraries.

- We updated the versions of the following external libraries which can be downloaded via the YAMBO configure script: SLEPC, PETSC, HDF5, NetCDF and Scalapack.
- The YAMBO fortran interface with the SLEPC and PETSC library was upgraded, since the previous version was not compatible with the newer libraries. Moreover work is currently in progress (development branch not yet merged to main branch) to further extend the interface. At present YAMBO uses SLEPC to solve the BSE

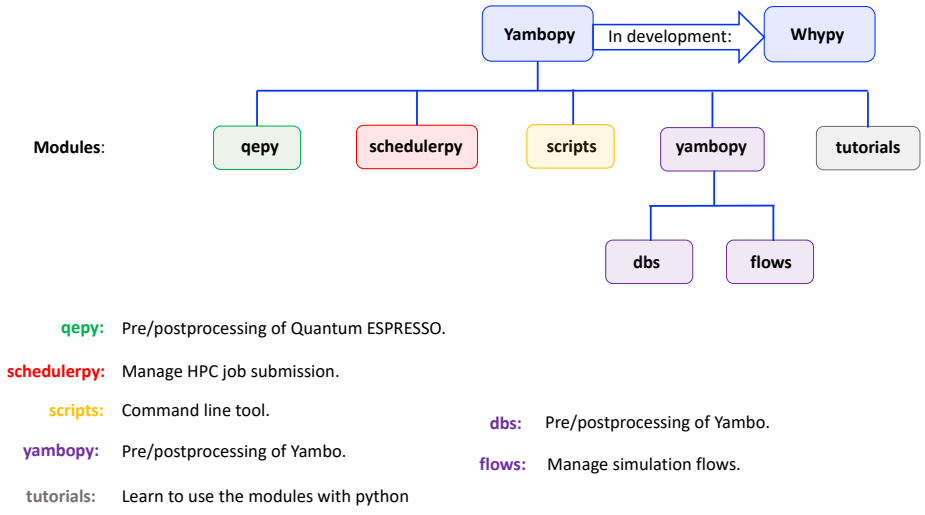

Figure 4: Scheme of the `yambopy/whypy` modular structure.

eigenvalue problem with a recursive "Krylov-Schur" algorithm. The ongoing development includes the use of an improved version with a "bcgs+jacobi" preconditionig scheme, its extension to the non-Hermitian case and the use of alternative algorithms such as "Generalized Davidson".

- We added an interface to the Futile MAX library and to the yaml libraries which are exploited to produce reports in Yaml format. The present implementation is still preliminary. It can be activated compiling the code with the option `-enable-yaml-output`.
- We added pnetcdf to the libraries which can be automatically downloaded and compiled via the YAMBO configure. pnetcdf+NetCDF is an alternative to the HDF5+netCDF scheme to produce parallel I/O. We are currently testing its usability in a development branch.

Whypy. Whypy is a python package that allows for pre- and postprocessing of Quantum ESPRESSO (QE) and Yambo data as well as managing job runs with these two codes. Its functionalities include:

- Input file parsing and generation
- Output parsing of human-readable, xml and netCDF databases
- Data analysis and visualisation
- Submit and manage simulations, both on local machines and HPC facilities
- Simulation flows
- Hackability: user-specific functionalities can easily be added into the code

`Yambopy` refers to the currently released package, while `Whypy` is the name of the development version. `Yambopy` was originally started by Henrique Miranda and

then co-developed by himself, Fulvio Paleari, Alejandro Molina-Sànchez and Alexandre Morlet. It has been featured in the hands-on sessions of the CECAM/Psi-k Yambo School in Lausanne (April 2017) and the ICTP Yambo School in Trieste (January 2020).

Its purpose is to help QE/YAMBO users to automatise several repetitive processes (for example convergence runs for materials science calculations) and to provide a single interface for the various executables that comprise the QE/YAMBO packages. It consists of a series of modules containing several python classes performing specific operations. These modules can be imported into python scripts managed by the user. The current development process aims at expanding the functionalities `whypy` provides, both in terms of data analysis and available workflows, as well as a focus on simulation management and data retrieval for many-body calculations via databases. A better integration with the `AiiDA` package, meant to be a higher level layer wrt `whypy`, is also foreseen – an interface is already present. In short, the main developmental goal is transforming the `whypy` package into a python-powered, all-purposes and easy-to-use driver for QE and YAMBO.

Test-suite. The YAMBO `test-suite` represent a key tool for the code development. It is hosted on a dedicated repository and:

- it can run up to 4000 test runs of the code covering almost all the code features.
- the parallel environment can be fully controlled. The test-suite can run: in serial, in parallel using a default workload distribution or looping over the parallel configurations, by using `OpenMP` and/or `CUDA`.
- it is completely automatized and adjusted.
- it can be controlled remotely.
- the results are automatically uploaded on a public webpage.

The `test-suite` has been deeply revised and improved. We have created different branches in order to boost its development.

The Git repositories. In order to accommodate the large development of the Yambo source the Github Yambo Project has been enriched of new components:

- yambo-libraries: Yambo derived libraries organized in branches.
- yambo-minipy: minimal set of phyton scripts to automatize different operations of the code.
- tutorials: comprehensive set of tutorials. These are fully documented on the Yambo wiki page. They contains a detailed description of the code and a guided tour in most of the code features.
- whypy: Whypy main development repository.

YAMBO v5.0. YAMBO 5.0 is a release with a major number change. This is due to a number of different reasons. First of all the user interface significantly changed. We have improved the user interface and in particular the keywords used by YAMBO to generate the input file. For example to perform a BSE simulation now the user can use, in place of the old

```
yambo -o b -k sex -y d
```

the new command

```
yambo -optics b -kernel sex -Ksolver d.
```

The old command line is still supported.

Moreover all features implemented in the YAMBO code were carefully checked, with applications on real materials before their public release. In addition tutorials were prepared to make easy for the user to learn how to use the new features. This implies, however, that some of the coded features wait a long time before being officially released. With version 5.0 we decided to adopt a two step strategy. All developed features are immediately made public and, moreover, the features which have gone through extensive validation and for which the documentation is available are marked as supported. This implies a number of new features entered in version 5.0. In the YAMBO code:

- finite-q BSE
- fxc TDDFT kernel obtained via MBPT
- spectral functions beyond the QP approximation
- ACFDT approach to total energy
- Extended features for electron phonon
- Magneto-Optical properties beyond the independent particles approximation (yambo_kerr)
- Fully self-consistent solution of the many-body problem with static self-energies like COHSEX (yambo_sc)

In the ypp utility:

- automatic detection of high-symmetry points when plotting bands,
- calculation of the Eliashberg electron-phonon terms,
- extraction of carriers occupations from real-time simulations
- interpolation of excitonic bands

Due to the significant changes in version 5.0 we expect the release will need some extra time to be fully stabilised and become robust for production runs. For this reason we will guarantee the support of previous releases with bug-fixes. YAMBO 4.5.0 was released at M12. Since then we have further tagged 4.5.1 (Feb 2020), 4.5.2 (Jul 2020), 4.5.3 (Aug 2020), as well as 4.3.3 and 4.4.1.

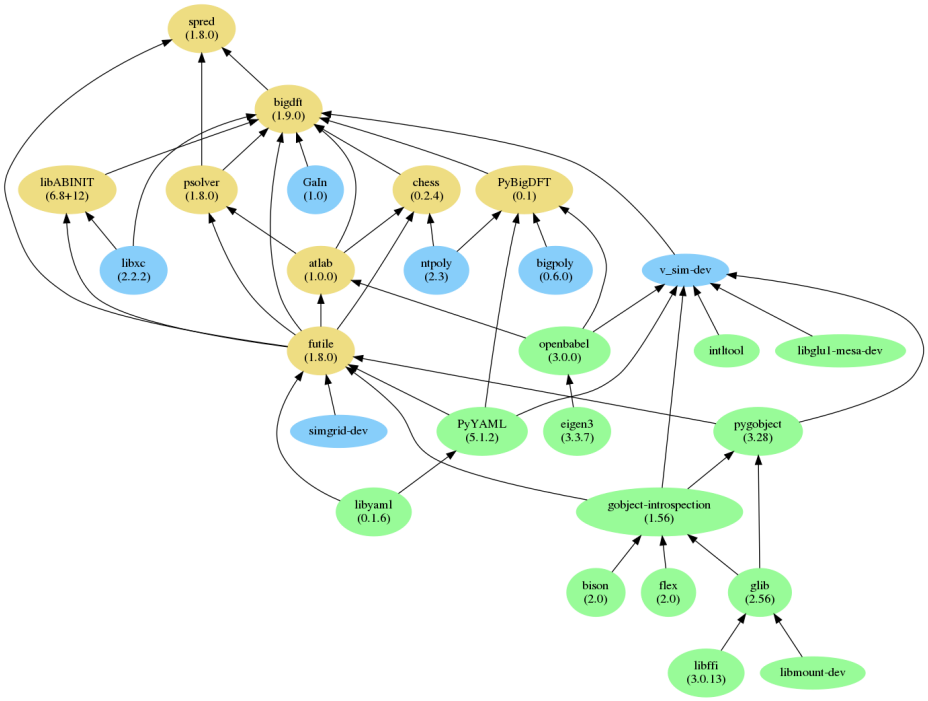

Figure 5: The compilation dependencies of the libraries in BigDFT suite. Upstream packages are indicated in blue and green, for domain-specific and system libraries, respectively. The packages of the consortium are depicted in yellow.

2.5 BigDFT

The code has further enhanced its modularity. Presently, the compilation of the BigDFT suite is performed via a stacked layer of multiple libraries, most of them being developed outside of the BigDFT consortium. We present in Fig. 5 an example of such compilation. In the following paragraphs we will illustrate the status of the development of the libraries that are internal to the BigDFT developers consortium.

The version 1.9.1 of the BIGDFT code has been released in December, 2020. With this version, also the 1.9.2 and 1.9.3 releases are under preparation, and in their beta stage. Several discussions have been made in the context of the restructuring of the code that has to be performed such as to go in the direction of a programmatical approach to the inner part of the SCF loop.

FUTILE and PSolver

See Secs. 3.6 and 3.7 in Libraries.

atlab

The library has been also release with a `cmake` build system, which at present second the already available *autotools* module. Both the approaches are therefore available. Like the FUTILE library, we are presently working in a python module that should abstract some of the operation on input positions and provides that to higher-level libraries like

```
from BigDFT import Calculators as C, Inputfiles as I
single_point=C.SystemCalculator()
inp = I.Inputfile()
inp.set_xc('LDA')
inp.write_orbitals_on_disk()
log=single_point.run(input=inp,posinp='mol.xyz')
print (log.energy)
```

Figure 6: `SystemCalculator`: the Aiida `CalcJob` equivalent

PyBigDFT.

libconv

The `libconv` library is now stabilised and ready for integration in the code. The code generation has also been tested on architectures like the Fugaku supercomputer.

PyBigDFT

As already presented in the M12 deliverable, this package is a collection of Python Modules that are conceived for pre- and post- processing of BigDFT input and output files. Such modules are supposed to enhance the BigDFT experience by high-level approach. Also, calculators and workflows are supposed to be created and inspected with modules of the PyBigDFT package. This package is conceived as a set of Python modules to manipulate complex simulation setups in a HPC framework. Recent advances in PyBigDFT have enabled the implementation and usage of new functionalities of BigDFT.

- The AiiDA BigDFT plugin has been inserted in PyBigDFT. It enables the remote, asynchronous execution of a PyBigDFT workflow from a Jupyter notebook. For each new production result of the BigDFT consortium, the calculations are triggered and pre-postprocessed from PyBigDFT.
- The PyBigDFT API has been carefully checked with respect to the compatibility with Python3 and Python2. Yet, the Python2 support will be soon declared as obscolescent. Flake8 execution scripts are inserted in the continuous integration of the library.
- Validation and verification techniques as per WP5 are now triggered entirely from PyBigDFT.
- The PyBigDFT tools analysis has been coupled with established packages for the simulation of biological systems that enabled the analysis of production results like the complexity reduction framework explained in the WP6 demonstrators related to BigDFT.

In Figs. 6,7,8 we present some snippets showing how to employ the APIs of PyBigDFT in conjunction with AiiDA calculators.

We have implemented the “traditional” flavour of AiiDA plugin.

```
pip install aiida-bigdft
```

```
from BigDFT import Datasets as D
hgrid_cv=D.Dataset('h_set')
for h in [0.5,0.45,0.4,0.35,0.3]:
    inp.set_hgrid(h)
    hgrid_cv.append_run(id={'h':h},input=inp,runner=single_point)
results=hgrid_cv.run()
energs=hgrid_cv.fetch_results(attribute='energy')
```

Figure 7: `Dataset`: a small equivalent of a Aiida `WorkChain`

```
from BigDFT import AiidaCalculator as A
study=A.AiidaCalculator(code="bigdft@localhost",
         num_machines=1,mpiprocs_per_machine=1,
         omp=1,walltime=3600)
%load_ext jupyternotify
%notify
hgrid_cv.wait()
>>> '0 processes still running'
```

Figure 8: `AiidaCalculator` ☛ used to **remotely submit** the job

bundler

The Bundler package has now been merged with the jhbuild upstream version and unified to the ESL package. Such package is defined from a fork of the `Jhbuild` package,[1] that has been conceived in the context of GNOME developers consortium. We are considering the possiblity of submitting a Merge Request to the jhbuild developers to include our needs in the GNOME development process. This package can now used as a ground basis to develop a common infrastructure to compile and link together libraries for electronic structure codes. We are still foreseeing new modification in its structure to enable, the possiblity of a easier usage by the end user.

sphinx-fortran

The project has been made compatible with python3 and it is now used in the Continuous Integration to build the documentation of the corresponding packages documented in the sources.

[1] https://developer.gnome.org/jhbuild/

2.6 CP2K

CP2K code is heavily relying on the external libraries for its performance portability as the code itself is written in a generic way without any architecture specific implementations. In particular, DBCSR and ScaLAPACK (as a source of distributed matrix-matrix implementation `pdgemm`) libraries must exhibit a decent performance on a given platform in order for CP2K to achieve a fast execution in $O(N)$ and RPA types of calculation. AS such, the effort of optimising and porting CP2K to new architectures was channelled to the performance tuning of DBCSR and COSMA libraries. COSMA is a new library developed at ETHZ which implements communication-optimal matrix-matrix algorithm and provides a corresponding `pdgemm` wrapper natively used by CP2K.

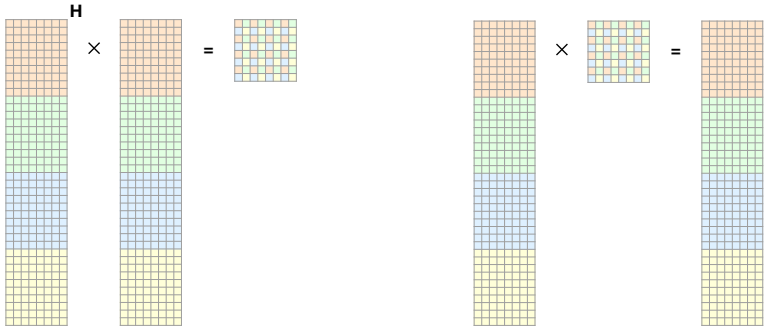

Figure 9: Stripe-Stripe-Block (left) and Stripe-Block-Stripe (right) distributed matrix multiplications supported by SPLA.

3 Libraries

3.1 SPLA

SPLA (Specialized Parallel Linear Algebra) domain specific library has been developed at CSCS. The library takes care of the two special matrix-matrix multiplications arising in the iterative solvers for plane-wave DFT codes (see Fig. 9 for the details). In the first case, two tall-and-skinny matrices representing wave-functions are multiplied together to form a matrix of inner products. Wave-functions are distributed in stripes between all available MPI ranks. The final matrix has a 2D block-cyclic ScaLAPCK distribution. In the second scenario, a transformation of wave-functions is performed using a 2D block-cyclic distributed matrix.

SPLA provides functions for the above mentioned types of distributed matrix multiplications with specific matrix distributions, which cannot be used directly with a ScaLAPACK interface. All computations can optionally utilize GPUs through CUDA or ROCm, where matrices can be located either in host or device memory. SPLA is written in C++ with MPI and OpenMP programming models and provides a C-interface which can be used in the Fortran codes with the help of the `ISO_C_BINDING` module. SPLA is using CMake for building, it has a GitHub page,[2], a documentation page,[3] and a CI/CD pipeline. The initial performance of the SPLA library was measured in a synthetic test of three consequtive matrix-matrix multiplications with (M,N,K) = (10000, {4000, 8000, 12000}, 1000000) dimensions (see Fig. 9).

3.2 SIRIUS

SIRIUS API was refactored to contain only Fortran subroutines with the optional error code parameter as last argument. If error happens on the SIRIUS side and the error code parameter was provided by the main Fortran program, the error code will be returned back and the calling program has to react. If no error code was provided, the library will terminate with the error message. Also, the Fortran API is now generated from the YAML markup, written inside a special comment block of SIRIUS. Example of such

[2] https://github.com/eth-cscs/spla
[3] https://spla.readthedocs.io/en/latest/

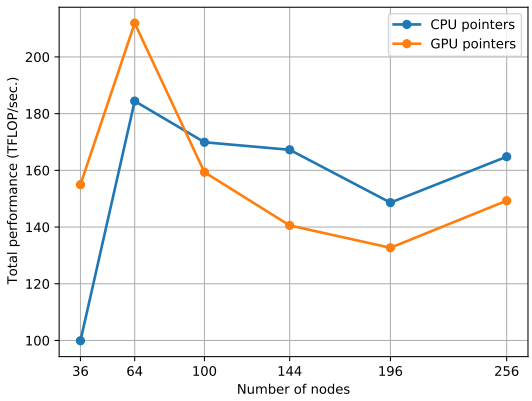

Figure 10: Initial performance of the SPLA in the matrix-matrix multiplication test on the hybrid partition of Piz Daint. A GPU back-end was used. CPU and GPU pointers of A, B, and C matrices were tested.

markup is listed below.

Markup description of the **sirius_context_initialized** *API function:*

```
@api begin
sirius_context_initialized:
  doc: Check if the simulation context is initialized.
  arguments:
    handler:
      type: void*
      attr: in, required
      doc: Simulation context handler.
    status:
      type: bool
      attr: out, required
      doc: Status of the library (true if initialized)
    error_code:
      type: int
      attr: out, optional
      doc: Error code.
@api end
*/
```

Generated definition of the Fortran interface for **sirius_context_initialized** *API function:*

```
!
!> @brief Check if the simulation context is initialized.
!> @param [in] handler Simulation context handler.
!> @param [out] st Status of the library (true if initialized)
!> @param [out] error_code Error code.
subroutine sirius_context_initialized(handler,st,error_code)
implicit none
!
type(C_PTR), target, intent(in) :: handler
logical, target, intent(out) :: st
integer, optional, target, intent(out) :: error_code
!
type(C_PTR) :: handler_ptr
type(C_PTR) :: st_ptr
logical(C_BOOL), target :: st_c_type
type(C_PTR) :: error_code_ptr
...
```

3.3 SpFFT

SpFFT is A 3D FFT library for sparse frequency domain data written in C++ with support for MPI, OpenMP, CUDA and ROCm. SpFFT library is designed to work with electronic structure codes. The library is published on a GitHub repository,[4] its development is fully finished and the library is ready for production.

3.4 COSMA

COSMA is a parallel, high-performance, GPU-accelerated, matrix-matrix multiplication algorithm that is communication-optimal for all combinations of matrix dimensions, number of processors and memory sizes, without the need for any parameter tuning. COSMA library is published on GitHub,[5] its development is fully finished and the library is ready for production.

3.5 DBCSR

DBCSR is a library designed to efficiently perform sparse matrix-matrix multiplication, among other operations. It is MPI and OpenMP parallel and can exploit Nvidia and AMD GPUs via CUDA and HIP. DBCSR library is hosted on GitHub[6] and is ready for production. In the M19-M24 period of the project the work on improving documentation, CMake build system, improving unit test coverage and unifying CUDA and ROCm backends has been accomplished. A work on the exploration and utilization of tensor cores (NVIDIA A100 cards) has been started.

3.6 FUTILE library

This library is in production stage since several months already. It has already been employed in several codes like YAMBO and FLAME. We are presently working on the release of a python package which includes the modules which are associated to the operations performed by futile.

3.7 PSolver

The PSolver library is also released in production stage and can be installed independently.

3.8 DevXlib

DevXlib has been maintained and further updated to consolidate the explicit handling of memory transfer (host/dev/dev/host,dev/dev,host/host) and the seamless GPU-acceleration of simple computational kernels. Besides a stable branch, that is explicitly used by QUANTUM ESPRESSO and YAMBO, a development branch has been dedicated to the support of different programming models, notably OpenACC, OpenMP5, and some explicit handling of the CUDA library. At the moment work on possible usage styles and

[4] https://github.com/eth-cscs/SpFFT
[5] https://github.com/eth-cscs/COSMA/
[6] https://github.com/cp2k/dbcsr

interplay of these models is ongoing, the final target being to support more than one back-end for GPU-acceleration (with the idea of running on NVIDIA but also on AMD and INTEL GPUs).

3.9 XC_lib

The calculation of the *exchange and correlation* (XC) energies and potentials is a frequent and compute-intensive task that implies iterations of some given functional expressions over many thousands of grid-points. To make the XC part architecture agnostic, it has been isolated, encapsulated and refactored as an autonomous library which will be used in future releases of QUANTUM ESPRESSO. The new library `XC_lib` collects all the functional expressions classified according to their family and type (LDA, GGA, MGGA for exchange and correlation). Internal driver routines use such modules to calculate the value of the XC energy and potential for each point of the input charge density grid (and relative gradient and laplacian, depending on the family). Drivers for the derivative of the XC potential are included too.

The API exposed a few general wrapper routines that may access the internal drivers and, if linked at build time, those of the `Libxc` library [2] and return the computed XC energy and potential arrays as output. This setup should allow one to easily incorporate other XC external libraries, if needed, just by extending the scheme adopted for `Libxc`. Due to the highly parallelizability of the XC machinery, the porting to heterogeneous architectures based on accelerator is relatively straightforward and has been partially done. Different programming paradigms are currently explored in order to get the highest portability. A testing program that allows one to check every part of the library and to get information about the available DFTs is under completion and it is included in `XC_lib` too.

4 Conclusions and ongoing work

We have presented the M24 releases of the MAX flagship codes. In these new stable versions the codes have benefited from the many development actions undertaken by WPs 1,2 and 3. In this document we have mainly focused on the achievements in modularisation, interoperability, and library re-usage. It is though important to remark in these conclusions that, even if reported in different documents, all development actions have proceeded in a coordinated way. They are reciprocally instrumental to each other, all aiming at the common goal of porting the MAX community codes to pre-exascale (and exascale, in the longer run) machines and prepare them for intensive and automated computations.

In all the codes the modularity has been significantly enhanced. There is a clear separation between the code parts that implement the different logical levels and functionalities, as identified in the Software Development Plan (SDP) [3]. Many of the planned libraries have reached a stable stage. They are now available for testing and re-usage. Indeed in these new releases many MAX codes are using libraries extracted from other MAX codes, also for the production versions, indicating that the modularisation and cross exchange of software components is working. Moreover, MAX libraries have also demonstrated an impact outside the consortium. The testing and experimental usage of the libraries is giving an important feedback for improving the interfaces. For instance the `SIRIUS` development platform presents in this release a set of refactored Fortran APIs.

Apart from the actions of WPs 1,2,3 –directly related with the development– also the work of other WPs has influenced the outcome of these releases. The development and testing of computational kernels is a joint action of the code developers and WP4 experts. WP4 has also been crucial as a link with hardware and software vendors. The new features and the improvements in interoperability have been decided and implemented in close collaboration with WPs 5 and 6. We would also like to thank WP9 for the help during these last months in organising the webinar series that allowed us to present many of the novelties and improvements reported here to a wide audience of users. We want finally to stress the many contributions coming from the developers. It is also for the sake of these community contributions that the effort in keeping a unique code base is crucial, and will probably be the most important challenge of the development activities of this next year.

Acronyms

DFPT Density Functional Perturbation Theory. 9, 10

TDDFPT Time Dependent Density Functional Perturbation Theory [4]. 9

References

[1] Baroni, S. *et al.* Second report on softwarearchitecture and implementation planning. Deliver-able D1.3 of the H2020 CoE MaX (final version asof 31/05/2020). EC grant agreement no: 824143,SISSA, Trieste, Italy (2020). URL http://www.max-centre.eu/sites/default/files/D1.3%20Second%20report%20on%20software%20architecture%20and%20implementation%20planning.pdf.

[2] URL https://tddft.org/programs/libxc/.

[3] Baroni, S. *et al.* First report on software architecture and implementation plan. Deliverable D1.1 of the H2020 CoE MaX (final version as of 30/03/2019). EC grant agreement no: 824143, SISSA, Trieste, Italy. (2019).

[4] Rocca, D., Gebauer, R., Saad, Y. & Baroni, S. Turbo charging time-dependent density-functional theory with Lanczos chains. *J. Chem. Phys.* **128**, 154105 (2008).