



# **ELSI Interface (Development Version) User's Guide**

The ELSI Team  
[elsi-interchange.org](http://elsi-interchange.org)

May 13, 2018

# Contents

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Introduction</b>  | <b>3</b>  |
| 1.1      | The Cubic Wall of Kohn-Sham Density-Functional Theory . . . . .                      | 3         |
| 1.2      | ELSI, the ELectronic Structure Infrastructure . . . . .                              | 4         |
| 1.2.1    | Design Tenets of ELSI . . . . .  | 4         |
| 1.3      | Kohn-Sham Solver Libraries Supported by ELSI . . . . .                               | 5         |
| 1.3.1    | ELPA: Eigenvalue soLvers for Petaflop-Applications . . . . .                         | 5         |
| 1.3.2    | libOMM: Orbital Minimization Method . . . . .  | 5         |
| 1.3.3    | PEXSI: Pole EXpansion and Selected Inversion . . . . .                               | 5         |
| 1.4      | Acknowledgments . . . . .  | 6         |
| <b>2</b> | <b>Installation of ELSI</b>  | <b>7</b>  |
| 2.1      | Overview . . . . .   | 7         |
| 2.2      | Prerequisites . . . . .  | 7         |
| 2.3      | CMake Basics . . . . .   | 7         |
| 2.4      | Configuration . . . . .  | 8         |
| 2.4.1    | Compilers . . . . .  | 8         |
| 2.4.2    | Solvers . . . . .  | 9         |
| 2.4.3    | Build Targets . . . . .  | 9         |
| 2.4.4    | List of All Options . . . . .  | 10        |
| 2.4.5    | “Toolchain” Files . . . . .  | 11        |
| 2.5      | Importing ELSI into KS-DFT Codes . . . . .   | 11        |
| 2.5.1    | Linking a KS-DFT Code against ELSI . . . . .   | 11        |
| 2.5.2    | Importing ELSI into a KS-DFT Code . . . . .  | 11        |
| <b>3</b> | <b>The ELSI API</b>  | <b>12</b> |
| 3.1      | Overview of the ELSI API . . . . .   | 12        |
| 3.2      | Description of All ELSI APIs . . . . .   | 12        |
| 3.2.1    | Initializing ELSI . . . . .  | 12        |
| 3.2.2    | Setting Up MPI . . . . .   | 14        |
| 3.2.3    | Setting Up Matrix Formats . . . . .  | 14        |
| 3.2.4    | Setting Up Multiple k-points and/or Spin Channels . . . . .                          | 15        |
| 3.2.5    | Finalizing ELSI . . . . .  | 16        |
| 3.3      | Solving Eigenvalues and Eigenvectors . . . . .                                       | 16        |
| 3.4      | Computing Density Matrices . . . . .   | 17        |
| 3.5      | Customizing ELSI . . . . .   | 18        |
| 3.5.1    | Customizing the ELSI Interface . . . . .   | 18        |
| 3.5.2    | Customizing the ELPA Solver . . . . .  | 20        |
| 3.5.3    | Customizing the libOMM Solver . . . . .  | 20        |
| 3.5.4    | Customizing the PEXSI Solver . . . . .   | 20        |
| 3.6      | Getting Additional Results from ELSI . . . . .                                       | 22        |
| 3.6.1    | Getting Results from the ELSI Interface . . . . .                                    | 23        |
| 3.6.2    | Getting Results from the PEXSI Solver . . . . .                                      | 23        |
| 3.7      | Demonstration Pseudo-Code . . . . .  | 24        |
| 3.7.1    | 2D Block-Cyclic Distributed Dense Matrix + ELSI Eigensolver Interface . . . . .      | 24        |
| 3.7.2    | 1D Block Distributed CSC Sparse Matrix + ELSI Eigensolver Interface . . . . .        | 24        |
| 3.7.3    | 1D Block-Cyclic Distributed CSC Sparse Matrix + ELSI Eigensolver Interface . . . . . | 25        |

|       |   |    |
|-------|---|----|
| 3.7.4 | 2D Block-Cyclic Distributed Dense Matrix + ELSI Density Matrix Interface . . . . .      | 26 |
| 3.7.5 | 1D Block Distributed CSC Sparse Matrix + ELSI Density Matrix Interface . . . . .        | 26 |
| 3.7.6 | 1D Block-Cyclic Distributed CSC Sparse Matrix + ELSI Density Matrix Interface . . . . . | 27 |
| 3.7.7 | Multiple k-points Calculations . . . . .  | 28 |
| 3.8   | C/C++ Interface . . . . .   | 28 |

# 1 Introduction

## 1.1 The Cubic Wall of Kohn-Sham Density-Functional Theory

Molecular and materials simulations based on Kohn-Sham density-functional theory (KS-DFT) [1] are widely used to provide atomic-scale insights, understanding, and predictions across a wide range of disciplines in the sciences and in engineering.

In KS-DFT [1], the many-electron problem for the Born-Oppenheimer electronic ground state is reduced to a system of single particle equations known as the Kohn-Sham equations

$$\hat{h}^{\text{KS}}\psi_l = \epsilon_l\psi_l, \quad (1.1)$$

where  $\psi_l$  and  $\epsilon_l$  are Kohn-Sham orbitals and their associated eigenenergies, and  $\hat{h}^{\text{KS}}$  denotes the Kohn-Sham Hamiltonian. In practice,  $N_{\text{basis}}$  basis functions  $\phi_i(\mathbf{r})$  are employed to expand the Kohn-Sham orbitals:

$$\psi_l(\mathbf{r}) = \sum_{j=1}^{N_{\text{basis}}} c_{jl}\phi_j(\mathbf{r}). \quad (1.2)$$

Using non-orthogonal basis functions (e.g., Gaussian functions, Slater functions, numeric atom-centered orbitals) in Eq. 1.2 converts Eq. 1.1 to a generalized eigenvalue problem that can be expressed in the following matrix form:

$$\mathbf{H}\mathbf{c} = \mathbf{\epsilon}\mathbf{S}\mathbf{c}. \quad (1.3)$$

Here,  $\mathbf{H}$  and  $\mathbf{S}$  are the Hamiltonian matrix and the overlap matrix, whose elements can be computed through numerical integrations. The matrix  $\mathbf{c}$  and diagonal matrix  $\mathbf{\epsilon}$  contain the eigenvectors and eigenvalues, respectively, of the eigensystem.

When using orthonormal basis sets (e.g., plane waves, multi-resolution wavelets), the eigenproblem described in Eq. 1.3 reduces to a standard form where  $\mathbf{S}$  is an identity matrix.

The explicit solution of Eq. 1.3 yields the Kohn-Sham orbitals  $\psi_l$ , from which the electron density  $n(\mathbf{r})$  can be computed following an orbital-based method that scales as  $O(N^2)$ :

$$n(\mathbf{r}) = \sum_{j=1}^{N_{\text{basis}}} f_l \psi_l^*(\mathbf{r}) \psi_l(\mathbf{r}), \quad (1.4)$$

where  $f_l$  denotes the occupation number of each orbital. In an actual computation, it is sufficient to perform the summation only for the occupied ( $f_l > 0$ ) orbitals. The ratio of occupied orbitals to the total number of basis functions can be below 1% for plane wave basis sets, whereas with some localized basis sets, fewer basis functions are required, leading to a larger fraction of occupied states typically between 10% and 40%.

An alternative method that scales as  $O(N)$  can be employed for localized basis functions:

$$n(\mathbf{r}) = \sum_{i,j}^{N_{\text{basis}}} \phi_i^*(\mathbf{r}) n_{ij} \phi_j(\mathbf{r}), \quad (1.5)$$

with  $n_{ij}$  being the elements of the density matrix that need to be computed before the density update.

From a viewpoint of computational complexity, almost all standard pieces of solving the Kohn-Sham equations can be formulated in a linear scaling fashion with respect to the system size. The only piece that can not be easily addressed in an  $O(N)$  fashion is the Kohn-Sham eigenvalue problem.

Traditional solutions to the Kohn-Sham eigenvalue problem rely on diagonalization of the Hamiltonian matrix to obtain the Kohn-Sham orbitals  $\psi_l$ . The associated computational cost is  $O(N^3)$ , regardless of the diagonalization algorithm employed. Thanks to a low prefactor, these methods are advantageous to use for systems comprised of up to roughly a few thousands of atoms, which account for the bulk of KS-DFT applications. Systems beyond a few thousands of atoms, however, cannot be efficiently handled by diagonalization due to its cubic scaling nature. Alternatively, approaches with a lower scaling factor, e.g. linear scaling methods, usually directly compute the density matrix as a function of the Hamiltonian matrix. The large prefactor of these methods introduces a significant overhead for small systems, hindering the establishment of lower scaling approaches as mainstream methods of the field. In addition, such alternatives are typically restricted to certain classes of systems or problems. The transition from cubic scaling methods to linear scaling ones is therefore not trivial to automate. In practice, the KS eigenvalue problem remains to be a bottleneck of KS-DFT simulations on current HPC architectures and for system sizes significantly exceeding several thousands of atoms.

## 1.2 ELSI, the ELectronic Structure Infrastructure

ELSI unifies the community effort in overcoming the cubic-wall problem of KS-DFT by bridging the divide between developers of electronic structure solvers and KS-DFT codes. Via a unified interface, ELSI gives KS-DFT developers easy access to multiple solvers that solve or circumvent the Kohn-Sham eigenproblem efficiently. Solvers are treated on equal footing within ELSI, giving solver developers a unified platform for implementation and benchmarking across codes and physical systems. Solvers may be switched dynamically in an SCF cycle, allowing the KS-DFT developer to mix-and-match strengths of different solvers. Solvers can work cooperatively with one another within ELSI, allowing for acceleration greater than either solver can achieve individually. Most importantly, ELSI exists as a community for KS-DFT and solver developers to interact and work together to improve performance of solvers, with monthly web meetings to discuss progress on code development, yearly on-site “connector meetings”, and planned webinars and workshops.

The current development version of ELSI supports ELPA [3, 4], libOMM [2], and PEXSI [5, 6] solvers. Codes currently integrated with ELSI include DFTB+ [8], DGDFT [9], FHI-aims [10], NWChem [11] via Global Arrays, and SIESTA [12].

### 1.2.1 Design Tenets of ELSI

**Versatility:** ELSI supports real-valued and complex-valued density matrix, eigenvalue, and eigenvector calculations. A unified software interface designed for rapid integration into a variety of KS-DFT codes is provided. Fortran and C/C++ interfaces are provided.

**Flexibility:** ELSI supports both dense and sparse matrices as input/output. Supported matrix distribution layouts include 2D block-cyclic distribution, 1D block-cyclic distribution, and 1D block distribution. In situations where the input/output matrix format requested by the calling KS-DFT code and the format used internally by the requested solver are different, conversion and redistribution of matrices will be performed automatically.

**Scalability:** The solver libraries collected in ELSI are highly scalable. For instance, ELPA can scale to  $\sim 100k$  CPU cores given a sufficiently large problem to solve, and PEXSI, with its efficient two-level parallelism, easily scales to  $\sim 500k$  CPU cores.

**Portability:** ELSI and its redistributed library source packages have been confirmed to work on commonly-used HPC architectures (Cray, IBM, Intel, NVIDIA) using major compilers (Cray, GNU, IBM, Intel, PGI).

## 1.3 Kohn-Sham Solver Libraries Supported by ELSI

The current development version of ELSI stably supports ELPA [3, 4], libOMM [2], and PEXSI [5, 6] solvers. Since this user’s guide focuses on the ELSI interface layer, given below are very brief introductions of the solvers. For more details and technical descriptions of each solver, the reader is referred to the original publications, e.g., those cited here, as well as an overview paper of the 2017.05 version of ELSI [13].

### 1.3.1 ELPA: Eigenvalue solvers for Petaflop-Applications

ELPA [3, 4] solves the Kohn-Sham eigenvalue problem in Eq. 1.3 by direct diagonalization.

In ELPA, the generalized eigenproblem is first transformed to the standard form by using Cholesky factorization of the overlap matrix. Then, the standard eigenproblem is either directly reduced to the tridiagonal form, using similar algorithms as implemented in ScaLAPACK eigensolvers, or first reduced to a banded intermediate form, then to the tridiagonal form. After the eigenvectors of the tridiagonal system are solved, they are back-transformed to the original form in either one step or two steps, depending on the algorithm employed in the diagonalization. Compared to the traditional one-step approach, although the two-step alternative introduces two additional steps (one forward transformation and one backward), it has been shown to enable faster computation and better parallel scalability on present-day computers. Specifically, the matrix-vector operations (BLAS level-2 routines) in the one-step diagonalization can be mostly replaced by more efficient matrix-matrix operations (BLAS level-3 routines) in the two-step version. The additional effort needed by back-transforming the eigenvectors is often alleviated in electronic structure calculations due to the fact that only a relatively small fraction of the eigensolutions need to be retrieved.

### 1.3.2 libOMM: Orbital Minimization Method

Instead of direct diagonalization, the orbital minimization method (OMM) relies on an iterative algorithm to minimize an unconstrained energy functional using a set of auxiliary Wannier functions that expand the occupied subspace of the system. The minimized OMM energy functional can be shown to be equal to the sum of the eigenvalues of the occupied Kohn-Sham orbitals. Furthermore, the Wannier functions are driven towards perfect orthonormality at this minimum. The density matrix is then constructed from the final Wannier functions. Although this density matrix is sufficient for the electron density update following Eq. 1.5, without knowledge of individual eigenstates, the OMM method cannot handle systems with fractional occupation numbers.

Compared to other minimization methods with the orthonormality constraint of eigenstates, the advantage of OMM is that it only requires an unconstrained minimization without an explicit orthonormalization step. This makes the OMM a good candidate for linear scaling DFT. However, in order to do so, it is necessary to spatially confine the Wannier functions, which introduces a number of technical difficulties. Recently, the OMM method has been reimplemented in the libOMM library [2] as a cubic scaling solver. The minimization of the OMM energy functional is carried out in libOMM by using the conjugate-gradient (CG) method with dense linear algebra operations, which guarantees that the functional converges to the correct minimum. The performance of libOMM mainly depends on the convergence rate of the OMM minimization.

### 1.3.3 PEXSI: Pole EXpansion and Selected Inversion

The pole expansion and selected inversion (PEXSI) method [5, 6] provides another alternative way for solving the Kohn-Sham electronic structure without diagonalization. As a Fermi operator expansion (FOE) based method, PEXSI expands the density matrix using a pole expansion. The number of terms needed by this expansion is proportional to  $\log(\beta\Delta E)$ , where  $\beta = 1/(k_B T)$ ,  $k_B$  is the Boltzmann constant,  $T$  is the electronic temperature, and  $\Delta E$  is the spectral radius. This logarithmic scaling makes the pole expansion a highly efficient approach to expand the Fermi operator. With the expansion method proposed by Moussa [7], about  $\sim 20$  poles are sufficient for the result obtained from PEXSI to be fully comparable to that obtained from diagonalization.

In PEXSI, selected elements of the density matrix, corresponding to the non-zero pattern of  $\mathbf{H}$  and  $\mathbf{S}$ , are evaluated using a parallel selected inversion algorithm. Its computational cost (for semilocal DFT) depends on the dimensionality of the system:  $O(N)$  for 1D systems,  $O(N^{1.5})$  for 2D systems, and  $O(N^2)$  for 3D systems. This favorable scaling hinges

on the sparse character of the Hamiltonian and overlap matrices, but not on any fundamental assumption about the localization properties of the single particle density matrix, i.e., this method is applicable to metallic systems as well as insulating and semi-conducting ones.

Designed in a multi-level parallelism structure, the PEXSI method is highly scalable, and can make efficient use of  $10,000 \sim 100,000$  processors on high performance computers.

## 1.4 Acknowledgments

ELSI is a National Science Foundation Software Infrastructure for Sustained Innovation - Scientific Software Integration (SI2-SSI) supported software infrastructure project. The ELSI Interface software and this User's Guide are based upon work supported by the National Science Foundation under Grant Number 1450280. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of the National Science Foundation.

## 2 Installation of ELSI

### 2.1 Overview

The ELSI package contains the ELSI Interface software as well as redistributed source code for the solver libraries ELPA (version 2016.11.001), libOMM, and PEXSI (version 1.0.0). We highly encourage all users to request access to our Git-Lab server (<http://git.elsi-interchange.org/elsi-devel>), where we regularly update ELSI between releases while preserving stability.

Starting from the May 2018 release, the installation of ELSI makes use of the CMake software.

### 2.2 Prerequisites

To build ELSI, the minimum requirements are:

`CMake` [version 3.0 or newer]  
`Fortran compiler` [with Fortran 2003]  
`C compiler` [with C99]  
`MPI`

In addition, building the PEXSI solver (highly recommended) requires:

`C++ compiler` [with C++ 11]

To build ELSI as a shared library, or to build the ELSI test programs, linear algebra libraries should be provided for ELSI to link against:

`BLAS`, `LAPACK`, `BLACS`, `ScaLAPACK`

By default, solver libraries and their dependencies redistributed through ELSI will be built. Optionally, they may be substituted by user's optimized versions:

`ELPA` [version 2017.05.003 or newer]  
`libOMM`  
`MatrixSwitch`  
`PT-SCOTCH` [version 6.0.5a]  
`SuperLU_DIST` [version 5.3.0]

### 2.3 CMake Basics

This section covers some basics of using CMake. Users who are familiar with CMake may safely skip this section.

The typical workflow of using CMake to build ELSI looks like:



```

$ ls

CMakeLists.txt  external/  src/  test/  ...

$ mkdir build
$ cd build
$ cmake [options] ..

...
...
-- Generating done
-- Build files have been written to: /current/dir

$ make [-j np]
$ make install

```

whenever CMake is invoked, one of the command line arguments must point to the path where the top level CMakeLists.txt file exists, hence the “..” in the above example.

An option can be defined by adding

```
-DKeyword=Value
```

to the command line when invoking CMake. If “Keyword” is of type boolean, its “Value” may be “ON” or “OFF”. If “Keyword” is a list of libraries or include directories, its items should be separated with “;” (semicolon) or “ ” (space).

For example,

```

-DCMAKE_INSTALL_PREFIX=/path/to/install/elsi
-DCMAKE_C_COMPILER=gcc
-DENABLE_TESTS=OFF
-DENABLE_PEXSI=ON
-DINC_PATHS="/path/to/include;/another/path/to/include"
-DLIBS="library1 library2 library3"

```

Available options are introduced in the next sections.

## 2.4 Configuration

### 2.4.1 Compilers

CMake automatically detects and sets compilers. The choices made by CMake often work, but not necessarily guarantee the optimal performance. In some cases, the compilers picked up by CMake may not be the ones desired by the user. To build ELSI, it is mandatory that the user explicitly sets the identification of the compilers:

```

-DCMAKE_Fortran_COMPILER=YOUR_MPI_FORTRAN_COMPILER
-DCMAKE_C_COMPILER=YOUR_MPI_C_COMPILER
-DCMAKE_CXX_COMPILER=YOUR_MPI_CXX_COMPILER

```

Please note that the C++ compiler is not needed if building ELSI without PEXSI.

In addition, it is highly recommended to specify the compiler flags, in particular the optimization flags:

```

-DCMAKE_Fortran_FLAGS=YOUR_FORTRAN_COMPILE_FLAGS
-DCMAKE_C_FLAGS=YOUR_MPI_C_COMPILE_FLAGS
-DCMAKE_CXX_FLAGS=YOUR_MPI_CXX_COMPILE_FLAGS

```

## 2.4.2 Solvers

The ELPA, libOMM, and PEXSI solver libraries, as well as the SuperLU\_DIST and PT-SCOTCH libraries (both required by PEXSI), are redistributed with the current ELSI package.

The redistributed version of ELPA comes with a few “kernels” specifically written to take advantage of processor architecture (e.g. vectorization instruction set extensions). A kernel may be chosen by the [ELPA2\\_KERNEL](#) keyword. Available options are:

```
-DELPA2_KERNEL=BGQ
-DELPA2_KERNEL=AVX
-DELPA2_KERNEL=AVX2
-DELPA2_KERNEL=AVX512
```

for the IBM Blue Gene Q, Intel AVX, Intel AVX2, and Intel AVX512 architectures, respectively. In ELPA, these kernels are employed to accelerate the calculation of eigenvectors, which is often a computational bottleneck when calculating a large percentage of eigenvectors. If this is the case in the user’s application, it is highly recommended that the user selects the kernel most suited to their system architecture.

Experienced users are encouraged to link the ELSI interface against external, better optimized solver libraries. Relevant options for this purpose are:

```
-DUSE_EXTERNAL_ELPA=ON
-DUSE_EXTERNAL_OMM=ON
-DUSE_EXTERNAL_SUPERLU=ON
```

The external libraries and the include paths should be set via the following three keywords:

```
-DLIB_PATHS=[ALL_DIRECTORIES_CONTAINING_YOUR_EXTERNAL_LIBRARIES]
-DINC_PATHS=[ALL_INCLUDE_DIRECTORIES_OF_YOUR_EXTERNAL_LIBRARIES]
-DLIBS=[NAMES_OF_ALL_YOUR_EXTERNAL_LIBRARIES]
```

Each of the above keywords is a space-separated or semicolon-separated list. If an external library depends on additional libraries, [LIBS](#) should include all the relevant libraries. For instance, [LIBS](#) should include the ELPA library and CUDA libraries when using an external ELPA compiled with CUDA support.

Please note that in the current version of ELSI, an external version of PEXSI is not officially supported, and the internal version of PEXSI is not enabled by default. It may be enabled by specifying:

```
-DENABLE_PEXSI=ON
```

## 2.4.3 Build Targets

By default, a static library (libelsi.a) will be created as the target of the compilation. Building ELSI as a shared library may be enabled by:

```
-DBUILD_SHARED_LIBS=ON
```

Building ELSI test programs may be enabled by:

```
-DENABLE_TESTS=ON
```

In either case, linear algebra libraries, BLAS, LAPACK, BLACS, and ScaLAPACK, should be valid in the [LIBS](#) keyword.

If test programs are enabled, the installation of ELSI may be verified by

```
$ ...
$ make [-j np]
$ make test
```

or

```
$ ...
$ make [-j np]
$ ctest
```

Note that the tests may not run if launching MPI jobs is prohibited on the user's working platform.

## 2.4.4 List of All Options

The options accepted by the ELSI CMake build system are listed here in alphabetical order. Some additional explanations are made below the table.

| Option                                 | Type    | Default    | Explanation                               |
|--|---------|------------|---|
| <a href="#">ADD_UNDERSCORE</a>         | boolean | ON         | Suffix C functions with an underscore     |
| <a href="#">CMAKE_C_COMPILER</a>       | string  | none       | MPI C compiler                            |
| <a href="#">CMAKE_C_FLAGS</a>          | string  | none       | C flags                                   |
| <a href="#">CMAKE_CXX_COMPILER</a>     | string  | none       | MPI C++ compiler                          |
| <a href="#">CMAKE_CXX_FLAGS</a>        | string  | none       | C++ flags                                 |
| <a href="#">CMAKE_Fortran_COMPILER</a> | string  | none       | MPI Fortran compiler                      |
| <a href="#">CMAKE_Fortran_FLAGS</a>    | string  | none       | Fortran flags                             |
| <a href="#">CMAKE_INSTALL_PREFIX</a>   | path    | /usr/local | Path to install ELSI                      |
| <a href="#">ELPA2_KERNEL</a>           | string  | none       | ELPA2 kernel                              |
| <a href="#">ENABLE_C_TESTS</a>         | boolean | OFF        | Build C test programs                     |
| <a href="#">ENABLE_PEXSI</a>           | boolean | OFF        | Enable PEXSI support                      |
| <a href="#">ENABLE_TESTS</a>           | boolean | OFF        | Build Fortran test programs               |
| <a href="#">INC_PATHS</a>              | string  | none       | Include directories of external libraries |
| <a href="#">LIB_PATHS</a>              | string  | none       | Directories containing external libraries |
| <a href="#">LIBS</a>                   | string  | none       | External libraries                        |
| <a href="#">USE_EXTERNAL_ELPA</a>      | boolean | OFF        | Use external ELPA                         |
| <a href="#">USE_EXTERNAL_OMM</a>       | boolean | OFF        | Use external libOMM and MatrixSwitch      |
| <a href="#">USE_EXTERNAL_SUPERLU</a>   | boolean | OFF        | Use external SuperLU_DIST                 |

### Comments

**1) [ADD\\_UNDERSCORE](#):** In the PEXSI and SuperLU\_DIST code redistributed through ELSI, there are calls to functions of the linear algebra libraries, e.g. “dgemm”. If [ADD\\_UNDERSCORE](#) is “ON”, the code will call “dgemm\_” instead of “dgemm”. Turn this keyword on if routines are suffixed with “\_” in external linear algebra libraries. Turn it off if routines are not suffixed with “\_”.

**2) [ELPA2\\_KERNEL](#):** As explained in 2.4.2, there are a number of computational kernels available with the ELPA solver. Choose from “BGQ” (IBM Blue Gene Q), “AVX” (Intel AVX), “AVX2” (Intel AVX2), and “AVX512” (Intel AVX512).

**3) External libraries:** ELSI redistributes source code of ELPA, libOMM, PEXSI, SuperLU\_DIST, and PT-SCOTCH libraries, which by default will be built together with the ELSI interface. Experienced users are encouraged to link the ELSI interface against external, better optimized solver libraries by setting [USE\\_EXTERNAL\\_XXX](#) and providing valid “XXX” library through [INC\\_PATHS](#), [LIB\\_PATHS](#), and [LIBS](#). Please note that in the current version of ELSI, an external version of PEXSI is not officially supported.

### 2.4.5 “Toolchain” Files

It is sometimes convenient to edit the settings in a “toolchain” file that can be read by CMake:

```
-DCMAKE_TOOLCHAIN_FILE=YOUR_TOOLCHAIN_FILE
```

Example “toolchains” are provided in the “./toolchains” directory of the ELSI package, which the user may use as templates to create new ones.

## 2.5 Importing ELSI into KS-DFT Codes

### 2.5.1 Linking a KS-DFT Code against ELSI

ELSI places its library and include files in the `lib` and `include` subfolders, respectively, of the ELSI root directory. An example set of compiler flags to link a generic KS-DFT code against a standard installation of ELSI are:

```
ELSI_INCLUDE = -I/PATH/TO/BUILD/ELSI/include
ELSI_LIB      = -L/PATH/TO/BUILD/ELSI/lib -lelsi \
               -lfortjson -lOMM -lMatrixSwitch -lelpa -lpexsi -lsuperlu_dist \
               -lptscotchparmetis -lptscotch -lptscotcherr \
               -lscotchmetis -lscotch -lscotcherr
```

These flags may vary based on the user’s needs. Linking ELSI against preinstalled solver libraries will require the user modify these flags accordingly.

If PEXSI is not enabled, an example set of the resulting compiler flags would be:

```
ELSI_INCLUDE = -I/PATH/TO/BUILD/ELSI/include
ELSI_LIB      = -L/PATH/TO/BUILD/ELSI/lib -lelsi \
               -lfortjson -lOMM -lMatrixSwitch -lelpa
```

### 2.5.2 Importing ELSI into a KS-DFT Code

ELSI may be used in the KS-DFT code by importing the appropriate header file. For codes written in Fortran, this is done by using the ELSI module

```
USE ELSI
```

For codes written in C, the ELSI wrapper may be imported by including the header file

```
#include <elsi.h>
```

These import statements give the KS-DFT code access to the ELSI interface. In the next chapter, we will describe the API for the ELSI interface.

# 3 The ELSI API

## 3.1 Overview of the ELSI API

In this chapter, we present the public-facing API for the ELSI Interface. We anticipate that fine details of this interface may change slightly in the future, but the fundamental structure of the interface layer is expected to remain consistent.

ELSI provides a C interface in addition to the native Fortran interface. The vast majority of this chapter, while written from a Fortran-ic standpoint, applies equally to both interfaces. Information specifically about the C wrapper for ELSI may be found in Section 3.8.

To allow multiple instances of ELSI to co-exist within a single calling code, we define an `elsi_handle` data type to encapsulate the state of an ELSI instance, i.e., all runtime parameters associated with the ELSI instance.

An `elsi_handle` instance is initialized with the `elsi_init` subroutine and is subsequently passed to all other ELSI subroutine calls.

## 3.2 Description of All ELSI APIs

In this section, we introduce the interface subroutines of ELSI. While this section serves as a reference to all subroutines, the user is encouraged to explore the demonstration codes of the ELSI interface in Section 3.7.

### 3.2.1 Initializing ELSI

The ELSI interface must be initialized via the `elsi_init` subroutine before any other ELSI subroutine may be called.

`elsi_init`(handle, solver, parallel\_mode, matrix\_format, n\_basis, n\_electron, n\_state)

| Argument                   | Data Type                      | in/out | Explanation  |
|----------------------------|--------------------------------|--------|--|
| <code>handle</code>        | <code>type(elsi_handle)</code> | out    | Handle to the current ELSI instance.   |
| <code>solver</code>        | integer                        | in     | The desired KS solver. Accepted values are: 1 (ELPA), 2 (LIBOMM), and 3 (PEXSI).   |
| <code>parallel_mode</code> | integer                        | in     | The type of parallelization used. See comment 3. Accepted values are: 0 (SINGLE_PROC) and 1 (MULTI_PROC).  |
| <code>matrix_format</code> | integer                        | in     | The matrix format used for the matrices handled by ELSI (Hamiltonian, overlap, density matrix, etc.) See comment 1. Accepted values are: 0 (BLACS_DENSE), 1 (PEXSI_CSC), and 2 (SIESTA_CSC). |
| <code>n_basis</code>       | integer                        | in     | Number of basis functions, i.e. the global size of the Hamiltonian matrix.   |
| <code>n_electron</code>    | real double                    | in     | Number of electrons.   |
| <code>n_state</code>       | integer                        | in     | Number of states, as interpreted by the solver. See comment 2.   |

#### Comments

1) **matrix\_format**: **BLACS\_DENSE**(0) refers a dense matrix format in a 2-dimensional block-cyclic distribution, i.e. the BLACS standard. **PEXSI\_CSC**(1) refers to a compressed sparse column (CSC) matrix format in a 1-dimensional block distribution. **SIESTA\_CSC**(2) refers to a compressed sparse column (CSC) matrix format in a 1-dimensional block-cyclic distribution. As the Hamiltonian, overlap, and density matrices are symmetric (Hermitian), compressed sparse row (CSR) matrix format is effectively supported.

2) **n.state**: If ELPA is the chosen solver, this parameter specifies the number of eigenstates to solve by the eigensolver. If libOMM is the chosen solver, **n.state** must be exactly the number of occupied states, as libOMM cannot handle fractional occupation numbers[2]. PEXSI does not make use of this parameter and a dummy value may be passed.

3) **parallel\_mode**: The two allowed values of **parallel\_mode**, 0 (**SINGLE\_PROC**) and 1 (**MULTI\_PROC**), allow for three parallelization strategies commonly employed by KS-DFT codes. See below.

3a) **SINGLE\_PROC**: Solves the KS eigenproblem following a LAPACK-like fashion. This option may only be selected when ELPA is chosen as the solver. This allows the following parallelization strategy:

---

### **SINGLE\_PROC** Example:

Every MPI task independently handles a group of k-points uniquely assigned to it.

Example number of k-points: 16

Example number of MPI tasks: 4

MPI task 0 handles k-points 1, 2, 3, 4 sequentially;

MPI task 1 handles k-points 5, 6, 7, 8 sequentially;

MPI task 2 handles k-points 9, 10, 11, 12 sequentially;

MPI task 3 handles k-points 13, 14, 15, 16 sequentially.

### **Pseudocode 1:**

```
elsi_init(elsi_h, ..., parallel_mode=0, ...)
...
for i_kpt = 1, n_kpoints_local, 1 do
    elsi_ev_{real|complex}(elsi_h, ham_this_kpt, ovlp_this_kpt, eval_this_kpt, evec_this_kpt)
end
```

---

3b) **MULTI\_PROC**: Solves the KS eigenproblem following a ScaLAPACK-like fashion. This allows the usage of the following two parallelization strategies:

---

### **MULTI\_PROC** Example:

Groups of MPI tasks coordinate to handle the same k-point, uniquely assigned to that group.

Example number of k-points: 4

Example number of MPI tasks: 16

MPI tasks 0, 1, 2, 3 cooperatively handle k-point 1;

MPI tasks 4, 5, 6, 7 cooperatively handle k-point 2;

MPI tasks 8, 9, 10, 11 cooperatively handle k-point 3;

MPI tasks 12, 13, 14, 15 cooperatively handle k-point 4.

## Pseudocode 2:

```
elsi_init(elsi_h, ..., parallel_mode=1, ...)
elsi_set_mpi(elsi_h, my_mpi_comm)
...
elsi_ev_{real|complex}(elsi_h, my_ham, my_ovlp, my_eval, my_evec)

or

elsi_init(elsi_h, ..., parallel_mode=1, ...)
elsi_set_mpi(elsi_h, my_mpi_comm)
elsi_set_kpoint(elsi_h, n_kpt, my_kpt, my_weight)
elsi_set_mpi_global(elsi_h, mpi_comm_global)
...
elsi_dm_complex(elsi_h, my_ham, my_ovlp, my_dm, global_energy)
```

In the last example, please note that when there is more than one k-point, a global MPI communicator must be provided for inter-k-points communications. See the documentation of [elsi\\_set\\_kpoint](#), [elsi\\_set\\_spin](#), and [elsi\\_set\\_mpi\\_global](#).

### 3.2.2 Setting Up MPI

The MPI communicator used by ELSI is passed into ELSI by the calling code via the [elsi\\_set\\_mpi](#) subroutine. When there is more than one k-point and/or spin channel, this communicator will be used only for one task defined

```
elsi_set_mpi(handle, mpi_comm)
```

| Argument                 | Data Type         | in/out | Explanation                          |
|--------------------------|-------------------|--------|--------------------------------------|
| <a href="#">handle</a>   | type(elsi_handle) | inout  | Handle to the current ELSI instance. |
| <a href="#">mpi_comm</a> | integer           | in     | MPI communicator.                    |

### 3.2.3 Setting Up Matrix Formats

When using the BLACS matrix format (BLACS\_DENSE), BLACS parameters are passed into ELSI via the [elsi\\_set\\_blacs](#) subroutine. The matrix format used internally in the ELSI interface and the ELPA solver requires the block sizes of the 2-dimensional block-cyclic distribution are the same in the row and column directions. It is necessary to call this subroutine before calling any solver interface that makes use of BLACS\_DENSE matrix format.

```
elsi_set_blacs(handle, blacs_context, block_size)
```

| Argument                      | Data Type         | in/out | Explanation  |
|-------------------------------|-------------------|--------|--|
| <a href="#">handle</a>        | type(elsi_handle) | inout  | Handle to the current ELSI instance.   |
| <a href="#">blacs_context</a> | integer           | in     | BLACS context.   |
| <a href="#">block_size</a>    | integer           | in     | Block size of the 2D block-cyclic distribution, specifying both row and column directions. |

When using the sparse matrix formats, namely 1D block distributed compressed sparse column (PEXSI\_CSC) or 1D block-cyclic distributed compressed sparse column (SIESTA\_CSC), the sparsity pattern should be passed into ELSI via the [elsi\\_set\\_csc](#) subroutine. It is necessary to call this subroutine before calling any solver interface that makes use of the sparse matrix formats.

`elsi_set_csc(handle, global_nnz, local_nnz, local_col, row_index, col_pointer)`

| Argument                 | Data Type                      | in/out | Explanation   |
|--------------------------|--------------------------------|--------|---|
| <code>handle</code>      | <code>type(elsi_handle)</code> | inout  | Handle to the current ELSI instance.                            |
| <code>global_nnz</code>  | integer                        | in     | Global number of non-zeros.                                     |
| <code>local_nnz</code>   | integer                        | in     | Local number of non-zeros.                                      |
| <code>local_col</code>   | integer                        | in     | Local number of matrix columns.                                 |
| <code>col_pointer</code> | integer, rank-1 array          | in     | Local column pointer array. Dimension: <code>local_nnz</code> . |
| <code>row_index</code>   | integer, rank-1 array          | in     | Local row index array. Dimension: <code>local_col+1</code> .    |

When using the 1D block distributed compressed sparse column (PEXSI\_CSC) format, the block size of the 1D distribution cannot be set by the user. This is because the PEXSI solver requires that the block size must be `n_basis / n_procs`, where `n_basis` is the total number of basis functions, and `n_procs` is the number of MPI tasks assigned to this problem. When using the 1D block-cyclic distributed compressed sparse column (SIESTA\_CSC) format, the block size of the 1D distribution must be set by the user.

`elsi_set_csc_blk(handle, block_size)`

| Argument                | Data Type                      | in/out | Explanation                                     |
|-------------------------|--------------------------------|--------|---|
| <code>handle</code>     | <code>type(elsi_handle)</code> | inout  | Handle to the current ELSI instance.            |
| <code>global_nnz</code> | integer                        | in     | Block size of the 1D block-cyclic distribution. |

### 3.2.4 Setting Up Multiple k-points and/or Spin Channels

When there is more than one k-point and/or spin channel in the simulating system, the ELSI interface can be configured to support parallel treatment of the k-points and/or spin channels. This is achieved by calling `elsi_set_kpoint` and `elsi_set_spin` to pass the required information into ELSI, and calling `elsi_set_mpi_global` to provide ELSI a global MPI communicator, which is used for a few communications among all the k-points and spin channels. Note that the current ELSI interface only supports the case where the eigenproblems for all the k-points and spin channels are fully parallelized, i.e., there is no MPI task that handles more than one k-point and/or spin channel.

`elsi_set_kpoint(handle, n_kpt, i_kpt, weight)`

| Argument            | Data Type                      | in/out | Explanation                                     |
|---------------------|--------------------------------|--------|---|
| <code>handle</code> | <code>type(elsi_handle)</code> | inout  | Handle to the current ELSI instance.            |
| <code>n_kpt</code>  | integer                        | in     | Total number of k-points.                       |
| <code>i_kpt</code>  | integer                        | in     | Index of the k-point handled by this MPI task.  |
| <code>weight</code> | integer                        | in     | Weight of the k-point handled by this MPI task. |

`elsi_set_spin(handle, n_spin, i_spin)`

| Argument            | Data Type                      | in/out | Explanation   |
|---------------------|--------------------------------|--------|---|
| <code>handle</code> | <code>type(elsi_handle)</code> | inout  | Handle to the current ELSI instance.                |
| <code>n_spin</code> | integer                        | in     | Total number of spin channels.                      |
| <code>i_spin</code> | integer                        | in     | Index of the spin channel handled by this MPI task. |

`elsi_set_mpi_global(handle, mpi_comm_global)`

| Argument                     | Data Type                      | in/out | Explanation   |
|------------------------------|--------------------------------|--------|---|
| <code>handle</code>          | <code>type(elsi_handle)</code> | inout  | Handle to the current ELSI instance.  |
| <code>mpi_comm_global</code> | integer                        | in     | Global MPI communicator used for communications among all k-points and spin channels. |



### 3.2.5 Finalizing ELSI

When an ELSI instance is no longer needed, its associated handle should be cleaned up by calling `elsi_finalize`.

`elsi_finalize(handle)`

| Argument            | Data Type                      | in/out | Explanation                          |
|---------------------|--------------------------------|--------|--------------------------------------|
| <code>handle</code> | <code>type(elsi_handle)</code> | inout  | Handle to the current ELSI instance. |

## 3.3 Solving Eigenvalues and Eigenvectors

The following subroutines return all the eigenvalues and (a subset of) eigenvectors of the provided H and S matrices. Only eigensolvers may be selected as the desired solver when using these subroutines.

`elsi_ev_real(handle, ham, ovlp, eval, evec)`

| Argument            | Data Type                      | in/out | Explanation  |
|---------------------|--------------------------------|--------|--|
| <code>handle</code> | <code>type(elsi_handle)</code> | inout  | Handle to the current ELSI instance.   |
| <code>ham</code>    | real double, rank-2 array      | inout  | Real Hamiltonian matrix in 2D block-cyclic dense format. See comment 1.                      |
| <code>ovlp</code>   | real double, rank-2 array      | inout  | Real overlap matrix (or its Cholesky factor) in 2D block-cyclic dense format. See comment 1. |
| <code>eval</code>   | real double, rank-1 array      | inout  | Eigenvalues. See comment 2.  |
| <code>evec</code>   | real double, rank-2 array      | out    | Real eigenvectors in 2D block-cyclic dense format. See comment 3.                            |

`elsi_ev_complex(handle, ham, ovlp, eval, evec)`

| Argument            | Data Type                      | in/out | Explanation   |
|---------------------|--------------------------------|--------|---|
| <code>handle</code> | <code>type(elsi_handle)</code> | inout  | Handle to the current ELSI instance.  |
| <code>ham</code>    | complex double, rank-2 array   | inout  | Complex Hamiltonian matrix in 2D block-cyclic dense format. See comment 1.                      |
| <code>ovlp</code>   | complex double, rank-2 array   | inout  | Complex overlap matrix (or its Cholesky factor) in 2D block-cyclic dense format. See comment 1. |
| <code>eval</code>   | real double, rank-1 array      | inout  | Eigenvalues. See comments 2.  |
| <code>evec</code>   | complex double, rank-2 array   | out    | Complex eigenvectors in 2D block-cyclic dense format. See comment 3.                            |

`elsi_ev_real_sparse(handle, ham, ovlp, eval, evec)`

| Argument            | Data Type                      | in/out | Explanation  |
|---------------------|--------------------------------|--------|--|
| <code>handle</code> | <code>type(elsi_handle)</code> | inout  | Handle to the current ELSI instance.                                   |
| <code>ham</code>    | real double, rank-1 array      | inout  | Real Hamiltonian matrix in 1D block or block-cyclic CSC sparse format. |
| <code>ovlp</code>   | real double, rank-1 array      | inout  | Real overlap matrix in 1D block or block-cyclic CSC sparse format.     |
| <code>eval</code>   | real double, rank-1 array      | inout  | Eigenvalues. See comment 2.  |
| <code>evec</code>   | real double, rank-2 array      | out    | Real eigenvectors in 2D block-cyclic dense format. See comment 3.      |

`elsi_ev_complex_sparse(handle, ham, ovlp, eval, evec)`

| Argument            | Data Type                      | in/out | Explanation   |
|---------------------|--------------------------------|--------|---|
| <code>handle</code> | <code>type(elsi_handle)</code> | inout  | Handle to the current ELSI instance.                                      |
| <code>ham</code>    | complex double, rank-1 array   | inout  | Complex Hamiltonian matrix in 1D block or block-cyclic CSC sparse format. |
| <code>ovlp</code>   | complex double, rank-1 array   | inout  | Complex overlap matrix in 1D block or block-cyclic CSC sparse format.     |
| <code>eval</code>   | real double, rank-1 array      | inout  | Eigenvalues. See comment 2.   |
| <code>evec</code>   | complex double, rank-2 array   | out    | Complex eigenvectors in 2D block-cyclic dense format. See comment 3.      |

## Comments

1) The Hamiltonian matrix will be destroyed by ELPA during computation. ELPA will overwrite the overlap matrix in its initial execution with the Cholesky factor, which will be reused by subsequent subroutine calls to `elsi_ev_real` or `elsi_ev_complex`. When using `elsi_ev_real_sparse`, the Cholesky factor (which is not sparse) is stored internally in the BLACS\_DENSE format. Starting from the second call to `elsi_ev_real_sparse`, the input sparse overlap matrix will not be referenced.

2) When using the ELPA solver, `elsi_ev_real`, `elsi_ev_complex`, `elsi_ev_real_sparse`, and `elsi_ev_complex_sparse` always compute all the eigenvalues, regardless of the choice of `n_state` specified in `elsi_init`. The dimension of `eval` thus should always be `n.basis`.

3) When using the ELPA solver, `elsi_ev_real`, `elsi_ev_complex`, `elsi_ev_real_sparse`, and `elsi_ev_complex_sparse` compute a subset of all eigenvectors. The number of eigenvectors to compute is specified by the keyword `n_state` in `elsi_init`. However, the local `eigenvectors` array should always be initialized to correspond to a global array of size `n.basis`  $\times$  `n.basis`, whose extra part is used as working space in ELPA. Note that when using `elsi_ev_real_sparse` and `elsi_ev_complex_sparse`, the eigenvectors are returned in a dense format (BLACS\_DENSE), as they are not necessarily sparse.

## 3.4 Computing Density Matrices

The following subroutines return the density matrix computed from the provided H and S matrices, as well as the energy corresponding to the occupied eigenstates. When the selected solver is ELPA, ELSI will internally construct the density matrix using the eigenvalues and eigenvectors returned by ELPA.

`elsi_dm_real(handle, ham, ovlp, dm, energy)`

| Argument            | Data Type                      | in/out | Explanation  |
|---------------------|--------------------------------|--------|--|
| <code>handle</code> | <code>type(elsi_handle)</code> | inout  | Handle to the current ELSI instance.   |
| <code>ham</code>    | real double, rank-2 array      | inout  | Real Hamiltonian matrix in 2D block-cyclic dense format.                                 |
| <code>ovlp</code>   | real double, rank-2 array      | inout  | Real overlap matrix (or Cholesky factor) in 2D block-cyclic dense format. See comment 1. |
| <code>dm</code>     | real double, rank-2 array      | out    | Real density matrix in 2D block-cyclic dense format.                                     |
| <code>energy</code> | real double                    | out    | Energy corresponding to the occupied eigenstates.  |

[elsi\\_dm\\_complex](#)(handle, ham, ovlp, dm, energy)

| Argument               | Data Type                    | in/out | Explanation   |
|------------------------|------------------------------|--------|---|
| <a href="#">handle</a> | type(elsi_handle)            | inout  | Handle to the current ELSI instance.  |
| <a href="#">ham</a>    | complex double, rank-2 array | inout  | Complex Hamiltonian matrix in 2D block-cyclic dense format.                                     |
| <a href="#">ovlp</a>   | complex double, rank-2 array | inout  | Complex overlap matrix (or its Cholesky factor) in 2D block-cyclic dense format. See comment 1. |
| <a href="#">dm</a>     | complex double, rank-2 array | out    | Complex density matrix in 2D block-cyclic dense format.   |
| <a href="#">energy</a> | real double                  | out    | Energy corresponding to the occupied eigenstates.   |

[elsi\\_dm\\_real\\_sparse](#)(handle, ham, ovlp, dm, energy)

| Argument               | Data Type                 | in/out | Explanation  |
|------------------------|---------------------------|--------|--|
| <a href="#">handle</a> | type(elsi_handle)         | inout  | Handle to the current ELSI instance.   |
| <a href="#">ham</a>    | real double, rank-1 array | inout  | Non-zero values of the real Hamiltonian matrix in 1D block or block-cyclic CSC format. |
| <a href="#">ovlp</a>   | real double, rank-1 array | inout  | Non-zero values of the real overlap matrix in 1D block or block-cyclic CSC format.     |
| <a href="#">dm</a>     | real double, rank-1 array | out    | Non-zero values of the real density matrix in 1D block or block-cyclic CSC format.     |
| <a href="#">energy</a> | real double               | out    | Energy corresponding to the occupied eigenstates.                                      |

[elsi\\_dm\\_complex\\_sparse](#)(handle, ham, ovlp, dm, energy)

| Argument               | Data Type                    | in/out | Explanation   |
|------------------------|------------------------------|--------|---|
| <a href="#">handle</a> | type(elsi_handle)            | inout  | Handle to the current ELSI instance.  |
| <a href="#">ham</a>    | complex double, rank-1 array | inout  | Non-zero values of the complex Hamiltonian matrix in 1D block or block-cyclic CSC format. |
| <a href="#">ovlp</a>   | complex double, rank-1 array | inout  | Non-zero values of the complex overlap matrix in 1D block or block-cyclic CSC format.     |
| <a href="#">dm</a>     | complex double, rank-1 array | out    | Non-zero values of the complex density matrix in 1D block or block-cyclic CSC format.     |
| <a href="#">energy</a> | real double                  | out    | Energy corresponding to the occupied eigenstates.   |

## Comments

1) If the chosen solver is ELPA or libOMM, the Hamiltonian matrix will be destroyed during the computation. ELPA will overwrite the overlap matrix in its initial execution with the Cholesky factor, which will be reused by subsequent subroutine calls to [elsi\\_dm\\_real](#).

## 3.5 Customizing ELSI

In ELSI, reasonable default values have been provided for a number of parameters used in the ELSI interface the the supported solvers. However, no set of default parameters can adequately cover all use cases. All parameters can be overridden through ELSI setter subroutines, as described in the following subsections.

### 3.5.1 Customizing the ELSI Interface

In all the subroutines listed below, the first argument (input and output) is an `elsi_handle`. The second argument (input) of each subroutine is the name of parameter to set.

Note that logical variables are not used in all ELSI API. Integers are used to represent logical, with 0 being false and any positive integer being true.

`elsi_set_output(handle, out_level)`  
`elsi_set_output_log(handle, out_log)`  
`elsi_set_write_unit(handle, write_unit)`  
`elsi_set_unit_ovlp(handle, unit_ovlp)`  
`elsi_set_zero_def(handle, zero_def)`  
`elsi_set_sing_check(handle, sing_check)`  
`elsi_set_sing_tol(handle, sing_tol)`  
`elsi_set_sing_stop(handle, sing_stop)`  
`elsi_set_uplo(handle, uplo)`  
`elsi_set_mu_broaden_scheme(handle, mu_broaden_scheme)`  
`elsi_set_mu_mp_order(handle, mu_mp_order)`  
`elsi_set_mu_broaden_width(handle, mu_broaden_width)`  
`elsi_set_mu_tol(handle, mu_tol)`

| Argument                       | Data Type   | Default    | Explanation   |
|--------------------------------|-------------|------------|---|
| <code>out_level</code>         | integer     | 0          | Output level of the ELSI interface. 0: no output. 1: standard ELSI output. 2: 1 + info from the solvers. 3: 2 + additional debug info.                                  |
| <code>out_log</code>           | integer     | 0          | If not 0, a separate log file in JSON format will be written out.   |
| <code>write_unit</code>        | integer     | 6          | The unit used in ELSI to write out information.   |
| <code>unit_ovlp</code>         | integer     | 0          | If not 0, the overlap matrix will be treated as an identity (unit) matrix in ELSI and the solvers. See comment 1.   |
| <code>zero_def</code>          | real double | $10^{-15}$ | When converting a matrix from dense to sparse format, values below this threshold will be discarded.  |
| <code>sing_check</code>        | integer     | 0          | If not 0, the singularity check of the overlap matrix will be performed. See comment 2.   |
| <code>sing_tol</code>          | real double | $10^{-5}$  | Eigenfunctions of the overlap matrix with eigenvalues smaller than this threshold will be removed to avoid ill-conditioning. See comment 1.                             |
| <code>sing_stop</code>         | integer     | 0          | If not 0, the code always stops if the overlap matrix is detected to be singular. See comment 1.  |
| <code>uplo</code>              | integer     | 0          | The input matrix format. 0: Input matrix is full, not triangular. This is currently the only supported option.  |
| <code>mu_broaden_scheme</code> | integer     | 0          | The broadening scheme employed to compute the occupation numbers and the Fermi level. 0: Gaussian. 1: Fermi-Dirac. 2: Methfessel-Paxton. 4: Marzari-Vanderbilt.         |
| <code>mu_mp_order</code>       | integer     | 0          | The order of the Methfessel-Paxton broadening scheme. No effect if Methfessel-Paxton is not the chosen broadening scheme.   |
| <code>mu_broaden_width</code>  | real double | 0.01       | The broadening width employed to compute the occupation numbers and the Fermi level. See comment 3.   |
| <code>mu_tol</code>            | real double | $10^{-13}$ | The convergence tolerance (in terms of the absolute error in electron count) of the bisection algorithm employed to compute the occupation numbers and the Fermi level. |

## Comments

1) If the input overlap matrix is an identity matrix, all settings related to the singularity (ill-conditioning) check take no effect.

2) If the singularity check is not disabled, in the first iteration of each SCF cycle, possible singularity of the overlap matrix is checked by computing all its eigenvalues. If there is any eigenvalue smaller than `sing_tol`, the matrix is considered to be singular.

3) In all supported broadening schemes, there is a term  $(\epsilon - E_F)/W$  in the distribution function, where  $\epsilon$  is the energy of an eigenstate, and  $E_F$  is the Fermi level. The `broadening_width` parameter should be set to  $W$ , in the unit of  $\epsilon$  and  $E_F$ .

### 3.5.2 Customizing the ELPA Solver

`elsi_set_elpa_solver(handle, elpa_solver)`

| Argument                 | Data Type | Default | Explanation  |
|--------------------------|-----------|---------|--|
| <code>elpa_solver</code> | integer   | 2       | 1: ELPA 1-stage solver. 2: ELPA 2-stage solver. 2 is usually faster and more scalable. |

### 3.5.3 Customizing the libOMM Solver

`elsi_set_omm_flavor(handle, omm_flavor)`

`elsi_set_omm_n_elpa(handle, omm_n_elpa)`

`elsi_set_omm_tol(handle, omm_tol)`

| Argument                | Data Type   | Default    | Explanation   |
|-------------------------|-------------|------------|---|
| <code>omm_flavor</code> | integer     | 0          | Method to perform OMM minimization. See comment 1.            |
| <code>omm_n_elpa</code> | integer     | 6          | Number of ELPA steps before libOMM. See comment 2.            |
| <code>omm_tol</code>    | real double | $10^{-12}$ | Convergence tolerance of orbital minimization. See comment 3. |

#### Comments

1) `omm_flavor`: Allowed choices are `0` for basic minimization of a generalized eigenproblem and `2` for a Cholesky factorization of the overlap matrix transforming the generalized eigenproblem to the standard form. Usually `2` leads to a faster convergence of the OMM energy functional minimization, at the price of transforming the eigenproblem. When using sufficiently many steps of ELPA to stabilize the SCF cycle, `0` is probably a better choice to finish the remaining SCF cycle. See also comment 2 below.

2) `omm_n_elpa`: It has been demonstrated that OMM is optimal at later stages of an SCF cycle where the electronic structure is closer to its expected local minimum, requiring only one CG iteration to converge the minimization of the OMM energy functional. Accordingly, it is recommended to use ELPA initially, then switching to libOMM after `omm_n_elpa` SCF steps.

3) `omm_tol`: A large minimization tolerance of course leads to a faster convergence, however unavoidably with a lower accuracy. `omm_tol` should be tested and chosen to balance the desired accuracy and computation time of the calling code.

### 3.5.4 Customizing the PEXSI Solver

`elsi_set_pexsi_n_pole(handle, pexsi_n_pole)`

`elsi_set_pexsi_n_mu(handle, pexsi_n_mu)`

`elsi_set_pexsi_np_per_pole(handle, pexsi_np_per_pole)`

`elsi_set_pexsi_np_symbo(handle, pexsi_np_symbo)`

`elsi_set_pexsi_ordering(handle, pexsi_ordering_method)`

`elsi_set_pexsi_temp(handle, pexsi_temp)`

`elsi_set_pexsi_gap(handle, pexsi_gap)`

`elsi_set_pexsi_delta_e(handle, pexsi_delta_e)`  
`elsi_set_pexsi_mu_min(handle, pexsi_mu_min)`  
`elsi_set_pexsi_mu_max(handle, pexsi_mu_max)`  
`elsi_set_pexsi_inertia_tol(handle, pexsi_inertia_tol)`

| Argument                       | Data Type   | Default | Explanation  |
|--------------------------------|-------------|---------|--|
| <code>pexsi_n_pole</code>      | integer     | 20      | Number of poles used by PEXSI. See comment 1.                  |
| <code>pexsi_n_mu</code>        | integer     | 2       | Number of mu points used by PEXSI. See comment 1.              |
| <code>pexsi_np_per_pole</code> | integer     | -       | Number of MPI tasks assigned to each mu point. See comment 2.  |
| <code>pexsi_np_symbo</code>    | integer     | 1       | Number of MPI tasks for symbolic factorization. See comment 3. |
| <code>pexsi_ordering</code>    | integer     | 0       | Matrix reordering method. See comment 3.                       |
| <code>pexsi_temp</code>        | real double | 0.002   | Temperature. See comment 4.                                    |
| <code>pexsi_gap</code>         | real double | 0       | Spectral gap. See comment 5.                                   |
| <code>pexsi_delta_e</code>     | real double | 10      | Spectral radius. See comment 6.                                |
| <code>pexsi_mu_min</code>      | real double | -10     | Minimum value of mu. See comment 7.                            |
| <code>pexsi_mu_max</code>      | real double | 10      | Maximum value of mu. See comment 7.                            |
| <code>pexsi_inertia_tol</code> | real double | 0.05    | Stopping criterion of inertia counting. See comment 7.         |

## Comments

1) Starting from PEXSI v1.0.0, the pole expansion is carried out by using the method proposed by Moussa. Usually, 20 poles are enough to get an accuracy that is comparable with the result obtained from diagonalization. Also introduced in PEXSI v1.0.0 is a robust and fast (no longer iterative) algorithm to determine the chemical potential, which performs Fermi operator expansion at several chemical potential values (referred to as “points” by PEXSI developers) in an SCF step, then interpolates the results at all points to get the final answer. The `pexsi_n_mu` setter controls the number of chemical potential “points” to be evaluated. In most cases, 2 points followed by a simple linear interpolation often yield reasonable results.

In short, we recommend `pexsi_n_pole` = 20 and `pexsi_n_mu` = 2.

2) `pexsi_np_per_pole`: PEXSI has, by construction, a 3-level parallelization: the 1st level independently handles all the poles in parallel; within each pole, the 2nd level evaluates the Fermi operator at all the chemical potential points in parallel; finally, within each point, parallel selected inversion is performed as the 3rd level. The value of `pexsi_np_per_pole` is the number of MPI tasks assigned to a single chemical potential point, for the parallel selected inversion at that point. Ideally, the total number of MPI tasks should be `pexsi_np_per_pole`  $\times$  `pexsi_n_mu`  $\times$  `pexsi_n_pole`, that is, all the three levels of the parallelization in PEXSI are fully exploited. In case that this is not feasible, PEXSI can also process the poles in serial, while the points must be handled simultaneously. The user should make sure that the total number of MPI tasks is a multiple of the number of MPI tasks per point times the number of points. The code will stop if this requirement is not fulfilled.

Assuming the ideal case, the default value for `pexsi_np_per_pole` is:  $n_{\text{total\_mpi}} / (n_{\text{pole}} \times n_{\text{mu}})$ .

If the dense density matrix solver interfaces are called, the input and output matrices should be 2D-block-cyclic-distributed among all available MPI tasks. In contrast, note that if the sparse density matrix solver interfaces are called, the input and output matrices should be 1D-block-distributed on the first `pexsi_np_per_pole` MPI tasks. PEXSI will then take over and automatically perform any necessary conversions to fit its need.

3) `pexsi_np_symbo`: The number of MPI tasks used for symbolic factorization cannot be larger than a magic number, depending on the matrix and the machine. The default value is set to 1 for safety. It is worth testing and increasing this number. The default matrix reordering method in PEXSI is set to parallel ordering using PTScotch or ParMETIS (0). The sequential ordering (1) may be necessary in rare cases.

4) `pexsi_temp`: This value corresponds to the  $1/k_B T$  term in the Fermi-Dirac distribution function.

5) `pexsi_gap`: The PEXSI method does not require a gap. If an estimate of the gap is unavailable, the default value usually works.

6) `pexsi_delta_e`: This is the spectral width of the eigensystem, i.e., the difference between the largest and smallest eigenvalues. Use the default value if no access to a better estimate.

7) The newly designed chemical potential determination algorithm relies on inertia counting to narrow down the chemical potential searching interval in the first few SCF steps. The `pexsi_inertia_tol` setter controls the stopping criterion of the inertia counting procedure. With a small interval obtained from the inertia counting step, PEXSI then selects a number of points in this interval to perform Fermi operator calculations, based on which a final chemical potential will be determined. The trick of this algorithm is that the chemical potential interval of the current SCF step can be used as a descent guess in the next SCF step. Therefore, the mechanism to choose input values for `pexsi_mu_min` and `pexsi_mu_max` is two-fold. For the first SCF iteration, they should be set to safe values that guarantee the true chemical potential lies in this interval. Then, for the  $n^{th}$  SCF step, `pexsi_mu_min` should be set to  $(\mu_{min}^{n-1} + \Delta V_{min})$ , `pexsi_mu_max` should be set to  $(\mu_{max}^{n-1} + \Delta V_{max})$ . Here,  $\mu_{min}^{n-1}$  and  $\mu_{max}^{n-1}$  are the lower bound and the upper bound of the chemical potential that are determined by PEXSI in the  $(n-1)^{th}$  SCF step. They can be retrieved by calling `elsi_get_pexsi_mu_min` and `elsi_get_pexsi_mu_max`, respectively (see Section 3.6.2). Suppose the effective potential (Hartree potential, exchange-correlation potential, and external potential) is stored in an array  $V$ , whose dimension is the number of grid points. From one SCF iteration to the next,  $\Delta V$  denotes the potential change, and  $\Delta V_{min}$  and  $\Delta V_{max}$  are the minimum and maximum values in the array  $\Delta V$ , respectively. The whole process is summarized in the following pseudo-code.

---

```
mu_min = -10.0
mu_max = 10.0
 $\Delta V_{min}$  = 0.0
 $\Delta V_{max}$  = 0.0

while SCF not converged do
    Update Hamiltonian

    elsi_set_pexsi_mu_min(elsi_h, mu_min +  $\Delta V_{min}$ )
    elsi_set_pexsi_mu_max(elsi_h, mu_max +  $\Delta V_{max}$ )

    elsi_dm_{real|complex}(elsi_h, ham, ovlp, dm, energy)

    elsi_get_pexsi_mu_min(elsi_h, mu_min)
    elsi_get_pexsi_mu_max(elsi_h, mu_max)

    Update electron density
    Update potential

     $\Delta V_{min}$  = minval( $V_{new}$  -  $V_{old}$ )
     $\Delta V_{max}$  = maxval( $V_{new}$  -  $V_{old}$ )

    Check SCF convergence
end
```

---

## 3.6 Getting Additional Results from ELSI

In Section 3.3 and Section 3.4, the interfaces to compute and return the eigensolutions and the density matrices have been introduced. Internally, ELSI and the solvers perform additional calculations whose results may not necessarily be useful for a normal SCF calculation, but can be valuable in some cases. One example is the energy-weighted density matrix that is employed to evaluate the Pulay forces during a geometry optimization calculation. The ELSI getter subroutines are used to retrieve such additional results from ELSI, as will be described in the following subsections.

### 3.6.1 Getting Results from the ELSI Interface

In all the subroutines listed below, the first argument (input and output) is an `elsi_handle`. The second argument (output) of each subroutine is the name of parameter to get.

`elsi_get_n_sing(handle, n_sing)`

`elsi_get_mu(handle, mu)`

`elsi_get_entropy(handle, ts)`

`elsi_get_edm_real(handle, edm_real)`

`elsi_get_edm_complex(handle, edm_complex)`

`elsi_get_edm_real_sparse(handle, edm_real_sparse)`

`elsi_get_edm_complex_sparse(handle, edm_complex_sparse)`

| Argument                        | Data Type                    | Explanation   |
|---------------------------------|------------------------------|---|
| <code>n_sing</code>             | integer                      | Number of eigenvalues of the overlap matrix that are smaller than the singularity tolerance. See Section 3.5.1. |
| <code>mu</code>                 | real double                  | Chemical potential. See comment 1.  |
| <code>ts</code>                 | real double                  | Entropy. See comment 1.   |
| <code>edm_real</code>           | real double, rank-2 array    | Real energy-weighted density matrix in 2D block-cyclic dense format. See comment 2.                             |
| <code>edm_complex</code>        | complex double, rank-2 array | Complex energy-weighted density matrix in 2D block-cyclic dense format. See comment 2.                          |
| <code>edm_real_sparse</code>    | real double, rank-1 array    | Non-zero values of the real density matrix in 1D block CSC format. See comment 2.                               |
| <code>edm_complex_sparse</code> | complex double, rank-1 array | Non-zero values of the complex density matrix in 1D block CSC format. See comment 2.                            |

#### Comments

1) In ELSI, the chemical potential will only be available if one of the density matrix solver interfaces has been called, with either ELPA or PEXSI being the chosen solver. The chemical potential can be retrieved by calling `elsi_get_mu`. The entropy will only be available if one of the density matrix solver interfaces has been called with ELPA being the chosen solver. The user should avoid calling the subroutine when the chemical potential or the entropy is not ready.

2) In general, the energy-weighted density matrix is only needed in a late stage of an SCF cycle to evaluate forces. It is, therefore, not calculated when any of the density matrix solver interface is called. When the energy-weighted density matrix is actually needed, it can be requested by calling the `elsi_get_edm` subroutines. Note that these subroutines all have the requirement that the corresponding `elsi_dm` subroutine must have been invoked. For instance, `elsi_get_edm_real_sparse` only makes sense if `elsi_dm_real_sparse` has been successfully executed.

### 3.6.2 Getting Results from the PEXSI Solver

`elsi_get_pexsi_mu_min(handle, pexsi_mu_min)`

`elsi_get_pexsi_mu_max(handle, pexsi_mu_max)`

| Argument                  | Data Type   | Explanation                         |
|---------------------------|-------------|-------------------------------------|
| <code>pexsi_mu_min</code> | real double | Minimum value of mu. See comment 1. |
| <code>pexsi_mu_max</code> | real double | Maximum value of mu. See comment 1. |

#### Comments

1) Please refer to the 7<sup>th</sup> comment in Section 3.5.4 for the chemical potential determination algorithm implemented in PEXSI and ELSI.



## 3.7 Demonstration Pseudo-Code

The typical workflow of ELSI within a KS-DFT code is demonstrated by the following pseudo-code. The ELSI interface routines and the operations to be performed by a KS-DFT code are marked with blue and red, respectively.

### 3.7.1 2D Block-Cyclic Distributed Dense Matrix + ELSI Eigensolver Interface

---

SCF initialize

```
elsi_init(elsi_h, ELPA, MULTIPROC, BLACS_DENSE, n_basis, n_electron, n_state)
elsi_set_mpi(elsi_h, mpi_comm)
elsi_set_blacs(elsi_h, blacs_ctxt, block_size)

while SCF not converged do
    Update Hamiltonian

    elsi_ev_{real|complex}(elsi_h, ham, ovlp, eval, evec)

    Update electron density
    Check SCF convergence
end

elsi_finalize(elsi_h)
```

---

### 3.7.2 1D Block Distributed CSC Sparse Matrix + ELSI Eigensolver Interface

---

SCF initialization

```
elsi_init(elsi_h, ELPA, MULTIPROC, PEXSL_CSC, n_basis, n_electron, n_state)
elsi_set_mpi(elsi_h, mpi_comm)
elsi_set_blacs(elsi_h, blacs_ctxt, block_size)
elsi_set_csc(elsi_h, global_nnz, local_nnz, local_col, row_index, col_pointer)

while SCF not converged do
    Update Hamiltonian

    elsi_ev_{real|complex}_sparse(elsi_h, ham, ovlp, eval, evec)

    Update electron density
    Check SCF convergence
end

elsi_finalize(elsi_h)
```

---

#### Comments

- 1) The SINGLE\_PROC parallel mode is not compatible with the PEXSL\_CSC matrix format.
- 2) Although the input Hamiltonian and overlap matrices are in a sparse format, the calculated eigenvectors are not in the same sparsity pattern. In fact, they are usually not sparse. Therefore, the eigenvectors are returned in the

BLACS\_DENSE format, which is required to be properly set up before calling `elsi_ev_{real|complex}`.

### 3.7.3 1D Block-Cyclic Distributed CSC Sparse Matrix + ELSI Eigensolver Interface

---

#### SCF initialization

```
elsi_init(elsi_h, ELPA, MULTI_PROC, SIESTA_CSC, n_basis, n_electron, n_state)
elsi_set_mpi(elsi_h, mpi_comm)
elsi_set_blacs(elsi_h, blacs_ctxt, block_size)
elsi_set_csc(elsi_h, global_nnz, local_nnz, local_col, row_index, col_pointer)
elsi_set_csc_blk(elsi_h, block_size)

while SCF not converged do
    Update Hamiltonian

    elsi_ev_{real|complex}_sparse(elsi_h, ham, ovlp, eval, evec)

    Update electron density
    Check SCF convergence
end

elsi_finalize(elsi_h)
```

---

#### Comments

- 1) The SINGLE\_PROC parallel mode is not compatible with the SIESTA\_CSC matrix format.
- 2) Although the input Hamiltonian and overlap matrices are in a sparse format, the calculated eigenvectors are not in the same sparsity pattern. In fact, they are usually not sparse. Therefore, the eigenvectors are returned in the BLACS\_DENSE format, which is required to be properly set up before calling `elsi_ev_{real|complex}`.

### 3.7.4 2D Block-Cyclic Distributed Dense Matrix + ELSI Density Matrix Interface

---

#### SCF initialization

```
elsi_init(elsi_h, LIBOMM, MULTIPROC, BLACS_DENSE, n_basis, n_electron, n_state)
elsi_set_mpi(elsi_h, mpi_comm)
elsi_set_blacs(elsi_h, blacs_ctxt, block_size)

while SCF not converged do
    Update Hamiltonian

    elsi_dm_{real|complex}(elsi_h, ham, ovlp, dm, energy)
    elsi_get_edm_{real|complex}(elsi_h, edm)

    Update electron density
    Check SCF convergence
end

elsi_finalize(elsi_h)
```

---

#### Comments

- 1) The SINGLE\_PROC parallel mode is not compatible with the ELSI density matrix interface.
- 2) The returned energy corresponds to the eigenenergies of the occupied Kohn-Sham orbitals.

### 3.7.5 1D Block Distributed CSC Sparse Matrix + ELSI Density Matrix Interface

---

#### SCF initialization

```
elsi_init(elsi_h, PEXSI, parallel_mode, PEXSI_CSC, n_basis, n_electron, n_state)
elsi_set_mpi(elsi_h, mpi_comm)
elsi_set_csc(elsi_h, global_nnz, local_nnz, local_col, row_index, col_pointer)

while SCF not converged do
    Update Hamiltonian

    elsi_dm_{real|complex}_sparse(elsi_h, ham, ovlp, dm, energy)
    elsi_get_edm_{real|complex}_sparse(elsi_h, edm)

    Update electron density
    Check SCF convergence
end

elsi_finalize(elsi_h)
```

---

#### Comments

- 1) The SINGLE\_PROC parallel mode is not compatible with the ELSI density matrix interface.

- 2) Refer to the 7<sup>th</sup> comment in Section 3.5.4 for the chemical potential determination algorithm implemented in PEXSI.
- 3) The returned energy corresponds to the eigenenergies of the occupied Kohn-Sham orbitals.

### 3.7.6 1D Block-Cyclic Distributed CSC Sparse Matrix + ELSI Density Matrix Interface

---

#### SCF initialization

```
elsi_init(elsi_h, PEXSI, parallel_mode, SIESTA_CSC, n_basis, n_electron, n_state)
elsi_set_mpi(elsi_h, mpi_comm)
elsi_set_csc(elsi_h, global_nnz, local_nnz, local_col, row_index, col_pointer)
elsi_set_csc_blk(elsi_h, block_size)
```

```
while SCF not converged do
```

```
    Update Hamiltonian
```

```
    elsi_dm_{real|complex}_sparse(elsi_h, ham, ovlp, dm, energy)
    elsi_get_edm_{real|complex}_sparse(elsi_h, edm)
```

```
    Update electron density
```

```
    Check SCF convergence
```

```
end
```

```
elsi_finalize(elsi_h)
```

---

#### Comments

- 1) The SINGLE\_PROC parallel mode is not compatible with the ELSI density matrix interface.
- 2) Refer to the 7<sup>th</sup> comment in Section 3.5.4 for the chemical potential determination algorithm implemented in PEXSI.
- 3) The returned energy corresponds to the eigenenergies of the occupied Kohn-Sham orbitals.

### 3.7.7 Multiple k-points Calculations

---

#### SCF initialization

```
elsi_init(elsi_h, ELPA, parallel_mode, BLACS_DENSE, n_basis, n_electron, n_state)
elsi_set_mpi(elsi_h, mpi_comm)
elsi_set_blacs(elsi_h, blacs_ctxt, block_size)

elsi_set_kpoint(elsi_h, n_kpt, i_kpt, weight)
elsi_set_mpi_global(elsi_h, mpi_comm_global)

while SCF not converged do
    Update Hamiltonian

    elsi_dm_{real|complex}(elsi_h, ham, ovlp, dm, energy)
    elsi_get_edm_{real|complex}(elsi_h, edm)

    Update electron density
    Check SCF convergence
end

elsi_finalize(elsi_h)
```

---

#### Comments

- 1) When there are multiple k-points, other than setting up the k-points and a global MPI communicator, there is no change in the way ELSI solver interfaces are called.
- 2) Calculations with two spin channels can be set up similarly.
- 3) One density matrix will be returned for each k-point. The KS-DFT code will take over to assemble the real-space density matrix. The returned energy, however, is already summed over all k-points with respect to the weight of each k-point.

## 3.8 C/C++ Interface

ELSI is written in Fortran, but many KS-DFT codes use C/C++. ELSI provides a C interface around the core Fortran ELSI code, which can be called from a C or C++ program. Each C wrapper function corresponds to a Fortran subroutine, where we have prefixed the original Fortran subroutine name with `c_` for clarity and consistency. Argument lists are identical to the associated native Fortran subroutine. For the complete definition of the C interface, the user is encouraged to look at the `elsi.h` header file directly.

# Bibliography

- [1] W. Kohn and L.J. Sham, Self-consistent equations including exchange and correlation effects, *Physical Review*, 140, 1133-1138 (1965).
- [2] F. Corsetti, The orbital minimization method for electronic structure calculations with finite-range atomic basis sets, *Computer Physics Communications*, 185, 873-883 (2014).
- [3] T. Auckenthaler et al., Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations, *Parallel Computing*, 37, 783-794 (2011).
- [4] A. Marek et al., The ELPA library: Scalable parallel eigenvalue solutions for electronic structure theory and computational science, *Journal of Physics: Condensed Matter*, 26, 213201 (2014).
- [5] L. Lin et al., Fast algorithm for extracting the diagonal of the inverse matrix with application to the electronic structure analysis of metallic systems, *Communications in Mathematical Sciences*, 7, 755-777 (2009).
- [6] L. Lin et al., Accelerating atomic orbital-based electronic structure calculation via pole expansion and selected inversion, *Journal of Physics: Condensed Matter*, 25, 295501 (2013).
- [7] J. E. Moussa, Minimax rational approximation of the Fermi-Dirac distribution, *The Journal of Chemical Physics*, 145, 164108 (2016).
- [8] B. Aradi et al., DFTB+, a sparse matrix-based implementation of the DFTB method, *Journal of Physical Chemistry A*, 111, 5678 (2007).
- [9] W. Hu et al., DGDFT: A massively parallel method for large scale density functional theory calculations, *The Journal of Chemical Physics*, 143, 124110 (2015).
- [10] V. Blum et al., Ab initio molecular simulations with numeric atom-centered orbitals, *Computer Physics Communications*, 180, 2175-2196 (2009).
- [11] M. Valiev et al., NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations, *Computer Physics Communications*, 181, 1477-1489 (2010).
- [12] J.M. Soler et al., The SIESTA method for ab initio order-N materials simulation, *Journal of Physics: Condensed Matter*, 14, 2745-2779 (2002).
- [13] V. Yu et al., ELSI: A unified software interface for Kohn-Sham electronic structure solvers, *Computer Physics Communications*, <http://dx.doi.org/10.1016/j.cpc.2017.09.007>.

# License and Copyright

ELSI Interface software is licensed under the 3-clause BSD license:

---

Copyright (c) 2015-2018, the ELSI team. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- 1) Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- 2) Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- 3) Neither the name of the "Electronic Structure Infrastructure (ELSI)" project nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL COPYRIGHT HOLDER BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

---

The source code of ELPA (LGPL), libOMM (2-clause BSD), PEXSI (3-clause BSD), and SuperLU\_DIST (3-clause BSD) are redistributed within ELSI. Individual license of each library can be found in the corresponding subfolder.