ELSI Interface
Development Version

User's Guide

The ELSI Team

http://elsi-interchange.org

August 19, 2019

Contents

1 Introduction

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

In Kohn-Sham density-functional theory (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

$$\hat{h}^{\mathrm{KS}}\psi_l = \epsilon_l \psi_l, \tag{1.1}$$

where ψ_l and ϵ_l are Kohn-Sham orbitals and their associated eigenenergies, and $\hat{h}^{\mathrm{KS}}$ denotes the Kohn-Sham Hamiltonian, which includes the kinetic energy, the average electrostatic potential of the electron density and of the nuclei (i.e. the Hartree potential), the exchange-correlation potential, and possible additional potential terms from external electromagnetic fields. These terms depend on the electron density $\boldsymbol{n}$, which is determined by the Kohn-Sham orbitals ψ_l. These terms also enter the Hamiltonian $\hat{h}^{\mathrm{KS}}$, which determines the Kohn-Sham orbitals ψ_l.

Due to this circular dependency, the Kohn-Sham equations are in fact a non-linear optimization problem, and therefore must be solved iteratively. The most commonly used method is the self-consistent field (SCF) approach. It usually starts from an initial guess of the electron density, from which the kinetic energy, electrostatic potential, exchange-correlation potential, and external potential are computed, forming the Kohn-Sham Hamiltonian. Then, the Kohn-Sham orbitals (wavefunctions) are solved from the Hamiltonian, and new electron density is computed from the Kohn-Sham orbitals. To achieve self-consistency, the electron density is updated in every SCF iteration until converged to an acceptable level.

In almost all practical approaches, N_{basis} basis functions $\phi_i(\boldsymbol{r})$ are employed to approximately expand the Kohn-Sham orbitals:

$$\psi_l(\boldsymbol{r}) = \sum_{j=1}^{N_{\mathrm{basis}}} c_{jl}\phi_j(\boldsymbol{r}). \tag{1.2}$$

The choice of basis set is one of the critical decisions in the design of an electronic structure code. Using non-orthogonal basis functions (e.g., Gaussian functions, Slater functions, numeric atom-centered orbitals) in 1.2 converts 1.1 to a generalized eigenvalue problem

$$\sum_j h_{ij}c_{jl} = \epsilon_l \sum_j s_{ij}c_{jl}, \tag{1.3}$$

where h_{ij} and s_{ij} are the elements of the Hamiltonian matrix $\boldsymbol{H}$ and the overlap matrix $\boldsymbol{S}$, which can be computed through numerical integrations:

$$\begin{aligned} h_{ij} &= \int d^3r[\phi_i^*(\boldsymbol{r})\hat{h}^{\mathrm{KS}}\phi_j(\boldsymbol{r})], \\ s_{ij} &= \int d^3r[\phi_i^*(\boldsymbol{r})\phi_j(\boldsymbol{r})]. \end{aligned} \tag{1.4}$$

1.3 can thus be expressed in the following matrix form

$$\boldsymbol{HC} = \boldsymbol{SC}\boldsymbol{\epsilon}. \tag{1.5}$$

Here, the matrix $\boldsymbol{C}$ and diagonal matrix $\boldsymbol{\epsilon}$ contain the eigenvectors and eigenvalues, respectively, of the eigensystem of the matrices $\boldsymbol{H}$ and $\boldsymbol{S}$.

When using orthonormal basis sets (e.g. plane waves, multi-resolution wavelets), the eigenproblem described in 1.5 reduces to a standard form where $s_{ij} = \delta_{ij}$.

The explicit solution of 1.3 or 1.5 yields the Kohn-Sham orbitals ψ_i, from which the electron density $n(\boldsymbol{r})$ can be computed following an orbital-based method:

$$n(\boldsymbol{r}) = \sum_{l=1}^{N_{\text{basis}}} f_l \psi_l^*(\boldsymbol{r}) \psi_l(\boldsymbol{r}), \tag{1.6}$$

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.

An alternative method can be employed for localized basis functions:

$$n(\boldsymbol{r}) = \sum_{i,j}^{N_{\text{basis}}} \phi_i^*(\boldsymbol{r}) p_{ij} \phi_j(\boldsymbol{r}), \tag{1.7}$$

with p_{ij} being the elements of the density matrix $\boldsymbol{P}$ that need to be computed before the density update:

$$p_{ij} = \sum_{l=1}^{N_{\text{basis}}} f_l c_{il} c_{jl}. \tag{1.8}$$

From a viewpoint of computational complexity, with localized basis functions, 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 remaining bottleneck for semilocal functionals is the eigenproblem described in Eqs. 1.3 and 1.5. The density matrix is directly accessible through methods other than diagonalization, therefore it is not always necessary to explicitly solve the eigenproblem. Which algorithm to use depends on many factors such as the choice of basis set, and the system and characters of the physical systems. In an SCF calculation, the eigenproblem needs to be tackled repeatedly. If this step is treated with the most efficient algorithm, the whole SCF calculation can be greatly accelerated.

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 version of ELSI supports ELPA [2, 3], libOMM [4], PEXSI [5, 6], EigenExa [7, 8], SLEPc-SIPs [9, 10], and NTPoly [11] solvers. Codes currently integrated with ELSI include DFTB+ [12], DGDFT [13], FHI-aims [14], and SIESTA [15].

- **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 electronic structure 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 used by the electronic structure 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 a hundred thousand CPU cores given a sufficiently large problem to solve, and PEXSI, with its efficient two-level parallelism, easily scales to tens of thousands of 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 Solver Libraries Supported by ELSI

Solvers supported in the current version of ELSI are: ELPA [2, 3], libOMM [4], PEXSI [5, 6], EigenExa [7, 8], SLEPc-SIPc [9, 10], and NTPoly [11]. The table below summarizes the supported data type, input/output matrix format, supported calculation type, and possible outputs of the solvers.

Solver	Data type	Matrix format	Spin/k-point	Output
ELPA	real/complex	dense/sparse	yes/yes	eigenvalues, eigenvectors, density matrix, energy-weighted density matrix, chemical potential, electronic entropy
libOMM	real/complex	dense/sparse	yes/yes	density matrix, energy-weighted density matrix
PEXSI	real/complex	dense/sparse	yes/yes	density matrix, energy-weighted density matrix, chemical potential
EigenExa	real	dense/sparse	yes/yes	eigenvalues, eigenvectors, density matrix, energy-weighted density matrix, chemical potential, electronic entropy
SLEPc-SIPs	real	dense/sparse	no/no	eigenvalues, eigenvectors, density matrix, energy-weighted density matrix, chemical potential, electronic entropy
NTPoly	real/complex	dense/sparse	yes/yes	density matrix, energy-weighted density matrix, chemical potential

What follows is a brief introduction of the solvers currently supported in ELSI. For detailed technical descriptions of the solvers, the reader is referred to the original publications of the solvers, e.g., those in the reference list of this document.

1.3.1 ELPA

The explicit solution of a generalized or standard eigenproblem is a well-studied task. The generalized eigenproblem in 1.5 is first transformed to the standard form, e.g., by Cholesky decomposition of the overlap matrix $\boldsymbol{S}$:

$$\boldsymbol{S} = \boldsymbol{L}\boldsymbol{L}^*, \tag{1.9}$$

where $\boldsymbol{L}$ is a lower triangular matrix. Applying $\boldsymbol{L}$ to $\boldsymbol{H}$ and $\boldsymbol{C}$ in the following way

$$\begin{aligned} \tilde{\boldsymbol{H}} &= \boldsymbol{L}^{-1}\boldsymbol{H}(\boldsymbol{L}^*)^{-1}, \\ \tilde{\boldsymbol{C}} &= \boldsymbol{L}^*\boldsymbol{C}, \end{aligned} \tag{1.10}$$

transforms 1.5 to a standard eigenproblem

$$\tilde{\boldsymbol{H}}\tilde{\boldsymbol{C}} = \tilde{\boldsymbol{C}}\boldsymbol{\epsilon}. \tag{1.11}$$

This standard eigenproblem is solved by further transforming it to a tridiagonal form

$$\boldsymbol{T} = \boldsymbol{Q}\tilde{\boldsymbol{H}}\boldsymbol{Q}^*, \tag{1.12}$$

where $\boldsymbol{Q}$ is a transformation matrix, and $\boldsymbol{T}$ is a tridiagonal matrix whose eigenvalues and eigenvectors are computed by, e.g., the divide-and-conquer approach or the MRRR method. This procedure is called "diagonalization", as the full matrix is reduced to a (tri)diagonal form.

The massively parallel direct eigensolver ELPA [2, 3] facilitates the direct solution of symmetric or Hermitian eigenproblems on high-performance computers by adopting a two-stage diagonalization algorithm, which first reduces the full matrix to a banded intermediate form, then to the tridiagonal form:

$$\begin{aligned} \boldsymbol{B} &= \boldsymbol{Q}_1\tilde{\boldsymbol{H}}\boldsymbol{Q}_1^*, \\ \boldsymbol{T} &= \boldsymbol{Q}_2\boldsymbol{B}\boldsymbol{Q}_2^*. \end{aligned} \tag{1.13}$$

where $\boldsymbol{Q}_1$ and $\boldsymbol{Q}_2$ are transformation matrices used in the two-stage diagonalization; $\boldsymbol{B}$ is a banded matrix; and $\boldsymbol{T}$ is a tridiagonal matrix. Compared to the one-stage diagonalization (1.12), the two-stage approach introduces two additional

steps. Still, the two-stage approach 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 1.12 can be mostly replaced by more efficient matrix-matrix operations (BLAS level-3 routines) in 1.13. The computational workload associated with the back-transformation of the eigenvectors is greatly alleviated if only a small fraction of the eigenvectors representing the lowest eigenstates is required, and by architecture-specific linear-algebra "kernels" provided with the ELPA library.

1.3.2 libOMM

Instead of diagonalizing the $N_{\text{basis}} \times N_{\text{basis}}$ eigenproblem, the orbital minimization method (OMM) minimizes an unconstrained energy functional using a set of auxiliary Wannier functions. At the minimum of the OMM energy functional, the Wannier functions can be used to construct the density matrix. Specifically, N_{W} non-orthogonal Wannier functions χ_k are employed to represent the occupied subspace of a system with N_{electron} electrons:

$$\chi_k = \sum_{j=1}^{N_{\text{basis}}} W_{kj}\phi_j. \tag{1.14}$$

For non-spin-polarized systems, the index k runs from 1 to $N_{\text{W}} = N_{\text{electron}}/2$. Then the matrices $\boldsymbol{H}$ and $\boldsymbol{S}$ are transformed into the occupied subspace

$$\begin{aligned} \boldsymbol{H}_{\text{omm}} &= \boldsymbol{W}^* \boldsymbol{H} \boldsymbol{W}, \\ \boldsymbol{S}_{\text{omm}} &= \boldsymbol{W}^* \boldsymbol{S} \boldsymbol{W}, \end{aligned} \tag{1.15}$$

where $\boldsymbol{W}$ is the coefficient matrix of the Wannier functions, whose dimension is $N_{\text{basis}} \times N_{\text{W}}$; $\boldsymbol{H}_{\text{omm}}$ and $\boldsymbol{S}_{\text{omm}}$ are $N_{\text{W}} \times N_{\text{W}}$ matrices. The OMM energy functional can then be evaluated from $\boldsymbol{H}_{\text{omm}}$ and $\boldsymbol{S}_{\text{omm}}$:

$$E[\boldsymbol{W}] = 4\text{Tr}[\boldsymbol{H}_{\text{omm}}] - 2\text{Tr}[\boldsymbol{S}_{\text{omm}}\boldsymbol{H}_{\text{omm}}]. \tag{1.16}$$

This energy functional, when minimized with respect to the coefficients of Wannier functions $\boldsymbol{W}$, is equal to the band structure energy, i.e. the sum of the energies of all eigenstates, weighted with their respective occupation numbers. Furthermore, the Wannier functions are driven towards orthonormality at this minimum. The density matrix is then constructed from the Wannier functions that minimize $E[\boldsymbol{W}]$.

Different from the originally proposed linear scaling OMM method, the OMM implementation in the libOMM library [4] is a cubic scaling density matrix solver. Theoretically, this implementation has a smaller prefactor than the direct diagonalization method. In libOMM, the minimization of the OMM energy functional is carried out with the conjugate-gradient (CG) method, whose performance mainly depends on the convergence rate of the minimization.

1.3.3 PEXSI

The pole expansion and selected inversion (PEXSI) method [5, 6] expands the density matrix $\boldsymbol{P}$ with rational functions:

$$\boldsymbol{P} = \sum_l \text{Im}\left(\frac{\omega_l}{\boldsymbol{H} - (z_l + \mu)\boldsymbol{S}}\right), \tag{1.17}$$

where μ is the chemical potential of the system; $\{z_l\}$ and $\{\omega_l\}$ are complex shifts and weights of the expansion terms. About 20 poles are usually sufficient for the result obtained from PEXSI to be fully comparable to that obtained from diagonalization. These poles can be processed in parallel, making PEXSI a highly scalable method on high performance computers.

Only selected elements of the object $(\boldsymbol{H} - (z_l + \mu)\boldsymbol{S})^{-1}$ corresponding to non-zero elements of $\boldsymbol{H}$ and $\boldsymbol{S}$ are computed with the parallel selected inversion method. The computational complexity of Eq. 1.17 depends on the dimensionality of the system: O(N), O($\text{N}^{1.5}$), and O(N^2) for 1D, 2D, and 3D systems, respectively. This favorable scaling does not rely on the existence of an energy gap. The PEXSI method is thus generally applicable to insulating as well as metallic systems, which differentiates PEXSI from traditional linear scaling algorithms.

1.3.4 EigenExa

The EigenExa library consists of two massively parallel implementations of direct, dense eigensolver. The eigen_s solver in EigenExa adopts the conventional one-stage (tri)diagonalization algorithm as in Eq. 1.12. Reimplemented from scratch, eigen_s exhibits a parallel performance superior to the ScaLAPACK solvers and comparable to the one-stage ELPA solver [7, 8]. Nevertheless, its performance is limited by the relatively inefficient matrix-vector operations (BLAS level-2 routines).

The eigen_sx solver in EigenExa features a novel pentadiagonalization approach that transforms the full matrix in Eq. 1.11 to a pentadiagonal form

$$\boldsymbol{P} = \boldsymbol{Q}\tilde{\boldsymbol{H}}\boldsymbol{Q}^*, \tag{1.18}$$

where $\boldsymbol{Q}$ is a transformation matrix, and $\boldsymbol{P}$ is a pentadiagonal matrix whose eigenvalues and eigenvectors are computed by a divide-and-conquer algorithm generalized from the divide-and-conquer algorithm for tridiagonal matrices. Compared to the tridiagonalization algorithm in eigen_s, The pentadiagonalization algorithm in eigen_sx makes more use of matrix-matrix operations (level-3 BLAS routines), leading to an improvement in its efficiency despite the slightly increased workload in the divide-and-conquer solve [7, 8]. Compared to the two-stage ELPA solver, eigen_sx does not have an advantage in the forward transformation, but it avoids the forward and backward transformations between a banded matrix and a tridiagonal matrix, which stands as a significant computational burden in ELPA when a large part of the eigenspectrum is of interest.

At present, the EigenExa library only provides routines for solving standard eigenproblems. When solving generalized eigenproblems with EigenExa, ELSI relies on routines from ELPA to carry out the eigenproblem transformation in Eqs. 1.9 and 1.10.

1.3.5 SLEPc-SIPs

The shift-and-invert spectral transformation method, implemented in the SLEPc library [9], transforms the eigenproblem 1.5 by shifting the eigenspectrum:

$$(\boldsymbol{H} - \boldsymbol{\sigma}\boldsymbol{S})\boldsymbol{C} = \boldsymbol{S}\boldsymbol{C}(\boldsymbol{\epsilon} - \boldsymbol{\sigma}), \tag{1.19}$$

where $\boldsymbol{\sigma}$ is a diagonal matrix with diagonal elements all equal to the shift σ. This shifted eigenproblem is converted to the standard form by inverting $(\boldsymbol{H} - \boldsymbol{\sigma}\boldsymbol{S})$ and $(\boldsymbol{\epsilon} - \boldsymbol{\sigma})$:

$$(\boldsymbol{H} - \boldsymbol{\sigma}\boldsymbol{S})^{-1}\boldsymbol{S}\boldsymbol{C} = (\boldsymbol{\epsilon} - \boldsymbol{\sigma})^{-1}\boldsymbol{C}, \tag{1.20}$$

If the shift can be chosen to be close to the target eigenvalue, 1.20 makes the magnitude of the transformed eigenvalues large, accelerating the convergence of the iterative Krylov-Schur eigensolver used in SLEPc.

On top of the basic shift-and-invert, the shift-and-invert parallel spectral transformation (SIPs) method [10] partitions the eigenspectrum of a given eigenproblem into N_{slice} slices. Accordingly, the processes involved in the calculation are split into N_{slice} groups, each of which solves one slice independently. Within the slices, carefully selected shifts are applied to the original problem. With this layer of parallelism across slices, the SLEPc-SIPs solver has the potential to exhibit enhanced scalability over direct diagonalization methods, especially when the load balance across slices can be guaranteed. Indeed, this has been reported to happen with very sparse Hamiltonian and overlap matrices out of density-functional tight-binding (DFTB) calculations [10].

1.3.6 NTPoly

Density matrix purification is an established way to achieve linear scaling KS-DFT. Assuming an orthogonal basis set, the density matrix $\boldsymbol{P}$ should satisfy the following conditions:

$$\begin{aligned} \boldsymbol{P} &= \boldsymbol{P}^*, \\ \mathrm{Tr}(\boldsymbol{P}) &= N_{\text{electron}}, \\ \boldsymbol{P} &= \boldsymbol{P}^2. \end{aligned} \tag{1.21}$$

An initial guess of such a matrix can be obtained by scaling the Hamiltonian matrix to make sure its eigenvalues lie in between 0 and 1. Then, the "purification" method iteratively updates the density matrix until it converges to a certain criterion. The converged density matrix satisfies the three conditions in 1.21.

The process of density matrix purification can be written in the general form

$$\boldsymbol{P}_{\mathrm{n}+1} = f(\boldsymbol{P}_{\mathrm{n}}), \tag{1.22}$$

where $\boldsymbol{P}_{\mathrm{n}}$ is the density matrix in the n^{th} purification iteration, $\boldsymbol{P}_{\mathrm{n}+1}$ is the density matrix in the $(\mathrm{n}+1)^{\mathrm{th}}$ iteration, and $f(\boldsymbol{P})$ is usually a matrix polynomial, which can be calculated by matrix-matrix multiplications.

Various algorithms have been developed to carry out the density matrix purification efficiently, such as the canonical purification [16], the trace resetting purification methods [17], and the generalized canonical purification [18]. These methods are implemented in the NTPoly library [11] using its distributed-memory sparse matrix-matrix multiplication kernel. Given sufficiently sparse matrices, the computational complexity of density matrix purification with NTPoly is O(N) for insulating systems.

1.4 Citing ELSI

Key concepts of ELSI and the first version of its implementation are described in the following paper [19]:

V. W-z. Yu, F. Corsetti, A. García, W. P. Huhn, M. Jacquelin, W. Jia, B. Lange, L. Lin, J. Lu, W. Mi, A. Seifitokaldani, Á. Vázquez-Mayagoitia, C. Yang, H. Yang, and V. Blum, ELSI: A Unified Software Interface for Kohn-Sham Electronic Structure Solvers, Computer Physics Communications, 222, 267-285 (2018).

In addition, an incomplete list of publications describing the solvers supported in ELSI may be found in the bibliography of this document. Please consider citing these articles when publishing results obtained with ELSI.

1.5 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 Prerequisites

The ELSI package contains the ELSI interface software as well as redistributed source code for the solver libraries ELPA (version 2016.11.001), libOMM, PEXSI (version 1.2.0), and NTPoly (version 2.3.1). The installation of ELSI makes use of the CMake software. Minimum requirements include:

```
CMake  [minimum version 3.0; newer version recommended]
Fortran compiler  [Fortran 2003 compliant]
C compiler  [C99 compliant]
MPI  [MPI-3 recommended]
```

Building the PEXSI solver (highly recommended) requires:

```
C++ compiler  [C++ 11 compliant]
```

Building the EigenExa solver requires:

```
EigenExa  [version 2.3 or newer]
```

Building the SLEPc-SIPs solver requires:

```
SLEPc  [version 3.9 or newer]
PETSc  [version 3.9 or newer, with SuperLU_DIST, MUMPS, ParMETIS, and PT-SCOTCH enabled]
```

Linear algebra libraries should be provided for ELSI to link against:

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

By default, the redistributed ELPA, libOMM, and NTPoly solvers will be built. If PEXSI is enabled during configuration, the redistributed PEXSI library and its dependencies, namely the SuperLU_DIST and PT-SCOTCH libraries, will be built as well. Optionally, the redistributed ELPA, libOMM, PEXSI, SuperLU_DIST, PT-SCOTCH, and NTPoly libraries may be substituted by user's optimized versions.

2.2 Quick Start

We recommend preparing configuration settings in a toolchain file that can be read by CMake. Edit one of the templates provided in the "toolchains" directory of the ELSI package. As an example, a minimal Intel toolchain looks like

```
# Modify contents in red if necessary
set(CMAKE_Fortran_COMPILER "mpiifort" CACHE STRING "MPI Fortran compiler")
set(CMAKE_Fortran_FLAGS "-O3 -ip -fp-model precise" CACHE STRING "Fortran flags")
set(LIB_PATHS "$ENVMKLROOT/lib/intel64" CACHE STRING "External library paths")
set(LIBS "mkl_scalapack_lp64 mkl_blacs_intelmpi_lp64 mkl_intel_lp64 mkl_sequential mkl_core"
    CACHE STRING "External libraries")
```

This will build ELSI with the redistributed ELPA, libOMM, and NTPoly solvers. To enable more solvers, check out the examples in the "toolchains" directory, and refer to a complete list of options in 2.3.5.

Once a toolchain file is ready, follow the steps below:

```
$ cd elsi-interface
$ ls

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

$ mkdir build
$ cd build
$ cmake -DCMAKE_TOOLCHAIN_FILE=YOUR_TOOLCHAIN_FILE ..

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

$ make [-j np]
$ [make install]
```

"YOUR_TOOLCHAIN_FILE" should be the user's toolchain file. Commands in square brackets are optional.

If the compilation succeeds, the next step would be reading the code examples in the "test" directory of the ELSI package, which showcase the use of ELSI in C and Fortran programs.

2.3 Configuration

2.3.1 Compilers

CMake automatically detects compilers. The choices made by CMake often work, but they do not necessarily lead to 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 in CMAKE_Fortran_COMPILER, CMAKE_C_COMPILER, and CMAKE_CXX_COMPILER. Please note that the C++ compiler is not needed when building ELSI without PEXSI.

In addition, it is highly recommended to specify the compiler flags in CMAKE_Fortran_FLAGS, CMAKE_C_FLAGS, and CMAKE_CXX_FLAGS.

2.3.2 Solvers

The ELPA, libOMM, PEXSI, and NTPoly 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), which may be chosen by the ELPA2_KERNEL keyword. Available options are AVX, AVX2, and AVX512, for architectures supporting Intel AVX, AVX2, and AVX512 instruction sets, respectively. In ELPA, these kernels are employed to accelerate the calculation of eigenvectors, which is often a bottleneck when calculating a large portion of the eigenspectrum.

The PEXSI, EigenExa, and SLEPc-SIPs solvers are not enabled by default. They may be activated by the keywords ENABLE_PEXSI, ENABLE_EIGENEXA, and ENABLE_SIPS, respectively. PEXSI 1.2.0, EigenExa 2.3 and 2.4, SLEPc 3.9, 3.10, and 3.11 have been tested with this version of ELSI. Older/newer versions may or may not be compatible. The PETSc library, required by SLEPc, must be compiled with MPI support, and (at least) with packages SuperLU_DIST, MUMPS, ParMETIS, and PT-SCOTCH enabled.

Experienced users are encouraged to link ELSI against external, better optimized solver libraries. The keywords USE_EXTERNAL_ELPA, USE_EXTERNAL_OMM, USE_EXTERNAL_PEXSI, and USE_EXTERNAL_NTPOLY control the usage of externally compiled ELPA, libOMM, PEXSI, and NTPoly, respectively.

All external libraries and include paths should be set via INC_PATHS, LIB_PATHS, and LIBS, each of which is a list of items separated with " " (space) or ";" (semicolon). If an external library depends on additional libraries, LIBS should include all the relevant dependencies. For instance, LIBS should include the ELPA library and CUDA libraries when using an external ELPA compiled with GPU (CUDA) support.

2.3.3 Shared Library

Building ELSI as a shared library may be enabled by BUILD_SHARED_LIBS.

2.3.4 Tests

Building ELSI test programs may be enabled by ENABLE_TESTS. Then, the compilation may be verified by "`make test`" or "`ctest`". Note that the tests may not run if launching MPI jobs is prohibited on the user's working platform.

2.3.5 List of All Configure 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
ADD_UNDERSCORE	boolean	ON	Suffix C functions with an underscore
BUILD_SHARED_LIBS	boolean	OFF	Build ELSI as a shared library
CMAKE_C_COMPILER	string	none	MPI C compiler
CMAKE_C_FLAGS	string	none	C flags
CMAKE_CXX_COMPILER	string	none	MPI C++ compiler
CMAKE_CXX_FLAGS	string	none	C++ flags
CMAKE_Fortran_COMPILER	string	none	MPI Fortran compiler
CMAKE_Fortran_FLAGS	string	none	Fortran flags
CMAKE_INSTALL_PREFIX	path	/usr/local	Path to install ELSI
ELPA2_KERNEL	string	none	ELPA2 kernel
ENABLE_C_TESTS	boolean	OFF	Build C test programs
ENABLE_EIGENEXA	boolean	OFF	Enable EigenExa support
ENABLE_PEXSI	boolean	OFF	Enable PEXSI support
ENABLE_SIPS	boolean	OFF	Enable SLEPc-SIPs support
ENABLE_TESTS	boolean	OFF	Build Fortran test programs
INC_PATHS	string	none	Include directories of external libraries
LIB_PATHS	string	none	Directories containing external libraries
LIBS	string	none	External libraries
MPIEXEC_NP	string	mpirun -n 4	Command to run tests in parallel with MPI
MPIEXEC_1P	string	mpirun -n 1	Command to run tests in serial with MPI
SCOTCH_LAST_RESORT	string	none	Command to invoke PT-SCOTCH header generator
USE_EXTERNAL_ELPA	boolean	OFF	Use external ELPA
USE_EXTERNAL_NTPOLY	boolean	OFF	Use external NTPoly
USE_EXTERNAL_OMM	boolean	OFF	Use external libOMM and MatrixSwitch
USE_EXTERNAL_PEXSI	boolean	OFF	Use external PEXSI (if PEXSI enabled)
USE_MPI_IALLGATHER	boolean	ON	Use non-blocking collective MPI functions

Remarks

1) ADD_UNDERSCORE: In the redistributed PEXSI and SuperLU_DIST code, there are calls to functions from the linear algebra libraries, e.g. "dgemm". If ADD_UNDERSCORE is "ON", the code will call "dgemm_" instead of "dgemm". Turn this keyword off if routines are not suffixed with "_" in the linear algebra libraries.

2) CMAKE_INSTALL_PREFIX: ELSI may be installed to the location specified in CMAKE_INSTALL_PREFIX by "`make install`".

3) ELPA2_KERNEL: There are a number of computational kernels available with the ELPA solver. Choose from "AVX" (Intel AVX), "AVX2" (Intel AVX2), and "AVX512" (Intel AVX512). See 2.3.2 for more information.

4) SCOTCH_LAST_RESORT: The compilation of the PT-SCOTCH library is a multi-step process. First, two auxiliary executables are created. Then, header files of the library are generated on-the-fly by the two executables. Finally, the main source files of the library are compiled with the generated header files included. The header generation step may fail on platforms where directly running an executable is prohibited, e.g. login nodes of a supercomputer. Often this can be circumvented by requesting an interactive session to a compute node and compiling the code there, or by submitting the compilation as a job to the queuing system. However, this may still fail on platforms where an executable compiled with MPI must be launched by an MPI job launcher (aprun, mpirun, srun, etc). If the standard compilation of PT-SCOTCH fails due to this reason, the user may set SCOTCH_LAST_RESORT to the command that starts an MPI job with one MPI task, e.g. "`mpirun -n 1`". This command will be used to launch the auxiliary executables to generate necessary header files for PT-SCOTCH.

5) External libraries: ELSI redistributes source code of ELPA, libOMM, NTPoly, 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. See 2.3.2 for more information.

6) USE_MPI_IALLGATHER: NTPoly makes use of non-blocking collective MPI functions such as `MPI_Iallgatherv` to overlap its computation and communication. If these MPI functions are not available in the user's MPI version, set USE_MPI_IALLGATHER to 'OFF'. Using this flag may lead to reduced performance. Upgrade MPI if possible.

2.4 Importing ELSI into Third-Party Code Projects

2.4.1 Linking against ELSI: CMake

A CMake configuration file called `elsiConfig.cmake` should be generated after ELSI is successfully installed. This file contains all the information about how the ELSI library and its dependencies should be included in an external project. For a project using CMake, only two lines are required to find and link to ELSI:

```
find_package(elsi REQUIRED)
target_link_libraries(my_project PRIVATE elsi::elsi)
```

If a minimum version of ELSI is required, this information may be passed to "`find_package`" by, e.g.:

```
find_package(elsi 2.0 REQUIRED)
```

In this case, if ELSI version 1.x is found, CMake stops with an appropriate error message.

2.4.2 Linking against ELSI: Makefile

For a project using makefiles, an example set of compiler flags to link against ELSI would be:

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

Enabling/disabling PEXSI, EigenExa, SLEPc-SIPs or linking ELSI against pre-installed solver libraries will require the user modify these flags accordingly.

2.4.3 Using ELSI

ELSI may be used in an electronic structure 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 electronic structure 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. While this chapter serves as a reference to the ELSI subroutines, the user is encouraged to explore the demonstration pseudo-codes of ELSI in 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.

ELSI provides a C interface in addition to the native Fortran interface. The vast majority of this chapter, while written from a Fortran standpoint, applies equally to both interfaces. Information specifically about the C wrapper for ELSI may be found in 3.9.

In the source code of ELSI, there may exist subroutines that are not documented as public API here. Usage of those undocumented subroutines is not recommended, as they are usually experimental and subject to modification or removal without notice.

3.2 Setting Up ELSI

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
handle	type(elsi_handle)	out	Handle to ELSI.
solver	integer	in	Desired solver. Accepted values are: 0 (AUTO), 1 (ELPA), 2 (libOMM), 3 (PEXSI), 4 (EigenExa), 5 (SLEPc-SIPs), and 6 (NTPoly). See remark 1.
parallel_mode	integer	in	Parallelization mode. Accepted values are: 0 (SINGLE_PROC) and 1 (MULTI_PROC). See remark 4.
matrix_format	integer	in	Matrix format. Accepted values are: 0 (BLACS_DENSE), 1 (PEXSI_CSC), 2 (SIESTA_CSC), and 3 (GENERIC_COO). See remark 2.
n_basis	integer	in	Number of basis functions, i.e. global size of Hamiltonian.
n_electron	real double	in	Number of electrons.
n_state	integer	in	Number of states. See remark 3.

Remarks

1) solver: Refer to 1.3 for supported features of each solver. The AUTO(0) option attempts to automate the solver selection procedure based on benchmarks performed and experiences gained in the ELSI project. User-supplied information

may assist in finding the optimal solver. In particular, see elsi_set_dimensionality and elsi_set_energy_gap in 3.5. Simply put, the solver selection favors ELPA for small-and-medium-sized problems, PEXSI for large, sparse, low-dimensional problems, and NTPoly for extra-large, sparse systems with a decent energy gap.

2) matrix_format: BLACS_DENSE(0) refers to 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. GENERIC_COO(3) refers to a coordinate (COO) sparse matrix format in an arbitrary distribution. Please refer to 3.2.3 for specifications of these matrix formats.

3) n_state: If ELPA, EigenExa, or SLEPc-SIPs is the chosen solver, this parameter specifies the number of eigenstates to solve. EigenExa internally computes all the eigenstates unless n_state is 0. When n_state is larger than 0 and smaller than n_basis, ELSI simply discards the unwanted solutions. libOMM, PEXSI and NTPoly do not make use of this parameter, thus a dummy value may be passed.

4) parallel_mode: The two allowed values of parallel_mode, 0 (SINGLE_PROC) and 1 (MULTI_PROC), allow for three parallelization strategies commonly employed by electronic structure 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. Every MPI task independently handles a group of $\boldsymbol{k}$-points uniquely assigned to it.

- Example: 16 $\boldsymbol{k}$-points, 4 MPI tasks.
- MPI task 0 handles $\boldsymbol{k}$-points 1, 2, 3, 4 sequentially;
- MPI task 1 handles $\boldsymbol{k}$-points 5, 6, 7, 8 sequentially;
- MPI task 2 handles $\boldsymbol{k}$-points 9, 10, 11, 12 sequentially;
- MPI task 3 handles $\boldsymbol{k}$-points 13, 14, 15, 16 sequentially.

```
call elsi_init (eh, ..., parallel_mode=0, ...)
...
do i_kpt = 1, n_kpt_local
   call elsi_ev_{real|complex} (eh, ham_this_kpt, ovlp_this_kpt, eval_this_kpt, evec_this_kpt)
end do
```

3b) MULTI_PROC: Solves the KS eigenproblem following a ScaLAPACK-like fashion. This allows the usage of the following parallelization strategy:

Groups of MPI tasks coordinate to handle the same $\boldsymbol{k}$-point, uniquely assigned to that group.

- Example: 4 $\boldsymbol{k}$-points, 16 MPI tasks.
- MPI tasks 0, 1, 2, 3 cooperatively handle $\boldsymbol{k}$-point 1;
- MPI tasks 4, 5, 6, 7 cooperatively handle $\boldsymbol{k}$-point 2;
- MPI tasks 8, 9, 10, 11 cooperatively handle $\boldsymbol{k}$-point 3;
- MPI tasks 12, 13, 14, 15 cooperatively handle $\boldsymbol{k}$-point 4.

```
call elsi_init (eh, ..., parallel_mode=1, ...)
call elsi_set_mpi (eh, my_mpi_comm)
call elsi_set_kpoint (eh, n_kpt, my_kpt, my_weight)
call elsi_set_mpi_global (eh, mpi_comm_global)
...
call elsi_{ev|dm}_{real|complex} (eh, my_ham, my_ovlp, ...)
```

Please note that when there is more than one $\boldsymbol{k}$-point, a global MPI communicator must be provided for inter-$\boldsymbol{k}$-point communications. See 3.2.4 for elsi_set_kpoint, elsi_set_spin, and elsi_set_mpi_global, which are used to set up a calculation with two spin channels and/or multiple $\boldsymbol{k}$-points.

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 $\boldsymbol{k}$-point and/or spin channel, this communicator will be used only for solving one problem corresponding to one $\boldsymbol{k}$-point and one spin channel. See 3.2.4 for details.

elsi_set_mpi(handle, mpi_comm)

Argument	Data Type	in/out	Explanation
handle	type(elsi_handle)	inout	Handle to ELSI.
mpi_comm	integer	in	MPI communicator.

3.2.3 Setting Up Matrix Formats

Four matrix formats are currently supported by ELSI, namely 2D block-cyclic distributed dense matrix format (BLACS_DENSE), 1D block distributed compressed sparse column format (PEXSI_CSC), 1D block-cyclic distributed compressed sparse column format, (SIESTA_CSC), arbitrarily distributed coordinate sparse format (GENERIC_COO).

When using the BLACS_DENSE format, 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 the BLACS_DENSE format.

elsi_set_blacs(handle, blacs_ctxt, block_size)

Argument	Data Type	in/out	Explanation
handle	type(elsi_handle)	inout	Handle to ELSI.
blacs_ctxt	integer	in	BLACS context.
block_size	integer	in	Block size of the 2D block-cyclic distribution, specifying both row and column directions.

When using the PEXSI_CSC or SIESTA_CSC format, 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 CSC sparse matrix formats.

elsi_set_csc(handle, global_nnz, local_nnz, local_col, row_idx, col_ptr)

Argument	Data Type	in/out	Explanation
handle	type(elsi_handle)	inout	Handle to ELSI.
global_nnz	integer	in	Global number of non-zeros.
local_nnz	integer	in	Local number of non-zeros.
local_col	integer	in	Local number of matrix columns.
row_idx	integer, rank-1 array	in	Local row index array. Dimension: local_nnz.
col_ptr	integer, rank-1 array	in	Local column pointer array. Dimension: local_col+1.

The block size of the PEXSI_CSC format cannot be set by the user. This is because the PEXSI solver requires that the block size must be floor(N_basis/N_procs), where floor(x) is the greatest integer less than or equal to x, N_basis and N_procs are the number of basis functions and the number of MPI tasks, respectively. The block size of the SIESTA_CSC must be explicitly set by calling elsi_set_csc_blk.

elsi_set_csc_blk(handle, block_size)

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

In most cases, input and output matrices should be distributed across all MPI tasks. The only exception is when using the PEXSI solver, one of the sparse density matrix interfaces (elsi_dm_real_sparse or elsi_dm_complex_sparse), and the PEXSI_CSC matrix format. In this case, an additional parameter, pexsi_np_per_pole, must be set by the user. Input and output matrices should be 1D-block-distributed among the first pexsi_np_per_pole MPI tasks (not all the MPI tasks). Please also read the 2nd remark in 3.5.4 for more information.

When using the GENERIC_COO format, the sparsity pattern should be passed into ELSI via the elsi_set_coo subroutine. It is necessary to call this subroutine before calling any solver interface that makes use of the COO sparse matrix format.

elsi_set_coo(handle, global_nnz, local_nnz, row_idx, col_idx)

Argument	Data Type	in/out	Explanation
handle	type(elsi_handle)	inout	Handle to ELSI.
global_nnz	integer	in	Global number of non-zeros.
local_nnz	integer	in	Local number of non-zeros.
row_idx	integer, rank-1 array	in	Local row index array. Dimension: local_nnz.
col_idx	integer, rank-1 array	in	Local column index array. Dimension: local_nnz.

The distribution of matrix elements in the GENERIC_COO format is arbitrary. Both sorted and unsorted inputs are supported.

3.2.4 Setting Up Multiple $\boldsymbol{k}$-points and/or Spin Channels

When there is more than one $\boldsymbol{k}$-point and/or spin channel in the simulating system, the ELSI interface can be set up to support parallel calculation of the $\boldsymbol{k}$-points and/or spin channels. The base case is a system isolated in space, e.g. free atoms, molecules, clusters, without spin-polarization. In this case, there is one eigenproblem in each iteration of an SCF cycle. When a spin-polarized periodic system is considered, 1.3 should have an index α denoting the spin channel, and an index $\boldsymbol{k}$ denoting points in reciprocal space:

$$\boldsymbol{H}_{\boldsymbol{k}}^{\alpha}\boldsymbol{C}_{\boldsymbol{k}}^{\alpha} = \boldsymbol{S}_{\boldsymbol{k}}\boldsymbol{C}_{\boldsymbol{k}}^{\alpha}\boldsymbol{\epsilon}_{\boldsymbol{k}}^{\alpha}. \tag{3.1}$$

In total, there are $N_{\mathrm{kpt}} \times N_{\mathrm{spin}}$ eigenproblems to solve. They can be solved in an embarrassingly parallel fashion. In ELSI, eigenproblems in 3.1 are considered as equivalent "unit tasks". The available computer processes are divided into $N_{\mathrm{kpt}} \times N_{\mathrm{spin}}$ groups, each of which is responsible for one unit task.

To set up the ELSI interface for a calculation with more than one $\boldsymbol{k}$-point and/or more than one spin channel, the elsi_set_kpoint and/or elsi_set_spin subroutines are called to pass the required information into ELSI. The MPI communicator for each unit task is passed into ELSI by calling elsi_set_mpi. In addition, a global MPI communicator for all tasks is passed into ELSI by calling elsi_set_mpi_global. Note that the current ELSI interface only supports the case where the eigenproblems for all the $\boldsymbol{k}$-points and spin channels are fully parallelized, i.e., there is no MPI task handling more than one $\boldsymbol{k}$-point and/or more than one spin channel. In ELSI, the two spin channels are always coupled by a uniform chemical potential. The distribution of electrons among the two channels, and thus the net spin moment of the system, cannot be specified. Calculations with a fixed, user-specified spin moment can be performed by initializing two independent ELSI instances for the two spin channels.

In this version of ELSI, the SLEPc-SIPs eigensolver is not supported in spin-polarized and/or periodic calculations.

elsi_set_kpoint(handle, n_kpt, i_kpt, weight)

Argument	Data Type	in/out	Explanation
handle	type(elsi_handle)	inout	Handle to ELSI.
n_kpt	integer	in	Total number of $\boldsymbol{k}$-points.
i_kpt	integer	in	Index of the $\boldsymbol{k}$-point handled by this MPI task.
weight	integer	in	Weight of the $\boldsymbol{k}$-point handled by this MPI task.

elsi_set_spin(handle, n_spin, i_spin)

Argument	Data Type	in/out	Explanation
handle	type(elsi_handle)	inout	Handle to ELSI.
n_spin	integer	in	Total number of spin channels.
i_spin	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
handle	type(elsi_handle)	inout	Handle to ELSI.
mpi_comm_global	integer	in	Global MPI communicator used for communications among all $\boldsymbol{k}$-points and spin channels.

3.2.5 Reinitializaing ELSI

When a geometry update takes place in geometry optimization or molecular dynamics calculations, the overlap matrix changes due to the movement of localized basis functions. Calling **elsi_reinit** instructs ELSI to flush geometry-related variables and arrays that cannot be used in the new geometry step, e.g., the overlap matrix and its sparsity pattern. Other runtime parameters are kept within the ELSI instance and reused throughout multiple geometry steps. Note that the chemical potential determination in PEXSI must be restarted for every new geometry. See the 5^{th} remark in 3.5.4 for details.

elsi_reinit(handle)

3.2.6 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
handle	type(elsi_handle)	inout	Handle to ELSI.

3.3 Solving Eigenvalues and Eigenvectors

The following subroutines return all the eigenvalues and a subset of eigenvectors of the provided generalized eigenproblem defined by H and S matrices. For standard eigenproblems, please see elsi_set_unit_ovlp in 3.5.1. ELPA, EigenExa, or SLEPc-SIPs may be selected as the solver when using these subroutines.

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

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

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

Argument	Data Type	in/out	Explanation
handle	type(elsi_handle)	inout	Handle to ELSI.
ham	complex double, rank-2 array	inout	Complex Hamiltonian matrix in 2D block-cyclic dense format. See remark 1.
ovlp	complex double, rank-2 array	inout	Complex overlap matrix (or its Cholesky factorization) in 2D block-cyclic dense format. See remark 1.
eval	real double, rank-1 array	inout	Eigenvalues. See remark 2.
evec	complex double, rank-2 array	out	Complex eigenvectors in 2D block-cyclic dense format. See remark 3.

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

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

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

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

Remarks

1) The Hamiltonian matrix will be destroyed by ELPA and EigenExa during computation. When using elsi_ev_real or elsi_ev_complex, the overlap matrix will be overridden by its Cholesky factorization. When using elsi_ev_real_sparse or elsi_ev_complex_sparse, the Cholesky factorization (not sparse) is stored internally in the BLACS_DENSE format.

2) The dimension of eval should always be n_basis, regardless of the choice of n_state specified in elsi_init.

3) 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 × n_basis, whose extra part is used as work space. When using elsi_ev_real_sparse and elsi_ev_complex_sparse, the eigenvectors are returned in a dense format (BLACS_DENSE), as they are in general not 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 band structure energy.

elsi_dm_real(handle, ham, ovlp, dm, bs_energy)

Argument	Data Type	in/out	Explanation
handle	type(elsi_handle)	inout	Handle to ELSI.
ham	real double, rank-2 array	inout	Real Hamiltonian matrix in 2D block-cyclic dense format.
ovlp	real double, rank-2 array	inout	Real overlap matrix (or Cholesky factorization) in 2D block-cyclic dense format. See remark 1.
dm	real double, rank-2 array	out	Real density matrix in 2D block-cyclic dense format.
energy	real double	out	Band structure energy.

elsi_dm_complex(handle, ham, ovlp, dm, energy)

Argument	Data Type	in/out	Explanation
handle	type(elsi_handle)	inout	Handle to ELSI.
ham	complex double, rank-2 array	inout	Complex Hamiltonian matrix in 2D block-cyclic dense format.
ovlp	complex double, rank-2 array	inout	Complex overlap matrix (or its Cholesky factorization) in 2D block-cyclic dense format. See remark 1.
dm	complex double, rank-2 array	out	Complex density matrix in 2D block-cyclic dense format.
energy	real double	out	Band structure energy.

elsi_dm_real_sparse(handle, ham, ovlp, dm, energy)

Argument	Data Type	in/out	Explanation
handle	type(elsi_handle)	inout	Handle to ELSI.
ham	real double, rank-1 array	inout	Non-zero values of the real Hamiltonian matrix in 1D block CSC, 1D block-cyclic CSC, or generic COO sparse format.
ovlp	real double, rank-1 array	inout	Non-zero values of the real overlap matrix in 1D block CSC, 1D block-cyclic CSC, or generic COO sparse format.
dm	real double, rank-1 array	out	Non-zero values of the real density matrix in 1D block CSC, 1D block-cyclic CSC, or generic COO sparse format.
energy	real double	out	Band structure energy.

elsi_dm_complex_sparse(handle, ham, ovlp, dm, energy)

Argument	Data Type	in/out	Explanation
handle	type(elsi_handle)	inout	Handle to ELSI.
ham	complex double, rank-1 array	inout	Non-zero values of the complex Hamiltonian matrix in 1D block CSC, 1D block-cyclic CSC, or generic COO sparse format.
ovlp	complex double, rank-1 array	inout	Non-zero values of the complex overlap matrix in 1D block CSC, 1D block-cyclic CSC, or generic COO sparse format.
dm	complex double, rank-1 array	out	Non-zero values of the complex density matrix in 1D block CSC, 1D block-cyclic CSC, or generic COO sparse format.
energy	real double	out	Band structure energy.

Remarks

1) When using elsi_dm_real or elsi_dm_complex with ELPA, libOMM, or EigenExa, the Hamiltonian matrix will be destroyed during the computation. The overlap matrix will be used to store its Cholesky factorization, which will be reused until the overlap matrix changes.

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. Parameters that can be overridden are 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 ELSI API. Integers are used to represent logical, with 0 being false and any positive integer being true.

elsi_set_output(handle, output_level)

elsi_set_output_unit(handle, output_unit)

elsi_set_output_log(handle, output_log)

elsi_set_save_ovlp(handle, save_ovlp)

elsi_set_unit_ovlp(handle, unit_ovlp)

elsi_set_zero_def(handle, zero_def)

elsi_set_illcond_check(handle, illcond_check)

elsi_set_illcond_tol(handle, illcond_tol)

elsi_set_spin_degeneracy(handle, spin_degeneracy)

elsi_set_energy_gap(handle, energy_gap)

elsi_set_spectrum_width(handle, spectrum_width)

elsi_set_dimensionality(handle, dimensionality)

elsi_set_mu_broaden_scheme(handle, mu_broaden_scheme)

elsi_set_mu_broaden_width(handle, mu_broaden_width)

elsi_set_mu_tol(handle, mu_tol)

elsi_set_mu_mp_order(handle, mu_mp_order)

elsi_set_write_unit(handle, write_unit)

elsi_set_sing_check(handle, sing_check)

elsi_set_sing_tol(handle, sing_tol)

Argument	Data Type	Default	Explanation
output_level	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.
output_unit	integer	6	Unit used in ELSI to write out information.
output_log	integer	0	If not 0, a separate log file in JSON format will be written out.
save_ovlp	integer	0	If not 0, the overlap matrix will be saved for extrapolation of density matrix or eigenvectors to a new geometry.
unit_ovlp	integer	0	If not 0, the overlap matrix will be treated as an identity (unit) matrix in ELSI and the solvers. See remark 1.
zero_def	real double	10^{-15}	When converting a matrix from dense to sparse format, values below this threshold will be discarded.
illcond_check	integer	0	If not 0, the eigenvalues of the overlap matrix will be calculated in order to check if it is ill-conditioned. See remark 2.
illcond_tol	real double	10^{-5}	Eigenfunctions of the overlap matrix with eigenvalues smaller than this threshold will be removed to avoid ill-conditioning. See remark 2.
spin_degeneracy	real double	2.0/n_spin	Spin degeneracy that controls the maximum number of electrons on a state.
energy_gap	real double	0	Energy gap. See remark 3.
spectrum_width	real double	10^3	Width of the eigenspectrum. See remark 3.
dimensionality	integer	3	Dimensionality (1, 2, or 3) of the simulating system. Only used for automatic solver selection.
mu_broaden_scheme	integer	0	Broadening scheme employed to compute the occupation numbers and the Fermi level. 0: Gaussian. 1: Fermi-Dirac. 2: Methfessel-Paxton. 4: Marzari-Vanderbilt.
mu_broaden_width	real double	0.01	Broadening width employed to compute the occupation numbers and the Fermi level. See remark 4.
mu_tol	real double	10^{-13}	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.
mu_mp_order	integer	0	Order of the Methfessel-Paxton broadening scheme. No effect if Methfessel-Paxton is not the chosen broadening scheme.
write_unit	integer	6	Deprecated. Use elsi_set_output_unit instead.
sing_check	integer	0	Deprecated. Use elsi_set_illcond_check instead.
sing_tol	real double	10^{-5}	Deprecated. Use elsi_set_illcond_tol instead.

Remarks

1) If the overlap matrix is set to be an identity matrix, all settings related to the singularity (ill-conditioning) check take no effect. The ovlp argument passed into elsi_ev_real, elsi_ev_complex, elsi_ev_real_sparse, elsi_ev_complex_sparse, elsi_dm_real, elsi_dm_complex, elsi_dm_real_sparse, and elsi_dm_complex_sparse will not be referenced.

2) If the ill-conditioning check is not disabled, in the first iteration of each SCF cycle, all eigenvalues of the overlap matrix is computed. If there is any eigenvalue smaller than illcond_tol, the matrix is considered to be ill-conditioned.

3) spectrum_width and energy_gap refer to the width and the gap of the eigenspectrum. Simply use the default values if there is no better estimate.

4) In all supported broadening schemes, there is a term $(\epsilon - E_\mathrm{F})/W$ in the distribution function, where ϵ is the energy of an eigenstate, and E_F is the Fermi level. The broadening_width parameter should be W in the same unit of ϵ and E_F.

3.5.2 Customizing the ELPA Solver

elsi_set_elpa_solver(handle, elpa_solver)

elsi_set_elpa_n_single(handle, elpa_n_single)

elsi_set_elpa_gpu(handle, elpa_gpu)

elsi_set_elpa_gpu_kernels(handle, elpa_gpu_kernels)

elsi_set_elpa_autotune(handle, elpa_autotune)

Argument	Data Type	Default	Explanation
elpa_solver	integer	2	1: ELPA 1-stage solver. 2: ELPA 2-stage solver. The latter is usually faster and more scalable.
elpa_n_single	integer	0	Number of SCF steps using single precision ELPA to solve standard eigenproblems. See remark 1.
elpa_gpu	integer	0	If not 0, try to enable GPU-acceleration in ELPA. See remark 2.
elpa_gpu_kernels	integer	0	If not 0, try to enable GPU-acceleration and GPU kernels in ELPA. See remark 2.
elpa_autotune	integer	1	If not 0, try to enable auto-tuning of runtime parameters in ELPA. See remark 3.

Remarks

1) elpa_n_single: If single precision arithmetic is available in an externally complied ELPA library, it may be enabled by setting elpa_n_single to a positive integer, then the standard eigenprolems in the first elpa_n_single SCF steps will be solved with single precision. The transformations between generalized eigenproblem and the standard form are always performed with double precision. Although this keyword accelerates the solution of standard eigenproblems, the overall SCF convergence may be slower, depending on the physical system and the SCF settings used in the electronic structure code. This keyword is ignored if single precision calculations are not available, which is the case if the internal version of ELPA is used, or if an external ELPA has not been complied with single precision support.

2) elpa_gpu and elpa_gpu_kernels: If GPU-acceleration is available in an externally compiled ELPA library, it may be enabled by setting elpa_gpu to a non-zero integer. Setting elpa_gpu itself does not enable the GPU kernels for eigenvector back-transformation. To enable the GPU kernels, use elpa_gpu_kernels. These two keywords are ignored if GPU-acceleration is not available, which is the case if the internal version of ELPA is used, or if an external ELPA has not been complied with GPU support.

3) elpa_autotune: If auto-tuning of runtime parameters is available in an externally complied ELPA library, it may be enabled by setting elpa_autotune to a nonzero integer. This keyword is ignored if auto-tuning is not available, which is the case if the internal version of ELPA is used.

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
omm_flavor	integer	0	Method to perform OMM minimization. See remark 1.
omm_n_elpa	integer	6	Number of SCF steps using ELPA. See remark 2.
omm_tol	real double	10^{-12}	Convergence tolerance of orbital minimization. See remark 3.

Remarks

1) omm_flavor: Allowed choices are 0 for a 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 (Cholesky) 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 (basic) is probably a better choice to finish the remaining SCF cycle. See also remark 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_temp(handle, pexsi_temp)

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)

elsi_set_pexsi_gap(handle, pexsi_gap)

elsi_set_pexsi_delta_e(handle, pexsi_delta_e)

Argument	Data Type	Default	Explanation
pexsi_n_pole	integer	20	Number of poles used by PEXSI. See remark 1.
pexsi_n_mu	integer	2	Number of mu points used by PEXSI. See remark 1.
pexsi_np_per_pole	integer	-	Number of MPI tasks assigned to each mu point. See remark 2.
pexsi_np_symbo	integer	1	Number of MPI tasks for symbolic factorization. See remark 3.
pexsi_temp	real double	0.002	Temperature. See remark 4.
pexsi_mu_min	real double	-10.0	Minimum value of mu. See remark 5.
pexsi_mu_max	real double	10.0	Maximum value of mu. See remark 5.
pexsi_inertia_tol	real double	0.05	Stopping criterion of inertia counting. See remark 5.
pexsi_gap	real double	0.0	Deprecated. Use elsi_set_energy_gap instead. See remark 6.
pexsi_delta_e	real double	10.0	Deprecated. Use elsi_set_spectrum_width instead.

Remarks

1) In PEXSI, 20 poles are usually sufficient to get an accuracy that is comparable with the result obtained from diagonalization. The chemical potential is determined by performing Fermi operator expansion at several chemical potential values (referred to as "points" by PEXSI developers) in an SCF step, then interpolating the results at all points to the final answer. The pexsi_n_mu parameter controls the number of chemical potential "points" to be evaluated. 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 parallelism: 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 × pexsi_n_mu × pexsi_n_pole, i.e., all the three levels of parallelism are fully exploited. In case that this is not feasible, PEXSI can also process the poles in serial, whereas all the chemical potential points must be evaluated simultaneously. The user should make sure that the total number of MPI tasks is divisible by the product of the number of MPI tasks per pole and the number of points. The code will stop if this requirement is not fulfilled.

When using the BLACS_DENSE or SIESTA_CSC matrix formats, pexsi_np_per_pole is automatically determined to balance the three levels of parallelism in PEXSI. Input and output matrices should be distributed across all MPI tasks in either a 2D block-cyclic distribution (BLACS_DENSE) or a 1D block-cyclic distribution (SIESTA_CSC).

Please note that when using the PEXSI_CSC matrix format together with the PEXSI solver, input and output matrices should be distributed among the first pexsi_np_per_pole MPI tasks (not all the MPI tasks) in a 1D block distribution. The block size of the distribution must be floor(N_basis/N_procs), where floor(x) is the greatest integer less than or equal to x, N_basis and N_procs are the number of basis functions and the number of MPI tasks, respectively.

when using the PEXSI_CSC matrix format with the ELPA, libOMM, EigenExa, SLEPc-SIPs, or NTPoly solver, input and output matrices should be distributed across all the MPI tasks in a 1D block distribution. Again, the block size of the distribution must be floor(N_basis/N_procs).

3) pexsi_np_symbo: Unless there is a memory bottleneck, using 1 MPI task for matrix reordering and symbolic factorization is favorable. When running in serial, the matrix reordering in PT-SCOTCH or ParMETIS introduces a minimal number of "fill-ins" to the factorized matrices. Using more MPI tasks introduces more fill-ins. As the matrix reordering and symbolic factorization are performed only once per SCF cycle (with a fixed overlap matrix), using 1 MPI task should not affect the overall timing too much. On the other hand, more fill-ins lead to slower numerical factorization in every SCF step. In addition, the number of MPI tasks used for matrix reordering and symbolic factorization cannot be too large. Otherwise, the symbolic factorization may fail. Therefore, the default number of MPI tasks for symbolic factorization is 1. It is worth testing and increasing this number for large-scale calculations.

4) pexsi_temp: This value corresponds to the $1/k_{\mathrm{B}}T$ term (not T) in the Fermi-Dirac distribution function.

```
do geometry update
  mu_min = -10.0
  mu_max = 10.0
  delta_V_min = 0.0
  delta_V_max = 0.0

  do SCF cycle
    Update Hamiltonian

    call elsi_set_pexsi_mu_min (eh, mu_min + delta_V_min)
    call elsi_set_pexsi_mu_max (eh, mu_max + delta_V_max)

    call elsi_dm_{real|complex} (eh, ham, ovlp, dm, bs_energy)

    call elsi_get_pexsi_mu_min (eh, mu_min)
    call elsi_get_pexsi_mu_max (eh, 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 do
end do
```

5) The chemical potential determination in PEXSI relies on inertia counting to narrow down the chemical potential searching interval in the first few SCF steps. The pexsi_inertia_tol parameter 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 of each geometry step, 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 $(\mu_{\mathrm{min}}^{\mathrm{n\text{-}1}} + \Delta V_{\mathrm{min}})$, pexsi_mu_max should be set to $(\mu_{\mathrm{max}}^{\mathrm{n\text{-}1}} + \Delta V_{\mathrm{max}})$. Here, $\mu_{\mathrm{min}}^{\mathrm{n\text{-}1}}$ and $\mu_{\mathrm{max}}^{\mathrm{n\text{-}1}}$ are the lower bound and the upper bound of the chemical potential that are determined by PEXSI in the $(\mathrm{n}\text{-}1)^{\mathrm{th}}$ SCF step. They can be retrieved by calling elsi_get_pexsi_mu_min and elsi_get_pexsi_mu_max, respectively (see 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, ΔV denotes the potential change, and $\Delta V_{\min}$ and $\Delta V_{\max}$ are the minimum and maximum values in the array ΔV, respectively. The whole process is summarized in the pseudo-code above. The (re-)initialization and finalization of ELSI are omitted.

6) pexsi_gap: The PEXSI method does not require an energy gap. If no knowledge is available, the default value usually works.

3.5.5 Customizing the EigenExa Solver

elsi_set_eigenexa_method(handle, eigenexa_method)

Argument	Data Type	Default	Explanation
eigenexa_method	integer	2	1: Tridiagonalization solver eigen_s. 2: Pentadiagonalization solver eigen_sx. The latter is usually faster and more scalable.

3.5.6 Customizing the SLEPc-SIPs Solver

elsi_set_sips_ev_min(handle, ev_min)

elsi_set_sips_ev_max(handle, ev_max)

elsi_set_sips_n_elpa(handle, sips_n_elpa)

elsi_set_sips_n_slice(handle, sips_n_slice)

elsi_set_sips_interval(handle, sips_lower, sips_upper)

Argument	Data Type	Default	Explanation
ev_min	real double	-2.0	Lower bound of eigenspectrum. See remark 1.
ev_max	real double	2.0	Upper bound of eigenspectrum. See remark 1.
sips_n_elpa	integer	0	Number of SCF steps using ELPA. See remark 2.
sips_n_slice	integer	1	Number of slices. See remark 3.
sips_lower	real double	-2.0	Deprecated. Use elsi_set_sips_ev_min instead.
sips_upper	real double	2.0	Deprecated. Use elsi_set_sips_ev_max instead.

Remarks

1) ev_min and ev_max: SLEPc-SIPs relies on some inertia counting steps to estimate the lower and upper bounds of the spectrum. Only eigenvalues within this interval, and their associated eigenvectors, will be solved. The inertia-counting-based eigenvalue searching starts from the interval determined by ev_min and ev_max. Depending on the results of inertia counting, this interval may expand or shrink to make sure that the 1^{st} to the $\text{n\_state}^{\text{th}}$ eigenvalues are all within this interval. If a good estimate of the lower or upper bounds of the eigenspectrum is available, it should be set by elsi_set_sips_ev_min or elsi_set_sips_ev_max.

2) sips_n_elpa: The performance of SLEPc-SIPs mainly depends on the load balance across slices. Optimal performance is expected if the desired eigenvalues are evenly distributed across slices. In an SCF calculation, eigenvalues obtained in the current SCF step can be used as an approximated distribution of eigenvalues in the next SCF step. This approximation should become better as the SCF cycle approaches its convergence. On the other hand, at the beginning of an SCF cycle, the load balance is only coarsely checked by inertia calculations. Using the direct eigensolver ELPA in the first sips_n_elpa SCF steps can circumvent the load imbalance of spectrum slicing in the initial SCF steps.

3) sips_n_slice: SLEPc-SIPs partitions the eigenspectrum into slices and solves the slices in parallel. The sips_n_slice parameter controls the number of slices to use in SLEPc-SIPs. The default value, 1, should always work, but by no means leads to the optimal performance of the solver. There are some general rules to set this parameter. Firstly, as a requirement of the SLEPc library, the total number of MPI tasks must by divisible by sips_n_slice. Secondly, setting sips_n_slice to be equal to the number of computing nodes (not MPI tasks) usually yields better performance, as the communication between nodes is minimized in this case. The optimal value of sips_n_slice depends on the actual problem as well as the computing hardware.

3.5.7 Customizing the NTPoly Solver

elsi_set_ntpoly_method(handle, ntpoly_method)

elsi_set_ntpoly_filter(handle, ntpoly_filter)

elsi_set_ntpoly_tol(handle, ntpoly_tol)

Argument	Data Type	Default	Explanation
ntpoly_method	integer	2	Method to perform density matrix purification. See remark 1.
ntpoly_filter	real double	10^{-15}	When performing sparse matrix multiplications, values below this filter will be discarded. See remark 2.
ntpoly_tol	real double	10^{-8}	Convergence tolerance of purification. See remark 2.

Remarks

1) ntpoly_method: Allowed choices are 0 for the canonical purification, 1 for the trace correcting purification, 2 for the 4th order trace resetting purification, and 3 for the generalized hole-particle canonical purification.

2) ntpoly_filter and ntpoly_tol control the accuracy and computational cost of the density matrix purification methods. Tight choices of ntpoly_filter and ntpoly_tol, e.g. the default values here, lead to highly accurate results that are comparable to the results obtained from diagonalization. However, linear scaling can only be achieved with a relatively large ntpoly_filter such as 10^{-6}. Correspondingly, ntpoly_tol may be set to 10^{-3}. Note that the purification may not converge if ntpoly_filter is too large relative to ntpoly_tol. Setting ntpoly_filter to be $\leq 10^{-3}\times$ ntpoly_tol is safe in most cases.

3.6 Getting Additional Results from ELSI

In 3.3 and 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 only be useful at a certain stage of an SCF calculation. One example is the energy-weighted density matrix that is employed to evaluate the Pulay forces during a geometry optimization calculation. The subroutines introduced in the following subsections are used to retrieve such additional results from ELSI.

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_initialized(handle, handle_init)

elsi_get_version(handle, major, minor, patch)

elsi_get_datestamp(handle, date_stamp)

elsi_get_n_illcond(handle, n_illcond)

elsi_get_ovlp_ev_min(handle, ev_min)

elsi_get_ovlp_ev_max(handle, ev_max)

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)

elsi_get_n_sing(handle, n_sing)

Argument	Data Type	Explanation
handle_init	integer	0 if the ELSI handle has not been initialized; 1 if initialized.
major	integer	Major version number.
minor	integer	Minor version number.
patch	integer	Patch level.
date_stamp	integer	Date stamp of ELSI (yyyymmdd).
n_illcond	integer	Number of eigenvalues of the overlap matrix that are smaller than the ill-conditioning tolerance. See 3.5.1.
ev_min	real double	Lowest eigenvalue of the overlap matrix. See remark 1.
ev_max	real double	Highest eigenvalue of the overlap matrix. See remark 1.
mu	real double	Chemical potential. See remark 2.
ts	real double	Entropy. See remark 2.
edm_real	real double, rank-2 array	Real energy-weighted density matrix in 2D block-cyclic dense format. See remark 3.
edm_complex	complex double, rank-2 array	Complex energy-weighted density matrix in 2D block-cyclic dense format. See remark 3.
edm_real_sparse	real double, rank-1 array	Non-zero values of the real density matrix in 1D block CSC format. See remark 3.
edm_complex_sparse	complex double, rank-1 array	Non-zero values of the complex density matrix in 1D block CSC format. See remark 3.
n_sing	integer	Deprecated. Use elsi_get_n_illcond instead.

Remarks

1) In ELSI, ill-conditioning check of the overlap matrix is enabled by default when ELPA is the chosen solver. It may be disabled by calling elsi_set_illcond_check, and is automatically disabled when the chosen solver is not ELPA. ev_min and ev_max are computed only if ill-conditioning check is enabled. Otherwise the return value may be zero.

2) In ELSI, the chemical potential will only be available if one of the density matrix solver interfaces has been called, with ELPA, PEXSI, EigenExa, or NTPoly 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 or EigenExa being the chosen solver. The user should avoid calling the subroutine when the chemical potential or the entropy is not ready.

3) 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. These subroutines 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
pexsi_mu_min	real double	Minimum value of mu. See remark 1.
pexsi_mu_max	real double	Maximum value of mu. See remark 1.

Remarks

1) Please refer to the 5^{th} remark in 3.5.4 for the chemical potential determination algorithm in PEXSI and ELSI.

3.6.3 Extrapolation of Wavefunctions and Density Matrices

In a single point total energy calculation, a simple way to construct an initial guess of the electron density is using the superposition of free atom densities. In geometry calculations, the initial guess in the $(n+1)^{th}$ geometry step can be made better than free atom superposition, by reusing the wavefunctions or density matrix calculated in the n^{th} geometry step. However, due to the movement of atoms and localized basis functions around them, wavefunctions obtained in the n^{th} geometry step are no longer orthonormalized in the $(n+1)^{th}$ geometry step. Similarly, density matrix from the n^{th} geometry step does not satisfy the conditions in Eq. 1.21 with respect to the new overlap matrix.

The following subroutines orthonormalize eigenvectors (coefficients of wavefunctions) in the n^{th} geometry step with respect to the overlap matrix in the $(n+1)^{th}$ geometry step with a Gram-Schmidt algorithm.

elsi_orthonormalize_ev_real(handle, ovlp, evec)

Argument	Data Type	in/out	Explanation
handle	type(elsi_handle)	inout	Handle to ELSI.
ovlp	real double, rank-2 array	in	Real overlap matrix in 2D block-cyclic dense format.
evec	real double, rank-2 array	inout	Real eigenvectors in 2D block-cyclic dense format.

elsi_orthonormalize_ev_complex(handle, ovlp, evec)

Argument	Data Type	in/out	Explanation
handle	type(elsi_handle)	inout	Handle to ELSI.
ovlp	complex double, rank-2 array	in	Complex overlap matrix in 2D block-cyclic dense format.
evec	complex double, rank-2 array	inout	Complex eigenvectors in 2D block-cyclic dense format.

elsi_orthonormalize_ev_real_sparse(handle, ovlp, evec)

Argument	Data Type	in/out	Explanation
handle	type(elsi_handle)	inout	Handle to ELSI.
ovlp	real double, rank-1 array	in	Real overlap matrix in 1D block CSC, 1D block-cyclic CSC, or generic COO sparse format.
evec	real double, rank-2 array	inout	Real eigenvectors in 2D block-cyclic dense format. See remark 1.

elsi_orthonormalize_ev_complex_sparse(handle, ovlp, evec)

Argument	Data Type	in/out	Explanation
handle	type(elsi_handle)	inout	Handle to ELSI.
ovlp	complex double, rank-1 array	in	Complex overlap matrix in 1D block CSC, 1D block-cyclic CSC, or generic COO sparse format.
evec	complex double, rank-2 array	inout	Complex eigenvectors in 2D block-cyclic dense format. See remark 1.

Remarks

1) When using elsi_orthonormalize_ev_real_sparse and elsi_orthonormalize_ev_complex_sparse, the eigenvectors are stored in a dense format (BLACS_DENSE), as they are in general not sparse.

The following subroutines extrapolate density matrix in the n^{th} geometry step to satisfy conditions 1.21 with respect to the overlap matrix in the $(n+1)^{th}$ geometry step.

elsi_extrapolate_dm_real(handle, ovlp, dm)

Argument	Data Type	in/out	Explanation
handle	type(elsi_handle)	inout	Handle to ELSI.
ovlp	real double, rank-2 array	inout	Real overlap matrix in 2D block-cyclic dense format. See remark 1.
dm	real double, rank-2 array	inout	Real density matrix in 2D block-cyclic dense format. See remark 2.

elsi_extrapolate_dm_complex(handle, ovlp, dm)

Argument	Data Type	in/out	Explanation
handle	type(elsi_handle)	inout	Handle to ELSI.
ovlp	complex double, rank-2 array	inout	Complex overlap matrix in 2D block-cyclic dense format. See remark 1.
dm	complex double, rank-2 array	inout	Complex density matrix in 2D block-cyclic dense format. See remark 2.

elsi_extrapolate_dm_real_sparse(handle, ovlp, dm)

Argument	Data Type	in/out	Explanation
handle	type(elsi_handle)	inout	Handle to ELSI.
ovlp	real double, rank-1 array	inout	Real overlap matrix in 1D block CSC, 1D block-cyclic CSC, or generic COO sparse format. See remark 1.
dm	real double, rank-1 array	out	Real density matrix in 1D block CSC, 1D block-cyclic CSC, or generic COO sparse format. See remark 3.

elsi_extrapolate_dm_complex_sparse(handle, ovlp, dm)

Argument	Data Type	in/out	Explanation
handle	type(elsi_handle)	inout	Handle to ELSI.
ovlp	complex double, rank-1 array	inout	Complex overlap matrix in 1D block CSC, 1D block-cyclic CSC, or generic COO sparse format. See remark 1.
dm	complex double, rank-1 array	out	Complex density matrix in 1D block CSC, 1D block-cyclic CSC, or generic COO sparse format. See remark 3.

Remarks

1) ovlp: This should be the overlap matrix in the $(n+1)^{th}$ geometry step. elsi_set_save_ovlp must have been called to store the overlap matrix in the n^{th} geometry step internally. Depending on the chosen solver, ovlp may be overridden by its Cholesky factorization, which will be reused by subsequent calls to the solver interfaces.

2) dm: Input should be the density matrix in the n^{th} geometry step. Output is the extrapolated density matrix.

3) dm: With the sparse density matrix extrapolation interface, the density matrix in the n^{th} geometry step is stored internally. Output is the extrapolated density matrix.

3.7 Parallel Matrix I/O

To test the solvers in ELSI, it is convenient to use matrices generated from actual electronic structure calculations. There exist a number of libraries for high-performance parallel I/O that are particularly capable of reading and writing a large amount of data with hierarchical structures and complex metadata. However, the data structure in ELSI is simply arrays that represent matrices, with a few integers to define the dimension of the matrices. In order to circumvent the

development and performance overhead associated with a high level I/O library, the parallel I/O functionality defined in the MPI standard is directly used to read and write matrices in ELSI.

When ELSI runs in parallel with multiple MPI tasks, the matrices are distributed across tasks. The choice of writing the distributed matrices into N_{procs} separate files, where N_{procs} is the number of MPI tasks, is not promising due to the difficulty of managing and post-processing a large number of files. The implementation of matrix I/O in ELSI adopts collective MPI I/O routines to write data to (read data from) a single binary file, as if the data was gathered onto a single MPI task then written to one file (read from one file by one MPI task then scattered to all tasks). The optimal I/O performance, both with MPI I/O and in general, is often obtained by making large and contiguous requests to access the file system, rather than small, non-contiguous, or random requests. Therefore, before being written to file, matrices are always redistributed to a 1D block distribution. This guarantees that each MPI task writes a contiguous trunk of data to a contiguous piece of file. Similarly, matrices read from file are in a 1D block distribution, and can be redistributed automatically if needed. A matrix is always stored in the CSC format in an ELSI matrix file. A dense matrix is automatically converted to the CSC format before writing to file, and can be converted back after being read from file.

Next, we present the API for parallel matrix I/O.

3.7.1 Setting Up Matrix I/O

An `elsi_rw_handle` must be initialized via the elsi_init_rw subroutine before any other matrix I/O subroutine may be called. This `elsi_rw_handle` is subsequently passed to all other matrix I/O subroutine calls.

elsi_init_rw(handle, task, parallel_mode, n_basis, n_electron)

Argument	Data Type	in/out	Explanation
handle	type(elsi_rw_handle)	out	Handle to matrix I/O instance.
task	integer	in	Matrix I/O task to perform. Accepted values are: 0 (READ_MATRIX) and 1(WRITE_MATRIX).
parallel_mode	integer	in	Parallelization mode. The only accepted value is 1 (MULTI_PROC) for now.
n_electron	real double	in	Number of electrons. See remark 1.
n_basis	integer	in	Number of basis functions, i.e. global size of matrix.

Remarks

1) n_electron: Matrices written out with ELSI matrix I/O are usually from actual electronic structure calculations. Having the number of electrons available makes the matrix file useful for testing density matrix solvers such as PEXSI. Therefore, it is recommended to set the correct number of electrons when initializing an matrix I/O handle, although setting it to an arbitrary number will not affect the matrix I/O operation.

2) n_basis: This can be set to an arbitrary value if task is 0 (READ_MATRIX). Its value will be read from file when calling elsi_read_mat_dim or elsi_read_mat_dim_sparse.

The MPI communicator which encloses the MPI tasks to perform the matrix I/O operation needs to be passed into ELSI via the elsi_set_rw_mpi subroutine.

elsi_set_rw_mpi(handle, mpi_comm)

Argument	Data Type	in/out	Explanation
handle	type(elsi_rw_handle)	inout	Handle to matrix I/O instance.
mpi_comm	integer	in	MPI communicator.

When reading or writing a dense matrix, BLACS parameters are passed into ELSI via the elsi_set_rw_blacs subroutine.

elsi_set_rw_blacs(handle, blacs_ctxt, block_size)

Argument	Data Type	in/out	Explanation
handle	type(elsi_rw_handle)	inout	Handle to matrix I/O instance.
blacs_ctxt	integer	in	BLACS context.
block_size	integer	in	Block size of the 2D block-cyclic distribution, specifying both row and column directions.

When writing a sparse matrix, its dimensions are passed into ELSI via the elsi_set_rw_csc subroutine. The only sparse matrix format currently supported by ELSI matrix I/O is the PEXSI_CSC format. When reading a sparse matrix, there is no need to call this subroutine. The relevant parameters will be read from file when calling elsi_read_mat_dim or elsi_read_mat_dim_sparse.

elsi_set_rw_csc(handle, global_nnz, local_nnz, local_col)

Argument	Data Type	in/out	Explanation
handle	type(elsi_rw_handle)	inout	Handle to matrix I/O instance.
global_nnz	integer	in	Global number of non-zeros.
local_nnz	integer	in	Local number of non-zeros.
local_col	integer	in	Local number of matrix columns.

When a matrix I/O instance is no longer needed, its associated handle should be cleaned up by calling elsi_finalize_rw.

elsi_finalize_rw(handle)

Argument	Data Type	in/out	Explanation
handle	type(elsi_rw_handle)	inout	Handle to matrix I/O instance.

3.7.2 Writing Matrices

The following two subroutines write a dense matrix to file. Before writing a dense matrix, MPI and BLACS should be set up properly using elsi_set_rw_mpi and elsi_set_rw_blacs.

elsi_write_mat_real(handle, filename, mat)

Argument	Data Type	in/out	Explanation
handle	type(elsi_rw_handle)	in	Handle to matrix I/O instance.
filename	string	in	Name of file to write.
mat	real double, rank-2 array	in	Local matrix in 2D block-cyclic dense format.

elsi_write_mat_complex(handle, filename, mat)

Argument	Data Type	in/out	Explanation
handle	type(elsi_rw_handle)	in	Handle to matrix I/O instance.
filename	string	in	Name of file to write.
mat	complex double, rank-2 array	in	Local matrix in 2D block-cyclic dense format.

The following two subroutines write a sparse matrix to file. Before writing a sparse matrix, MPI and CSC matrix format should be set up properly using elsi_set_rw_mpi and elsi_set_rw_csc.

elsi_write_mat_real_sparse(handle, filename, row_idx, col_ptr, mat)

Argument	Data Type	in/out	Explanation
handle	type(elsi_rw_handle)	in	Handle to matrix I/O instance.
filename	string	in	Name of file to write.
row_idx	integer, rank-1 array	in	Local row index array.
col_ptr	integer, rank-1 array	in	Local column pointer array.
mat	real double, rank-1 array	in	Local non-zero values in 1D block CSC format.

elsi_write_mat_complex_sparse(handle, filename, row_idx, col_ptr, mat)

Argument	Data Type	in/out	Explanation
handle	type(elsi_rw_handle)	in	Handle to matrix I/O instance.
filename	string	in	Name of file to write.
row_idx	integer, rank-1 array	in	Local row index array.
col_ptr	integer, rank-1 array	in	Local column pointer array.
mat	complex double, rank-1 array	in	Local non-zero values in 1D block CSC format.

When writing a dense matrix to file, values smaller than a predefined threshold will be discarded. The default value of this threshold is 10^{-15}. It can be overridden via elsi_set_rw_zero_def.

elsi_set_rw_zero_def(handle, zero_def)

Argument	Data Type	in/out	Explanation
handle	type(elsi_rw_handle)	inout	Handle to matrix I/O instance.
zero_def	real double	in	When writing a dense matrix to file, values below this threshold will be discarded.

An array of eight user-defined integers can be optionally set up via elsi_set_rw_header. This array will be attached to the matrix file written out by the above subroutines. When reading a matrix file, this array may be retrieved via elsi_get_rw_header.

elsi_set_rw_header(handle, header)

Argument	Data Type	in/out	Explanation
handle	type(elsi_rw_handle)	inout	Handle to matrix I/O instance.
header	integer, rank-1 array	in	An array of eight integers.

3.7.3 Reading Matrices

The following subroutines read a dense or sparse matrix from file. While writing a matrix to file can be done in one step, it is easier to read a matrix from file in two steps, i.e., first read the dimension of the matrix and allocate memory accordingly, then read the actual data of the matrix.

The following three subroutines read a dense matrix from file. Before reading a dense matrix, MPI and BLACS should be set up properly using elsi_set_rw_mpi and elsi_set_rw_blacs. elsi_read_mat_dim is used to read the dimension of a matrix, including the number of electrons in the physical system (for testing purpose), the global size of the matrix, and the local size of the matrix. Memory needs to be allocated according to the return values of local_row and local_col. Then elsi_read_mat_real or elsi_read_mat_complex may be called to read a real or complex matrix, respectively.

elsi_read_mat_dim(handle, filename, n_electron, n_basis, local_row, local_col)

Argument	Data Type	in/out	Explanation
handle	type(elsi_rw_handle)	inout	Handle to matrix I/O instance.
filename	string	in	Name of file to read.
n_electron	real double	out	Number of electrons.
n_basis	integer	out	Number of basis functions, i.e. global size of matrix.
local_row	integer	out	Local number of matrix rows.
local_col	integer	out	Local number of matrix columns.

elsi_read_mat_real(handle, filename, mat)

Argument	Data Type	in/out	Explanation
handle	type(elsi_rw_handle)	inout	Handle to matrix I/O instance.
filename	string	in	Name of file to read.
mat	real double, rank-2 array	out	Local matrix in 2D block-cyclic distribution.

elsi_read_mat_complex(handle, filename, mat)

Argument	Data Type	in/out	Explanation
handle	type(elsi_rw_handle)	inout	Handle to matrix I/O instance.
filename	string	in	Name of file to read.
mat	complex double, rank-2 array	out	Local matrix in 2D block-cyclic distribution.

The following three subroutines read a sparse matrix from file. Before reading a sparse matrix, MPI should be set up properly using elsi_set_rw_mpi. elsi_read_mat_dim_sparse is used to read the dimension of a matrix, including the number of electrons in the physical system (for testing purpose), the global size of the matrix, and the local size of the matrix. Memory needs to be allocated according to the return values of local_nnz and local_col. Then elsi_read_mat_real_sparse or elsi_read_mat_complex_sparse may be called to read a real or complex matrix, respectively.

elsi_read_mat_dim_sparse(handle, filename, n_electron, n_basis, global_nnz, local_nnz, local_col)

Argument	Data Type	in/out	Explanation
handle	type(elsi_rw_handle)	inout	Handle to matrix I/O instance.
filename	string	in	Name of file to read.
n_electron	real double	out	Number of electrons.
n_basis	integer	out	Number of basis functions, i.e. global size of matrix.
global_nnz	integer	out	Global number of non-zeros.
local_nnz	integer	out	Local number of non-zeros.
local_col	integer	out	Local number of matrix columns.

elsi_read_mat_real_sparse(handle, filename, row_idx, col_ptr, mat)

Argument	Data Type	in/out	Explanation
handle	type(elsi_rw_handle)	inout	Handle to matrix I/O instance.
filename	string	in	Name of file to read.
row_idx	integer, rank-1 array	out	Local row index array.
col_ptr	integer, rank-1 array	out	Local column pointer array.
mat	real double, rank-1 array	out	Local non-zero values in 1D block CSC format.

elsi_read_mat_complex_sparse(handle, filename, row_idx, col_ptr, mat)

Argument	Data Type	in/out	Explanation
handle	type(elsi_rw_handle)	inout	Handle to matrix I/O instance.
filename	string	in	Name of file to read.
row_idx	integer, rank-1 array	out	Local row index array.
col_ptr	integer, rank-1 array	out	Local column pointer array.
mat	complex double, rank-1 array	out	Local non-zero values in 1D block CSC format.

An array of eight user-defined integers can be optionally set up via elsi_set_rw_header. This array will be attached to the matrix file written out by the above subroutines. When reading a matrix file, this array may be retrieved via elsi_get_rw_header.

elsi_get_rw_header(handle, header)

Argument	Data Type	in/out	Explanation
handle	type(elsi_rw_handle)	inout	Handle to matrix I/O instance.
header	integer, rank-1 array	out	An array of eight integers.

3.8 Example Pseudo-Code

Typical workflow of ELSI within an electronic structure code is demonstrated by the following pseudo-code. In the "test" directory of the ELSI package, there are also examples that showcase the usage of ELSI in C and Fortran.

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

```
SCF initialize

call elsi_init (eh, ELPA, MULTI_PROC, BLACS_DENSE, n_basis, n_electron, n_state)
call elsi_set_mpi (eh, mpi_comm)
call elsi_set_blacs (eh, blacs_ctxt, block_size)

do SCF cycle
  Update Hamiltonian

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

  Update electron density
  Check SCF convergence
end do

call elsi_finalize(eh)
```

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

```
SCF initialize

call elsi_init (eh, ELPA, MULTI_PROC, SIESTA_CSC, n_basis, n_electron, n_state)
call elsi_set_mpi (eh, mpi_comm)
call elsi_set_blacs (eh, blacs_ctxt, block_size)
call elsi_set_csc (eh, global_nnz, local_nnz, local_col, row_idx, col_ptr)
call elsi_set_csc_blk (eh, block_size_csc)

do SCF cycle
  Update Hamiltonian

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

  Update electron density
  Check SCF convergence
end do

call elsi_finalize(eh)
```

Remarks

1) Eigenvectors are returned in the BLACS_DENSE format, which is required to be properly set up.

Arbitrarily Distributed COO Sparse Matrix + ELSI Eigensolver Interface

```
SCF initialize

call elsi_init (eh, ELPA, MULTI_PROC, GENERIC_COO, n_basis, n_electron, n_state)
call elsi_set_mpi (eh, mpi_comm)
call elsi_set_blacs (eh, blacs_ctxt, block_size)
call elsi_set_coo (eh, global_nnz, local_nnz, row_idx, col_idx)

do SCF cycle
  Update Hamiltonian

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

  Update electron density
  Check SCF convergence
end do

call elsi_finalize(eh)
```

Remarks

1) Eigenvectors are returned in the BLACS_DENSE format, which is required to be properly set up.

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

```
SCF initialize

call elsi_init (eh, OMM, MULTI_PROC, BLACS_DENSE, n_basis, n_electron, n_state)
call elsi_set_mpi (eh, mpi_comm)
call elsi_set_blacs (eh, blacs_ctxt, block_size)

do SCF cycle
  Update Hamiltonian

  call elsi_dm_{real|complex} (eh, ham, ovlp, dm, bs_energy)

  Update electron density
  Check SCF convergence
end do

call elsi_finalize(eh)
```

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

```
SCF initialize

call elsi_init (eh, PEXSI, MULTI_PROC, SIESTA_CSC, n_basis, n_electron, n_state)
call elsi_set_mpi (eh, mpi_comm)
call elsi_set_csc (eh, global_nnz, local_nnz, local_col, row_idx, col_ptr)
call elsi_set_csc_blk (eh, block_size)

do SCF cycle
  Update Hamiltonian

  call elsi_dm_{real|complex}_sparse (eh, ham, ovlp, dm, bs_energy)
  call elsi_get_edm_{real|complex}_sparse (eh, edm)

  Update electron density
  Check SCF convergence
end do

call elsi_finalize(eh)
```

Remarks

1) Refer to the 5$^{\text{th}}$ remark in 3.5.4 for the chemical potential determination algorithm in PEXSI.

Arbitrarily Distributed COO Sparse Matrix + ELSI Density Matrix Interface

```
SCF initialize

call elsi_init (eh, PEXSI, MULTI_PROC, GENERIC_COO, n_basis, n_electron, n_state)
call elsi_set_mpi (eh, mpi_comm)
call elsi_set_coo (eh, global_nnz, local_nnz, row_idx, col_idx)

do SCF cycle
  Update Hamiltonian

  call elsi_dm_{real|complex}_sparse (eh, ham, ovlp, dm, bs_energy)
  call elsi_get_edm_{real|complex}_sparse (eh, edm)

  Update electron density
  Check SCF convergence
end do

call elsi_finalize(eh)
```

Remarks

1) Refer to the 5$^{\text{th}}$ remark in 3.5.4 for the chemical potential determination algorithm in PEXSI.

Multiple k-points Calculations

```
SCF initialize

call elsi_init (eh, NTPOLY, MULTI_PROC, BLACS_DENSE, n_basis, n_electron, n_state)
call elsi_set_mpi (eh, mpi_comm)
call elsi_set_blacs (eh, blacs_ctxt, block_size)
call elsi_set_kpoint (eh, n_kpt, i_kpt, i_wt)
call elsi_set_mpi_global (eh, mpi_comm_global)

do SCF cycle
  Update Hamiltonian

  call elsi_dm_{real|complex} (eh, ham, ovlp, dm, bs_energy)
  call elsi_get_edm_{real|complex} (eh, edm)

  Update electron density
  Check SCF convergence
end do

call elsi_finalize(eh)
```

Remarks

1) When there are multiple $\boldsymbol{k}$-points, there is no change in the way ELSI solver interfaces are called.

2) The electronic structure code needs to assemble the real-space density from the density matrices returned for the $\boldsymbol{k}$-points. The returned band structure energy, however, is already summed over all $\boldsymbol{k}$-points with respect to the weight of each $\boldsymbol{k}$-point. Refer to 3.2.4 for more information.

3) Spin-polarized calculations may be set up similarly.

Geometry Relaxation Calculations

```
SCF initialize

call elsi_init (eh, ...)
call elsi_set_* (eh, ...)

do geometry
  do SCF cycle
    Update Hamiltonian

    call elsi_{ev|dm}_{real|complex} (eh, ham, ovlp, ...)

    Update electron density
    Check SCF convergence
  end do

  Update geometry (overlap)

  call elsi_reinit (eh)
end do

call elsi_finalize(eh)
```

3.9 C/C++ Interface

ELSI is written in Fortran. A C interface around the core Fortran code is provided, 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] T. Auckenthaler et al., Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations, Parallel Computing, 37, 783-794 (2011).

[3] 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).

[4] F. Corsetti, The orbital minimization method for electronic structure calculations with finite-range atomic basis sets, Computer Physics Communications, 185, 873-883 (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] T. Imamura et al., Development of a high-Performance eigensolver on a peta-scale next-generation supercomputer system, Progress in Nuclear Science and Technology, 2, 643-650 (2011).

[8] T. Fukaya and T. Imamura, Performance evaluation of the EigenExa eigensolver on Oakleaf-FX: tridiagonalization versus pentadiagonalization, In 2015 IEEE International Parallel and Distributed Processing Symposium Workshop, 960-969 (2015).

[9] V. Hernandez et al., SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems, ACM Transactions on Mathematical Software, 31, 351-362 (2005).

[10] M. Keceli et al., Shift-and-invert parallel spectral transformation eigensolver: Massively parallel performance for density-functional based tight-binding, Journal of Computational Chemistry, 37, 448-459 (2016).

[11] W. Dawson and T. Nakajima, Massively parallel sparse matrix function calculations with NTPoly, Computer Physics Communications, 225, 154 (2018).

[12] B. Aradi et al., DFTB+, a sparse matrix-based implementation of the DFTB method, Journal of Physical Chemistry A, 111, 5678 (2007).

[13] W. Hu et al., DGDFT: A massively parallel method for large scale density functional theory calculations, The Journal of Chemical Physics, 143, 124110 (2015).

[14] V. Blum et al., Ab initio molecular simulations with numeric atom-centered orbitals, Computer Physics Communications, 180, 2175-2196 (2009).

[15] J.M. Soler et al., The SIESTA method for ab initio order-N materials simulation, Journal of Physics: Condensed Matter, 14, 2745-2779 (2002).

[16] A.H.R. Palser and D.E. Manolopoulos, Canonical purification of the density matrix in electronic-structure theory, Physical Review B, 58, 12704-12711 (1998).

[17] A.M.N. Niklasson, Expansion algorithm for the density matrix, Physical Review B, 66, 155115 (2002).

[18] L.A. Truflandier et al., Communication: Generalized canonical purification for density matrix minimization, The Journal of Chemical Physics, 144, 091102 (2016).

[19] V. Yu et al., ELSI: A unified software interface for Kohn-Sham electronic structure solvers, Computer Physics Communications, 222, 267-285 (2018).

License and Copyright

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

```
Copyright (c) 2015-2019, 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 2016.11.001 (LGPL3), libOMM (BSD2), NTPoly 2.3.1 (MIT), PEXSI 1.2.0 (BSD3), PT-SCOTCH 6.0.0 (CeCILL-C), and SuperLU_DIST 6.1.1 (BSD3) are redistributed through this version of ELSI. Individual license of each library can be found in the corresponding subfolder.