ELPA Manual

User's Guide and Best Practices

Version 2025.01.001

A. Marek, P. Karpov, T. Melson,
The ELPA developer team
Max Planck Computing and Data Facility

February 6, 2025

Contents

1 About ELPA

The computation of a subset or all eigenvalues and eigenvectors of a Hermitian matrix has high relevance for various scientific disciplines. Typically, direct solvers are used for the calculation of a significant part of the eigensystem. For large problems, solving for the eigensystem with the existing solvers can become the computational bottleneck.

With the aim of developing and implementing an efficient eigenvalue solver for petaflop applications, ELPA (Eigenvalue soLvers for Petaflop Applications) was born, and today it has become a modern library for direct, efficient, and scalable solution of eigenvalue problems involving dense, Hermitian matrices.

The ELPA library was originally created by the ELPA consortium consisting of the following organizations:

- Max Planck Computing and Data Facility (MPCDF), formerly known as Rechenzentrum Garching der Max-Planck-Gesellschaft (RZG),

- Bergische Universität Wuppertal, Lehrstuhl für angewandte Informatik,

- Technische Universität München, Lehrstuhl für Informatik mit Schwerpunkt Wissenschaftliches Rechnen,

- Fritz-Haber-Institut der Max-Planck-Gesellschaft, Berlin, Abt. Theorie,

- Max-Plack-Institut für Mathematik in den Naturwissenschaften, Leipzig, Abt. Komplexe Strukturen in Biologie und Kognition, and

- IBM Deutschland GmbH

We emphatically acknowledge code contributions from other developers.

ELPA uses the distributed matrix layout of ScaLAPACK, but replaces the solution steps with subroutines of its own. Two variants of the solver are available: a one-step, and a two-step solver hereinafter referred to as ELPA1 and ELPA2, respectively.

1.1 How to obtain ELPA

ELPA is an open source project. Its source code is freely available at `https://gitlab.mpcdf.mpg.de/elpa/elpa`. It is distributed under the terms of the GNU Lesser General Public License version 3 as published by the Free Software Foundation. A mirror of the above repository is also available on GitHub, which is mainly for opening issues and merge requests as well as contributions from the developer's community: `https://github.com/marekandreas/elpa`. Additionally, ELPA can be obtained from the following sources:

- Official release tarball from the ELPA webpage

- As a packaged software for several Linux distributions (e.g., Debian, Fedora, OpenSUSE)

1.2 Terms of use

ELPA can be freely obtained, used, modified and redistributed under the terms of the GNU Lesser General Public License version 3.

No other conditions have to be met. Nonetheless, we would be grateful if you consider citing the following articles:

1. If you use ELPA in general:

- T. Auckenthaler, V. Blum, H. J. Bungartz, T. Huckle, R. Johanni, L. Krämer, B. Lang, H. Lederer, and P. R. Willems, "Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations", Parallel Computing 37, 783 (2011). doi:10.1016/j.parco.2011.05.002.

- A. Marek, V. Blum, R. Johanni, V. Havu, B. Lang, T. Auckenthaler, A. Heinecke, H. J. Bungartz, and H. Lederer, "The ELPA library: scalable parallel eigenvalue solutions for electronic structure theory and computational science", Journal of Physics Condensed Matter, 26 (2014). doi:10.1088/0953-8984/26/21/213201

2. If you use the GPU version of ELPA:

- P. Kus, A. Marek, and H. Lederer, "GPU Optimization of Large-Scale Eigenvalue Solver", In: F. Radu, K. Kumar, I. Berre, J. Nordbotten, I. Pop (eds) Numerical Mathematics and Advanced Applications ENUMATH 2017. ENUMATH 2017. Lecture Notes in Computational Science and Engineering, vol 126. Springer, Cham

- V. Yu, J. Moussa, P. Kus, A. Marek, P. Messmer, M. Yoon, H. Lederer, and V. Blum, "GPU-Acceleration of the ELPA2 Distributed Eigensolver for Dense Symmetric and Hermitian Eigenproblems", Computer Physics Communications, 262, (2021)

3. If you use the new API and/or autotuning:

- P. Kus, A. Marek, S. S. Koecher, H. H. Kowalski, Ch. Carbogno, Ch. Scheurer, K. Reuter, M. Scheffler, and H. Lederer, "Optimizations of the Eigenvaluesolvers in the ELPA Library", Parallel Computing 85, 167 (2019). doi:10.1016/j.parco.2019.04.003.

4. If you use the new support for skew-symmetric matrices:

- P. Benner, C. Draxl, A. Marek, C. Penke, and C. Vorwerk, "High Performance Solution of Skew-symmetric Eigenvalue Problems with Applications in Solving the Bethe-Salpeter Eigenvalue Problem", Parallel Computing 96, 102639 (2020). doi:10.1016/j.parco.2020.102639.

1.3 Current release

The current ELPA release is 2025.01.001. It supports the API version 20250131. The oldest API version supported by the current release is version 20170403. On more information on the API versions, please have a look at Section 5.1

2 Quick start guide

This section gives a very short overview on how to use ELPA from a Fortran, C, or C++ application. Before showing the respective examples, a few things should be noted:

- ELPA uses the same block-cyclic matrix distribution as ScaLAPACK, so if you already have an application that uses ScaLAPCK eigensolvers, then converting it to use ELPA is just a matter of adding a couple of lines of code.

- It is assumed that the ELPA library is already installed on your system, either by a system administrator or by you via a system package or manually. For manual installation, please have a look at the Section 3.

- It is assumed that you can link your application against the installed ELPA library. If you need instructions on how to do that, please have a look at the Section 4.

- The examples provide a simple explanation on how to use ELPA within your application. They neglect all options about tailoring ELPA to your specific needs and about how to achieve the best performance possible. For quite a number of control options, we have chosen *reasonable* but maybe *not perfect* defaults. If you want to tune the usage and the performance of ELPA to your specific needs, please have a look at the Section 5.2 on key-value pairs and at the Section 6.1 on autotuning.

- ELPA can be used with Nvidia, AMD, and Intel GPUs, see Section 5.6. Make sure to install the appropriate GPU drivers on the machine.

2.1 Sequence of steps to use ELPA

To use the ELPA library in your code, follow these steps:

1. Include the header files (for C or C++ applications) or use the ELPA module (for Fortran applications).

2. Define a handle for an ELPA object.

3. Initialize the ELPA library.

4. Allocate the ELPA object.

5. Set the *mandatory parameters*, namely as the matrix size, the number of eigenvectors to be calculated, the block size of the BLACS block-cyclic distribution of the matrix, and additional parameters for the MPI setup. Note that these parameters are *fixed* for the lifetime of the ELPA object.

6. Set up the ELPA object via the `setup` method. This finalizes the setting of the *mandatory parameters* and they cannot be changed anymore for the lifetime of the given ELPA object.

7. Set some *runtime options*.

   - If GPUs should be used: Set one of the keywords `nvidia-gpu`, `amd-gpu`, or `intel-gpu` via the ELPA `set` method and call the `setup_gpu` method to finalize setup of GPU.

   - Set any other combination of *runtime options* (see Sec. 5.2) to control the ELPA runtime behavior to your preference.

8. Call one of ELPA's math functions. Examples for most commonly used routines are `eigenvectors`, `eigenvalues`, `generalized_eigenvectors`, and `generalized_eigenvalues` (see Sec. 5.3).

9. When finished: Deallocate the ELPA object and unintialize the ELPA library.

Below we provide *minimalistic* examples of how to call ELPA from a Fortran or C/C++ application. These examples are, however, not self-contained. They will only compile and run if they are embedded into an existing MPI application. Also the user has to create the matrix for which the (generalized) eigenvalue problem should be solved, which in case of an MPI application, must be distributed in a BLACS block-cyclic distribution, as it must be done for ScaLAPACK. For standalone examples, see the test programs in the `test` directory of ELPA's source code.

Please note that in the examples below, we first set the *mandatory parameters* (Sec. 5.2.1), initialize ELPA object, and then set the *runtime options* (Sec. 5.2.2). For the runtime options we set only GPU-related ones, but no other options for control and tuning of the ELPA library. See the discussion at the beginning of Sec. 5.2 for the difference between the mandatory parameters and the runtime options.

Also note that the *ELPA API version* is set to 20250131. The API version defines the set of key-value pairs which can be used to control the ELPA library and also defines the procedures provided by the library. For more details please have a look at Section 5.1.

2.2   Fortran example

```fortran
! Step 1: use the ELPA module
use elpa

! Step 2: define a handle for the ELPA object
class(elpa_t), pointer :: elpaInstance
integer :: status
! We urge the user to always check the error codes of all ELPA functions!

! Step 3: initialize the ELPA library
status = elpa_init(20250131)
if (status /= ELPA_OK) then
    print *, "ELPA API version not supported"
    ! Handle this error in your application
endif

! Step 4: allocate the ELPA object
elpaInstance => elpa_allocate(status)
! Check status code, e.g. with
if (status /= ELPA_OK) then
    print *, "Could not allocate ELPA instance"
    ! Handle this error in your application
endif

! Step 5: set mandatory parameters describing the matrix
! and its MPI distribution
```

```fortran
call elpaInstance%set("na", na, status)
if (status /= ELPA_OK) then
    print *, "Could not set parameter na"
    ! Handle this error in your application
endif
call elpaInstance%set("nev", nev, status)
! Check status code ...
call elpaInstance%set("local_nrows", na_rows, status)
! Check status code ...
call elpaInstance%set("local_ncols", na_cols, status)
! Check status code ...
call elpaInstance%set("nblk", nblk, status)
! Check status code ...
call elpaInstance%set("mpi_comm_parent", MPI_COMM_WORLD, status)
! Check status code ...
call elpaInstance%set("process_row", my_prow, status)
! Check status code ...
call elpaInstance%set("process_col", my_pcol, status)
! Check status code ...

! Step 6: set up the elpa object, finalize setting of mandatory parameters
status = elpaInstance%setup()
if (status /= ELPA_OK) then
    print *, "Could not setup the ELPA object"
    ! Handle this error in your application
endif

! Step 7: set runtime options, e.g. GPU settings
call elpaInstance%set("nvidia-gpu", 1, status) ! 1=on, 0=off
! Check status code ...

! If desired, set other tunable runtime options...

! Finalize the GPU setup. Needed only when using GPUs
status = elpaInstance%setup_gpu()
! Check status code ...

! Step 8: Solve the eigenvalue problem to obtain eigenvalues and eigenvectors
call elpaInstance%eigenvectors(a, ev, q, status)
! Check status code ...

! Step 9: cleanup ELPA
call elpa_deallocate(elpaInstance, status)
! Check status code ...
call elpa_uninit(status)
```

2.3   C/C++ example

```c
// Step 1: include the ELPA header file
#include <elpa/elpa.h>

// Step 2: define a handle for the ELPA object
```

```c
elpa_t elpaInstance;
int status;
// We urge the user to always check the error code of all ELPA functions!

// Step 3: initialize the ELPA library
status = elpa_init(20250131);
if (status != ELPA_OK) {
    fprintf(stderr, "ELPA API version not supported");
    // Handle this error in your application
}

// Step 4: allocate the ELPA object
elpaInstance = elpa_allocate(&status);
if (status != ELPA_OK) {
    fprintf(stderr, "Could not allocate ELPA instance");
    // Handle this error in your application
}

// Step 5: set mandatory parameters describing the matrix
// and its MPI distribution
elpa_set(elpaInstance, "na", na, &status);
// Check status code ...
elpa_set(elpaInstance, "nev", nev, &status);
// Check status code ...
elpa_set(elpaInstance, "local_nrows", na_rows, &status);
// Check status code ...
elpa_set(elpaInstance, "local_ncols", na_cols, &status);
// Check status code ...
elpa_set(elpaInstance, "nblk", nblk, &status);
// Check status code ...
elpa_set(elpaInstance, "mpi_comm_parent", MPI_COMM_WORLD, &status);
// Check status code ...
elpa_set(elpaInstance, "process_row", my_prow, &status);
// Check status code ...
elpa_set(elpaInstance, "process_col", my_pcol, &status);
// Check status code ...

// Step 6: set up the elpa object, finalize setting of mandatory parameters
status = elpa_setup(elpaInstance);
if (status != ELPA_OK) {
    fprintf(stderr, "Could not set up the ELPA object");
    // Handle this error in your application
}

// Step 7: set runtime options, e.g. GPU settings
elpa_set(elpaInstance, "nvidia-gpu", 1, &status); // 1=on, 0=off
// Check status code ...

// If desired, set other tunable runtime options...

// Finalize the GPU setup. Needed only when using GPUs
```

```
status = elpa_setup_gpu(elpaInstance);
// Check status code ...

// Step 8: solve the eigenvalue problem to obtain eigenvalues and eigenvectors
elpa_eigenvectors(elpaInstance, a, ev, q, &status);
// Check status code ...

// Step 9: cleanup ELPA
elpa_deallocate(elpaInstance, &status);
// Check status code ...
elpa_uninit(&status);
```

3 Installation guide

The build system of ELPA is the standard GNU Autotools (autoconf and automake installation infrastructure) and consists of the following steps:

- `./configure`

- `make`

- `make check`

- `make install`

Note that the `configure` script is included in the official ELPA release tarballs, which can be obtained from the ELPA website. The `configure` script is most likely **not** included if you obtain the ELPA sources by other means, in particular, if you use a Git clone of the ELPA repository or you download a tarball from the Git repository. To generate the `configure` in such cases, you must run the shell script `./autogen.sh`.

We describe the ELPA dependencies in Section 3.1 and then elaborate on the individual installation steps in Sections 3.2-3.3. A minimal complete installation example is given in Section 3.4. Finally, Section 3.5 gives some hints on the installation troubleshooting.

3.1 Dependencies

In order to build ELPA the following prerequisites and dependencies must be met:

1. Build tools:
   GNU Autotools (`autoconf`, `automake`, and `libtool`) must be installed. A minimal version of 2.71 for `autoconf` is needed in order to build ELPA with modern compilers (for example, Intel Fortran compiler `ifx`).

2. Compilers:
   ELPA is written in Fortran, C, and C++. The GPU versions are written in CUDA, HIP, or SYCL. Thus you need several compilers to build ELPA.

   (a) Fortran compiler: a recent Fortran compiler is needed. It must fully support the Fortran 2003 and parts of the Fortran 2008 standard. To achieve the best performance if possible the most recent compilers should be used. In case of the GNU compiler, at least version 10 is required.

   (b) C compiler: a recent C compiler is needed. The compiler must at least support the C11 standard

(c) C++ compiler: a recent C compiler is needed. The compiler must at least support the C++11. **Note** that to build with SYCL, the C++17 standard must be supported and used.

Some compilers (e.g., clang) are not regularly tested in building ELPA. Every modern compiler should, however, be able to compile the code.

In case of GPU build, additional compilers are needed depending on which version of the GPU support should be build. We recommend using the most recent as possible GPU software stack version.

- Nvidia-GPUs: The CUDA software stack must be installed and the `nvcc` compiler is needed.

- AMD-GPUs: The ROCm software stack must be installed and the `hip` compiler is needed.

- Intel-GPUs: The Intel oneAPI compilers `icx` and `ifx` must be installed.

3. External libraries: Some external libraries are needed at build and runtime:

- the Basic Linear Algebra Subroutines (BLAS)

- the Linear Algebra Package (LAPACK)

If ELPA is build for parallel distributed runs (which is the preferred case), in addition

- the Basic Linear Algebra Communication Subroutines (BLACS)

- ScaLAPACK

- Message Passing Interface (MPI)

are needed. Depending on whether you want to build ELPA in the Nvidia, AMD, or Intel GPU version, some additional libraries might be needed:

- Nvidia-GPUs: cublas, cusolver and potentially NCCL

- AMD-GPU: rocblas and rocsolver

- Intel-GPUs: oneAPI MKL

ELPA can be configured to run sequentially as well as in parallel on shared- and/or distributed-memory systems. The shared-memory parallel algorithm uses OpenMP threads, while the programming model of the distributed algorithm is based on the message passing library (MPI). In addition, the hybrid models MPI+OpenMP, MPI+GPU, and MPI+OpenMP+GPU are also supported. For details on the installation process and the necessary configure options, please see Sec. 3.2. **Note** that the **sequential** build option is only meant for installation on desktop or laptop machines. Such a build of ELPA provides you the full API, such that you can develop applications with ELPA, but obviously the performance of such a build will be very suboptimal.

3.2 Configuration

Running the `configure` script is the first step of the installation procedure. Note that if this script is not present in the ELPA root folder, that is, if you obtained the ELPA sources from a Git clone as in the example below, it can be easily created using `autogen.sh`, otherwise you can skip the following step.

```
git clone https://gitlab.mpcdf.mpg.de/elpa/elpa.git
cd elpa
```

```
./autogen.sh
```

It is best practice to run the configuration in a subdirectory in order to keep the source directory clean:

```
mkdir build
cd build
../configure [options]
```

3.2.1 Compiler and linker variables for configure

We observe that most problems with building ELPA arise from a misunderstanding how to pass flags to the compilers, the linker, and how to specify the link line for the libraries which ELPA needs as external dependencies. Thus, we want to mention here how one can typically control these when calling a configure script.

Note these variables represent a very generic concept which applies to all builds with `autoconf` tools, independent of the ELPA library.

FC	The Fortran compiler to use. **Examples:** `FC="mpif90"` (for GNU) or `FC="mpiifx"` (for Intel) Note, that this variable must point to the Fortran compiler executable you want to invoke. Thus in case of an MPI build, it must be the MPI Fortran compiler. In case of a serial build, it must be the Fortran compiler.
CC	The C compiler to use. **Examples:** `CC="mpigcc"` or `CC="mpiicx"`
CXX	The C++ compiler to use. **Examples:** `CXX="mpicxx"` or `CXX="mpiicpx"`
CPP	The C preprocessor to use. Explicitly set this if you encounter errors during the build. **Examples:** `CPP="gcc -E"`
FCFLAGS	All flags that must be passed to the Fortran compiler to control the compiler's behavior. **Example:** `FCFLAGS="-O2 -mavx"` Note that setting the Fortran optimization and vectorization flags via `FCFLAGS` (as well C/C++ flags via `CFLAGS`/`CXXFLAGS`, see below) is of **utmost** importance to obtain a good ELPA performance. We elaborate on the optimization and vectorization flags in Sec. 3.2.2.
CFLAGS	All flags that must be passed to the C compiler to control the compiler's behaviour. **Example:** `CFLAGS="-O2 -mavx"`
CXXFLAGS	All flags which must be passed to the C++ compiler to control the compiler's behaviour. **Example:** `CXXFLAGS="-O2 -mavx"`
LDFLAGS	All flags which must be passed to the linker to control the linker's behaviour. **Example:** `LDFLAGS="-Wl,-rpath,/absolute_path_to_a_library"`

`LIBS`	External libraries that you want to link with.
	Example:
	`LIBS="-Wl,-L/absolute_path_to_a_library -llibrary"`

In addition to these standard variables of `autotools`, the ELPA configure honors some special variables:

`SCALAPACK_FCFLAGS`	Additional Fortran compiler flags for ScaLAPACK usage.
	Example:
	`SCALAPACK_FCFLAGS="-I$MKL_HOME/include/intel64/lp64"`
	Note, that this variable is a convenience feature.
	You can also pass these flags to the Fortran compiler via the `FCFLAGS` variable (see above).

`SCALAPACK_LDFLAGS`	Additional linker flags for ScaLAPACK usage.
	Example:
	`SCALAPACK_LDFLAGS="-L$MKL_HOME/lib/intel64 \`
	`  -lmkl_scalapack_lp64 -lmkl_intel_lp64 -lmkl_sequential \`
	`  -lmkl_core -lmkl_blacs_intelmpi_lp64 -lpthread -lm \`
	`  -Wl,-rpath,$MKL_HOME/lib/intel64"`
	Note, that this variable is a convenience feature. You can also pass these flags to the linker via the `LDFLAGS` and `LIBS` variables (see above).

`NVCC`	CUDA compiler command for processing `.cu` files.
	Default: `nvcc`

`NVCCFLAGS`	CUDA compiler flags.
	Default: `-O2`

`HIPCC`	HIP compiler command for processing `.hip` files.
	Default: `hipcc`

`HIPFLAGS`	HIP compiler flags.
	Default: `-O2`

`PYTHON_CONFIG`	Path to `python-config`.

`PYTHON_INCLUDE`	Include flags for Python.

`NUMPY_INCLUDE`	Include flags for NumPy.

Below in Sec. 3.2.3, we list and categorize the important options to configure ELPA. A full list of all available options can be obtained with `./configure --help`.

The configure options control which features are available at runtime. Whether a feature is actually used for the solution of the eigenproblem, depends on the ELPA settings chosen in your application. For example, if a specific kernel is enabled by a configuration option and it is not set as default, it must be activated with the `solver` setting (cf. Sec. 5.2.2) in order to be used for the computation.

3.2.2   Compiler flags for vectorization and optimization

In this section we give some hints on how to set the compiler vectorization and optimization flags, which are of the **utmost** importance for the ELPA performance.

Since (combinations of) these flags depend on the used compiler, its version, and the target hardware architecture, it is beyond the scope of this manual to give their comprehensive description

and we refer further to the compiler/your HPC center documentation. In this section we still give some useful guides for these flags.

Vectorization
: The vectorization capabilities of the CPU should be fully exploited. Consult the documentation of your compiler to find the appropriate `FCFLAGS/CFLAGS/CXXFLAGS` for your system. It is important to enable the appropriate ELPA kernels (cf. Sec. 3.2.3) together with the correct compiler flags.
For GNU and Intel compilers, consider the `-march` and `-x` flags, respectively.
Examples: `FCFLAGS="-march=skylake-avx512 ..."` (GNU)
`FCFLAGS="-xCORE-AVX512 ..."` (Intel)

Optimization
: Compiler optimization should be switched on to enhance performance. You are adviced to select the highest optimization level that yields correct results. If the optimization is too aggressive, the calculated eigenvalues and eigenvectors are inaccurate. We do not generally recommend an optimization level, because this depends on the compiler and its version. Be aware that compiler vendors occasionally change the optimization strategies included in a certain level. Please always verify the correctness of your configuration with `make check` (see Sec. 3.3). Note that specifying the `-O` flag alone will typically *not* enable vectorization.
Examples: For the GNU compilers (up to version 12) and the *classic* Intel compilers (`icc` and `ifort` up to Intel oneAPI version 2021.6), we recommend `-O2` or `-O3`. This is *not* valid for the new Intel `icx` and `ifx` compilers, which we have not yet tested thoroughly.

Threading
: If ELPA should be run in hybrid parallelization with MPI and OpenMP, we recommend to link against the threaded math library.
Example: `-lmkl_intel_thread` instead of `-lmkl_sequential` for Intel MKL

Floating-point calculations
: To obtain accurate computational results, you should consider setting the flags for controlling the floating-point handling.
Example: For the Intel compiler, consider changing the value of `-fp-model`. Setting `-fp-model=precise` will enable the most accurate calculations, however, with a potential performance penalty.

Some further useful hints can be found in the ELPA configuration examples, Sec. 3.2.4.

3.2.3 Configure options

In addition to the variables described in Sec. 3.2.1 the build of ELPA, can be controlled by adding options to the configure command line. Here, one has to distinguish between

- standard configure options, offered by configure and

- ELPA specific configure options.

Listing here all the *standard configure options* is beyond the scope of this documentation. Most of these options are also only recommended for very experienced users. Concerning the *ELPA specific configure options*, in this section we will focus here on most common ones, while other "expert" configure options are listed and explained in the Appendix A.

In any case, all configure options can be listed via the `./configure --help` command.

The general syntax for optional flags is `--enable-feature` or `--enable-feature=yes` for enabling the "feature" and `--disable-feature` or `--enable-feature=no` for disabling it. For some of the flags, the syntax is `--with-feature=yes` to enable and `--with-feature=no` to disable or `--with-feature=value` to specify a special flavor of a feature.

> **Hint**
>
> It is **strongly recommended** to always include the option
> `--enable-option-checking=fatal`, which aborts the configuration if any other option is unknown or invalid.

1. Controlling the installation directories

`--prefix`	Installation directory for architecture-independent files. Use this option if you want to install without root privileges. **Example:** `--prefix="$HOME/soft/elpa"` **Default:** `/usr/local`
`--exec-prefix`	Installation directory for architecture-dependent files. **Default:** same as `--prefix`
	Further options are available for controlling the individual subdirectories. See `./configure --help` for a detailed list.

2. Controlling the API provided by ELPA

`--disable-skew-symmetric-support`	Do not support skew-symmetric matrices. Removes `skew_eigenvectors`, `skew_eigenvalues`, etc. from the API. **Default:** enabled
`--enable-python`	Build and install the Python wrapper. **Default:** disabled

3. Controlling MPI

`--with-mpi=[yes|no]`	Enable MPI parallelization. Note that the MPI parallelization should only be switched off for very good reasons and that ELPA execution is then limitied to one compute node! **Default:** `yes`
`--disable-mpi-module`	Replace the Fortran MPI module `use mpi` with `include "mpif.h"`. Use this option only if your MPI library does not provide a Fortran module. Typically error messages are "no module mpi" or "cannot open module mpi". **Default:** enabled (= use mpi module)
`--disable-detect-mpi-launcher`	Disable automatic detection of the MPI launcher. **Default:** enabled (= detect launcher)
`--enable-mpi-launcher=[mpiexec|mpiexec.hydra|mpirun|srun]`	

Use the specified MPI launcher for running the test suite (`make check`) on HPC systems that do not allow interactive MPI runs. Must be combined with `--disable-detect-mpi-launcher`.

Example: With `--disable-detect-mpi-launcher` `--enable-mpi-launcher=srun` you can call `make check` from a SLURM script on a system that supports `srun`.

Default: detect automatically (see `--disable-detect-mpi-launcher`)

4. Controlling OpenMP

`--enable-openmp`   Compile with OpenMP threading parallelism. Note that independent of whether ELPA has been built with threading support, you can always use multi-threading for your math library if ELPA is properly linked against its threaded version. See also Sec. 5.4.2.
Default: disabled

5. Availability of ELPA2 compute kernels

Note that at the end of the configuration, a list of all enabled kernels will be displayed.

`--disable-generic-kernels`   Do not build generic kernels compatible with all platforms. Note that the performance of these kernels will be inferior to other vectorized kernels.
Default: enabled

`--disable-sse-kernels`   Do not build SSE kernels.
Default: enabled

`--disable-sse-assembly-kernels`
Do not build SSE kernels written in assembly.
Default: enabled

`--disable-avx-kernels`   Do not build AVX kernels for Intel Sandy Bridge and later.
Default: enabled

`--disable-avx2-kernels`   Do not build AVX2 kernels for Intel Haswell and later.
Default: enabled

`--disable-avx512-kernels`   Do not build AVX-512 kernels for Intel Knights Landing and later.
Default: enabled

`--enable-vsx-kernels`   Build VSX kernels for IBM POWER7 and later.
Default: disabled

`--enable-sparc64-kernels`   Build kernels for processors supporting SPARC64 (SPARC V9).
Default: disabled

`--enable-bgp-kernels`   Build kernels for IBM Blue Gene/P.
Default: disabled

`--enable-bgq-kernels`   Build kernels for IBM Blue Gene/Q.

Default: disabled

`--enable-neon-arch64-kernels`
Build kernels for ARM using Neon (Advanced SIMD) instructions.
Default: disabled

`--enable-sve128-kernels`   Build 128-bit SVE kernels for ARM processors.
Default: disabled

`--enable-sve256-kernels`   Build 256-bit SVE kernels for ARM processors.
Default: disabled

`--enable-sve512-kernels`   Build 512-bit SVE kernels for ARM processors.
Default: disabled

`--with-fixed-real-kernel=KERNEL`
Build only a single specific real kernel and make it default. Avialable kernels are: `generic`, `generic_simple`, `generic_simple_block4`, `generic_simple_block6`, `sparc64_block2`, `sparc64_block4`, `sparc64_block6`, `neon_arch64_block2`, `neon_arch64_block4`, `neon_arch64_block6`, `vsx_block2`, `vsx_block4`, `vsx_block6`, `sse_block2`, `sse_block4`, `sse_block6`, `sse_assembly`, `sve128_block2`, `sve128_block4`, `sve128_block6`, `avx_block2`, `avx_block4`, `avx_block6`, `avx2_block2`, `avx2_block4`, `avx2_block6`, `sve256_block2`, `sve256_block4`, `sve256_block6`, `avx512_block2`, `avx512_block4`, `avx512_block6`, `sve512_block2`, `sve512_block4`, `sve512_block6`, `bgp`, `bgq`, `nvidia_gpu`, `amd_gpu`, `intel_gpu_sycl`, `nvidia_sm80_gpu`.

`--with-fixed-complex-kernel=KERNEL`
Build only a single specific complex kernel and make it default. Avialable kernels are: `generic`, `generic_simple`, `neon_arch64_block1`, `neon_arch64_block2`, `sse_block1`, `sse_block2`, `sse_assembly`, `sve128_block1`, `sve128_block2`, `avx_block1`, `avx_block2`, `avx2_block1`, `avx2_block2`, `sve256_block1`, `sve256_block2`, `avx512_block1`, `avx512_block2`, `sve512_block1`, `sve512_block2`, `bgp`, `bgq`, `nvidia_gpu`, `amd_gpu`, `intel_gpu_sycl`, `nvidia_sm80_gpu`.

`--with-default-real-kernel=KERNEL`
Set a specific real kernel as default. See `--with-fixed-real-kernel` for a complete list of avaliable kernels.
Default: `real_avx512_block2`

`--with-default-complex-kernel=KERNEL`
Set a specific complex kernel as default. See `--with-fixed-complex-kernel` for a complete list of

avaliable kernels.
Default: `complex_avx512_block1`

`--enable-heterogenous-cluster-support`
Experimental! Select a kernel supported by all CPUs in a heterogenous cluster. Currently, only available for Intel CPUs.
Default: disabled

6. Controlling the AMD GPU version

`--enable-amd-gpu-kernels` Build kernels for AMD GPUs.
Default: disabled

If this option is enabled, then the details of the AMD GPU version can be further controlled with

`--enable-gpu-streams=[amd|no]`
Use Cuda or HIP streams in Nvidia or AMD GPU versions, respectively.
Default: `amd` (enabled)

`--with-AMD-gpu-support-only=[yes|no]`
Experimental! Build real and complex AMD GPU kernels only. If enabled, no other kernels will be available at runtime.
Default: `no`

`--with-rocsolver=[yes|no]` Use AMD rocSOLVER library.
Default: `yes`

`--enable-marshalling-hipblas-library`
Use indirection layer hipBLAS instead of rocBLAS.
Default: disabled

`--enable-hipcub` Use reductions from hipCUB in AMD GPU kernels.
Default: disabled

`--enable-gpu-ccl=[rccl|no]`
Use NCCL or RCCL communication libraries in Nvidia or AMD GPU versions, respectively.
Default: `no` (disabled)

7. Controlling the Intel GPU version

`--enable-intel-gpu-sycl-kernels`
Build kernels for Intel GPUs using SYCL. Requires `--enable-intel-gpu-backend=sycl`.
Default: disabled

`--enable-intel-gpu-backend=[sycl|openmp]`
Build GPU code for Intel GPUs and select either SYCL or OpenMP as the backend.
Default: disabled (= no Intel GPU kernels)

If ELPA is configured to use SYCL then one can further control the build with

```
--with-INTEL-gpu-support-only=[yes|no]
```
> **Experimental!** Build real and complex Intel GPU kernels only. If enabled, no other kernels will be available at runtime.
> **Default:** `no`

8. Controlling the Nvidia GPU version

   In the past, when only an Nvidia GPU version was available, Nvidia GPU builds were triggered by

   `--enable-gpu` **Deprecated.** Build kernels for GPUs. Please use explicit options for the various vendors instead.
   Default: disabled

Instead, nowadays, the Nvidia GPU build must be enabled with one of the two following options:

```
--enable-nvidia-gpu-kernels
```
> Build kernels for Nvidia GPUs. Use `--with-NVIDIA-GPU-compute-capability` to set the compute capability for best performance.
> **Default:** disabled

```
--enable-nvidia-sm80-gpu-kernels
```
> Build kernels for Nvidia GPUs supporting the compute capability 8.0, for example, Nvidia A100.
> **Default:** disabled

If the Nvidia GPU build is enabled, then it can be further controlled with the arguments:

`--with-cuda-path=PATH` Path where CUDA is installed.
Default: detect automatically

`--enable-cuda-aware-mpi` **Experimental!** Use CUDA-aware MPI features to enhance performance. Requires an MPI library that integrates with CUDA, for example, OpenMPI.
Default: disabled

```
--enable-gpu-streams=[nvidia|no]
```
> Use CUDA or HIP streams in Nvidia or AMD GPU versions, respectively.
> **Default:** `nvidia` (enabled)

```
--enable-gpu-ccl=[nccl|no]
```
> Use NCCL or RCCL communication libraries for Nvidia or AMD GPU versions, respectively.
> **Default:** `no` (disabled)

`--with-nccl-path=PATH`   Path where NCCL is installed.
Default: detect automatically

`--with-NVIDIA-gpu-support-only=[yes|no]`
Build real and complex Nvidia GPU kernels only. If enabled, no other kernels will be available at runtime.
Default: `no`

`--with-NVIDIA-sm_80-gpu-support-only=[yes|no]`
Build real and complex Nvidia GPU kernels for compute capability 8.0 only. If enabled, no other kernels will be available at runtime.
Default: `no`

`--with-NVIDIA-GPU-compute-capability=VALUE`
Use compute capability `VALUE` for Nvidia GPU kernels.
Default: `sm_35` (= 3.5)

`--enable-NVIDIA-gpu-memory-debug`
Output memory information of Nvidia GPU devices. The script at `utils/memory/check_memory.py` can be used to process the output.
Default: disabled

`--with-cusolver=[yes|no]`   Use Nvidia cuSolver library.
Default: `yes`

`--enable-nvidia-cub`   Use reductions from CUB in real Nvidia GPU kernel.
Default: disabled

`--enable-nvtx`   Build and install NVTX wrappers for profiling the GPU code.
Default: disabled

9. Controlling performance-related options

`--disable-autotuning`   Disable autotuning. See Sec. 6.1 for a detailed explanation.
Default: enabled

`--with-papi=[yes|no]`   Use PAPI to measure and print FLOP counts. Only available if `--enable-timings` is set.
Default: `no`

`--with-likwid=[yes|no|PATH]`
Use LIKWID to measure the performance of some solver parts. If set to `yes` and the library can not be found, you can set the `PATH` explicitly.
Default: `no`

`--disable-assumed-size`   Do not use assumed-size Fortran arrays.
Default: enabled

10. Controlling output

`--disable-timings`   Disable timings measurement with the API functions `timer_start`, `timer_stop`, `get_time`, and the output of

`print_times`. If disabled, some ELPA features like autotuning will not work.
Default: enabled

`--enable-redirect`   **For test programs.** Redirect stdout and stderr of each MPI task to a separate file in the subdirectory `mpi_stdout`.
Default: disabled

11. Controlling precision

`--disable-single-precision` Disable single precision and build for double precision only.
Default: enabled

`--enable-64bit-integer-math-support`
Support 64-bit integers in the math libraries BLAS, LAPACK, and ScaLAPACK. Combine this option only with the appropriate link line to the math library, e.g., by choosing the suffix `_ilp64` for Intel MKL.
Default: disabled

`--enable-64bit-integer-mpi-support`
Support 64-bit integers in the MPI library. Make sure to link against the appropriate MPI library.
Default: disabled

12. Controlling Fortran features

`--disable-Fortran2008-features`
Do not use Fortran 2008 features. Use this option if your compiler does not support the Fortran 2008 standard.
Default: enabled

`--enable-ifx-compiler`   Modify the code to build with older `ifx` compiler versions. This flag is not needed anymore in recent Intel oneAPI versions.
Default: disabled

13. Controlling which test programs will be build

`--disable-c-tests`   Do not build C tests.
Default: enabled

`--disable-cpp-tests`   Do not build C++ tests.
Default: enabled

`--enable-scalapack-tests` Build ScaLAPACK test cases for performance comparison.
Default: disabled

`--enable-python-tests`   Enable Python tests. Only available if `--enable-python` is set.
Default: disabled

`--with-test-programs=[all|gpu|cpu|none]` Build the test programs for cpu, gpu, all, or none.
Default: all

3.2.4 `configure` examples

The following examples should provide an overview of how to configure ELPA. They are, however, not meant to be copied and pasted for a production-ready build. They have to be adapted to the respective system and must be optimized for best performance.

- **OpenMP, GNU compilers**

To configure a threaded build without MPI support on your personal linux workstation, you can try using this command ("\" symbol is used for a line break and can be omitted):

```
../configure --prefix=$HOME/soft/elpa CC=gcc CXX=g++ FC=gfortran \
  CFLAGS="-O3 -march=native" FCFLAGS="-O3 -march=native" \
  --enable-option-checking=fatal --disable-avx512 --with-mpi=no --enable-openmp
```

We assume here that you have the current GNU compiler suite and all required libraries installed in the default location. These requirements are often met on the well-known linux distributions. If the math libraries can not be found automatically, you need to explicitly set the variables `SCALAPACK_FCFLAGS` and `SCALAPACK_LDFLAGS` (see Sec. 3.2.1) and other examples below.

- **MPI+OpenMP, GNU compilers**

Assuming the same system as in the previous example and after having installed an MPI library (for example, OpenMPI) in the default location, you can build ELPA with additional MPI support like this:

```
../configure --prefix=$HOME/soft/elpa CC=mpicc CXX=mpicxx FC=mpifort \
  CFLAGS="-O3 -march=native" FCFLAGS="-O3 -march=native" \
  --enable-option-checking=fatal --disable-avx512 --with-mpi=yes \
  --enable-openmp
```

- **MPI, Intel toolchain**

Here we present an example of a configure line for a system with an Intel CPU that does not support AVX-512 instructions. We are using the Intel classic compilers in combination with Intel MKL and Intel MPI ("Intel toolchain"):

```
../configure --prefix=$HOME/soft/elpa CC=mpiicc CXX=mpiicpc FC=mpiifort \
  CFLAGS="-O3 -xHost" FCFLAGS="-O3 -xHost" \
  SCALAPACK_FCFLAGS="-I${MKLROOT}/include/intel64/lp64" \
  SCALAPACK_LDFLAGS="-L${MKLROOT}/lib/intel64 -lmkl_scalapack_lp64 \
  -lmkl_intel_lp64 -lmkl_sequential -lmkl_core -lmkl_blacs_intelmpi_lp64 \
  -lpthread -lm -Wl,-rpath,${MKLROOT}/lib/intel64" \
  --enable-option-checking=fatal --with-mpi=yes --disable-avx512
```

Note that `-lmkl_intel_thread` should be used instead of `-lmkl_sequential` for the threaded build. `-lmkl_blacs_openmpi_lp64` should be used instead of `-lmkl_blacs_intelmpi_lp64` if you use Open MPI instead of Intel MPI. For the further details specific to Intel MKL we refer to Intel MKL Link Line Advisor [1].

At the time of writing, all necessary Intel tools including C, C++, and Fortran compilers, Intel MKL, and Intel MPI are available free of charge as a part of Intel oneAPI HPC Toolkit.

---

[1] https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl-link-line-advisor.html

- **Nvidia GPU + MPI, Intel toolchain**

If you want to use ELPA on a Nvidia GPU-accelerated system with Intel CPU supporting AVX-512 (Intel Knights Landing and later) and using Intel toolchain:

```
../configure --prefix=$HOME/soft/elpa CC=mpiicc FC=mpiifort CXX=mpiicpc \
  CFLAGS="-O3 -xCORE-AVX512 -I$MKLROOT/include/intel64/lp64 \
  -I$CUDA_HOME/include" \
  FCFLAGS="-O3 -xCORE-AVX512 -I$MKLROOT/include/intel64/lp64 \
  -I$CUDA_HOME/include" \
  LDFLAGS="-L$MKLROOT/lib/intel64 -lmkl_scalapack_lp64 -lmkl_intel_lp64 \
  -lmkl_sequential -lmkl_core -lmkl_blacs_intelmpi_lp64 -lpthread -lm \
  -Wl,-rpath,$MKLROOT/lib/intel64" \
  --enable-option-checking=fatal --with-mpi=yes --enable-nvidia-gpu-kernels \
  --with-NVIDIA-GPU-compute-capability=sm_80 --with-cuda-path=$CUDA_HOME \
  --with-cusolver=yes --with-gpu-streams=nvidia
```

Here `$CUDA_HOME` is the path to the CUDA installation directory.

- **Nvidia GPU + NCCL + OpenMPI, GNU compilers, Intel MKL**

```
../configure --prefix=$HOME/soft/elpa CC=mpicc FC=mpif90 CXX=mpicxx \
  CFLAGS="-O3 -march=skylake-avx512 -I$MKL_HOME/include/intel64/lp64 \
  -I$CUDA_HOME/include" \
  FCFLAGS="-O3 -march=skylake-avx512 -I$MKL_HOME/include/intel64/lp64 \
  -I$CUDA_HOME/include" \
  LDFLAGS="-L$MKL_HOME/lib/intel64 -lmkl_scalapack_lp64 -lmkl_gf_lp64 \
  -lmkl_sequential -lmkl_core -lmkl_blacs_openmpi_lp64 -lpthread \
  -Wl,-rpath,$MKL_HOME/lib/intel64" \
  --enable-option-checking=fatal --with-mpi=yes \
  --enable-nvidia-gpu \
  --with-NVIDIA-GPU-compute-capability=sm_80 \
  --with-cuda-path=$CUDA_HOME --enable-gpu-ccl=nccl --with-nccl-path=$NCCL_HOME
```

- **Intel GPU + MPI, Intel toolchain**

If you want to use ELPA on a Intel GPU-accelerated system on top of the Intel oneAPI toolchain:

```
../configure --prefix=$HOME/soft/elpa FC="mpiifort -fc=ifx" CC=mpiicx \
  CXX=mpiicpx CFLAGS="-O3 -xCORE-AVX512" CXXFLAGS="-O3 -xCORE-AVX512 -fsycl \
  -I$ONEAPI_ROOT/compiler/latest/linux/include/sycl \
  -I$ONEAPI_ROOT/mkl/latest/include" \
  FCFLAGS="-O3 -xCORE-AVX512 -fsycl \
  -I$ONEAPI_ROOT/compiler/latest/linux/include/sycl" \
  LIBS="-L$ONEAPI_ROOT/compiler/latest/linux/lib \
  -L$ONEAPI_ROOT/compiler/latest/linux/compiler/lib/intel64_lin -lsycl \
  -Wl,-rpath,$ONEAPI_ROOT/compiler/latest/linux/lib" \
  SCALAPACK_FCFLAGS="-I$ONEAPI_ROOT/mkl/latest/include/intel64/lp64 -fsycl" \
  SCALAPACK_LDFLAGS="-fsycl -L$ONEAPI_ROOT/mkl/latest/lib/intel64 -lmkl_sycl \
  -lmkl_scalapack_lp64 -lmkl_intel_lp64 -lmkl_sequential -lmkl_core \
  -lmkl_blacs_intelmpi_lp64 -lsycl -lOpenCL -lpthread -lm -ldl -lirng \
  -lstdc++ -Wl,-rpath,$ONEAPI_ROOT/mkl/latest/lib/intel64" \
  --enable-option-checking=fatal --enable-ifx-compiler --with-mpi=yes \
  --enable-sse-kernels --enable-sse-assembly-kernels --enable-avx-kernels \
  --enable-avx2-kernels --enable-avx512 --enable-intel-gpu-sycl-kernels \
```

```
   --enable-intel-gpu-backend=sycl --enable-single-precision
```

- **AMD GPU + MPI, Cray toolchain**

```
../configure CPP="gcc -E" CC=cc CXX=hipcc FC=ftn CFLAGS="-O3 -g" \
  CXXFLAGS="-O3 -g -std=c++17 -DROCBLAS_V3 -D__HIP_PLATFORM_AMD__ \
  --offload-arch=gfx90a" FCFLAGS="-O3 -g" LIBS="-lamdhip64 -fPIC" \
  --enable-option-checking=fatal --with-mpi=yes --disable-sse \
  --disable-sse-assembly --disable-avx --disable-avx2 --disable-avx512 \
  --enable-amd-gpu --enable-single-precision --enable-gpu-streams=amd \
  --enable-hipcub --disable-cpp-tests --with-rocsolver
```

- **AMD GPU + RCCL, Cray toolchain**

```
../configure CPP="gcc -E" CC=cc CXX=hipcc FC=ftn CFLAGS="-g -O3 -std=c++17" \
  FCFLAGS="-g -O3" CXXFLAGS="-DROCBLAS_V3 -D__HIP_PLATFORM_AMD__ \
  --offload-arch=gfx90a -g -O3 -std=c++17" LIBS="-lamdhip64 -fPIC" \
  --enable-option-checking=fatal --with-mpi=yes --disable-sse \
  --disable-sse-assembly --disable-avx --disable-avx2 --disable-avx512 \
  --enable-amd-gpu --enable-single-precision --enable-gpu-streams=amd \
  --enable-hipcub --disable-cpp-tests --with-rocsolver --enable-gpu-ccl=rccl \
  --with-rocsolver
```

3.3 Building

After the successful configuration with the appropriate options for your system, ELPA can be built with `make`. Depending on your machine, you can speed up this process with the command line argument `-j` followed by the number of cores to use for building.

When ELPA has been compiled and linked successfully, we recommend running the included test suite with `make check`. It supports the following options:

CHECK_LEVEL	If set to `extended`, run additional time-consuming tests. If set to `autotune`, run additional tests for verifying the autotuning feature of ELPA. **Default:** Run the basic test suite.
TASKS	Number of MPI tasks to use for testing. **Default:** 2
TEST_FLAGS	Tuple of matrix size, number of eigenvalues, and block size. **Example:** `TEST_FLAGS="150 100 32"` for a matrix of 150×150 requesting 100 eigenvalues using a block size of 32. Smaller matrices speed up the 'make check' test suite. **Default:** `"5000 150 16"`

3.4 Complete installation example

Here we present an example of a complete installation of ELPA for a linux workstation with Intel CPU, using the Intel toolchain.

```
git clone https://gitlab.mpcdf.mpg.de/elpa/elpa.git
cd elpa
./autogen.sh
```

```
mkdir build
cd build

../configure --prefix=$HOME/soft/elpa CC=mpiicc CXX=mpiicpc FC=mpiifort \
    CFLAGS="-O3 -xHost" FCFLAGS="-O3 -xHost" \
    SCALAPACK_FCFLAGS="-I${MKLROOT}/include/intel64/lp64" \
    SCALAPACK_LDFLAGS="-L${MKLROOT}/lib/intel64 -lmkl_scalapack_lp64 \
    -lmkl_intel_lp64 -lmkl_sequential -lmkl_core -lmkl_blacs_intelmpi_lp64 \
    -lpthread -lm -Wl,-rpath,${MKLROOT}/lib/intel64" \
    --enable-option-checking=fatal --with-mpi=yes --disable-avx512

make -j 8
make check -j 8 TEST_FLAGS="150 100 32"
make install
```

> **Hint**
>
> You can accelerate the test suite by customizing the value of `TEST_FLAGS` as shown above.

> **Hint**
>
> `make check` step is optional. If you don't want to run the tests, you can also speed up the installation and skip their build by using the configure option `--with-test-programs=no`.

If the installation was successful, ELPA is now installed in the directory `$HOME/soft/elpa` and you'll get a message how to link your application against ELPA. More details on compiling and linking against ELPA can be found in Sec. 4.

If the installation was not successful, we give some (although incomprehensive) hints on troubleshooting in the following section.

3.5 Troubleshooting

Errors can occur during one of these steps:

- `configure`
- `make`
- `make check`

If the error occured during `make` or `make check`, make sure to clear the build directory before re-running `make` or `make check` before reconfiguring ELPA with new flags.

3.5.1 Common `configure` problems

Most typical errors at the `configure` stage are related to missing dependencies. Please make sure that you have installed all required dependencies (see Sec. 3.1) and carefully read the error message, since it can give a hint which dependency is missing. For the extended error message, check the `config.log` file in the build directory.

Problem. If you use the GNU compilers and encounter the error "`initializer element is not constant`" or "`not specified in enclosing 'parallel'`" during build, this is most likely caused by an outdated gcc compiler version.

Solution. Make sure that you use gcc of at least version 10.

3.5.2   Common `make` problems

Problem. There is an error message related to MPI, e.g. `"no module mpi"`, `"cannot open module mpi"`, `"Could not resolve generic procedure mpi_send/mpi_recv/mpi_allreduce"`.

Solution. Try to reconfigure ELPA with the option `--disable-mpi-module`. This flag does not switch off MPI, it just affects internal working of ELPA, so that it does not use the Fortran MPI module, but rather get interfaces by `include mpif.h`.

3.5.3   Common `make check` problems

Extended error messages can be found in the `test-suite.log` file in the build directory. Note that the Fortran tests, e.g. `validate_complex_double_eigenvectors_...` are usually more expressive in describing the error than the corresponding C/C++ tests, e.g.
`validate_c_version_complex_double_eigenvectors_.../`
`validate_cpp_version_complex_double_eigenvectors_...`

Problem. Some HPC centers do not allow running MPI programs interactively. It therefore could happen that `make check` does not run at all on the machine on which you are installing ELPA.

Solution. If the HPC center supports SLURM with `srun`, you can reconfigure ELPA with the following options: `--disable-detect-mpi-launcher --enable-mpi-launcher=srun` (see Sec. 3.2.3) and then call `make check` from a SLURM script. Alternatively, consult the documentation of your HPC center on how to *interactively* run MPI programs.

Problem. `"Program Exception - illegal instruction"` error and/or errors occurring in `compute_hh_trafo`.

Solution. Try to reconfigure ELPA with disabled vectorization options, e.g. `--disable-avx512`.

Problem. `make check` is too slow.

Solution. Use the `-j` option for utilizing more cores and a smaller test matrix size to speed up the tests, e.g. `make check -j 8 TEST_FLAGS="150 100 32"` for using 8 cores and the matrix of size 150×150. You can also switch off certain ELPA APIs and tests by reconfiguring ELPA with the flags `"--disable-skew-symmetric-support --disable-c-tests --disable-cpp-tests"` or use the very minimal test set by configuring with the flag `--with-test-programs=no`.

4 Compiling and linking against ELPA

To link your application against your local installation of the ELPA library, you need to point the compiler to the correct include files (header file for C/C++ or module file for Fortran) and instruct the linker to find the library file.

4.1 Linking with `pkg-config`

The best option is to use the *package config tool*. Make sure to install the program `pkg-config` on your system. The following steps explain how to fetch the correct flags. Note that you usually forward them to the compiler and linker as `FCFLAGS`, `CFLAGS`, `CXXFLAGS`, and `LDFLAGS`.

1. Extend the `PKG_CONFIG_PATH` environment variable to point to the subfolder `lib/pkgconfig` (or `lib64/pkgconfig` on some systems) of your ELPA installation. Depending on your system and shell, this *might* look similar to this:

   ```
   export
       PKG_CONFIG_PATH=/absolute_path_to_elpa/lib/pkgconfig:$PKG_CONFIG_PATH
   ```

   where `/absolute_path_to_elpa` corresponds to the absolute path of the ELPA installation set by `--prefix` (e.g. `$HOME/soft/elpa` from Sec. 3.2.4)

2. To fetch the correct flags for Fortran (`FCFLAGS`), run the command

   ```
   pkg-config --variable=fcflags elpa
   ```

   or

   ```
   pkg-config --variable=fcflags elpa_openmp
   ```

   depending on your build.

3. To fetch the correct flags for C/C++ (`CFLAGS` or `CXXFLAGS`), run the command

   ```
   pkg-config --cflags elpa
   ```

   or

   ```
   pkg-config --cflags elpa_openmp
   ```

   depending on your build.

4. To fetch the correct linker flags (`LDFLAGS`), run the command

   ```
   pkg-config --libs elpa
   ```

   or

   ```
   pkg-config --libs elpa_openmp
   ```

   depending on your build.

Adding these flags to the build procedure of your application will link it against ELPA. It should be mentioned that these flags will include all necessary options for libraries that ELPA has been linked against during its build, especially the GPU, MPI, BLACS, BLAS, LAPACK, and ScaLAPACK libraries. If your application relies also on one or more of these libraries, the linkline is "shipped" with the ELPA linkline and explicit linking might not be necessary.

4.2 Linking without `pkg-config`

If you do not want to use the `pkg-config` tool, although we strongly recommend doing that, you can also set the flags manually. For most compilers, the C-include flag (added to `CFLAGS` or `CXXFLAGS`) should be

> `-I/absolute_path_to_elpa/include/build_specific_subdirectory`

and the Fortran module flag (added to `FCFLAGS`) should be

> `-I/absolute_path_to_elpa/include/build_specific_subdirectory/modules`

where, as before, `/absolute_path_to_elpa` corresponds to the absolute path of the ELPA installation set by `--prefix` (e.g. `$HOME/soft/elpa` from Sec. 3.2.4). The highlighted subdirectory `build_specific_subdirectory` is something like `elpa-2025.01.001`.

The linker flags (`LDFLAGS`) are typically

> `-L/absolute_path_to_elpa/lib -lelpa`

or

> `-L/absolute_path_to_elpa/lib -lelpa_openmp`

depending on your build. Make sure that you adapt the paths and flags accordingly. Note that unlike in the case of `pkg-config --libs`, here the `LDFLAGS` do not automatically contain links to external libraries (MPI, BLACS, etc.).

It might happen at *runtime* that the ELPA library cannot be found. In this case either set the `LD_LIBRARY_PATH` pointing to the ELPA library directory with (depending on your system and shell)

> `export LD_LIBRARY_PATH=/absolute_path_to_elpa/lib:$LD_LIBRARY_PATH`

or add an additional linker flag

> `-Wl,-rpath,/absolute_path_to_elpa/lib`

to the `LDFLAGS` when building your application. In the latter case, setting the `LD_LIBRARY_PATH` is not necessary anymore.

5 Calling ELPA

In this section, the ELPA Fortran API is explained first followed by illustrations of the steps involved to setup ELPA and use it from within an application code. For guidlines on using the Python API, please see Sec. 5.7.

5.1 API version

ELPA release	Release API version	Minimal supported API vesion
2025.01.001	20250131	20170403
2024.05.001	20241105	20170403
2024.03.001	20241103	20170403
2023.11.001	20231705	20170403
2023.05.001	20231705	20170403
2022.11.001	20221109	20170403
2022.05.001	20220510	20170403
2021.11.001	20211125	20170403
2021.05.001	20210430	20170403
2020.11.001	20200417	20170403
2020.05.001	20200417	20170403
2019.11.001	20191110	20170403
2019.05.001	20190524	20170403
2018.11.001	20181113	20170403
2018.05.001	20180525	20170403
2017.11.001	20171201	20170403
2017.05.001	20170403	20170403

Table 1: ELPA release versions and the corresponding API versions

Each ELPA release defines two version numbers for the API. First, the *release API version*, for the latest release also often referred to as *current API version*, and the *minimal API version* supported by this release. Obviously, the versioning scheme of ELPA API versions is monotonically increasing such that a natural ordering (lower API version means older) can be infered. An overview over the ELPA versions published and with the repective *release API version* and *minimal API version* is shown in the Table 1.

The *minimal API version* tells you whether there have been breaking changes in the API, i.e. whether downward compatibility only to a certain ELPA release (identified by the *release API version* of this old release being the same as the *minimal API version* of the newer release) is guranteed. Up to now this has been never happening for the ELPA library, but might potentially occur in the future.

A change in the *release API version*, implies that there have either been changes to the API or whether new *key-value pairs* (see Sec. 5.2) have been introduced. Typically, the *release API version* is increased if new procedures have been *added* with a release. If the *minimal API version* did not change from one release to the other, it also implies that nothing has been *removed* from the API. As mentioned, the API version of a new release will also be changed if new *key-value pairs* have been introduced, to allow for new funtionality or performance tuning. It is important to note that adding new key-value pairs does not introduce breaking changes, since an application making use of these new key-value pairs can be still linked against and run with older ELPA versions not supporting these keywords. The only change will be that the older ELPA library will ignore the new keywords but still run and produce correct results, albeit with maybe lower

performance than a newer ELPA release. Even removing key-value pairs would only introduce a "breaking change" insofar that the key-value combination would be ignored and performance might drop, but again, ELPA would continue to work and produce the correct results.

Nevertheless, it is recommended to upgrade your application to the latest versions of ELPA availabe and to initialize with the latest *release API version* since only this does gurantee you to obtain the best possible performance from the ELPA library.

Note that before the release of ELPA 2017.05.001 another API has been used and breaking API changes occured with every release. With the introduction of the API of release 2017.05.001 the API become much more expressive and stable.

For a given ELPA installation you can find out the supported API versions by either referring to the Table 1, or by inspecting the file `elpa_version.h` in your ELPA installation path.

5.2 Key-Value pairs

Every ELPA object is controlled via key-value pairs. Note, that ELPA knows two types of key-value pairs:

- **Mandatory parameters**: settings which are *fixed* for the lifetime of an ELPA object and *must* be set *before* calling the `setup` procedure, e.g. a matrix size. If you want to change any of these parameters, you have to create a new ELPA object. Note that as many ELPA objects as needed can be instanciated at the same time. These parameters are listed in Sec. 5.2.1.

- **Runtime options**: key-value pairs which control the runtime of the ELPA library for a given ELPA object. These options might either control the program flow, such as using GPUs or the 1-stage or 2-stage solver, or the performance of the ELPA library, by tuning the algorithmic execution to the hardware and problem size. Whether a key-value pair is available or has an effect, depends on the supported API version of the ELPA library used (see Sec. 5.1), the API version initialized, and also on the build options of the ELPA library. Runtime option values can be adjusted between calls to the ELPA math-routines. Most common runtime options are listed in Sec. 5.2.2, and some additional expert options are listed in Appendix. B.

The values of key-value pairs can be integers, floating-point (float or double) numbers, boolean flags (`0` or `1`), or special data types. The accepted values are specified below together with their default values if applicable.

5.2.1 Mandatory parameters

The following key-value pairs are mandatory parameters which must be set for each ELPA object, **before** calling the `setup` procedure and then cannot be changed anymore:

`na`	*Integer.* Global matrix size `na` × `na`.
`nev`	*Integer.* Number of eigenvalues and/or eigenvectors to be computed. $0 \leq$ `nev` $\leq$ `na`
`nblk`	*Integer.* Block size for the block-cyclic matrix layout. Must be a power of two. Typical values for CPU execution are 16 or 32. For GPU computations, 64 or larger are favorable and 1024 is the maximal allowed value. Note that the parameter's value should be chosen in accordance with the the most favorable BLACS distribution of your application.

`local_nrows`	*Integer.* Number of rows of the local matrix stored on the given MPI process. Can be determined using the ScaLAPACK function `numroc`.
`local_ncols`	*Integer.* Number of columns of the local matrix stored on the given MPI process. Can be determined using the ScaLAPACK function `numroc`.
`mpi_comm_parent`	*MPI_Comm.* Global MPI communicator comprising all MPI ranks used by ELPA. Mandatory if MPI is enabled.
`blacs_context`	*Integer.* The blacs context for the valid BLACS distribution as obtained from the BLACS funtions.

Note that it is mandatory the set the parameters `local_nrows`, `local_ncols` to describe the dimension of the local sub-matrices of the distributed global matrix of size $na \times na$. It is also mandatory to set the parameter `mpi_comm_parent` to provide the global MPI communicator of all ranks to be used in the calculations.

However, ELPA does also need the information how the MPI setup is spanning a 2D grid of row and column MPI processes. You have two possible ways how to provide this information to ELPA:

1. The splitting of the communicator `mpi_comm_parent` (typically that is `MPI_COMM_WORLD`) into the `mpi_comm_rows` and `mpi_comm_cols` communicators is done in your application *before* the ELPA object is setup. Then you can provide this communicators to ELPA. If you choose this option it is mandatory to set the following parameters:

`mpi_comm_rows`	*MPI_Comm.* MPI communicator for the MPI processes organized in rows.
`mpi_comm_cols`	*MPI_Comm.* MPI communicator for the MPI processes organized in columns.

2. ELPA should internally split the provided `mpi_comm_parent` communicator into the internally used `mpi_comm_rows` and `mpi_comm_cols` communicators. If you choose this option since you do not want to provide the `mpi_comm_rows` and `mpi_comm_cols` communicators, it is mandatory to set the following parameters:

`process_row`	*Integer.* The row id of the MPI rank in the row communicator as obtained from the BLACS routines.
`process_col`	*Integer.* The column id of the MPI rank in the column communicator as obtained from the BLACS routines.

Note that per instanciated ELPA object one has to decide for **one** of the two options discussed above. It is not allowed to provide a combination of the parameters from both options, since the `setup` method will not accept such input.

In addition to the above mentioned mandatory parameters for seting up the ELPA object, one can provide *additional* parameters to describe the MPI setup:

`num_processes`	*Integer.* Total number of MPI ranks in `mpi_comm_parent`.
`num_process_rows`	*Integer.* Total number of MPI ranks in `mpi_comm_rows`.
`num_process_cols`	*Integer.* Total number of MPI ranks in `mpi_comm_cols`.
`process_id`	*Integer.* Rank number of each MPI task in `mpi_comm_parent`.

Setting these parameters is not necessary, since ELPA can deduce them from the mandatory parameters and will set them internally if they are not provided by the user. However, it is recommended to set them, since we have observed that this helps users to organize their code and keep an understanding on how the ELPA object is set up.

5.2.2 Runtime options

The following parameters are optional.

5.2.2.a General runtime options

`omp_threads`	*Integer.* Number of OpenMP threads to use. Only relevant if ELPA has been configured with `--enable-openmp`. **Default: 1**
`solver`	Either `ELPA_SOLVER_1STAGE` or `ELPA_SOLVER_2STAGE`. Specify which solver to use: ELPA1 or ELPA2. This choice can influence the performance considerably. If unsure, measure and compare the runtime of both solvers. **Default:** `ELPA_SOLVER_1STAGE`
`real_kernel`	Real kernel to use if `solver` is set to `ELPA_SOLVER_2STAGE`. **Default:** set by configuration option `--with-default-real-kernel`
`complex_kernel`	Complex kernel to use if `solver` is set to `ELPA_SOLVER_2STAGE`. **Default:** set by configuration option `--with-default-complex-kernel`

5.2.2.b Runtime options for GPU

The following parameters are related to running ELPA on GPUs. All flags can be enabled or disabled by setting them to `1` or `0`, respectively.

`nvidia-gpu`	Enable GPU acceleration using Nvidia GPUs. Only available if ELPA has been configured with `--enable-nvidia-gpu`. **Default:** 0 (= disabled)
`amd-gpu`	Enable GPU acceleration using AMD GPUs. Only available if ELPA has been configured with `--enable-amd-gpu`. **Default:** 0 (= disabled)
`intel-gpu`	Enable GPU acceleration using Intel GPUs. Only available if ELPA has been configured with `--enable-intel-gpu-backend=sycl` and `--enable-intel-gpu-sycl-kernels`. Additionally, ELPA must be configured with the support of Intel ifx compiler `--enable-ifx-compiler`. **Default:** 0 (= disabled)
`use_gpu_id`	*Integer.* Specify which GPU should be used by the calling MPI task.

5.2.2.c Runtime options for debugging

The following switches control additional measurements or output, which can be conveniently used for debugging.

verbose	Print verbose information about calculations and errors. This option can be enabled without performance loss. **Default:** 0 (= disabled)
debug	Print debugging information. Additional checks are performed and additional timing information is gathered. Enabling this option decreases performance and is not recommended for production. **Default:** 0 (= disabled)
output_build_config	Print the options with which ELPA has been configured and built.
output_pinning_information	Print pinning information, i.e. association of OpenMP threads to cores. **Default:** 0 (= disabled)
print_flops	Enable printing FLOP rates. **Default:** 0 (= disabled)
timings	Enable the detailed timings measurement. Only available if ELPA has been configured with `--enable-timings`. This option can be enabled without performance loss. It should not be disabled if autotuning is used. Although it's a runtime option, it has to be enabled once before calling `elpa_setup()` to set up the timer, then `timings` can be switched on and off between the calls to ELPA solver routines like other runtime options. See also Sec. 6.3. **Default:** 1 (= enabled)
measure_performance	Measure FLOP rates together with the timings using PAPI. Only available if ELPA has been configured with `--enable-timings`. **Default:** 0 (= disabled)

5.3   Math routines provided by ELPA

ELPA provides numerous math routines needed for solving symmetric, or hermitian (generalized) eigenvalue problems. If not stated otherwise, all routines are available for real and complex double-precision calculations. If ELPA has been build with single-precision support, the routines are also available for real and complex single-precision datatypes.

In the following "all datatypes" means real and complex double and single-precision. Please note that all ELPA procedures have in common that they have a sligthly different synopsis depending whether ELPA is used form Fortran or C/C++. The difference, however, follows a single pattern:

- In Fortran programs ELPA procedures are always used in the form `yourELPAobjectInstance%procedurename`.

- In C/C++ programs ELPA procedures have always an *additional first argument* – the `handle` to `yourELPAobjectInstance` and procedure names are preceded with the prefix `elpa_`.

For simplicity, only the Fortran synopsis is shown here. More details, and also the C/C++ synopsis can be found in Appendix D.

5.3.1 Standard eigenvalue problem

> **Important note for the GPU users**
>
> The overloaded convenience functions, like `eigenvectors()` can only be used if the data has been allocated on the host. If the data has been allocated on the GPU device, an automatic destinction of datatypes is not possible and one has to use the explicit functions specifying the datatype, e.g. `eigenvectors_double()`.

For the *standard* eigenvalue problem the following routines are provided:

`eigenvalues(a, ev, error)`
Overloaded function (for all datatypes) that returns only the eigenvalues. Here:
a   is the *host* matrix,
ev is the *host* eigenvalue array,
error is the return code.

`eigenvectors(a, ev, q, error)`
Overloaded function (for all datatypes) that returns all eigenvalues and (part of) the eigenvectors. Here:
a   is the *host* matrix,
ev is the *host* eigenvalue array,
q   is the *host* matrix of eigenvectors,
error is the return code.

`eigenvalues_[double|single|double_complex|single_complex](a, ev, error)`
Explicit function (for all datatypes) that returns only the eigenvalues. Here:
a   is the *host/device* matrix,
ev is the *host/device* eigenvalue array,
error is the return code.

`eigenvectors_[double|single|double_complex|single_complex](a, ev, q, error)`
Explicit function (for all datatypes) that returns all eigenvalues and (part of) the eigenvectors. Here:
a   is the *host/device* matrix,
ev is the *host/device* eigenvalue array,
q   is the *host/device* matrix of eigenvectors,
error is the return code.

Note that if ELPA has been build with the support for real skew-symmetric matrices, then in addition the procedures `skew_eigenvalues`, `skew_eigenvalues_[double|float]`, `skew_eigenvectors` and `skew_eigenvectors_[double|float]` are available.

5.3.2 Generalized eigenvalue problem

> **Important note for the GPU users**
>
> The overloaded convenience functions, like `generalized_eigenvectors()` can only be used if the data has been allocated on the host. If the data has been allocated on the GPU device, an automatic destinction of datatypes is not possible and one has to use the explicit functions specifying the datatype, e.g. `generalized_eigenvectors_double()`.

> **Performance tip**
>
> For the best performance on GPU we recommend using the explicit interface e.g.
> `generalized_eigenvectors_double()` instead of `generalized_eigenvectors()` since the
> former excludes also some internal data transfers between the host and the device.

For the *generalized* eigenvalue problem $AQ = \lambda BQ$ the following routines are provided:

`generalized_eigenvalues(a, b, ev, is_already_decomposed, error)`

Overloaded function (for all datatypes) that only returns (part of) the eigenvalues. Here

a   is the *host* matrix A,

b   is the *host* matrix B,

ev is the *host* eigenvalue array,

`is_already_decomposed` allows one can skip the decomposition if the `b` matrix stays the same between subsequent calls,

`error` is the return code.

`generalized_eigenvectors(a, ev, b, q, is_already_decomposed, error)`

Overloaded function (for all datatypes) that returns all eigenvalues and (part of) the eigenvectors. Here

a   is the *host* matrix A,

b   is the *host* matrix B,

ev is the *host* eigenvalue array,

q   is the *host* matrix of eigenvectors,

`is_already_decomposed` allows one can skip the decomposition if the `b` matrix stays the same between subsequent calls,

`error` is the return code.

`generalized_eigenvalues_[double|single|double_complex|single_complex]`
`(a, b, ev, is_already_decomposed, error)`

Explicit function (for all datatypes) that returns only the eigenvalues. Here:

a   is the *host/device* matrix A,

b   is the *host/device* matrix B,

ev is the *host/device* eigenvalue array,

`is_already_decomposed` allows one can skip the decomposition if the `b` matrix stays the same between subsequent calls,

`error` is the return code.

`generalized_eigenvectors_[double|single|double_complex|single_complex]`
`(a, b, ev, q, is_already_decomposed, error)`

Explicit function (for all datatypes) that returns all eigenvalues and (part of) the eigenvectors. Here:

a   is the *A host/device* matrix,

b   is the *B host/device* matrix,

ev is the *host/device* eigenvalue array,

q   is the *host/device* matrix of eigenvectors,

`is_already_decomposed` allows one can skip the decomposition if the `b` matrix stays the same between subsequent calls,

`error` is the return code.

5.3.3 Auxillary routines

These auxillary routines are internally used by ELPA for transforming a generalized eigenvalue problem to a standard eigenvalue problem. Since these routines do offer GPU support (unlike in ScaLAPACK), and generally perform better also on CPUs than the respectice ScaLAPACK implementations, these routines are also available via the API. These procedures are:

`cholesky(a,error)`
>Overloaded function (for all datatypes) that returns the Cholesky decomposition for the *host* matrix `a`.

`cholesky_[double|float|double_complex|float_complex](a,error)`
>Explicit function (for all datatypes) that returns the Cholesky deomposition of the *host/device* matrix `a`.

`hermitian_multiply(uplo_a,uplo_c,ncb,a,b,nrows_b,ncols_b,c,nrows_c,ncols_c,error)`
>Overloaded function (for all datatypes) that multiplies the transposed/hermitian conjugated matrix A with matrix B and stores the result in matrix $C = A^{T/H}B$.
>Here:
>`uplo_a` is set to 'U' if A is upper triangular, 'L' if A is lower triangular, or anything else if A is a full matrix;
>`uplo_c` is set to 'U' if only the upper triangular part of C is needed, 'L' if only the lower triangular part of C is needed, or anything else full matrix C is needed;
>`ncb` is the number of columns of the global matrices `b` and `c`;
>`a` is the *host* matrix A,
>`b` is the *host* matrix B,
>`nrows_b` is the number of rows of matrix `b`;
>`ncols_b` is the number of columns of matrix `b`;
>`c` is the *host* matrix C,
>`nrows_c` is the number of rows of matrix `c`;
>`ncols_c` is the number of columns of matrix `c`;
>`error` is the return code.

`hermitian_multiply_[double|float|double_complex|float_complex]`
`(uplo_a,uplo_c,ncb,a,b,nrows_b,ncols_b,c,nrows_c,ncols_c,error)`
>Explicit function (for all datatypes) that multiplies the transposed/hermitian conjugated matrix `a` with matrix `b` and stores the results in matrix `c`. Arguments are the same as above except:
>`a` is the *host/device* matrix A,
>`b` is the *host/device* matrix B,
>`c` is the *host/device* matrix C.

`invert_triangular(a,error_elpa)`
>Overloaded function (for all datatypes) that inverts the upper triangular *host* matrix `a`.

`invert_triangular_[double|float|double_complex|float_complex](a,error_elpa)`

Explicit function (for all datatypes) that inverts the upper triangular *host/device* matrix `a`.

`solve_tridiagonal(d, e, q, error)`
Overloaded function (for all datatypes) that returns all eigenvalues and all eigenvectors of a real symmetric tridiagonal matrix. Here:
`d` is the *host* array of the diagonal elements of the matrix. On output: the eigenvalues of the matrix,
`e` is the *host host* array of the offdiagonal elements of the matrix.
`q` is the *host* matrix of eigenvectors,
`error` is the return code.

`pxgemm_multiply(trans_a,trans_b,ncb,a,b,nrows_b,ncols_b,c,nrows_c,ncols_c,error)`
Overloaded function (for all datatypes) that calculates the matrix product
$C = op(A)op(B)$. Here:
`trans_a` is set to 'N' if A is non-transposed: $op(A) = A$; to 'T' if A is transposed: $op(A) = A^T$; to 'H' if A is conjugate-transposed: $op(A) = A^H$;
`trans_b` is set to 'N' if B is non-transposed: $op(B) = B$; to 'T' if B is transposed: $op(B) = B^T$; to 'H' if B is conjugate-transposed: $op(B) = B^H$;
`ncb` is the number of columns of the global matrices `b` and `c`;
`a` is the *host* matrix A,
`b` is the *host* matrix B,
`nrows_b` is the number of rows of matrix `b`;
`ncols_b` is the number of columns of matrix `b`;
`c` is the *host* matrix C,
`nrows_c` is the number of rows of matrix `c`;
`ncols_c` is the number of columns of matrix `c`;
`error` is the return code.

`pxgemm_multiply_[double|float|double_complex|float_complex]`
`(trans_a,transa_b,ncb,a,b,nrows_b,ncols_b,c,nrows_c,ncols_c,error)`
Explicit function (for all datatypes) that calculates the matrix product
$C = op(A)op(B)$. Arguments are the same as above except:
`a` is the *host/device* matrix A,
`b` is the *host/device* matrix B,
`c` is the *host/device* matrix C.

Important note for the users

`pxgemm_multiply` IS AN EXPERIMENTAL ROUTINE. For now, only square matrices (`ncb=elpa_handle%na`), having the same block-cyclic distribution (`nrows_b=nrows_c=elpa_handle%local_nrows`, `ncols_b=ncols_c=elpa_handle%local_ncols`) are supported. Use with care.

5.4 Using ELPA without MPI

Performance tip

We strongly discourage using ELPA in a non-MPI mode for production runs.

5.4.1 Sequential mode

Although the main focus of ELPA is on massively parallel execution, to get acquainted with it, it can be useful to test ELPA in a sequential mode first. In this case, the following steps (already outlined in Sec. 2.1) have to be taken:

1. Use/include the elpa module

Fortran
| ```
use elpa
``` |

C/C++
| ```
#include <elpa/elpa.h>
``` |

2. Define a handle for an ELPA object

Fortran
```fortran
class(elpa_t), pointer :: elpaInstance
integer :: status
```

C/C++
```c
elpa_t elpaInstance;
int status;
```

3. Initialize ELPA by passing the API version that is going to be used (see Table 1)

Fortran
```fortran
status = elpa_init(20171201)
if (status /= ELPA_OK) then
  print *, "ELPA API version not supported"
  stop 1
endif
```

C/C++
```c
status = elpa_init(20171201);
if (status != ELPA_OK) {
    fprintf(stderr, "ELPA API version not supported");
    exit(1);
}
```

4. Allocate the ELPA object

Fortran
```fortran
elpaInstance => elpa_allocate(status)
if (status /= ELPA_OK) then
  print *, "Could not allocate ELPA instance"
  stop 1
endif
```

```c
elpaInstance = elpa_allocate(&status);
if (status != ELPA_OK) {
  fprintf(stderr, "Could not allocate ELPA instance");
  exit(1);
}
```

We recommend to always check the return status of the ELPA routines. For brevity we don't show this in the following steps, but status checks are always asssumed.

5. Specify the information about the input matrix via setting the *mandatory parameters*. For the sequential mode the dimensions of the local part of the matrix `local_nrows`×`local_ncols` are equal to these of the global matrix `na`×`na`. Also note that even though a BLACS grid as such is not used for sequential execution, the `nblk` parameter must be set to some non-zero value, e.g. to `na`.

Fortran

```fortran
! size of the input matrix is na x na
call elpaInstance%set("na", na, status)

! number of eigenvectors to be computed, 0 <= nev <= na
call elpaInstance%set("nev", nev, status)

! number of rows of the local part of the matrix
call elpaInstance%set("local_nrows", na, status)

! number of columns of the local part of the matrix
call elpaInstance%set("local_ncols", na, status)

! block size of the BLACS block-cyclic distribution
call elpaInstance%set("nblk", na, status)
```

C/C++

```c
// size of the input matrix is na x na
elpa_set(elpaInstance, "na", na, &status);

// number of eigenvectors to be computed, 0 <= nev <= na
elpa_set(elpaInstance, "nev", nev, &status);

// number of rows of the local part of the matrix
elpa_set(elpaInstance, "local_nrows", na, &status);

// number of columns of the local part of the matrix
elpa_set(elpaInstance, "local_ncols", na, &status);

// block size of the BLACS block-cyclic distribution
elpa_set(elpaInstance, "nblk", na, &status);
```

6. Call the `setup()` routine to complete the problem setup. This step finalizes the setting of *mandatory parameters* for the given ELPA object and they can not be changed in the future.

Fortran	C/C++
```status = elpaInstance%setup()```	```status = elpa_setup(elpaInstance);```

7. If desired, set any number of tunable *runtime options*. These can be changed between different calls of ELPA solver. A complete list of the runtime options can be found in Sec. 5.2.2.

Fortran
```
call elpaInstance%set("solver", ELPA_SOLVER_2STAGE, status)
```

C/C++
```
elpa_set(elpaInstance, "solver", ELPA_SOLVER_2STAGE, &status);
```

Performance tip

ELPA2 is usually the better choice than ELPA1 for performance on CPU

Fortran
```
! set the AVX BLOCK2 kernel; otherwise ELPA_2STAGE_REAL_DEFAULT is used
call elpaInstance%set("real_kernel",ELPA_2STAGE_REAL_AVX_BLOCK2, status)
```

C/C++
```
elpa_set(elpaInstance,"real_kernel",ELPA_2STAGE_REAL_AVX_BLOCK2,&status);
```

The concept of *kernel* is specific to ELPA2, and affects its most computationally intensive part. The default kernel depends on the flags provided during the `configure` step (e.g. `--enable-avx512-kernels`) and is printed out after the configure is finished (e.g. `real_avx512_block2 (default)`). The default kernel is usually the best choice, but if you are not sure, you can measure the performance of different kernels.

You can control the default values of the runtime options by setting the corresponding enviroment variables, for example:

```
export ELPA_DEFAULT_solver=ELPA_SOLVER_2STAGE
export ELPA_DEFAULT_real_kernel=ELPA_2STAGE_REAL_AVX_BLOCK2
```

Hence, if your code doesn't explicitly set certain ELPA runtime option, in this way you can change the its value without modifying your code.

8. Call the solver to obtain eigenvalues `ev(na)` and eigenvectors `q(na,na)`. The input matrix `a(na,na)` has to be initialized any time before this step and it's also a resposibility of the user to take care of allocation and deallocation of `a`, `ev`, and `q` arrays.

```
call elpaInstance%eigenvectors(a, ev, q, status)
```

```
elpa_eigenvectors(elpaInstance, a, ev, q, &status);
```

The `elpa_eigenvectors()` routine requires that the matrix of eigenvectors `q(na,na)` was of the same size as the input matrix `a(na,na)`, even in the case when only part of the eigenvectors is requested.

ELPA relies on the contigousness of the data, hence never use vectors of vectors in C++ to represent the 2D data!

ELPA always assumes the column-major ordering of the matrices. In C/C++, to avoid the confusion, we recommend to use 1D arrays/vectors.

9. Clean up by deallocating the ELPA object and uninitializing ELPA

```
call elpa_deallocate(elpaInstance)
call elpa_uninit(&status)
```

```
elpa_deallocate(elpaInstance, &status);
elpa_uninit(&status);
```

5.4.2 OpenMP mode

ELPA with OpenMP threads successfully scales only up to $\sim 2-8$ threads, depending on the problem parametrs and the hardware. If you want to utilize more CPUs, you should use ELPA with MPI (Sec. 5.5.1) or in MPI+OpenMP hybrid mode (Sec. 5.5.2).

To enable multi-threading, ELPA should be configured with the switch `--enable-openmp=yes`. Needless to say, your compiler should support OpenMP and the corresponding flags should be provided upon compilations of the users's code (e.g. `-fopenmp` for GCC and `-qopenmp` for Intel compilers).

If ELPA has been built with OpenMP threading support, you can specify the number of OpenMP threads that ELPA will use internally. The steps involved in setting up the problem are the same as for sequential case (see Sec. 5.4.1) with one additional step: to allocate OpenMP threads

for ELPA routines, it is **mandatory** to set the number of threads as a runtime parameter using the `set()` method in addition to setting it in the execution environment (via `export OMP_NUM_THREADS=...`):

```fortran
! set 4 threads for the elpa object
call elpaInstance%set("omp_threads", 4, status)
```

```c
elpa_set(elpaInstance, "omp_threads", 4, &status);
```

ELPA utilizes two different levels of parallelization with OpenMP threads: OpenMP parallelization of native ELPA routines ("ELPA-OpenMP") and the threading of the BLAS-like math library being in use ("BLAS-OpenMP"). The corresponding two kinds of parallelization regions are independent of each other and do not overlap. Since there is no nested OpenMP parallelization, in the optimal setting all the allocated treads either perform ELPA-OpenMP or BLAS-OpenMP work. Hence we recommend to set the OpenMP enviroment variable that prohibits the nested parallelization `OMP_MAX_ACTIVE_LEVELS=1`.

In order to utilize the "BLAS-OpenMP" parallelization, please ensure that you link ELPA against a BLAS/LAPACK library which does offer threading support; otherwise, a severe performance loss will be encountered. Please refer to the documentation of your math library for details on multi-threading support and how to activate it.

In particular, if Intel MKL is used, ELPA has to be linked with the threaded MKL library `-lmkl_intel_thread` (and not `-lmkl_sequential`). Then the "BLAS-OpenMP" number of threads can be controlled by `MKL_NUM_THREADS` environment variable that can be set by the user to any value $\leq$`$SLURM_CPUS_PER_TASK`. We recommend, however, to allow MKL to pick the number of threads dynamically by setting `MKL_DYNAMIC=TRUE`.

> **Performance tip**
>
> In production, for reasons of best performance, users should not build ELPA with OpenMP support if the BLAS/LAPACK library does not suppport threading parallelism.

The number of "ELPA-OpenMP" threads can be set via the `OMP_NUM_THREADS` variable. The corresponding dynamic threading `OMP_DYNAMIC` is currently not supported by ELPA.

Summarizing, the following settings are recommended for optimal performance:

```
export OMP_MAX_ACTIVE_LEVELS=1
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
export MKL_DYNAMIC=TRUE
```

5.5   Using ELPA with MPI

> **Performance tip**
>
> Since main scope of ELPA is massively parallel calculations, ELPA is not optimized for the use with only 1 MPI rank. Using at least 2 MPI ranks is strongly recommended.

5.5.1 Plain MPI mode

ELPA uses MPI to support the distributed-memory parallel execution model which also allows it to scale beyond one compute node. In this case, the distribution of the input matrix as well as the internal data follows the block-cyclic model, same as used by the BLACS and ScaLAPACK libraries. Consequently, before calling ELPA, the user has to set up the BLACS grid and initialize the input matrix accordingly.

1-4. These steps are the same as in Sec. 5.4.1. Here we initialize ELPA and allocate ELPA object. The additional initialization steps needed to set up the MPI and the BLACS grid are sketched in Appendix C.

5. Specify the information about the input matrix. Note that compared to Step 5 in Sec. 5.4.1, there are three additional parameters that must be set, namely the MPI parent communicator, as well as the row and column indices for every processor:

```fortran
Fortran

! size of the input matrix is na x na
call elpaInstance%set("na", na, status)

! number of eigenvectors to be computed, 0 <= nev <= na
call elpaInstance%set("nev", nev, status)

! number of rows of the local part of the distributed matrix
call elpaInstance%set("local_nrows", na_rows, status)

! number of columns of the local part of the distributed matrix
call elpaInstance%set("local_ncols", na_cols, status)

! block size of the BLACS block-cyclic distribution
call elpaInstance%set("nblk", nblk, status)

! the global MPI communicator
call elpaInstance%set("mpi_comm_parent", MPI_COMM_WORLD, status)

! row coordinate of MPI process
call elpaInstance%set("process_row", my_prow, status)

! column coordinate of MPI process
call elpaInstance%set("process_col", my_pcol, status)
```

```cpp
C/C++

elpa_set(elpaInstance, "na", na, &status);
elpa_set(elpaInstance, "nev", nev, &status);
elpa_set(elpaInstance, "local_nrows", na_rows, &status);
elpa_set(elpaInstance, "local_ncols", na_cols, &status);
elpa_set(elpaInstance, "nblk", nblk, &status);
elpa_set(elpaInstance, "mpi_comm_parent", MPI_Comm_c2f(MPI_COMM_WORLD),
    &status);
elpa_set(elpaInstance, "process_row", my_prow, &status);
```

```
    elpa_set(elpaInstance, "process_col", my_pcol, &status);
```

Note that for C/C++ case, the MPI communicator has to be converted to the Fortran integer type using `MPI_Comm_c2f()` function.

From here on, the remaining steps are the same as steps 5 through 8 as outlined in the previous section. For the sake of clarity and to avoid confusion, we include them here as well:

6. Call the `setup()` function to finalize the setting of *mandatory parameters* for the given ELPA object.

Fortran
```
status = elpaInstance%setup()
```

C/C++
```
status = elpa_setup(elpaInstance);
```

7. If desired, set any number of tunable *runtime options*.

Fortran
```
call elpaInstance%set("solver", ELPA_SOLVER_2STAGE, status)
call elpaInstance%set("real_kernel",ELPA_2STAGE_REAL_AVX_BLOCK2, status)
```

C/C++
```
elpa_set(elpaInstance, "solver", ELPA_SOLVER_2STAGE, &status);
elpa_set(elpaInstance,"real_kernel",ELPA_2STAGE_REAL_AVX_BLOCK2,&status);
```

8. Call the solver to obtain eigenvalues `ev(1:na)` and eigenvectors `q(na_rows,na_cols)` of matrix `a(na_rows,na_cols)`.

Fortran
```
call elpaInstance%eigenvectors(a, ev, q, status)
```

C/C++
```
elpa_eigenvectors(elpaInstance, a, ev, q, &status);
```

Here `a`, `q` are the *local* parts of the corresponding distributed matrices; `ev` is the *global* array of eigenvalues – it has to be allocated on each task and the result is available on each task.

> **Important note**
>
> The `elpa_eigenvectors()` routine requires that the local part of the matrix of eigenvectors `q(na_rows,na_cols)` was of the same size as the local part of the input matrix `a(na_rows,na_cols)`, even in the case when only part of the eigenvectors is requested.

> **Important note**
>
> ELPA always assumes the column-major ordering of the local parts of the matrices. In C/C++, to avoid the confusion, we recommend to use 1D arrays/vectors.

9. Clean up ELPA

```Fortran
call elpa_deallocate(elpaInstance, status)
call elpa_uninit(status)
```

```C/C++
elpa_deallocate(elpaInstance, &status);
elpa_uninit(&status);
```

For correctness, keep in mind to also call `mpi_finalize()` at the end of your program.

5.5.2 Hybrid MPI+OpenMP mode

The steps needed to set up the program combine those outlined in Sections 5.4.2 and 5.5.1. Additionally, in case of hybrid MPI and OpenMP execution, it is **mandatory** that your MPI library is thread-compliant, i.e. that it supports the threading levels `MPI_THREAD_SERIALIZED` or `MPI_THREAD_MULTIPLE` (support of `MPI_THREAD_FUNNELED` is not guaranteed). In this case, instead of calling `mpi_init`, you should call `mpi_init_thread`, e.g.:

```Fortran
integer :: thread_level
call MPI_Init_thread(MPI_THREAD_MULTIPLE, thread_level, mpierr)
```

```C/C++
int thread_level;
MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &thread_level);
```

You can check whether your MPI library is thread-compliant e.g. by running one of the ELPA test suite programs, which will warn you if this prerequisite is not met.

If your MPI library is not thread-compliant, ELPA will internally (independent of your applied setting) use only one OpenMP thread, and you will be informed at runtime with a warning. The number of threads used in a threaded implementation of your BLAS library will not be affected by this as long as these threads can be controlled through another method than specifying `OMP_NUM_THREADS` (for instance with Intel MKL library where you can specify `MKL_NUM_THREADS`).

For the optimal performance of ELPA in the hybrid MPI-OpenMP mode, it is important that the combination of the number of MPI tasks and OpenMP threads does not over-subscribe the compute nodes. Also, nested OpenMP regions should be disabled (see Sec. 5.4.2). Last but not least, please also make sure that the MPI tasks, as well as the OpenMP threads per task are pinned in an appropriate way defined for your system. Consequently, the following requirements should be fulfilled:

1. # MPI tasks per node $\times$ # OpenMP threads per task $\leq$ # cores per node

2. Set the number of ELPA-OpenMP threads via the `OMP_NUM_THREADS` variable

3. Set the number of BLAS-OpenMP threads for the math library. For Intel MKL, `MKL_NUM_THREADS` can be set to a value larger than 1 or, preferably, `MKL_DYNAMIC=TRUE` should be used.

4. Process/thread migration should be prevented via correct pinning of MPI tasks and OpenMP threads, but do **not** pin to *hyperthreads*

5.6 Using GPU acceleration

> **Performance tip**
>
> For production runs it is strongly recommeneded to use GPU in the hybrid MPI+GPU execution model with at least of total 2 MPI ranks.

Currently, Nvidia, AMD, and Intel GPUs are supported. You have to make sure that ELPA has been configured with GPU support as explained earlier in Sec. 3.2.

ELPA can be compiled with all parallelization models (MPI, OpenMP, GPU). However, at runtime only either GPU or OpenMP can be used. If both are enabled, then only GPU will be used. We recommend to use ELPA in the hybrid MPI+GPU execution model.

The initial steps needed to set up the program are similar to those outlined in Steps 1-6 of Sec. 5.4.1 for the sequential case or of Sec. 5.5.1 for the MPI case, which conclude by setting up the ELPA object `status=elpaInstance%setup()` and hence finilizing the values of mandatory parameters. Below we emphasize the differencies with respect to the Steps 7-8 specific to the GPU execution model.

7. Set the runtime options to use the GPU.

   Use of GPU in ELPA can be switched on by setting the corresponding runtime option (which can be done after all the mandatory parameters had been set and finalized with), e.g.:

   Fortran
   ```fortran
   ! Choose only one architecture: "nvidia-gpu", "amd-gpu", or "intel-gpu"
   ! 1=on, 0=off
   call elpaInstance%set("nvidia-gpu", 1, status)
   call elpaInstance%set("solver", ELPA_SOLVER_1STAGE, status)
   ```

   C/C++
   ```c
   elpa_set(elpaInstance, "nvidia-gpu", 1, &status);
   elpa_set(elpaInstance, "solver", ELPA_SOLVER_1STAGE, &status);
   ```

   > **Performance tip**
   >
   > ELPA1 is usually the better choice than ELPA2 for the performance on GPU, but if the local matrix size becomes very small, ELPA2 still can be faster.

   For MPI programs, one has to ensure that the number of MPI tasks per GPU device is constant across all GPUs. By default, ELPA will automatically assign each MPI task to a certain GPU device in a round-robin fashion. However, this assignment can also be done manually by setting the `use_gpu_id` runtime option, e.g.:

```
Fortran
! optional step: manually assign the GPU device to each the MPI task
my_gpu_id = mod(myrank, number_of_GPU_devices_per_node)
call elpaInstance%set("use_gpu_id", my_gpu_id, status)
```

```
C/C++
my_gpu_id = myrank%number_of_GPU_devices_per_node;
elpa_set(elpaInstance, "use_gpu_id", my_gpu_id, &status);
```

To finalize the GPU setup one has to call the routine:

```
Fortran
status = elpaInstance%setup_gpu()
```

```
C/C++
status = elpa_setup_gpu(elpaInstance);
```

8. Call the desired ELPA solver routine. There is a special ELPA API that explicitly specifies the data type and it can be used for both host- and device-allocated data:

```
Fortran
call elpaInstance%eigenvectors_double(a_host_or_dev, ev_host_or_dev, &
    z_host_or_dev, status)
```

```
C/C++
elpa_eigenvectors_double(elpaInstance, a_host_or_dev, ev_host_or_dev,
    z_host_or_dev, &status);
```

Here ELPA will automatically detect where the arrays `a_host_or_dev`, `ev_host_or_dev`, `z_host_or_dev` were allocated (either all on host or all on device) and perform the data transfers if needed.

If the data were allocated on host, one can also use the traditional ELPA API:

```
Fortran
call elpaInstance%eigenvectors(a_host, ev_host, z_host, status)
```

```
C/C++
elpa_eigenvectors(elpaInstance, a_host, ev_host, z_host, &status);
```

Here ELPA will automatically detect the *datatype* of the arrays `a_host`, `ev_host`, `z_host` *but they have to be allocated on the host.*

5.6.1 Using GPU streams

For Nvidia and AMD GPUs, it is recommended to use streams to achieve the best performance. They are enabled by default and no special action is needed to use them. If, for some special

reason, the user wants to disable the GPU streams, this has to be done at the `configure` stage by setting `--enable-gpu-streams=no` flag.

5.6.2 Using GPU solver libraries

For Nvidia and AMD GPUs, it is recommended to use solver libraries (cuSOLVER, rocSOLVER) to achieve the best performance for ELPA standard/generalized eigenproblem and auxillary routines.

For Nvidia and AMD GPUs, the solver libraries are enabled by default and no special action is needed to use them. If the user wants to disable the GPU solver libraries (e.g. when the solver libraries are not available), this has to be done at the `configure` stage by setting `--with-cusolver=no` or `--with-rocsolver=no`.

5.6.3 Using NCCL/RCCL communication libraries

To maximize the ELPA performance, it is recommended use vendor-specific communication libraries. The current release of ELPA supports NCCL for Nvidia GPUs and RCCL for AMD GPUs. They can be enabled during the `configure` step by adding the `-enable-gpu-ccl=nccl` or `rccl` flags respectively, for which also the GPU streams (Sec. 5.6.1) have to be enabled. Then ELPA will automatically use NCCL/RCCL for GPU runs.

Important note for the GPU users
If NCCL/RCCL is enabled, the number of MPI tasks per GPU device must be equal to one.

5.6.4 Using several MPI tasks per GPU

If ELPA was installed without NCCL/RCCL support, then, in principle, more than one MPI task per GPU device can be used.

For Nvidia GPUs the performance can be substantially improved if the Nvidia Multi-Process Service (MPS) is activated on each node. The MPS daemon must be started exactly **once** per node. Some batch submission systems take care of this automatically. Check with your system administrator if this feature is provided; otherwise, the following mechanism can be used to set up MPS properly.

In the submission script, here using SLURM just as an example, we call the mpi launcher to run a **wrapper script**. This way, in the wrapper script, the process IDs can be queried where only one process (e.g. process 0) sets up the MPS server:

1. In the job submission script:

```
# set up the environment
...
srun ./wrapper_script.sh
```

2. In the `wrapper_script.sh`:

```
#!/bin/bash
# only process 0 sets up the MPS server:
if [ $SLURM_LOCALID -eq 0 ]; then
  nvidia-cuda-mps-control -d
fi
```

```
# now launch the program
./<your_executable> <input_arguments>
```

More details on ELPA on GPUs and Nvidia MPS can be found here. Analogous service for Intel GPUs Compute Aggregation Layer (CAL) is also available but not yet have been tested by the ELPA team. For AMD GPUs no special service is needed and recent ROCm versions are recommended for good performace.

However, it can happen that the optimal number of MPI processes per GPU for ELPA differs from the optimal number of MPI processes per GPU for the parent application. In this case, the application may use two different MPI communicators, one internal to the application itself which handles all the available physical cores, and one for ELPA containing the optimal number of MPI processes per GPU device. Consequently, the entire input matrix should be redistributed over these MPI processes that call ELPA. When redistributing the matrix, care has to be taken so that the first row and the first column of the redistributed matrix are located on the 0-th processors row and the 0-th processors column, respectively. Of course, it has to be tested whether the additional performance achieved is actually worth the extra effort that goes to communicator splitting and data redistribution. In any case, ELPA works correctly with any desired number of MPI processes per GPU device, and the discussion above is only for the sake of improving the performance.

> **Performance tip**
>
> For optimal GPU performance, we recommend tuning the number of MPI processes per GPU. Typically, using 2–6 MPI processes per GPU device yields the best results, depending on the hardware. In particular, using all available MPI processes can be very inefficient!

5.6.5 Other tips for using ELPA-GPU

- Matrix size

  If matrix size is too small (e.g. $\lesssim 5000 \times 5000$ per GPU), the GPU version of ELPA may be not beneficial over the CPU version.

- ELPA1-GPU vs ELPA2-GPU

  For the GPU calculations, unlike the CPU ones, ELPA1 usually has significantly better performance than ELPA2. The only exception is when the local matrix size becomes very small (e.g. $\lesssim 500 \times 500$ per GPU), where ELPA2 can still be faster. But in such cases, already the CPU version of ELPA should be preferred.

- Explicitly set the compute capability

  If you still need ELPA2 with NVIDIA GPUs, make sure that the configure command also explicitly sets the compute capability variable to the highest level supported by your hardware. For instance, for A100 devices, it should be set to `sm_80`. Please also ensure that, at the end of the configure step, the ELPA2 GPU kernels are listed.

5.7 Using ELPA from Python

In order to use ELPA within your python code, a wrapper has to be generated, which allows you to import ELPA's functionality via a generated shared object. For the generation of the wrapper, several python packages are needed to be installed in your system, notably `mpi4py`, `cython`, and `pytest`. Then during ELPA's `configure` step the additional two flags have to be

provided: `--enable-python` and `--enable-python-tests`. Then, after compilation using `make`, run the install command

```
make install
```

which installs ELPA to `/absolute_path_to_elpa` specified by `--prefix` (as in Sec. 3.4). Upon successfull installation, the following message will be printed:

```
Libraries have been installed in:
   /absolute_path_to_elpa/lib/python3.x/site-packages/pyelpa
```

This path containing the `pyelpa` package must be included in your system's `PYTHONPATH` using e.g.

```
export PYTHONPATH=/absolute_path_to_elpa/lib/python3.x/site-packages:$PYTHONPATH
```

Note that 'x' refers to the minor version of your system's `python3` installation, e.g. `python3.10`. Note also that, similar to using ELPA from a C or Fortran program, the path to where ELPA's shared libraries are generated must be known to the loader at runtime. This can be done using either the `rpath` mechanism, or by adding the path to the `LD_LIBRARY_PATH`. The exact path is `/absolute_path_to_elpa/lib` as described in Sec. 4.2.

Now you should be able to import the shared object into your python code:

```
from pyelpa import DistributedMatrix
```

To actually use ELPA from python, there are a few steps to be taken to set up and solve the problem. The example code included in the ELPA repository under `python/examples/example.py` shows these steps. For the sake of brevity and to avoid repetition, it will not be included here. It is worthwhile to mention however, that as the example shows, there are two different ways to go over the elements of any matrix (in order to for example set up the input matrix). One is labelled to be the easiest yet less efficient where the elements are accessed individually one by one. The other method uses the block structure and is therefore more efficient. The difference in the efficiency of these methods would likely play a major role for accessing the elements of very large matrices.

6   Best practices

6.1   Autotuning for better performance

ELPA's autotuning engine is a powerful utility that can optimize a large number of tunable *runtime options*. Their optimal values can then be used for subsequent runs. This means that to obtain these values, ELPA needs to solve the problem once in order to test and compute the optimal tunable parameters. The first run will likely be sub-optimal however, and, therefore, autotuning is particularly promising if the problem has to be solved repeatedly as is the case of self-consistent methods for instance.

To use this feature, the application code must implement a few steps in similar way as explained earlier in Sec. 5. These steps are explained in the following paragraphs.

1. In your program decleration, declare the following two objects

    ```Fortran
    class(elpa_t), pointer :: elpaInstance
    class(elpa_autotune_t), pointer :: elpaTuneState
    ```

    ```C/C++
    elpa_t elpaInstance;
    elpa_autotune_t elpaTuneState;
    ```

2. Follow the steps needed to set up the problem as explained earlier in Sec. 5

3. Initialize the tuning object

    ```Fortran
    elpaTuneState => elpaInstance%autotune_setup(tuning_level, &
        tuning_domain, status)
    ```

    ```C/C++
    elpaTuneState = elpa_autotune_setup(elpaInstance, tuning_level,
        tuning_domain, &status);
    ```

    There are currently three possible values for the parameter `tuning_level`:

    - `ELPA_AUTOTUNE_FAST`, which includes tuning of the parameters related to the following items: `solver`, `real_kernel`, `complex_kernel`, `omp_threads`

    - `ELPA_AUTOTUNE_MEDIUM`, which, in addition to the above-mentioned parameters, includes the GPU-related ones (`gpu_tridiag`, `gpu_solve_tridi`, `gpu_trans_ev`, `gpu_bandred`, `gpu_trans_ev_tridi_to_band`, `gpu_trans_ev_band_to_full`) and `min_tile_size`

    - `ELPA_AUTOTUNE_EXTENSIVE`, includes all of the above parameters plus the ones related to the following items:
      various blocking factors (`blocking_in_band_to_full`, `blocking_in_multiply`, `blocking_in_cholesky`), `max_stored_rows`, `stripewidth_[real|complex]`,

```
    intermediate_bandwidth
```

Furthermore, there are parameters that are relevant to real or complex problems only, while others are relevant to any problem type. The `tuning_domain` parameter controls whether tuning will be performed for real (`ELPA_AUTOTUNE_DOMAIN_REAL`) problems or complex (`ELPA_AUTOTUNE_DOMAIN_COMPLEX`) problems or for both cases through the parameter choice `ELPA_AUTOTUNE_DOMAIN_ANY`.

The list of all tunable parameters can be obtained using a python script included in your current release. To do so, change directory to the following path

```
cd elpa_dir/utils/parse_index
```

where `elpa_dir` is the main directory that includes all ELPA source files. Next, call the parser, which prints a list of parameter names and their description to standard output:

```
python extract_options.py
```

A complete list of these parameters for the current release is included in Sec. 5.2 and Appendix B along with a discussion on potential impacts of certain parameters on correctness and/or performance wherever necessary.

At this stage, if you wish to remove any of the tunable parameters from the tuning process, you should explicitly set the desired value before going to the next step. The value of the removed parameter is fixed, which speeds up the autotuning process. For example,

Fortran
```fortran
call elpaInstance%set("solver", ELPA_SOLVER_2STAGE, status)
```

C/C++
```c
elpa_set(elpaInstance, "solver", ELPA_SOLVER_2STAGE, &status);
```

will remove the choice of solver from the set of tunable parameters.

4. Construct a loop in which the solver is iteratively called to solve the same problem until the tuning engine converges

   Remember to keep a copy of the input matrix which will have to be used to restore it because the solver overwrites the input.

Fortran
```fortran
iter_max=100

do iter = 1, iter_max
  ! logical :: unfinished,
  unfinished = elpaInstance%autotune_step(elpaTuneState, status)

  if (.not. unfinished) exit ! exit the loop if autotuning is finished

  ! Solve EV problem
  call elpaInstance%eigenvectors(a, ev, q, status)
```

```fortran
   ! Print the current autotune state
   call elpaInstance%autotune_print_state(elpaTuneState)

   ! restore the matrix
   a(:,:) = a_copy(:,:)
 end do
```

```c
  int iter_max = 100;

  for (int iter=1; iter <= iter_max; iter++) {
    int unfinished = elpa_autotune_step(elpaInstance, elpaTuneState,
        &status);

    if (unfinished == 0) break; // exit the loop if autotuning is finished

    // Solve EV problem
    elpa_eigenvectors(elpaInstance, a, ev, q, &status);

    // Print the current autotune state
    elpa_autotune_print_state(elpaInstance, elpaTuneState, &status);

    // restore the matrix
    for (int k = 0; k<na_rows*na_cols; k++) a[k] = a_copy[k];
  }
```

Every iteration of the loop will test a new combination of parameters and the autotuning engine will update the `elpaTuneState` object which carries both the currently testet state and the best state found so far.

5. Set and print the optimal settings

   Once the tuning is done, the converged parameters can be set as the best combination by calling the subroutine `elpaInstance%autotune_set_best()`. Afterwards, repeated calls to the solver will run using the optimal parameters.

Fortran

```fortran
call elpaInstance%autotune_save_state(elpaTuneState, "saved_state.txt", &
    status)

call elpaInstance%autotune_set_best(elpaTuneState, status)

! Print the best combination found by the autotuning
call elpaInstance%autotune_print_best(elpaTuneState, status)
```

C/C++

```c
elpa_autotune_save_state(elpaInstance, elpaTuneState, "saved_state.txt",
    &status);
```

```
elpa_autotune_set_best(elpaInstance, elpaTuneState, &status);

// Print the best combination found by the autotuning
elpa_autotune_print_best(elpaInstance, elpaTuneState, &status);
```

6. Finally, deallocate the objects and finalize the program

Fortran

```
call elpa_autotune_deallocate(elpaTuneState, status)
call elpa_deallocate(elpaInstance, status)
call elpa_uninit(status)
```

C/C++

```
elpa_autotune_deallocate(elpaTuneState, &status);
elpa_deallocate(elpaInstance, &status);
elpa_uninit(&status);
```

6.2   Choosing the optimal BLACS grid

The following information holds for all runs of ELPA as long as MPI is used, including also plain MPI and hybrid MPI+OpenMP runs.

For MPI runs, ELPA requires that matrices are distributed in a BLACS block-cyclic distribution. The BLACS matrix layout representation can be chosen to be either "row-major" or "column-major". The choice migth depend on the requirements of your application. ELPA works with both choices, but for the best performance, it might be necessary that both alternatives are tested. In case there is no special requirement from the application's perspecive, we recommend to use the "column-major" ordering.

Furthermore, the distribution of the MPI processes into a logical, 2D process grid should be specified. This setup is then used to address the BLACS block-cyclic distributed matrix with "row" and "column" processes. ELPA works correctly irrespective of the choice of the 2D processor grid, which is automatically deduced by ELPA from the underlying BLACS grid:

Fortran

```
call blacs_gridinit(ictxt, layout, np_rows, np_cols)
```

C/C++

```
Cblacs_gridinit(&ictxt, layout, np_rows, np_cols);
```

However, the choice of the BLACS matrix layout (column- or row-major) and the 2D BLACS processor grid dimensions (`np_rows, np_cols`) can affect the ELPA performance.

6.2.1   Optimal BLACS grid dimensions

As a rule of thumb, ELPA solvers work best if the 2D BLACS processor grid (internal to ELPA) is quadratic or at least as "quadratic" as possible. For example, using 16 MPI tasks, the setup

(MPI-rows `np_rows=8`, MPI-columns `np_cols=4`) works best. On the other hand, the following (`np_rows, np_cols`) setups work correctly but with less-than-optimal performance:

- (8,2)
- (2,8)
- (16,1) → very bad
- (1,16) → very bad

Especially, very elongated setups with only one process row/column should be avoided. This also implies that the runtime of the solution can be influenced by the number of MPI tasks employed: in some situations it might be beneficial to use less MPI tasks than there are cores available in order to ensure that a well-shaped, (almost-)quadratic 2D grid can be set up. For example, on a hypothetical machine with 13 cores, one should not use all 13 MPI tasks as the only possible combination of `np_rows` and `np_cols` are 1 and 13. Rather, one should use 12 MPI tasks and leave one core idle to obtain a better distribution of 4×3.

The impact is illustrated in Figure 1 where the run-time for the solution of a real matrix (size 10k) with varying number of MPI processes from 2 to 40 is shown. For prime numbers, only very elongated process grids are possible, and a dramatic performance drop can be seen. Note that in all these tests, the choice of the number of processor rows and columns is always as optimal as possible. Please also note that this setup has been tuned to magnify the effect of the processor grid, and the execution times do not correspond to the optimal run-time as ELPA was built with no optimizations for this test.

(a) ELPA 1-stage   (b) ELPA 2-stage
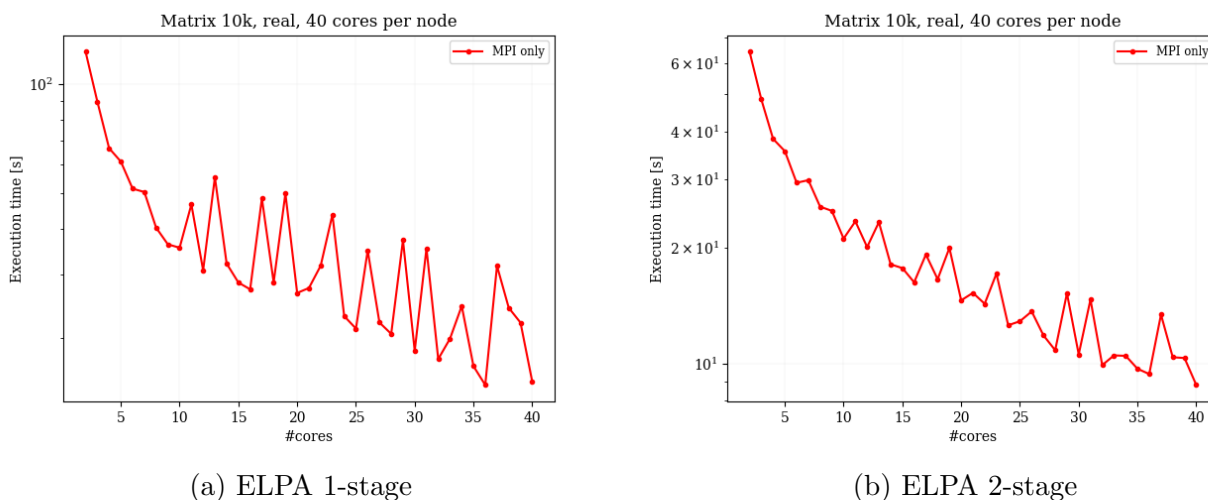
Figure 1: Performance impacts of number of MPI processes and hence the 2D processor grid dimensions

One notable exception to the "most quadratic" setup rule of thumb, is the case of using ELPA1, when the "most quadratic" setup gives the dimensions such that the greatest common divisor of `np_rows` and `np_cols` is 1 or a small number. For example, using 72 MPI tasks, the setup that maximizes the greatest common divisor (`np_rows=6, np_cols=12`) can be better than (`np_rows=8, np_cols=9`).

In case, when the external application has to run with a processor grid which is sub-optimal for ELPA, it might be beneficial to re-distribute the matrix to another processor grid (internal to ELPA) to obtain a better setup.

6.2.2 Optimal BLACS layout

The choice of the BLACS grid layout (column- or row-major) can also affect the ELPA performance both for square and rectangular BLACS grids.

As an example: using 8 MPI processes, a 2D grid can be chosen to have (`np_rows=4`, `np_cols=2`), or (`np_rows=2`, `np_cols=4`) with either column-major layout ("C") or row-major layout ("R"), hence the following combinations are possible:

- `np_rows=4`, `np_cols=2` for column-major layout ("C")

- `np_rows=2`, `np_cols=4` for column-major layout ("C")

- `np_rows=4`, `np_cols=2` for row-major layout ("R")

- `np_rows=2`, `np_cols=4` for row-major layout ("R")

The best setup can depend on many factors, such as the solver used (e.g. ELPA1 vs ELPA2), the hardware and the process pinning. As a rule of thumb, column-major layout ("C") should be preferred over row-major layout ("R"). If unsure, you can test different setups, either directly in your application or using ELPA test programs, which we descibe next.

6.2.3 ELPA test programs to find the best BLACS settings

ELPA comes with test programs located in the ELPA build folder. These programs are compiled when you run the `make` command during the ELPA installation and are located in the `.libs` subdirectory of the build directory; `.libs` has to be also put in `LD_LIBRARY_PATH`. These tests can show how performance is affected if the BLACS grid layout and grid dimensions are not set optimally. For example, you can run:

```
mpiexec -n 8 \
./validate_real_double_eigenvectors_2stage_default_kernel_random_all_layouts \
2000 2000 32
```

The test program calculates eigenvectors for real double-precision random matrix with its elements uniformly distrubuted on $[0, 1]$ interval, using ELPA2 solver with the default kernel and testing all BLACS layouts. Here the values "2000 2000 32" correspond to the matrix-size (`na`), the number of eigenvectors sought (`nev`), and the block size of BLACS block-cyclic distribution (`nblk`) respectively. Consequently, the timings for the solutions of the eigenvalue problem in all possible combinations of the layout and the 2D processor grid will be obtained.

> **Caution!**
>
> Run this test only for small matrix sizes, otherwise the total runtime will be very large, because all layouts will be tested.

The ELPA test programs also provide detailed information about the settings as shown in the excerpt below:

```
...
Matrix size: 2000
Num eigenvectors: 2000
Blocksize: 32
Num MPI proc: 8
Number of processor rows=2, cols=4, total=8
Process layout: C
```

```
| Random matrix block has been set up. (only processor 0 confirms this step)
| Random matrix block has been symmetrized

 The settings in the test program want to use
 ELPA_2STAGE_REAL_AVX2_BLOCK2 kernel
 (This might be overriden with some environment settings)
 /= Group                                          [s]      fraction
 |                                            ============ ============
 |_ e%eigenvectors()                             0.721352      1.000
...
```

By comparing the execution times for (e.g. 0.721352 seconds in the example above), you can find the best BLACS grid settings for your problem.

6.3 Track ELPA timings in your application

ELPA has an internal timer that tracks the time spent in the solver as well as in its individual substeps. To use the timer, several additional steps have to be made on top of the standard ELPA usage described in Sections 5.4-5.5

1. Timer has to be switched on once before the `elpa_setup` is called:

Fortran

```
call elpaInstance%set("timings", 1, status)
status = elpaInstance%setup()
```

C/C++

```
elpa_set(elpaInstance, "timings", 1, &status);
status = elpa_setup(elpaInstance);
```

2. Start the timer before calling the solver and stop it after

Fortran

```
call elpaInstance%timer_start("elpa_eigenvectors")
call elpaInstance%eigenvectors(a, ev, q, status)
call elpaInstance%timer_stop("elpa_eigenvectors")
```

C/C++

```
elpa_timer_start(elpaInstance, (char*) "elpa_eigenvectors");
elpa_eigenvectors(elpaInstance, a, ev, q, &status);
elpa_timer_stop(elpaInstance, (char*) "elpa_eigenvectors");
```

The label `"elpa_eigenvectors"` is arbitrary and can be replaced with any other string.

3. Finally, print out the obtainted timings

Fortran

```
call e%print_times("elpa_eigenvectors")
```

```
    elpa_print_times(elpaInstance, (char*) "elpa_eigenvectors");
```

An excerpt of a sample output is shown below:

```
/= Group                                    [s]      fraction
|                                    ============ ============
|_ elpa_eigenvectors                    11.026278     1.000
  |_ (own)                              0.000004      0.000
  |_ elpa_solve_evp_real_1stage_double  11.026274     1.000
   |_ (own)                             0.000011      0.000
   |_ mpi_communication                 0.000001      0.000
   |_ forward                           10.163898     0.922
    |_ (own)                            0.000004      0.000
    |_ tridiag_real_double              10.163894     1.000
     |_ (own)                            10.052890      0.989
     |_ elpa_transpose_vectors_real_double (999  0.075857       0.007
      |_ (own)                            0.041266      0.544
      |_ mpi_communication (24975x)       0.034591      0.456
     |_ elpa_reduce_add_vectors_real_double (47  0.035147       0.003
      |_ (own)                            0.026825      0.763
      |_ mpi_communication (9486x)        0.008322      0.237
   |_ solve                             0.720247      0.065
    |_ (own)                            0.000006      0.000
    |_ solve_tridi_double               0.720241      1.000
...
```

7   Troubleshooting

If you face any issues with using ELPA, the information in this section will help you find a solution.

7.1   Debugging information

It is very helpful to have debugging information for troubleshooting. To this end, please instruct ELPA to generate the extra details at run-time using the `set()` method after instantiating the elpa object as:

```Fortran
call elpaInstance%set("debug", 1, status)
```

```C/C++
elpa_set(elpaInstance, "debug", 1, &status);
```

Alternatively, if your code does not set the `debug` flag as described above, you can set the environment variable `export ELPA_DEFAULT_debug=1` either in your shell or in the slurm script before running the executable. In the event of an issue, please provide the developers with the reported debug information for troubleshooting. Please also follow the guidelines listed in Section 7.2.

7.2   Reporting bugs and issues

If you run into issues with using ELPA, you are welcome to contact us via elpa-library@mpcdf.mpg.de. However, please note that in order for us to to successfully find a solution as quick as possible, it is important that you provide the following information when you report an issue:

1. Information about the toolchain including which Fortran and C compiler and version as well as which math library were used. If applicable, also which MPI library and version, and which GPU compiler

2. The complete command that was used during the build process. Please **note** that it can be helpful to specify the configure flag `--enable-store-build-config` when configuring ELPA. It will compile the build configuration information into the library object, which can then be querried if needed

3. "config.log" file

4. Information about the input data including matrix type and size

5. Total of MPI processes, MPI per node, MPI tasks per GPU

6. The error message and any extra debug information generated as explained in Sec. 7.1

8 Contributions guide

It has been and continues to be a tremendous effort to develop and maintain the ELPA library. Every help to improve ELPA is highly appreciated.

To open pull requests and issues, please use the ELPA repository on GitHub:
`https://github.com/marekandreas/elpa`
(which is a public mirror of ELPA's official repo `https://gitlab.mpcdf.mpg.de/elpa/elpa`)

For recommendations and suggestions, both for improving the code and the documentation, you can also send an e-mail to elpa-library@mpcdf.mpg.de.

Appendices

A   Expert configure options

Here we list some additional "expert" flags that can be specified during `configure` step. These flags are listed for completeness; they are not needed in typical use cases.

`--enable-optional-argument-in-C-API`

Make the error argument in the C API optional.

Default: disabled

`--with-threading-support-check-during-build=[yes|no]`

Run a small program during configuration to check sufficient threading support of the MPI library. Disable only if launching this test program causes problems, for example, because you are not allowed to run an MPI program on the machine you are compiling ELPA on.

Default: `yes`

`--disable-runtime-threading-support-checks`

Use with caution! Do not verify the required threading support (`MPI_THREAD_SERIALIZED` or `MPI_THREAD_MULTIPLE`) of the MPI library at runtime. Disable only if you have verified the compatibility of the MPI library, otherwise ELPA will yield incorrect results without notification.

Default: enabled

`--disable-allow-thread-limiting`

Use with caution! Do not reduce the number of OpenMP threads to 1 if the MPI library does not offer sufficient threading support (`MPI_THREAD_SERIALIZED` or `MPI_THREAD_MULTIPLE`). Potentially causes incorrect results.

Default: enabled

`--disable-affinity-checking`

Do not run thread affinity checks.

Default: enabled

`--disable-band-to-full-blocking`

Use blocking implementation when transforming from band to full matrix.

Default: enabled

`--enable-autotune-redistribute-matrix`

Experimental! During autotuning, re-distribute the matrix across the MPI ranks to find the optimal block size in the block-cyclic distribution. Requires the corresponding ScaLAPACK functionality.

Default: disabled

`--enable-store-build-config`

Experimental! If enabled, the build config is stored as a binary blop into the ELPA library object file and can be retrieved later for debugging.

Default: disabled

B Expert key-value runtime option pairs for setting the ELPA object

Most commonly used rutime options are described in Sec. 5.2.2. Here we list additional runtime options that are considered to be expert settings. They are not needed in the typical use cases and documented here for completeness.

B.1 General runtime options

The following are general runtime options, some require deeper understanding and should only be used by experts.

`output_build_config`
: *Integer.* If set, and if ELPA has been build to support this the build-config is printed. This keyword is only available if ELPA has been build with `--enable-store-build-config`, otherwise the set and/or get methods return an `ELPA_ERROR_ENTRY_INVALID_VALUE` error.
Default: `0` (= disabled)
Auto-tunable: `no`

`output_pinning_information`
: *Integer.* If set, some information about the pinning of MPI tasks (and potentially OpenMP threads) to cores is printed.
Default: `0` (= disabled)
Auto-tunable: `no`

`matrix_order`
: Either `COLUMN_MAJOR_ORDER` or `ROW_MAJOR_ORDER`. Define the matrix layout to be used when the matrix is re-distributed during autotuning. Only relevant if ELPA has been configured with `--enable-autotune-redistribute-matrix`. In all other cases the matrix layout is automatically deduced by ELPA from the underlying BLACS grid and this parameter is ignored.
Default: `COLUMN_MAJOR_ORDER`
Auto-tunable: `no`

`internal_nblk`
: *Integer.* Block size for the block-cyclic matrix layout used for re-distribution during autotuning. Only relevant if ELPA has been configured with `--enable-autotune-redistribute-matrix`.
Default: `none`
Auto-tunable: `yes`

`gpu`
: **Deprecated.** Enable GPU acceleration using Nvidia GPUs. Please use explicit parameters for the various vendors instead, e.g. 'nvidia-gpu', 'amd-gpu', or 'intel-gpu', since this option is depricated and will be disabled in the future.
Default: `0` (= disabled)
Auto-tunable: `no`

`nvidia-gpu`
: Enable GPU acceleration using Nvidia GPUs.
Default: `0` (= disabled)
Auto-tunable: `yes`

`amd-gpu`
: Enable GPU acceleration using AMD GPUs.
Default: `0` (= disabled)

Auto-tunable: `yes`

`intel-gpu`	Enable GPU acceleration using Intel GPUs.
	Default: `0` (= disabled)
	Auto-tunable: `yes`

`sycl_show_all_devices`

Utilize ALL SYCL devices, not just Level Zero GPUs.
Default: `0` (= disabled)
Auto-tunable: `no`

B.2 Runtime options to control the standard solvers

`solver`	*Integer.* Allows choosing between the ELPA 1stage and 2stage solver. Possible values are "`ELPA_SOLVER_1STAGE`" or "`ELPA_SOLVER_1STAGE`. As a rule of thumb: use the 2stage solver for CPU computations and the 1stage solver for GPU computations (if the matrix size is at least 10000).
	Default: `ELPA_SOLVER_1STAGE`
	Auto-tunable: `yes`
`real_kernel`	*Integer.* Real kernel to use if `solver` is set to `ELPA_SOLVER_2STAGE`.
	Default: set by configuration option `--with-default-real-kernel`
	Auto-tunable: `no`
`complex_kernel`	Complex kernel to use if `solver` is set to `ELPA_SOLVER_2STAGE`.
	Default: set by configuration option `--with-default-complex-kernel`
	Auto-tunable: `no`
`check_pd`	*Integer.* If enabled, before computing the eigenvectors a check is done whether the input matrix is positiv definite (by checking that all eigenvalues are larger then a threshold. If this condition is not satisfied the solver returs without computing the eigenvectors.
	Default: off
	Auto-tunable: `no`
`thres_pd_double`	*Double.* The value of the threshold to be checked in conjunction with the "`check_pd`" keyword.
	Default: 0.00001
	Auto-tunable: `no`
`thres_pd_float`	*Float.* The value of the threshold to be checked in conjunction with the "`check_pd`" keyword.
	Default: 0.00001
	Auto-tunable: `no`
`bandwidth`	*Integer.* If set, the input matrix is assumed to be a band matrix with this bandwidth. Must be a multiple of `nblk`, but at least $2 \cdot$ `nblk`.
	Auto-tunable: `yes`

`intermediate_bandwidth`

Integer. For ELPA2. Intermediate bandwidth used for conversion to band matrix form.

Default: $\max\{64, \texttt{nblk}\}$ for real matrices, $\max\{32, \texttt{nblk}\}$ for complex matrices
Auto-tunable: yes

`min_tile_size`	*Integer.* Minimum tile size used in tridiagonalization (ELPA1) and band reduction (ELPA2). **Default:** $128 \cdot \max\{\texttt{np\_rows}, \texttt{np\_cols}\}$ **Auto-tunable:** yes

`blocking_in_band_to_full`

Integer. For ELPA2. Blocking factor when transforming from band to full matrix. Only relevant if ELPA has been configured with `--enable-band-to-full-blocking`.
Default: 3
Auto-tunable: yes

`max_stored_rows`	*Integer. For ELPA1.* Maximum number of rows stored in ELPA1 backtransformation. **Default:** 256 **Auto-tunable:** yes
`stripewidth_real`	*Integer. For ELPA2.* A parameter of ELPA2 backtransformation. Must be a multiple of 4. **Default:** 48 **Auto-tunable:** yes

`stripewidth_complex`

Integer. For ELPA2. A parameter of ELPA2 backtransformation. Must be a multiple of 8.
Default: 96
Auto-tunable: yes

`qr`	`0` or `1`. *For ELPA2.* Use QR decomposition. Only relevant for real matrices. **Default:** `0` (= disabled) **Auto-tunable:** no
`gpu_tridiag`	*For ELPA1.* Tridiagonalize matrix using GPUs. **Default:** `1` (= enabled) **Auto-tunable:** yes
`gpu_solve_tridi`	*Both ELPA1, ELPA2.* Solve the eigenproblem for tridiagonal matrix on GPUs. **Default:** `1` (= enabled) **Auto-tunable:** yes
`gpu_trans_ev`	*For ELPA1.* Compute eigenvector transformation from tridiagonal to full matrix representation on GPUs. **Default:** `1` (= enabled) **Auto-tunable:** yes
`gpu_bandred`	*For ELPA2.* Compute reduction to band matrix on GPUs. **Default:** `1` (= enabled) **Auto-tunable:** yes

`gpu_trans_ev_tridi_to_band`

For ELPA2. Compute eigenvector transformation from tridiagonal to
band matrix representation on GPUs.
Default: 1 (= enabled)
Auto-tunable: yes

`gpu_trans_ev_band_to_full`

For ELPA2. Compute eigenvector transformation from band to full
matrix representation on GPUs.
Default: 1 (= enabled)
Auto-tunable: yes

B.3   Runtime options to control (parts of) the generalized EVP solvers

Since during the generalized EVP the ELPA 1stage or 2stage solvers are called, the keywords
for the standard EVP also play a role in the computations of the general EVP. All GPU options
`gpu_`... are only relevant for the GPU-builds of ELPA; they are set to 1 by default if GPU device
is found and to 0 otherwise.

`cannon_for_generalized`

0 or 1. Use Cannon's algorithm for the generalized eigenvalue problem.
It's available only if number of columns of MPI grid is a multiple of
number of rows.
Default: 1 (= enabled for CPU-runs) or 0 (= disabled for GPU-runs)
Auto-tunable: no

`cannon_buffer_size`

Integer. If set, use this buffer size for Cannon's algorithm. Larger
buffers potentially accelerate the algorithm, but occupy more memory.
Only relevant if `cannon_for_generalized` is 1. `cannon_buffer_size`
$>= 0$ cannot be used in conjunction with GPUs runs.
Default: 0
Auto-tunable: no

`pxgemm_for_generalized`

0 or 1. Use `elpa_pxgemm_multiply` for the generalized eigenvalue
problem. **Default:** 0 (= disabled for CPU-runs) or 1 (= enabled for
GPU-runs)
Auto-tunable: no

`pxtrmm_for_generalized`

0 or 1. Use ScaLAPACK's `P?TRMM` for the generalized eigenvalue
problem. Can be used only for CPU-runs and relevant only if
`cannon_for_generalized=0` and `pxgemm_for_generalized=0`.
Default: 0 (= disabled)
Auto-tunable: no

`gpu_cannon`

Use Cannon's algorithm for generalized EVP on GPUs.
Default: 1 (= enabled)
Auto-tunable: yes

`gpu_cholesky`

Compute Cholesky factorization on GPUs.
Default: 1 (= enabled)
Auto-tunable: yes

`gpu_hermitian_multiply`

Perform "hermitian" matrix-matrix multiplications $C = A^H B$ on GPUs.
Default: `1` (= enabled)
Auto-tunable: `yes`

`gpu_invert_trm`   Compute inversion of upper triangular matrices on GPUs.
Default: `1` (= enabled)
Auto-tunable: `yes`

`gpu_pxgemm_multiply`
Perform matrix-matrix multiplications via `elpa_pxgemm_multiply` on GPUs.
Default: `1` (= enabled)
Auto-tunable: `yes`

`blocking_in_multiply`
Blocking used in hermitian multiply step.
Default: `31`
Auto-tunable: `yes`

`blocking_in_cholesky`
Blocking used in cholesky step.
Default: `128`
Auto-tunable: `yes`

B.4   Expert runtime options for collective MPI operations

The runtime options in this section control the communication pattern in ELPA. They allow switching from blocking to non-blocking communication (NBC) for collective operations for certain parts of the library. All flags are disabled by default and can be enabled by setting them to `1`.

`nbc_row_global_gather`
Use NBC for rows in `global_gather`.
Auto-tunable: `yes`

`nbc_col_global_gather`
Use NBC for columns in `global_gather`.
Auto-tunable: `yes`

`nbc_row_global_product`
Use NBC for rows in `global_product`.
Auto-tunable: `yes`

`nbc_col_global_product`
Use NBC for columns in `global_product`.
Auto-tunable: `yes`

`nbc_row_solve_tridi`
Use NBC for rows in `solve_tridi`.
Auto-tunable: `yes`

`nbc_row_transpose_vectors`
Use NBC for rows in `transpose_vectors`.
Auto-tunable: `yes`

`nbc_col_transpose_vectors`
Use NBC for columns in `transpose_vectors`.

Auto-tunable: yes

`nbc_row_herm_allreduce`
> Use NBC for rows in `herm_allreduce`.
> **Auto-tunable: yes**

`nbc_col_herm_allreduce`
> Use NBC for columns in `herm_allreduce`.
> **Auto-tunable: yes**

`nbc_row_sym_allreduce`
> Use NBC for rows in `sym_allreduce`.
> **Auto-tunable: yes**

`nbc_col_sym_allreduce`
> Use NBC for columns in `sym_allreduce`.
> **Auto-tunable: yes**

`nbc_row_elpa1_full_to_tridi`
> *For ELPA1.* Use NBC for rows in `tridiag`.
> **Auto-tunable: yes**

`nbc_col_elpa1_full_to_tridi`
> *For ELPA1.* Use NBC for columns in `tridiag`.
> **Auto-tunable: yes**

`nbc_row_elpa1_tridi_to_full`
> *For ELPA1.* Use NBC for rows in `trans_ev`.
> **Auto-tunable: yes**

`nbc_col_elpa1_tridi_to_full`
> *For ELPA1.* Use NBC for columns in `trans_ev`.
> **Auto-tunable: yes**

`nbc_row_elpa2_full_to_band`
> *For ELPA2.* Use NBC for rows in `bandred`.
> **Auto-tunable: yes**

`nbc_col_elpa2_full_to_band`
> *For ELPA2.* Use NBC for columns in `bandred`.
> **Auto-tunable: yes**

`nbc_all_elpa2_band_to_tridi`
> *For ELPA2.* Use NBC in `tridiag_band`.
> **Auto-tunable: yes**

`nbc_row_elpa2_tridi_to_band`
> *For ELPA2.* Use NBC for rows in `trans_ev_tridi_to_band`.
> **Auto-tunable: yes**

`nbc_col_elpa2_tridi_to_band`
> *For ELPA2.* Use NBC for columns in `trans_ev_tridi_to_band`.
> **Auto-tunable: yes**

`nbc_row_elpa2_band_to_full`
> *For ELPA2.* Use NBC for rows in `trans_ev_band_to_full`.
> **Auto-tunable: yes**

`nbc_col_elpa2_band_to_full`

>*For ELPA2.* Use NBC for columns in `trans_ev_band_to_full`.
>**Auto-tunable:** `yes`

`nbc_all_elpa2_redist_band`

>*For ELPA2.* Use NBC in `redist_band`.
>**Auto-tunable:** `yes`

`nbc_all_elpa2_main`

>*For ELPA2.* Use NBC in `elpa_solve_ev`.
>**Auto-tunable:** `yes`

C Initialization of MPI and BLACS

In this Appendix, we provide a minimal example of how to initialize MPI and BLACS for using ELPA. The example is written in Fortran, but the same principles apply to C and C++.

1. Use/include MPI module

Fortran	C/C++
```use mpi```	```#include <mpi.h>```

2. Declare variables for the BLACS context and the ScaLAPACK descriptor

Fortran	C/C++
```integer :: ictxt, sc_desc(9)```	```int ictxt, sc_desc[9];```

3. MPI Initialization

Fortran	C/C++
```call MPI_Init(mpierr)```	```MPI_Init(&argc, &argv);```

4. Select the number of processor rows and columns. The application has to decide how the input matrix should be distributed. The grid setup may be done in an arbitrary way as long as it is consistent, i.e. $0 \le$ `my_prow` $<$ `np_rows`, and $0 \le$ `my_pcol` $<$ `np_cols`, and every process has a unique (`my_prow`, `my_pcol`) coordinate pair. For details see the documentation of `BLACS_Gridinit` and `BLACS_Gridinfo` of your BLACS installation. For better performance, it is recommended to setup the grid such that it is as close to a square grid as possible.

   ```
   np_cols = some value
   np_rows = some value
   ```

5. Set up the BLACS context and MPI communicators. The BLACS context is only necessary for using the ScaLAPACK routines (e.g. `numroc`, see below). For ELPA itself, the MPI communicators along rows and columns are sufficient.

   Fortran
   ```
   call blacs_get(-1, 0, ictxt)
   call blacs_gridinit(ictxt, 'C', np_rows, np_cols)
   call blacs_gridinfo(ictxt, np_rows, np_cols, my_prow, my_pcol)
   ```

   C/C++
   ```
   Cblacs_get(-1, 0, &ictxt);
   Cblacs_gridinit(&ictxt, 'C', np_rows, np_cols);
   Cblacs_gridinfo(ictxt, &np_rows, &np_cols, &my_prow, &my_pcol);
   ```

   'R' or 'C' stands for Row/Column the ordering of the processes in the grid. ELPA works with either of them.

6. For your distributed matrix, compute the number of local rows and columns per MPI task, e.g. with the ScaLAPACK routine `numroc`:

Fortran
```fortran
na_rows = numroc(na, nblk, my_prow, 0, np_rows)
na_cols = numroc(na, nblk, my_pcol, 0, np_cols)
```

C/C++
```c
int izero = 0;
na_rows = numroc_(&na, &nblk, &my_prow, &izero, &np_rows);
na_cols = numroc_(&na, &nblk, &my_pcol, &izero, &np_cols);
```

7. Set up a BLACS descriptor for the target matrix

Fortran
```fortran
call descinit(descA, na, na, nblk, nblk, 0, 0, ictxt, na_rows, info)
if (info /= 0) then
  print *, "Invalid blacs-distribution. Abort!"
  stop 1
endif
```

C/C++
```c
descinit_(descA, &na, &na, &nblk, &nblk, &izero, &izero, &ictxt,
    &na_rows, &info);
if (info != 0) {
  printf("Invalid blacs-distribution. Abort!\n");
  exit(1);
}
```

For ELPA the following restrictions hold:

- block sizes in both directions must be identical (arguments 4 and 5)

- first row and column of the distributed matrix must be on `p_row=0, p_col=0` (arguments 6 and 7)

- the leading dimension of the local matrix must be equal to the number of local rows (argument 9)

- if the eigenvectors are to be calculated, the desciptor for the eigenvector matrix must be identical to the descriptor of the input matrix

D ELPA functions

In this Appendix, we list all ELPA math and auxillary functions and their arguments. This Appendix is a copy of the *man* pages provided with every ELPA installation. They can be invoked by a shell command from the `./man` folder that is located in the elpa root directory, for example:

```
git clone https://gitlab.mpcdf.mpg.de/elpa/elpa.git
cd elpa/man
ls # list all available man pages
man ./elpa_eigenvalues.3
```

for showing the man page for `eigenvalues()` routine.

D.1 `elpa2_print_kernels`

NAME

    elpa2_print_kernels – provides information, which ELPA2 kernels are available on this system.

SYNOPSIS

    elpa2_print_kernels

Description

    Provides information, which ELPA2 kernels are available on this system. It is possible to configure ELPA2 such, that different compute intensive 'ELPA2 kernels' can be chosen at runtime. The service binary elpa2_print_kernels will query the library and tell whether ELPA2 has been configured in this way, and if this is the case which kernels can be chosen at runtime. It will furthermore detail whether ELPA has been configured with OpenMP support.

Options

    none

Author

    A. Marek, MPCDF

Reporting bugs

    Report bugs to the ELPA mail elpa-library@mpcdf.mpg.de

SEE ALSO

    elpa_init(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3) **elpa_cholesky**(3) **elpa_invert_triangular**(3) **elpa_solve_tridiagonal**(3) **elpa_hermitian_multiply**(3) **elpa_uninit**(3) **elpa_deallocate**(3)

D.2 elpa_allocate

NAME
elpa_allocate – allocates an instance of the ELPA library

SYNOPSIS
FORTRAN INTERFACE
use elpa
class(elpa_t), pointer :: elpa_handle

elpa_handle => **elpa_allocate** (error)

With the definitions of the input and output variables:

class(elpa_t)   :: **elpa_handle**
Returns an instance of the ELPA object

integer, optional :: **error**
A returned error code

C/C++ INTERFACE
#include <elpa/elpa.h>
elpa_t elpa_handle;

elpa_t elpa_handle = **elpa_allocate**(**int** *error);

With the definitions of the input and output variables:

elpa_t **elpa_handle**; // returns a handle to the allocated ELPA object

int ***error**;   // a returned error code

DESCRIPTION
Allocate an ELPA object. The function **elpa_init**(3) must be called once
BEFORE **elpa_allocate** can be called.

SEE ALSO
elpa2_print_kernels(1) **elpa_init**(3) **elpa_set**(3) **elpa_setup**(3)
elpa_strerr(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3) **elpa_cholesky**(3)
elpa_invert_triangular(3) **elpa_solve_tridiagonal**(3)
elpa_hermitian_multiply(3) **elpa_uninit**(3) **elpa_deallocate**(3)

D.3 elpa_autotune_deallocate

NAME
      elpa_autotune_deallocate – deallocates an ELPA autotuning instance

SYNOPSIS
  FORTRAN INTERFACE
     use elpa
     class(elpa_t), pointer :: elpa_handle
     class(elpa_autotune_t), pointer :: autotune_handle

     call elpa_handle%**autotune_deallocate** (autotune_handle, error)

     With the definitions of the input and output variables:

     type(elpa_autotune_t) :: **autotune_handle**
         The ELPA autotuning object, created with **elpa_autotune_setup**(3)

     integer, optional   :: **error**
         The returned error code

  C/C++ INTERFACE
     #include <elpa/elpa.h>
     elpa_autotune_t autotune_handle;

     void **elpa_autotune_deallocate** (**elpa_autotune_t** autotune_handle, **int**
     *error);

     With the definitions of the input and output variables:

     elpa_autotune_t **autotune_handle**;
         The handle of an ELPA object, obtained before with
         elpa_autotune_setup(3)

     int ***error**;
         The returned error code

DESCRIPTION
     Deallocates an ELPA autotuning instance. *Prior* to calling the
     elpa_autotune_deallocate method, an ELPA autotuning object must have
     been created. See **elpa_autotune_setup**(3)

SEE ALSO
     elpa_autotune_step(3) **elpa_autotune_setup**(3) **elpa_autotune_deallocate(3)**

D.4   `elpa_autotune_load_state`

NAME
     elpa_autotune_load_state – loads a state of an ELPA autotuning object

SYNOPSIS
  FORTRAN INTERFACE
     use elpa
     class(elpa_t), pointer :: elpa_handle class(elpa_autotune_t), pointer ::
     autotune_handle

     call elpa_handle%**autotune_load_state** (autotune_handle, filename, error)

     With the definitions of the input and output variables:

     class(elpa_t)   :: **elpa_handle**
         An instance of the ELPA object

     class(elpa_autotune_t) :: **autotune_handle**
         An instance of the ELPA autotune object

     character(*)   :: **filename**
         The filename to be used for loading the settings

     integer, optional   :: **error**
         An error return code

  C/C++ INTERFACE
     #include <elpa/elpa.h>
     elpa_t elpa_handle;
     elpa_autotune_t autotune_handle;

     void **elpa_autotune_load_state**(**elpa_t** elpa_handle, **elpa_autotune_t**
     autotune_handle, **const char** *filename, **int** *error);

     With the definitions of the input and output variables:

     elpa_t   **elpa_handle**;
         The handle to the ELPA object

     elpa_autotune_t   **autotune_handle**;
         The handle to the ELPA autotune object

     const char   ***filename**;
         The filename to load the settings

     int   ***error**;
         The error return code

DESCRIPTION
     Loads a previously stored state of an autotune object. With the loaded,
     state the autotuning could be resumed.

SEE ALSO
     elpa_autotune_save_state(3)

D.5 `elpa_autotune_print_state`

NAME
      elpa_autotune_print_state – prints the current state of an ELPA
      autotuning object

SYNOPSIS
  FORTRAN INTERFACE
      use elpa
      class(elpa_t), pointer :: elpa_handle
      class(elpa_autotune_t), pointer :: autotune_handle

      call elpa_handle%**autotune_print_state** (autotune_handle, error)

      With the definitions of the input and output variables:

      class(elpa_t)   :: **elpa_handle**
            An instance of the ELPA object

      class(elpa_autotune_t) :: **autotune_handle**
            An instance of the ELPA autotune object

      integer, optional   :: **error**
            An error return code

  C/C++ INTERFACE
      #include <elpa/elpa.h>
      elpa_t elpa_handle;
      elpa_autotune_t autotune_handle;

      void **elpa_autotune_print_state**(**elpa_t** elpa_handle, **elpa_autotune_t**
      autotune_handle, **int** *error);

      With the definitions of the input and output variables:

      elpa_t **elpa_handle**;
            The handle to the ELPA object

      elpa_autotune_t **autotune_handle**;
            The handle to the ELPA autotune object

      int ***error**;
            The error return code

DESCRIPTION
      Prints the current state of an autotune object.

SEE ALSO
      elpa_autotune_save_state(3) **elpa_autotune_load_state**(3)

D.6 elpa_autotune_save_state

NAME

    elpa_autotune_save_state – saves the current state of an ELPA autotuning object

SYNOPSIS

  FORTRAN INTERFACE

```
use elpa
class(elpa_t), pointer :: elpa_handle
class(elpa_autotune_t), pointer :: autotune_handle

call elpa_handle%autotune_save_state (autotune_handle, filename, error)
```

With the definitions of the input and output variables:

class(elpa_t)   :: **elpa_handle**
    An instance of the ELPA object

class(elpa_autotune_t) :: **autotune_handle**
    An instance of the ELPA autotune object

character(*)   :: **filename**
    The filename to be used for storing the settings

integer, optional   :: **error**
    An error return code

  C/C++ INTERFACE

```
#include <elpa/elpa.h>
elpa_t elpa_handle;
elpa_autotune_t autotune_handle;

void elpa_autotune_save_state(elpa_t elpa_handle, elpa_autotune_t
autotune_handle, char *filename, int *error);
```

With the definitions of the input and output variables:

elpa_t **elpa_handle**;
    The handle to the ELPA object

elpa_autotune_t **autotune_handle**;
    The handle to the ELPA autotune object

char ***filename**;
    The filename to store the settings

int ***error**;
    The error return code

DESCRIPTION

    Saves the current state of an autotune object. The state can be restored with **elpa_autotune_load_state**(3) and the autotuning could be resumed.

SEE ALSO

    elpa_autotune_load_state(3)

D.7 `elpa_autotune_set_best`

NAME
>   elpa_autotune_set_best – sets the tunable parameters to the up-to-now
>   best solution
>   Before the autotuning options can be set, an autotuning step has to be
>   done **elpa_autotune_step**(3)

SYNOPSIS
>   **FORTRAN INTERFACE**
>   use elpa
>   class(elpa_t), pointer :: elpa_handle
>   class(elpa_autotune_t), pointer :: autotune_handle
>
>   call elpa_handle%**autotune_set_best** (autotune_handle)
>
>   With the definitions of the input and output variables:
>
>   type(elpa_autotune_t) :: **autotune_handle**
>   The ELPA autotuning object, created with **elpa_autotune_setup**(3)
>
>   **C/C++ INTERFACE**
>   #include <elpa/elpa.h>
>   elpa_t elpa_handle;
>   elpa_autotune_t autotune_handle;
>
>   void **elpa_autotune_set_best** (**elpa_t** elpa_handle, **elpa_autotune_t**
>   autotune_handle);
>
>   With the definitions of the input and output variables:
>
>   elpa_t **elpa_handle**;
>   The handle of an ELPA object, obtained before with
>   **elpa_allocate**(3)
>   elpa_autotune_t **autotune_handle**;
>   The handle of an ELPA object, obtained before with
>   **elpa_autotune_setup**(3)

DESCRIPTION
>   Sets the up-to-now best options for ELPA tunable parameters. *Prior* to
>   calling the elpa_autotune_set_best method, an ELPA autotuning step must
>   have been performed. See **elpa_autotune_set_best**(3)

SEE ALSO
>   **elpa_autotune_step**(3) **elpa_autotune_setup**(3) **elpa_autotune_deallocate(3)**

D.8 `elpa_autotune_setup`

NAME
    elpa_autotune_setup – creates an instance for autotuning of the ELPA
    library

    Before the autotuning object can be created, an instance of the ELPA
    library has to be setup, see e.g. **elpa_setup**(3)

SYNOPSIS
  FORTRAN INTERFACE
    use elpa
    class(elpa_t), pointer :: elpa_handle
    class(elpa_autotune_t), pointer :: autotune_handle

    autotune_handle= elpa_handle%**autotune_setup** (level, domain)

    With the definitions of the input and output variables:

    integer :: **level**
        The level of the autotuning, at the moment ELPA_AUTOTUNE_FAST is
        supported

    integer :: **domain**
        The domain (real or complex) of the autotuning, can be either
        ELPA_AUTOTUNE_DOMAIN_REAL or ELPA_AUTOTUNE_DOMAIN_COMPLEX

  C/C++ INTERFACE
    #include <elpa/elpa.h>
    elpa_t elpa_handle;
    elpa_autotune_t autotune_handle;

    elpa_autotune_t autotune_handle = **elpa_autotune_setup** (**elpa_t**
    elpa_handle, int level, int domain);

    With the definitions of the input and output variables:

    elpa_t **elpa_handle**;
        The handle of an ELPA object, obtained before with
        elpa_allocate(3)

    int **level**;
        The level of the autotuning, at the moment "ELPA_AUTOTUNE_FAST" is
        supported

    int **domain**;
        The domain (real or complex) of the autotuning, can be either
        "ELPA_AUTOTUNE_DOMAIN_REAL" and "ELPA_AUTOTUNE_DOMAIN_COMPLEX

    elpa_autotune_t **autotune_handle**;
        The created handle of the autotune object

DESCRIPTION
    Creates an ELPA autotuning object. *Prior* to calling the autotune_setup,
    an ELPA object must have been created. See **elpa_setup**(3)

SEE ALSO
    elpa_autotune_step(3) **elpa_autotune_set_best**(3)
    elpa_autotune_deallocate(3)

D.9 `elpa_autotune_step`

NAME
elpa_autotune_step – does one ELPA autotuning step
Before the autotuning step can be done, an instance of the ELPA autotune
object has to be created, see **elpa_autotune_setup**(3)

SYNOPSIS
FORTRAN INTERFACE
```
use elpa
class(elpa_t), pointer :: elpa_handle
class(elpa_autotune_t), pointer :: autotune_handle

unfinished = elpa_handle%autotune_step (autotune_handle)
```

With the definitions of the input and output variables:

type(elpa_autotune_t) :: **autotune_handle**
   The ELPA autotuning object, created with **elpa_autotune_setup**(3)

logical   :: **unfinished**
   Logical, specifying whether autotuning has finished (.false.) or
   not (.true.)

C/C++ INTERFACE
```
#include <elpa/elpa.h>
elpa_t elpa_handle;
elpa_autotune_t autotune_handle;

int unfinished = elpa_autotune_step (elpa_t elpa_handle, elpa_autotune_t
autotune_handle);
```

With the definitions of the input and output variables:

elpa_t **elpa_handle**;
   The handle of an ELPA object, obtained before with
   elpa_allocate(3)

elpa_autotune_t **autotune_handle**;
   The handle of the autotuning object, created with
   elpa_autotune_setup(3)

int **unfinished**;
   Integer, specifying whether autotuning has finished (0) or not (1)

DESCRIPTION
Performs an ELPA autotuning step. *Prior* to calling the autotune_step,
an ELPA autotune object must have been created. See
elpa_autotune_setup(3)

SEE ALSO
elpa_autotune_setup(3) **elpa_autotune_set_best**(3)
elpa_autotune_deallocate(3)

D.10 `elpa_cholesky`

NAME

elpa_cholesky – performs a Cholesky factorization of a real symmetric or complex hermitian positive-definite matrix: A = U^T*U or A = U^H*U, where U is an upper triangular matrix.

There are also variations of this routine that can accept not only host but also device pointers as input/output. Names of these routines explicitly contain the corresponding datatypes: elpa_cholesky_double, elpa_cholesky_float, elpa_cholesky_double_complex, elpa_cholesky_float_complex.

SYNOPSIS

FORTRAN INTERFACE

```
use elpa
class(elpa_t), pointer :: elpa_handle

call elpa_handle%cholesky (a, error)
```

With the definitions of the input and output variables:

datatype :: **a**
> On input: the local part of matrix A that should be decomposed.
> On output: the local part upper triangular matrix U. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The **datatype** of the matrix can be one of "real(kind=c_double)", "real(kind=c_float)", "complex(kind=c_double)", or "complex(kind=c_float)".

integer, optional :: **error**
> The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3)

C/C++ INTERFACE

```
#include <elpa/elpa.h>
elpa_t elpa_handle;

void elpa_cholesky(elpa_t elpa_handle, datatype *a, int *error);
```

With the definitions of the input and output variables:

elpa_t **elpa_handle**;
> The handle to the ELPA object

datatype ***a**;
> On input: the local part of matrix A that should be decomposed.
> On output: the local part of upper triangular matrix U. The dimensions of the matrix must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The **datatype** can be one of "double", "float", "double complex"/"std::complex<double>", "float complex"/"std::complex<float>" for C/C++.

int ***error**;
> The error code of the function. Should be "ELPA_OK". The error codes can be queried with **elpa_strerr**(3)

DESCRIPTION

      Computes the Cholesky decomposition of a real symmetric or complex
      hermitian positive-definite matrix. The functions **elpa_init**(3),
      elpa_allocate(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE*
      elpa_cholesky can be called.

SEE ALSO

      elpa_init(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3)
      elpa_eigenvalues(3) **elpa_eigenvectors**(3) **elpa_invert_triangular**(3)
      elpa_solve_tridiagonal(3) elpa_hermitian_multiply(3) elpa_uninit(3)
      elpa_deallocate(3)

D.11 `elpa_cholesky_double`

NAME
      elpa_cholesky_double – performs a Cholesky factorization of a real
      symmetric positive-definite matrix: A = U^T*U, where U is an upper
      triangular matrix.

SYNOPSIS
  FORTRAN INTERFACE
      use elpa
      class(elpa_t), pointer :: elpa_handle

      call elpa_handle%**cholesky_double** (a, error)

      With the definitions of the input and output variables:

      real(kind=c_double) :: **a**   OR   type(c_ptr) :: **a**
          On input: the host/device local part of matrix A that should be
          decomposed. On output: the host/device local part of upper
          triangular matrix U. The dimensions of matrix **a** must be set *BEFORE*
          with the methods **elpa_set**(3) and **elpa_setup**(3). In case of a GPU
          build **a** can be a pointer to the device memory.

      integer, optional :: **error**
          The return error code of the function. Should be "ELPA_OK". The
          error code can be queried with the function **elpa_strerr**(3)

  C/C++ INTERFACE
      #include <elpa/elpa.h>
      elpa_t elpa_handle;

      void **elpa_cholesky_double**(**elpa_t** elpa_handle, **double** *a, **int** *error);

      With the definitions of the input and output variables:

      elpa_t **elpa_handle**;
          The handle to the ELPA object

      double ***a**;
          On input: the host/device local part of matrix A that should be
          decomposed. On output: the host/device local part of upper
          triangular matrix U. The dimensions of the matrix must be set
          BEFORE with the methods **elpa_set**(3) and **elpa_setup**(3).

      int ***error**;
          The error code of the function. Should be "ELPA_OK". The error
          codes can be queried with **elpa_strerr**(3)

DESCRIPTION
      Computes the Cholesky decomposition of a real symmetric positive-
      definite matrix. The functions **elpa_init**(3), **elpa_allocate**(3),
      elpa_set(3), and **elpa_setup**(3) must be called *BEFORE*
      elpa_cholesky_double can be called.

SEE ALSO
      elpa_init(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3)
      elpa_eigenvalues(3) **elpa_eigenvectors**(3) **elpa_invert_triangular**(3)
      elpa_solve_tridiagonal(3) **elpa_hermitian_multiply(3) elpa_uninit(3)**
      elpa_deallocate(3)

D.12 `elpa_cholesky_double_complex`

NAME

      elpa_cholesky_double_complex – performs a Cholesky factorization of a complex hermitian positive-definite matrix: A = U^H*U, where U is an upper triangular matrix.

SYNOPSIS

  FORTRAN INTERFACE

      use elpa
      class(elpa_t), pointer :: elpa_handle

      call elpa_handle%**cholesky_double_complex** (a, error)

      With the definitions of the input and output variables:

      real(kind=c_double_complex) :: **a**   OR   type(c_ptr) :: **a**
          On input: the host/device local part of matrix A that should be decomposed. On output: the host/device local part of upper triangular matrix U. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). In case of a GPU build **a** can be a pointer to the device memory.

      integer, optional :: **error**
          The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3)

  C/C++ INTERFACE

      #include <elpa/elpa.h>
      elpa_t elpa_handle;

      // C:
      void **elpa_cholesky_double_complex**(elpa_t elpa_handle, **double complex** *a, **int** *error);

      // C++:
      void **elpa_cholesky_double_complex**(elpa_t elpa_handle, **std::complex<double>** *a, **int** *error);

      With the definitions of the input and output variables:

      elpa_t **elpa_handle**;
          The handle to the ELPA object

      double complex ***a**;   OR   std::complex<double> ***a**;
          On input: the host/device local part of matrix A that should be decomposed. On output: the host/device local part of upper triangular matrix U. The dimensions of the matrix must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3).

      int ***error**;
          The error code of the function. Should be "ELPA_OK". The error codes can be queried with **elpa_strerr**(3)

DESCRIPTION
      Computes the Cholesky decomposition of a complex hermitian positive-definite matrix. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE* **elpa_cholesky_double_complex** can be called.

SEE ALSO
      elpa_init(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3) **elpa_invert_triangular**(3) **elpa_solve_tridiagonal(3) elpa_hermitian_multiply(3) elpa_uninit(3) elpa_deallocate(3)**

D.13 `elpa_cholesky_float`

NAME

      elpa_cholesky_float – performs a Cholesky factorization of a real symmetric positive-definite matrix: $A = U^T U$, where U is an upper triangular matrix.

SYNOPSIS

  FORTRAN INTERFACE

      use elpa
      class(elpa_t), pointer :: elpa_handle

      call elpa_handle%**cholesky_float** (a, error)

      With the definitions of the input and output variables:

      real(kind=c_float) :: **a**   OR   type(c_ptr) :: **a**
          On input: the host/device local part of matrix A that should be decomposed. On output: the host/device local part of upper triangular matrix U. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). In case of a GPU build **a** can be a pointer to the device memory.

      integer, optional :: **error**
          The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3)

  C/C++ INTERFACE

      #include <elpa/elpa.h>
      elpa_t elpa_handle;

      void **elpa_cholesky_float**(**elpa_t** elpa_handle, **float** *a, **int** *error);

      With the definitions of the input and output variables:

      elpa_t **elpa_handle**;
          The handle to the ELPA object

      float ***a**;
          On input: the host/device local part of matrix A that should be decomposed. On output: the host/device local part of upper triangular matrix U. The dimensions of the matrix must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3).

      int ***error**;
          The error code of the function. Should be "ELPA_OK". The error codes can be queried with **elpa_strerr**(3)

DESCRIPTION

      Computes the Cholesky decomposition of a real symmetric positive-definite matrix. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE* **elpa_cholesky_float** can be called.

SEE ALSO

elpa_init(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3)
elpa_eigenvalues(3) **elpa_eigenvectors**(3) **elpa_invert_triangular**(3)
elpa_solve_tridiagonal(3) **elpa_hermitian_multiply(3) elpa_uninit(3)**
elpa_deallocate(3)

D.14 `elpa_cholesky_float_complex`

NAME

elpa_cholesky_float_complex – performs a Cholesky factorization of a complex hermitian positive-definite matrix: A = U^H*U, where U is an upper triangular matrix.

SYNOPSIS

FORTRAN INTERFACE

```
use elpa
class(elpa_t), pointer :: elpa_handle

call elpa_handle%cholesky_float_complex (a, error)
```

With the definitions of the input and output variables:

real(kind=c_float_complex) :: **a**   OR   type(c_ptr) :: **a**
    On input: the host/device local part of matrix A that should be decomposed. On output: the host/device local part of upper triangular matrix U. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). In case of a GPU build **a** can be a pointer to the device memory.

integer, optional :: **error**
    The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3)

C/C++ INTERFACE

```
#include <elpa/elpa.h>
elpa_t elpa_handle;

// C:
void elpa_cholesky_float_complex(elpa_t elpa_handle, float complex *a,
int *error);

// C++:
void elpa_cholesky_float_complex(elpa_t elpa_handle,
std::complex<float> *a, int *error);
```

With the definitions of the input and output variables:

elpa_t **elpa_handle**;
    The handle to the ELPA object

float complex ***a**;   OR   std::complex<float> ***a**;
    On input: the host/device local part of matrix A that should be decomposed. On output: the host/device local part of upper triangular matrix U. The dimensions of the matrix must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3).

int ***error**;
    The error code of the function. Should be "ELPA_OK". The error codes can be queried with **elpa_strerr**(3)

DESCRIPTION

Computes the Cholesky decomposition of a complex hermitian positive-
definite matrix. The functions **elpa_init**(3), **elpa_allocate**(3),
elpa_set(3), and **elpa_setup**(3) must be called *BEFORE*
elpa_cholesky_float_complex can be called.

SEE ALSO

 elpa_init(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3)
elpa_eigenvalues(3) **elpa_eigenvectors**(3) **elpa_invert_triangular**(3)
**elpa_solve_tridiagonal(3) elpa_hermitian_multiply(3) elpa_uninit(3)
elpa_deallocate(3)**

D.15 `elpa_deallocate`

NAME
elpa_deallocate – deallocates an instance of the ELPA library after usage

SYNOPSIS
　FORTRAN INTERFACE
```
use elpa
class(elpa_t), pointer :: elpa_handle

call elpa_deallocate (elpa_handle, error)

With the definitions of the input and output variables:


class(elpa_t)      :: elpa_handle
      The pointer to the instance of the ELPA library that is to be
      deallocated
integer, optional :: error
      The returned error code
```

　C/C++ INTERFACE
```
#include <elpa/elpa.h>
elpa_t elpa_handle;

void elpa_deallocate(elpa_t elpa_handle, int *error);

With the definitions of the input and output variables:


elpa_t  elpa_handle;
      The handle to the ELPA instance which should be deallocated.
int  *error;
      The returned error code
```

DESCRIPTION
Deallocate an ELPA object. The functions **elpa_init**(3) and **elpa_allocate**(3) must have been called *BEFORE* **elpa_deallocate** can be called.

SEE ALSO
elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3)
elpa_setup(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3)
elpa_cholesky(3) **elpa_invert_triangular**(3) **elpa_solve_tridiagonal**(3)
elpa_hermitian_multiply(3) **elpa_uninit**(3)

D.16 `elpa_eigenvalues`

NAME

elpa_eigenvalues – computes all eigenvalues λ for a real symmetric or complex hermitian eigenproblem: A*Q = λ*Q

There are also variations of this routine that can accept not only host but also device pointers as input/output. Names of these routines explicitly contain the corresponding datatypes: elpa_eigenvalues_double, elpa_eigenvalues_float, elpa_eigenvalues_double_complex, elpa_eigenvalues_float_complex.

SYNOPSIS

  FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa_handle

call elpa_handle%**eigenvalues** (a, ev, error)

With the definitions of the input and output variables:

class(elpa_t) :: **elpa_handle**
    An instance of the ELPA object.

datatype :: **a**
    The local part of matrix A for which the eigenvalues should be computed. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The **datatype** of the matrix can be one of "real(kind=c_double)", "real(kind=c_float)", "complex(kind=c_double)", or "complex(kind=c_float)". The global matrix has to be symmetric or hermitian, this is not checked by the routine.

datatype_real :: **ev**
    The global array where the eigenvalues λ will be stored in *ascending* order. The **datatype_real** of **ev** can be either "real(kind=c_double)" or "real(kind=c_float)", depending of the **datatype** of the matrix. Note that complex hermitian matrices also have real eigenvalues.

integer, optional :: **error**
    The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3)

  C/C++ INTERFACE

#include <elpa/elpa.h>
elpa_t elpa_handle;

void **elpa_eigenvalues**(**elpa_t** elpa_handle, **datatype** *a, **datatype_real** *ev, **int** *error);

With the definitions of the input and output variables:

elpa_t **elpa_handle**;
    The handle to the ELPA object

datatype ***a;**
>   The local part of matrix A for which the eigenvalues should be
>   computed. The dimensions of the matrix must be set *BEFORE* with the
>   methods **elpa_set**(3) and **elpa_setup**(3). The **datatype** can be one of
>   "double", "float", "double complex"/"std::complex<double>", "float
>   complex"/"std::complex<float>" for C/C++. The global matrix has to
>   be symmetric or hermitian, this is not checked by the routine.

datatype_real ***ev;**
>   The global array where the eigenvalues λ will be stored in
>   *ascending* order. The **datatype_real** of the vector **ev** can be either
>   "double" or "float", depending of the **datatype** of the matrix. Note
>   that complex hermitian matrices also have real eigenvalues.

int ***error;**
>   The error code of the function. Should be "ELPA_OK". The error
>   codes can be queried with **elpa_strerr**(3)

DESCRIPTION
>   Computes the eigenvalues of a real symmetric or complex hermitian
>   matrix. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and
>   **elpa_setup**(3) must be called *BEFORE* **elpa_eigenvalues** can be called.

SEE ALSO
>   **elpa2_print_kernels**(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3)
>   **elpa_setup**(3) **elpa_strerr**(3) **elpa_skew_eigenvalues**(3)
>   **elpa_eigenvectors**(3) **elpa_skew_eigenvectors**(3) **elpa_cholesky**(3)
>   **elpa_invert_triangular**(3) **elpa_solve_tridiagonal**(3) **elpa_eigenvalues**(3)
>   **elpa_uninit**(3) **elpa_deallocate**(3)

D.17   `elpa_eigenvalues_double`

NAME

   elpa_eigenvalues_double – computes all eigenvalues λ for a real
   symmetric eigenproblem: $A*Q = \lambda*Q$

SYNOPSIS

  FORTRAN INTERFACE

   use elpa
   class(elpa_t), pointer :: elpa_handle

   call elpa_handle%**eigenvalues_double** (a, ev, error)

   With the definitions of the input and output variables:

   class(elpa_t) :: **elpa_handle**
        An instance of the ELPA object.

   real(kind=c_double) :: **a**   OR   type(c_ptr) :: **a**
        The local part of the host/device matrix A for which the
        eigenvalues should be computed. The dimensions of matrix **a** must be
        set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The
        global matrix has to be symmetric, this is not checked by the
        routine. In case of a GPU build **a** can be a pointer to the device
        memory.

   real(kind=c_double) :: **ev**   OR   type(c_ptr) :: **ev**
        The global array where the eigenvalues λ will be stored in
        ascending order. In case of a GPU build **ev** can be a pointer to the
        device memory.

   integer, optional :: **error**
        The return error code of the function. Should be "ELPA_OK". The
        error code can be queried with the function **elpa_strerr**(3)

  C/C++ INTERFACE

   #include <elpa/elpa.h>
   elpa_t elpa_handle;

   void **elpa_eigenvalues_double**(**elpa_t** elpa_handle, **double** *a, **double** *ev,
   int *error);

   With the definitions of the input and output variables:

   elpa_t   **elpa_handle**;
        The handle to the ELPA object

   double   ***a**;
        The local part of the host/device matrix A for which the
        eigenvalues should be computed. The dimensions of matrix **a** must be
        set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The
        global matrix A has to be symmetric, this is not checked by the
        routine. In case of a GPU build **a** can be a pointer to the device
        memory.

   datatype_real   ***ev**;
        The array where the eigenvalues λ will be stored in *ascending*
        order. Eigenvalues will be stored in *ascending* order. In case of a

GPU build **ev** can be a pointer to the device memory.

int   ***error;**
      The error code of the function. Should be "ELPA_OK". The error
      codes can be queried with **elpa_strerr**(3)

DESCRIPTION
    Computes the eigenvalues of a double precision real symmetric matrix.
    The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and
    elpa_setup(3) must be called *BEFORE* **elpa_eigenvalues_double** can be
    called.

SEE ALSO
    elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3)
    elpa_setup(3) **elpa_strerr**(3) **elpa_skew_eigenvalues**(3)
    elpa_eigenvectors(3) **elpa_skew_eigenvectors**(3) **elpa_cholesky**(3)
    elpa_invert_triangular(3) **elpa_solve_tridiagonal**(3) **elpa_eigenvalues**(3)
    elpa_uninit(3) **elpa_deallocate**(3)

D.18 `elpa_eigenvalues_double_complex`

NAME
     elpa_eigenvalues_double_complex – computes all eigenvalues λ for a
     complex hermitian eigenproblem: A*Q = λ*Q

SYNOPSIS
  FORTRAN INTERFACE
     use elpa
     class(elpa_t), pointer :: elpa_handle

     call elpa_handle%**eigenvalues_double_complex** (a, ev, error)

     With the definitions of the input and output variables:

     class(elpa_t) :: **elpa_handle**
          An instance of the ELPA object.

     real(kind=c_double_complex) :: **a**   OR   type(c_ptr) :: **a**
          The local part of the host/device matrix A for which the
          eigenvalues should be computed. The dimensions of matrix **a** must be
          set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The
          global matrix has to be hermitian, this is not checked by the
          routine. In case of a GPU build **a** can be a pointer to the device
          memory.

     real(kind=c_double) :: **ev**   OR   type(c_ptr) :: **ev**
          The global array where the eigenvalues λ will be stored in
          ascending order. In case of a GPU build **ev** can be a pointer to the
          device memory. Note that complex hermitian matrices also have real
          eigenvalues.

     integer, optional :: **error**
          The return error code of the function. Should be "ELPA_OK". The
          error code can be queried with the function **elpa_strerr**(3)

  C/C++ INTERFACE
     #include <elpa/elpa.h>
     elpa_t elpa_handle;

     // C:
     void **elpa_eigenvalues_double_complex**(**elpa_t** elpa_handle, **double complex**
     *a, **double** *ev, **int** *error);

     // C++:
     void **elpa_eigenvalues_double_complex**(**elpa_t** elpa_handle,
     std::complex<double> *a, **double** *ev, **int** *error);

     With the definitions of the input and output variables:

     elpa_t **elpa_handle**;
          The handle to the ELPA object

     double complex ***a**;   OR   std::complex<double> ***a**;
          The local part of the host/device matrix A for which the
          eigenvalues should be computed. The dimensions of matrix **a** must be
          set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The

global matrix A has to be hermitian, this is not checked by the
routine. In case of a GPU build **a** can be a pointer to the device
memory.

double   ***ev;**
>   The global array where the eigenvalues λ will be stored in
>   *ascending* order. In case of a GPU build **ev** can be a pointer to the
>   device memory. Note that complex hermitian matrices also have real
>   eigenvalues.

int   ***error;**
>   The error code of the function. Should be "ELPA_OK". The error
>   codes can be queried with **elpa_strerr**(3)

DESCRIPTION
>   Computes the eigenvalues of a complex hermitian matrix. The functions
>   **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be
>   called *BEFORE* **elpa_eigenvalues_double_complex** can be called.

SEE ALSO
>   **elpa2_print_kernels**(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3)
>   **elpa_setup**(3) **elpa_strerr**(3) **elpa_skew_eigenvalues**(3)
>   **elpa_eigenvectors**(3) **elpa_skew_eigenvectors**(3) **elpa_cholesky**(3)
>   **elpa_invert_triangular**(3) **elpa_solve_tridiagonal**(3) **elpa_eigenvalues**(3)
>   **elpa_uninit**(3) **elpa_deallocate**(3)

D.19 `elpa_eigenvalues_float`

NAME

elpa_eigenvalues_float – computes all eigenvalues λ for a real symmetric eigenproblem: A*Q = λ*Q

SYNOPSIS

FORTRAN INTERFACE

```
use elpa
class(elpa_t), pointer :: elpa_handle
```

```
call elpa_handle%eigenvalues_float (a, ev, error)
```

With the definitions of the input and output variables:

```
class(elpa_t) :: elpa_handle
```
An instance of the ELPA object.

```
real(kind=c_float) :: a        OR     type(c_ptr) :: a
```
The local part of the host/device matrix A for which the eigenvalues should be computed. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The global matrix has to be symmetric, this is not checked by the routine. In case of a GPU build **a** can be a pointer to the device memory.

```
real(kind=c_float) :: ev        OR     type(c_ptr) :: ev
```
The global array where the eigenvalues λ will be stored in *ascending* order. In case of a GPU build **ev** can be a pointer to the device memory.

```
integer, optional :: error
```
The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3)

C/C++ INTERFACE

```
#include <elpa/elpa.h>
elpa_t elpa_handle;
```

```
void elpa_eigenvalues_float(elpa_t elpa_handle, float *a, float *ev, int *error);
```

With the definitions of the input and output variables:

```
elpa_t  elpa_handle;
```
The handle to the ELPA object

```
float  *a;
```
The local part of the host/device matrix A for which the eigenvalues should be computed. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The global matrix A has to be symmetric, this is not checked by the routine. In case of a GPU build **a** can be a pointer to the device memory.

```
datatype_real  *ev;
```
The array where the eigenvalues λ will be stored in *ascending* order. Eigenvalues will be stored in *ascending* order. In case of a

GPU build **ev** can be a pointer to the device memory.

int   ***error;**
      The error code of the function. Should be "ELPA_OK". The error
      codes can be queried with **elpa_strerr**(3)

DESCRIPTION
    Computes the eigenvalues of a float precision real symmetric matrix. The
    functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3)
    must be called *BEFORE* **elpa_eigenvalues_float** can be called.

SEE ALSO
    elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3)
    elpa_setup(3) **elpa_strerr**(3) **elpa_skew_eigenvalues**(3)
    elpa_eigenvectors(3) **elpa_skew_eigenvectors**(3) **elpa_cholesky**(3)
    elpa_invert_triangular(3) **elpa_solve_tridiagonal**(3) **elpa_eigenvalues**(3)
    elpa_uninit(3) **elpa_deallocate**(3)

D.20 `elpa_eigenvalues_float_complex`

NAME

      elpa_eigenvalues_float_complex – computes all eigenvalues λ for a
      complex hermitian eigenproblem: A*Q = λ*Q

SYNOPSIS

  FORTRAN INTERFACE

      use elpa
      class(elpa_t), pointer :: elpa_handle

      call elpa_handle%**eigenvalues_float_complex** (a, ev, error)

      With the definitions of the input and output variables:

      class(elpa_t) :: **elpa_handle**
          An instance of the ELPA object.

      real(kind=c_float_complex) :: **a**   OR   type(c_ptr) :: **a**
          The local part of the host/device matrix A for which the
          eigenvalues should be computed. The dimensions of matrix **a** must be
          set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The
          global matrix has to be hermitian, this is not checked by the
          routine. In case of a GPU build **a** can be a pointer to the device
          memory.

      real(kind=c_float) :: **ev**   OR   type(c_ptr) :: **ev**
          The global array where the eigenvalues λ will be stored in
          ascending order. In case of a GPU build **ev** can be a pointer to the
          device memory. Note that complex hermitian matrices also have real
          eigenvalues.

      integer, optional :: **error**
          The return error code of the function. Should be "ELPA_OK". The
          error code can be queried with the function **elpa_strerr**(3)

  C/C++ INTERFACE

      #include <elpa/elpa.h>
      elpa_t elpa_handle;

      // C:
      void **elpa_eigenvalues_float_complex**(elpa_t elpa_handle, **float complex**
      *a, **float** *ev, **int** *error);

      // C++:
      void **elpa_eigenvalues_float_complex**(elpa_t elpa_handle,
      std::complex<float> *a, **float** *ev, **int** *error);

      With the definitions of the input and output variables:

      elpa_t **elpa_handle**;
          The handle to the ELPA object

      float complex ***a**;   OR   std::complex<float> ***a**;
          The local part of the host/device matrix A for which the
          eigenvalues should be computed. The dimensions of matrix **a** must be
          set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The

           global matrix A has to be hermitian, this is not checked by the routine. In case of a GPU build **a** can be a pointer to the device memory.

   float ***ev;**
           The global array where the eigenvalues λ will be stored in *ascending* order. In case of a GPU build **ev** can be a pointer to the device memory. Note that complex hermitian matrices also have real eigenvalues.

   int ***error;**
           The error code of the function. Should be "ELPA_OK". The error codes can be queried with **elpa_strerr**(3)

DESCRIPTION

     Computes the eigenvalues of a complex hermitian matrix. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE* **elpa_eigenvalues_float_complex** can be called.

SEE ALSO

     elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3)
     elpa_setup(3) **elpa_strerr**(3) **elpa_skew_eigenvalues**(3)
     elpa_eigenvectors(3) **elpa_skew_eigenvectors**(3) **elpa_cholesky**(3)
     elpa_invert_triangular(3) **elpa_solve_tridiagonal**(3) **elpa_eigenvalues**(3)
     elpa_uninit(3) **elpa_deallocate**(3)

D.21 `elpa_eigenvectors`

NAME

elpa_eigenvectors – computes all eigenvalues and (part of) the eigenvectors for a real symmetric or complex hermitian eigenproblem: A*Q = λ*Q

There are also variations of this routine that can accept not only host but also device pointers as input/output. Names of these routines explicitly contain the corresponding datatypes: elpa_eigenvectors_double, elpa_eigenvectors_float, elpa_eigenvectors_double_complex, elpa_eigenvectors_float_complex.

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa_handle

call elpa_handle%**eigenvectors** (a, ev, q, error)

With the definitions of the input and output variables:

class(elpa_t) :: **elpa_handle**
An instance of the ELPA object.

datatype :: **a**
The local part of matrix A for which the eigenvalues and (part of) eigenvectors should be computed. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The **datatype** of the matrix can be one of "real(kind=c_double)", "real(kind=c_float)", "complex(kind=c_double)", or "complex(kind=c_float)". The global matrix has to be symmetric or hermitian, this is not checked by the routine.

datatype_real :: **ev**
The global array where the eigenvalues λ will be stored in *ascending* order. The **datatype_real** of **ev** can be either "real(kind=c_double)" or "real(kind=c_float)", depending of the **datatype** of the matrix. Note that complex hermitian matrices also have real eigenvalues.

datatype :: **q**
The storage space for the computed eigenvectors Q. The number of requested eigenvectors must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The **datatype** of the matrix can be one of "real(kind=c_double)", "real(kind=c_float)", "complex(kind=c_double)", or "complex(kind=c_float)".

integer, optional :: **error**
The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3).

C/C++ INTERFACE

#include <elpa/elpa.h>
elpa_t elpa_handle;

void **elpa_eigenvectors**(**elpa_t** elpa_handle, **datatype** *a, **datatype_real** *ev, **datatype** *q, **int** *error);

With the definitions of the input and output variables:

elpa_t **elpa_handle**;
>   The handle to the ELPA object

datatype ***a**;
>   The local part of matrix A for which the eigenvalues and (part of)
>   eigenvectors should be computed. The dimensions of the matrix must
>   be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The
>   **datatype** can be one of "double", "float", "double
>   complex"/"std::complex<double>", "float
>   complex"/"std::complex<float>" for C/C++. The global matrix has to
>   be symmetric or hermitian, this is not checked by the routine.

datatype_real ***ev**;
>   The global array where the eigenvalues λ will be stored in
>   *ascending* order. The **datatype_real** can be either "double" or
>   "float", depending of the **datatype** of the matrix. Note that the
>   eigenvalues of complex hermitian matrices are also real.

datatype ***q**;
>   The storage space for the computed eigenvectors Q. The number of
>   requested eigenvectors must be set *BEFORE* with the methods
>   **elpa_set**(3) and **elpa_setup**(3). The **datatype** can be one of
>   "double", "float", "double complex"/"std::complex<double>", "float
>   complex"/"std::complex<float>" for C/C++.

int ***error**;
>   The error code of the function. Should be "ELPA_OK". The error
>   codes can be queried with **elpa_strerr**(3)

DESCRIPTION

>   Computes the eigenvalues and (part of) the eigenvector spectrum of a
>   real symmetric or complex hermitian matrix. The functions **elpa_init**(3),
>   **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE*
>   **elpa_eigenvectors** can be called. In particular, the number of
>   eigenvectors to be computed, "nev", must be set with **elpa_set**(3).

SEE ALSO

>   **elpa2_print_kernels**(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3)
>   **elpa_setup**(3) **elpa_strerr**(3) **elpa_eigenvalues**(3)
>   **elpa_skew_eigenvalues**(3) **elpa_skew_eigenvectors**(3) **elpa_cholesky**(3)
>   **elpa_invert_triangular**(3) **elpa_solve_tridiagonal**(3)
>   **elpa_hermitian_multiply**(3) **elpa_uninit**(3) **elpa_deallocate**(3)

D.22 `elpa_eigenvectors_double`

NAME

    elpa_eigenvectors_double – computes all eigenvalues and (part of) the
    eigenvectors for a real symmetric eigenproblem: A*Q = λ*Q

SYNOPSIS

  FORTRAN INTERFACE

    use elpa
    class(elpa_t), pointer :: elpa_handle

    call elpa_handle%**eigenvectors_double** (a, ev, q, error)

    With the definitions of the input and output variables:

    class(elpa_t) :: **elpa_handle**
        An instance of the ELPA object.

    real(kind=c_double) :: **a**   OR   type(c_ptr) :: **a**
        The local part of the host/device matrix A for which all
        eigenvalues and (part of) eigenvectors should be computed. The
        dimensions of matrix **a** must be set *BEFORE* with the methods
        elpa_set(3) and **elpa_setup**(3). The global matrix has to be
        symmetric, this is not checked by the routine. In case of a GPU
        build **a** can be a pointer to the device memory.

    real(kind=c_double) :: **ev**   OR   type(c_ptr) :: **ev**
        The host/device vector **ev** where the eigenvalues will be stored in
        ascending order. In case of a GPU build **ev** can be a pointer to the
        device memory.

    real(kind=c_double) :: **q**   OR   type(c_ptr) :: **q**
        The host/device storage space for the computed eigenvectors. The
        number of requested eigenvectors must be set *BEFORE* with the
        methods **elpa_set**(3) and **elpa_setup**(3). In case of a GPU build **q**
        can be a pointer to the device memory.

    integer, optional :: **error**
        The return error code of the function. Should be "ELPA_OK". The
        error code can be queried with the function **elpa_strerr**(3)

  C/C++ INTERFACE

    #include <elpa/elpa.h>
    elpa_t elpa_handle;

    void **elpa_eigenvectors_double**(**elpa_t** elpa_handle, **double** *a, **double** *ev,
    double *q, **int** *error);

    With the definitions of the input and output variables:

    elpa_t **elpa_handle**;
        The handle to the ELPA object

    double ***a**;
        The local part of the host/device matrix A for which the
        eigenpairs should be computed. The dimensions of the matrix must
        be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The
        global matrix has to be symmetric, this is not checked by the

routine. In case of a GPU build **a** can be a pointer to the device
memory.

double   ***ev;**
> The host/device storage for the computed eigenvalues. Eigenvalues
> will be stored in *ascending* order. In case of a GPU build **ev** can
> be a pointer to the device memory.

double   ***q;**
> The host/device storage space for the computed eigenvectors. The
> number of requested eigenvectors must be set *BEFORE* with the
> methods **elpa_set**(3) and **elpa_setup**(3). In case of a GPU build **q**
> can be a pointer to the device memory.

int   ***error;**
> The error code of the function. Should be "ELPA_OK". The error
> codes can be queried with **elpa_strerr**(3)

DESCRIPTION

Computes the eigenvalues and (part of) the eigenvectors of a real
symmetric matrix. The functions **elpa_init**(3), **elpa_allocate**(3),
elpa_set(3), and **elpa_setup**(3) must be called *BEFORE*
elpa_eigenvectors_double can be called. In particular, the number of
eigenvectors to be computed, "nev", must be set with **elpa_set**(3).

SEE ALSO

elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3)
elpa_setup(3) **elpa_strerr**(3) **elpa_eigenvalues**(3)
elpa_skew_eigenvalues(3) **elpa_skew_eigenvectors**(3) **elpa_cholesky**(3)
elpa_invert_triangular(3) **elpa_solve_tridiagonal**(3)
elpa_hermitian_multiply(3) **elpa_uninit**(3) **elpa_deallocate**(3)

D.23 `elpa_eigenvectors_double_complex`

NAME

  elpa_eigenvectors_double_complex – computes all eigenvalues and (part
  of) the eigenvectors for a complex hermitian eigenproblem: A*Q = λ*Q

SYNOPSIS

  FORTRAN INTERFACE

  use elpa
  class(elpa_t), pointer :: elpa_handle

  call elpa_handle%**eigenvectors_double_complex** (a, ev, q, error)

  With the definitions of the input and output variables:

  class(elpa_t) :: **elpa_handle**
      An instance of the ELPA object.

  real(kind=c_double_complex) :: **a**   OR   type(c_ptr) :: **a**
      The local part of the host/device matrix A for which all
      eigenvalues and (part of) eigenvectors should be computed. The
      dimensions of matrix **a** must be set *BEFORE* with the methods
      elpa_set(3) and **elpa_setup**(3). The global matrix has to be
      symmetric, this is not checked by the routine. In case of a GPU
      build **a** can be a pointer to the device memory.

  real(kind=c_double) :: **ev**   OR   type(c_ptr) :: **ev**
      The host/device vector **ev** where the eigenvalues will be stored in
      ascending order. In case of a GPU build **ev** can be a pointer to the
      device memory. Note that complex hermitian matrices also have real
      eigenvalues.

  real(kind=c_double_complex) :: **q**   OR   type(c_ptr) :: **q**
      The host/device storage space for the computed eigenvectors. The
      number of requested eigenvectors must be set *BEFORE* with the
      methods **elpa_set**(3) and **elpa_setup**(3). In case of a GPU build **q**
      can be a pointer to the device memory.

  integer, optional :: **error**
      The return error code of the function. Should be "ELPA_OK". The
      error code can be queried with the function **elpa_strerr**(3)

  C/C++ INTERFACE

  #include <elpa/elpa.h>
  elpa_t elpa_handle;

  // C:
  void **elpa_eigenvectors_double_complex**(elpa_t elpa_handle, **double complex**
  *a, **double complex** *ev, **double complex** *q, **int** *error);

  // C++:
  void **elpa_eigenvalues_double_complex**(elpa_t elpa_handle,
  std::complex<double> *a, **double** *ev, **std::complex<double>** *q, **int**
  *error);

  With the definitions of the input and output variables:

elpa_t **elpa_handle**;
>   The handle to the ELPA object

double complex ***a**;   OR   std::complex<double> ***a**;
>   The local part of the host/device matrix A for which the
>   eigenpairs should be computed. The dimensions of the matrix must
>   be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The
>   global matrix has to be symmetric, this is not checked by the
>   routine. In case of a GPU build **a** can be a pointer to the device
>   memory.

double ***ev**;
>   The host/device storage for the computed eigenvalues. Eigenvalues
>   will be stored in *ascending* order. In case of a GPU build **ev** can
>   be a pointer to the device memory.

double complex ***q**;   OR   std::complex<double> ***q**;
>   The host/device storage space for the computed eigenvectors. The
>   number of requested eigenvectors must be set *BEFORE* with the
>   methods **elpa_set**(3) and **elpa_setup**(3). In case of a GPU build **q**
>   can be a pointer to the device memory.

int ***error**;
>   The error code of the function. Should be "ELPA_OK". The error
>   codes can be queried with **elpa_strerr**(3)

DESCRIPTION

Computes the eigenvalues and (part of) the eigenvectors of a complex
hermitian matrix. The functions **elpa_init**(3), **elpa_allocate**(3),
elpa_set(3), and **elpa_setup**(3) must be called *BEFORE*
elpa_eigenvectors_double_complex can be called. In particular, the
number of eigenvectors to be computed, "nev", must be set with
elpa_set(3).

SEE ALSO

elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3)
elpa_setup(3) **elpa_strerr**(3) **elpa_eigenvalues**(3)
elpa_skew_eigenvalues(3) **elpa_skew_eigenvectors**(3) **elpa_cholesky**(3)
elpa_invert_triangular(3) **elpa_solve_tridiagonal**(3)
elpa_hermitian_multiply(3) **elpa_uninit**(3) **elpa_deallocate**(3)

D.24 `elpa_eigenvectors_float`

NAME
> elpa_eigenvectors_float – computes all eigenvalues and (part of) the
> eigenvectors for a real symmetric eigenproblem: A*Q = λ*Q

SYNOPSIS
  FORTRAN INTERFACE
> use elpa
> class(elpa_t), pointer :: elpa_handle
>
> call elpa_handle%**eigenvectors_float** (a, ev, q, error)
>
> With the definitions of the input and output variables:
>
> class(elpa_t) :: **elpa_handle**
>> An instance of the ELPA object.
>
> real(kind=c_float) :: **a**   OR   type(c_ptr) :: **a**
>> The local part of the host/device matrix A for which all
>> eigenvalues and (part of) eigenvectors should be computed. The
>> dimensions of matrix **a** must be set *BEFORE* with the methods
>> **elpa_set**(3) and **elpa_setup**(3). The global matrix has to be
>> symmetric, this is not checked by the routine. In case of a GPU
>> build **a** can be a pointer to the device memory.
>
> real(kind=c_float) :: **ev**   OR   type(c_ptr) :: **ev**
>> The host/device vector **ev** where the eigenvalues will be stored in
>> *ascending* order. In case of a GPU build **ev** can be a pointer to the
>> device memory.
>
> real(kind=c_float) :: **q**   OR   type(c_ptr) :: **q**
>> The host/device storage space for the computed eigenvectors. The
>> number of requested eigenvectors must be set *BEFORE* with the
>> methods **elpa_set**(3) and **elpa_setup**(3). In case of a GPU build **q**
>> can be a pointer to the device memory.
>
> integer, optional :: **error**
>> The return error code of the function. Should be "ELPA_OK". The
>> error code can be queried with the function **elpa_strerr**(3)

  C/C++ INTERFACE
> #include <elpa/elpa.h>
> elpa_t elpa_handle;
>
> void **elpa_eigenvectors_float**(elpa_t elpa_handle, **float** *a, **float** *ev,
> **float** *q, **int** *error);
>
> With the definitions of the input and output variables:
>
> elpa_t **elpa_handle**;
>> The handle to the ELPA object
>
> float ***a**;
>> The local part of the host/device matrix A for which the
>> eigenpairs should be computed. The dimensions of the matrix must
>> be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The
>> global matrix has to be symmetric, this is not checked by the

routine. In case of a GPU build **a** can be a pointer to the device
memory.

float   ***ev;**
>   The host/device storage for the computed eigenvalues. Eigenvalues
>   will be stored in *ascending* order. In case of a GPU build **ev** can
>   be a pointer to the device memory.

float   ***q;**
>   The host/device storage space for the computed eigenvectors. The
>   number of requested eigenvectors must be set *BEFORE* with the
>   methods **elpa_set**(3) and **elpa_setup**(3). In case of a GPU build **q**
>   can be a pointer to the device memory.

int   ***error;**
>   The error code of the function. Should be "ELPA_OK". The error
>   codes can be queried with **elpa_strerr**(3)

DESCRIPTION
>   Computes the eigenvalues and (part of) the eigenvectors of a real
>   symmetric matrix. The functions **elpa_init**(3), **elpa_allocate**(3),
>   **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE*
>   **elpa_eigenvectors_float** can be called. In particular, the number of
>   eigenvectors to be computed, "nev", must be set with **elpa_set**(3).

SEE ALSO
>   **elpa2_print_kernels**(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3)
>   **elpa_setup**(3) **elpa_strerr**(3) **elpa_eigenvalues**(3)
>   **elpa_skew_eigenvalues**(3) **elpa_skew_eigenvectors**(3) **elpa_cholesky**(3)
>   **elpa_invert_triangular**(3) **elpa_solve_tridiagonal**(3)
>   **elpa_hermitian_multiply**(3) **elpa_uninit**(3) **elpa_deallocate**(3)

D.25 `elpa_eigenvectors_float_complex`

NAME

elpa_eigenvectors_float_complex – computes all eigenvalues and (part of)
the eigenvectors for a complex hermitian eigenproblem: A*Q = λ*Q

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa_handle

call elpa_handle%**eigenvectors_float_complex** (a, ev, q, error)

With the definitions of the input and output variables:

class(elpa_t) :: **elpa_handle**
    An instance of the ELPA object.

real(kind=c_float_complex) :: **a**   OR   type(c_ptr) :: **a**
    The local part of the host/device matrix A for which all
    eigenvalues and (part of) eigenvectors should be computed. The
    dimensions of matrix **a** must be set *BEFORE* with the methods
    elpa_set(3) and **elpa_setup**(3). The global matrix has to be
    symmetric, this is not checked by the routine. In case of a GPU
    build **a** can be a pointer to the device memory.

real(kind=c_float) :: **ev**   OR   type(c_ptr) :: **ev**
    The host/device vector **ev** where the eigenvalues will be stored in
    ascending order. In case of a GPU build **ev** can be a pointer to the
    device memory. Note that complex hermitian matrices also have real
    eigenvalues.

real(kind=c_float_complex) :: **q**   OR   type(c_ptr) :: **q**
    The host/device storage space for the computed eigenvectors. The
    number of requested eigenvectors must be set *BEFORE* with the
    methods **elpa_set**(3) and **elpa_setup**(3). In case of a GPU build **q**
    can be a pointer to the device memory.

integer, optional :: **error**
    The return error code of the function. Should be "ELPA_OK". The
    error code can be queried with the function **elpa_strerr**(3)

C/C++ INTERFACE

#include <elpa/elpa.h>
elpa_t elpa_handle;

// C:
void **elpa_eigenvectors_float_complex**(elpa_t elpa_handle, **float complex**
*a, **float complex** *ev, **float complex** *q, **int** *error);

// C++:
void **elpa_eigenvalues_float_complex**(elpa_t elpa_handle,
std::complex<float> *a, **float** *ev, **std::complex<float>** *q, **int** *error);

With the definitions of the input and output variables:

> elpa_t **elpa_handle**;
>> The handle to the ELPA object

> float complex ***a**;   OR   std::complex<float> ***a**;
>> The local part of the host/device matrix A for which the
>> eigenpairs should be computed. The dimensions of the matrix must
>> be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The
>> global matrix has to be symmetric, this is not checked by the
>> routine. In case of a GPU build **a** can be a pointer to the device
>> memory.

> float ***ev**;
>> The host/device storage for the computed eigenvalues. Eigenvalues
>> will be stored in *ascending* order. In case of a GPU build **ev** can
>> be a pointer to the device memory.

> float complex ***q**;   OR   std::complex<float> ***q**;
>> The host/device storage space for the computed eigenvectors. The
>> number of requested eigenvectors must be set *BEFORE* with the
>> methods **elpa_set**(3) and **elpa_setup**(3). In case of a GPU build **q**
>> can be a pointer to the device memory.

> int ***error**;
>> The error code of the function. Should be "ELPA_OK". The error
>> codes can be queried with **elpa_strerr**(3)

DESCRIPTION
>   Computes the eigenvalues and (part of) the eigenvectors of a complex
>   hermitian matrix. The functions **elpa_init**(3), **elpa_allocate**(3),
>   **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE*
>   **elpa_eigenvectors_float_complex** can be called. In particular, the number
>   of eigenvectors to be computed, "nev", must be set with **elpa_set**(3).

SEE ALSO
>   **elpa2_print_kernels**(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3)
>   **elpa_setup**(3) **elpa_strerr**(3) **elpa_eigenvalues**(3)
>   **elpa_skew_eigenvalues**(3) **elpa_skew_eigenvectors**(3) **elpa_cholesky**(3)
>   **elpa_invert_triangular**(3) **elpa_solve_tridiagonal**(3)
>   **elpa_hermitian_multiply**(3) **elpa_uninit**(3) **elpa_deallocate**(3)

D.26 `elpa_generalized_eigenvalues`

NAME

  elpa_generalized_eigenvalues – computes all eigenvalues of a generalized
  eigenvalue problem, A*Q = λ*B*Q, for real symmetric or complex hermitian
  matrices A, B.

  There are also variations of this routine that can accept not only host
  but also device pointers as input/output. Names of these routines
  explicitly contain the corresponding datatypes:
  elpa_generalized_eigenvalues_double, elpa_generalized_eigenvalues_float,
  elpa_generalized_eigenvalues_double_complex,
  elpa_generalized_eigenvalues_float_complex.

SYNOPSIS

  FORTRAN INTERFACE

  use elpa
  class(elpa_t), pointer :: elpa_handle

  call elpa_handle%**generalized_eigenvalues** (a, b, ev,
  is_already_decomposed, error)

  With the definitions of the input and output variables:

  class(elpa_t) :: **elpa_handle**
    An instance of the ELPA object.

  datatype :: **a**
    The local matrix **a** for which the eigenvalues should be computed.
    The dimensions of matrix **a** must be set *BEFORE* with the methods
    elpa_set(3) and **elpa_setup**(3). The **datatype** of the matrix can be
    one of "real(kind=c_double)", "real(kind=c_float)",
    "complex(kind=c_double)", or "complex(kind=c_float)".

  datatype :: **b**
    The local matrix **b** defining the generalized eigenvalue problem.
    The dimensions and datatype of the matrix **b** has to be the same as
    for matrix **a**.

  datatype_real :: **ev**
    The array where the eigenvalues λ will be stored in *ascending*
    order. The **datatype_real** of the vector **ev** can be either
    "real(kind=c_double)" or "real(kind=c_float)", depending of the
    datatype of the matrix. Note that complex hermitian matrices also
    have real eigenvalues.

  logical :: **is_already_decomposed**
    Has to be set to .false. for the first call with a given **b** and
    .true. for each subsequent call with the same **b**, since **b** then
    already contains decomposition and thus the decomposing step is
    skipped.

  integer, optional :: **error**
    The return error code of the function. Should be "ELPA_OK". The
    error code can be queried with the function **elpa_strerr**(3)

C/C++ INTERFACE
```
#include <elpa/elpa.h>
elpa_t elpa_handle;

void elpa_generalized_eigenvalues(elpa_t elpa_handle, datatype *a,
datatype *b, datatype_real *ev, int is_already_decomposed, int *error);
```

With the definitions of the input and output variables:

elpa_t **elpa_handle**;
> The handle to the ELPA object

datatype ***a**;
> The local matrix **a** for which the eigenvalues should be computed. The dimensions of the matrix must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The **datatype** can be one of "double", "float", "double complex"/"std::complex<double>", "float complex"/"std::complex<float>" for C/C++.

datatype * **b**;
> The matrix **b** defining the generalized eigenvalue problem. The dimensions and the **datatype** of the matrix **b** must be the same as matrix a.

datatype_real ***ev**;
> The array where the eigenvalues λ will be stored in *ascending* order. The **datatype_real** can be either "double" or "float". Note that the eigenvalues of complex hermitian matrices are also real.

int **is_already_decomposed**;
> Has to be set to 0 for the first call with a given **b** and 1 for each subsequent call with the same **b**, since **b** then already contains decomposition and thus the decomposing step is skipped.

int ***error**;
> The error code of the function. Should be "ELPA_OK". The error codes can be queried with **elpa_strerr**(3)

DESCRIPTION
> Computes the generalized eigenvalues for a real symmetric or complex hermitian generalized eigenproblem. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE* **elpa_generalized_eigenvalues** can be called. Unlike in the case of ordinary eigenvalue problem, the generalized problem calls some external ScaLAPACK routines. The user is responsible for initialization of the BLACS context, which then has to be passed to elpa by **elpa_set**(3) *BEFORE* **elpa_generalized_eigenvalues** can be called.

SEE ALSO
> **elpa2_print_kernels**(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3) **elpa_cholesky**(3) **elpa_invert_triangular**(3) **elpa_solve_tridiagonal**(3) **elpa_hermitian_multiply**(3) **elpa_uninit**(3) **elpa_deallocate**(3)

D.27   `elpa_generalized_eigenvalues_double`

NAME

elpa_generalized_eigenvalues_double – computes all eigenvalues of a generalized eigenvalue problem, A*Q = λ*B*Q, for real symmetric matrices A, B.

SYNOPSIS

FORTRAN INTERFACE

```
use elpa
class(elpa_t), pointer :: elpa_handle
```

call elpa_handle%**generalized_eigenvalues_double** (a, b, ev, is_already_decomposed, error)

With the definitions of the input and output variables:

class(elpa_t) :: **elpa_handle**
An instance of the ELPA object.

real(kind=c_double) :: **a**   OR   type(c_ptr) :: **a**
The local matrix **a** for which the eigenvalues should be computed. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). In case of a GPU build **a** can be a pointer to the device memory.

real(kind=c_double) :: **b**   OR   type(c_ptr) :: **b**
The local matrix **b** defining the generalized eigenvalue problem. The dimensions of the matrix **b** has to be the same as for matrix **a**. In case of a GPU build **a** can be a pointer to the device memory.

real(kind=c_double) :: **ev**   OR   type(c_ptr) :: **ev**
The array where the eigenvalues λ will be stored in *ascending* order. In case of a GPU build **a** can be a pointer to the device memory.

logical :: **is_already_decomposed**
Has to be set to .false. for the first call with a given **b** and .true. for each subsequent call with the same **b**, since **b** then already contains decomposition and thus the decomposing step is skipped.

integer, optional :: **error**
The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3)

C/C++ INTERFACE

```
#include <elpa/elpa.h>
elpa_t elpa_handle;
```

void **elpa_generalized_eigenvalues_double**(**elpa_t** elpa_handle, **double** *a, **double** *b, **double** *ev, **int** is_already_decomposed, **int** *error);

With the definitions of the input and output variables:

elpa_t **elpa_handle**;
> The handle to the ELPA object

double ***a**;
> The local matrix **a** for which the eigenvalues should be computed.
> The dimensions of the matrix must be set *BEFORE* with the methods
> **elpa_set**(3) and **elpa_setup**(3). The global matrix has to be
> symmetric, this is not checked by the routine. In case of a GPU
> build **a** can be a pointer to the device memory.

double * **b**;
> The local matrix **b** defining the generalized eigenvalue problem.
> The dimensions of the matrix **b** must be the same as matrix a. The
> global matrix has to be symmetric, this is not checked by the
> routine. In case of a GPU build **a** can be a pointer to the device
> memory.

double ***ev**;
> The array where the eigenvalues λ will be stored in *ascending*
> order. In case of a GPU build **a** can be a pointer to the device
> memory.

int **is_already_decomposed**;
> Has to be set to 0 for the first call with a given **b** and 1 for
> each subsequent call with the same **b**, since **b** then already
> contains decomposition and thus the decomposing step is skipped.

int ***error**;
> The error code of the function. Should be "ELPA_OK". The error
> codes can be queried with **elpa_strerr**(3)

DESCRIPTION
Computes the generalized eigenvalues for a real symmetric generalized
eigenproblem. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3),
and **elpa_setup**(3) must be called *BEFORE*
elpa_generalized_eigenvalues_double can be called. Unlike in the case of
ordinary eigenvalue problem, the generalized problem calls some external
ScaLAPACK routines. The user is responsible for initialization of the
BLACS context, which then has to be passed to elpa by **elpa_set**(3) *BEFORE*
elpa_generalized_eigenvalues_double can be called.

SEE ALSO
elpa_init(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3)
elpa_eigenvalues(3) **elpa_eigenvectors**(3) **elpa_cholesky**(3)
elpa_invert_triangular(3) **elpa_solve_tridiagonal**(3)
elpa_hermitian_multiply(3) **elpa_uninit**(3) **elpa_deallocate**(3)

D.28 `elpa_generalized_eigenvalues_double_complex`

NAME

elpa_generalized_eigenvalues_double_complex – computes all eigenvalues of a generalized eigenvalue problem, A*Q = λ*B*Q, for complex hermitian matrices A, B.

SYNOPSIS

FORTRAN INTERFACE

```
use elpa
class(elpa_t), pointer :: elpa_handle

call elpa_handle%generalized_eigenvalues_double_complex (a, b, ev,
is_already_decomposed, error)
```

With the definitions of the input and output variables:

class(elpa_t) :: **elpa_handle**
An instance of the ELPA object.

real(kind=c_double_complex) :: **a**   OR   type(c_ptr) :: **a**
The local matrix **a** for which the eigenvalues should be computed. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). In case of a GPU build **a** can be a pointer to the device memory.

real(kind=c_double_complex) :: **b**   OR   type(c_ptr) :: **b**
The local matrix **b** defining the generalized eigenvalue problem. The dimensions of the matrix **b** has to be the same as for matrix **a**. In case of a GPU build **a** can be a pointer to the device memory.

real(kind=c_double) :: **ev**   OR   type(c_ptr) :: **ev**
The array where the eigenvalues λ will be stored in *ascending* order. Note that the eigenvalues of complex hermitian matrices are also real. In case of a GPU build **a** can be a pointer to the device memory.

logical :: **is_already_decomposed**
Has to be set to .false. for the first call with a given **b** and .true. for each subsequent call with the same **b**, since **b** then already contains decomposition and thus the decomposing step is skipped.

integer, optional :: **error**
The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3)

C/C++ INTERFACE

```
#include <elpa/elpa.h>
elpa_t elpa_handle;

// C:
void elpa_generalized_eigenvalues_double_complex(elpa_t elpa_handle,
double complex *a, double complex *b, double *ev, int
is_already_decomposed, int *error);

// C++:
```

void **elpa_generalized_eigenvalues_double_complex**(**elpa_t** elpa_handle,
std::complex<double> *a, **std::complex<double>** *b, **double** *ev, **int**
is_already_decomposed, **int** *error);

With the definitions of the input and output variables:

elpa_t **elpa_handle**;
    The handle to the ELPA object

double complex ***a**;   OR   std::complex<double> ***a**;
    The local matrix **a** for which the eigenvalues should be computed.
    The dimensions of the matrix must be set *BEFORE* with the methods
    elpa_set(3) and **elpa_setup**(3). The global matrix has to be
    symmetric, this is not checked by the routine. In case of a GPU
    build **a** can be a pointer to the device memory.

double complex * **b**;   OR   std::complex<double> ***b**;
    The local matrix **b** defining the generalized eigenvalue problem.
    The dimensions of the matrix **b** must be the same as matrix a. The
    global matrix has to be symmetric, this is not checked by the
    routine. In case of a GPU build **a** can be a pointer to the device
    memory.

double ***ev**;
    The array where the eigenvalues λ will be stored in *ascending*
    order. Note that the eigenvalues of complex hermitian matrices are
    also real. In case of a GPU build **a** can be a pointer to the device
    memory.

int **is_already_decomposed**;
    Has to be set to 0 for the first call with a given **b** and 1 for
    each subsequent call with the same **b**, since **b** then already
    contains decomposition and thus the decomposing step is skipped.

int ***error**;
    The error code of the function. Should be "ELPA_OK". The error
    codes can be queried with **elpa_strerr**(3)

DESCRIPTION
    Computes the generalized eigenvalues for a complex hermitian generalized
    eigenproblem. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3),
    and **elpa_setup**(3) must be called *BEFORE*
    elpa_generalized_eigenvalues_double_complex can be called. Unlike in the
    case of ordinary eigenvalue problem, the generalized problem calls some
    external ScaLAPACK routines. The user is responsible for initialization
    of the BLACS context, which then has to be passed to elpa by **elpa_set**(3)
    BEFORE **elpa_generalized_eigenvalues_double_complex** can be called.

SEE ALSO
    elpa_init(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3)
    elpa_eigenvalues(3) **elpa_eigenvectors**(3) **elpa_cholesky**(3)
    elpa_invert_triangular(3) **elpa_solve_tridiagonal**(3)
    elpa_hermitian_multiply(3) **elpa_uninit**(3) **elpa_deallocate**(3)

D.29 `elpa_generalized_eigenvalues_float`

NAME

elpa_generalized_eigenvalues_float – computes all eigenvalues of a
generalized eigenvalue problem, A*Q = λ*B*Q, for real symmetric matrices
A, B.

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa_handle

call elpa_handle%**generalized_eigenvalues_float** (a, b, ev,
is_already_decomposed, error)

With the definitions of the input and output variables:

class(elpa_t) :: **elpa_handle**
An instance of the ELPA object.

real(kind=c_float) :: **a**   OR   type(c_ptr) :: **a**
The local matrix **a** for which the eigenvalues should be computed.
The dimensions of matrix **a** must be set *BEFORE* with the methods
elpa_set(3) and **elpa_setup**(3). In case of a GPU build **a** can be a
pointer to the device memory.

real(kind=c_float) :: **b**   OR   type(c_ptr) :: **b**
The local matrix **b** defining the generalized eigenvalue problem.
The dimensions of the matrix **b** has to be the same as for matrix **a**.
In case of a GPU build **a** can be a pointer to the device memory.

real(kind=c_float) :: **ev**   OR   type(c_ptr) :: **ev**
The array where the eigenvalues λ will be stored in *ascending*
order. In case of a GPU build **a** can be a pointer to the device
memory.

logical :: **is_already_decomposed**
Has to be set to .false. for the first call with a given **b** and
.true. for each subsequent call with the same **b**, since **b** then
already contains decomposition and thus the decomposing step is
skipped.

integer, optional :: **error**
The return error code of the function. Should be "ELPA_OK". The
error code can be queried with the function **elpa_strerr**(3)

C/C++ INTERFACE

#include <elpa/elpa.h>
elpa_t elpa_handle;

void **elpa_generalized_eigenvalues_float**(**elpa_t** elpa_handle, **float** *a,
float *b, **float** *ev, **int** is_already_decomposed, **int** *error);

With the definitions of the input and output variables:

elpa_t **elpa_handle**;
  The handle to the ELPA object

float ***a**;
  The local matrix **a** for which the eigenvalues should be computed.
  The dimensions of the matrix must be set *BEFORE* with the methods
  elpa_set(3) and **elpa_setup**(3). The global matrix has to be
  symmetric, this is not checked by the routine. In case of a GPU
  build **a** can be a pointer to the device memory.

float * **b**;
  The local matrix **b** defining the generalized eigenvalue problem.
  The dimensions of the matrix **b** must be the same as matrix a. The
  global matrix has to be symmetric, this is not checked by the
  routine. In case of a GPU build **a** can be a pointer to the device
  memory.

float ***ev**;
  The array where the eigenvalues λ will be stored in *ascending*
  order. In case of a GPU build **a** can be a pointer to the device
  memory.

int **is_already_decomposed**;
  Has to be set to 0 for the first call with a given **b** and 1 for
  each subsequent call with the same **b**, since **b** then already
  contains decomposition and thus the decomposing step is skipped.

int ***error**;
  The error code of the function. Should be "ELPA_OK". The error
  codes can be queried with **elpa_strerr**(3)

DESCRIPTION

Computes the generalized eigenvalues for a real symmetric generalized
eigenproblem. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3),
and **elpa_setup**(3) must be called *BEFORE*
elpa_generalized_eigenvalues_float can be called. Unlike in the case of
ordinary eigenvalue problem, the generalized problem calls some external
ScaLAPACK routines. The user is responsible for initialization of the
BLACS context, which then has to be passed to elpa by **elpa_set**(3) *BEFORE*
elpa_generalized_eigenvalues_float can be called.

SEE ALSO

 elpa_init(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3)
elpa_eigenvalues(3) **elpa_eigenvectors**(3) **elpa_cholesky**(3)
elpa_invert_triangular(3) **elpa_solve_tridiagonal**(3)
elpa_hermitian_multiply(3) **elpa_uninit**(3) **elpa_deallocate**(3)

D.30 `elpa_generalized_eigenvalues_float_complex`

NAME

      elpa_generalized_eigenvalues_float_complex – computes all eigenvalues of
a generalized eigenvalue problem, A*Q = λ*B*Q, for complex hermitian
matrices A, B.

SYNOPSIS

  FORTRAN INTERFACE

      use elpa
      class(elpa_t), pointer :: elpa_handle

      call elpa_handle%**generalized_eigenvalues_float_complex** (a, b, ev,
is_already_decomposed, error)

      With the definitions of the input and output variables:

      class(elpa_t) :: **elpa_handle**
          An instance of the ELPA object.

      real(kind=c_float_complex) :: **a**   OR   type(c_ptr) :: **a**
          The local matrix **a** for which the eigenvalues should be computed.
The dimensions of matrix **a** must be set *BEFORE* with the methods
elpa_set(3) and **elpa_setup**(3). In case of a GPU build **a** can be a
pointer to the device memory.

      real(kind=c_float_complex) :: **b**   OR   type(c_ptr) :: **b**
          The local matrix **b** defining the generalized eigenvalue problem.
The dimensions of the matrix **b** has to be the same as for matrix **a**.
In case of a GPU build **a** can be a pointer to the device memory.

      real(kind=c_float) :: **ev**   OR   type(c_ptr) :: **ev**
          The array where the eigenvalues λ will be stored in *ascending*
order. Note that the eigenvalues of complex hermitian matrices are
also real. In case of a GPU build **a** can be a pointer to the device
memory.

      logical :: **is_already_decomposed**
          Has to be set to .false. for the first call with a given **b** and
.true. for each subsequent call with the same **b**, since **b** then
already contains decomposition and thus the decomposing step is
skipped.

      integer, optional :: **error**
          The return error code of the function. Should be "ELPA_OK". The
error code can be queried with the function **elpa_strerr**(3)

  C/C++ INTERFACE

      #include <elpa/elpa.h>
      elpa_t elpa_handle;

      // C:
      void **elpa_generalized_eigenvalues_float_complex**(**elpa_t** elpa_handle,
float complex *a, **float complex** *b, **float** *ev, **int**
is_already_decomposed, **int** *error);

      // C++:

```
void elpa_generalized_eigenvalues_float_complex(elpa_t elpa_handle,
std::complex<float> *a, std::complex<float> *b, float *ev, int
is_already_decomposed, int *error);
```

With the definitions of the input and output variables:

elpa_t **elpa_handle**;
     The handle to the ELPA object

float complex ***a**;   OR   std::complex<float> ***a**;
     The local matrix **a** for which the eigenvalues should be computed.
     The dimensions of the matrix must be set *BEFORE* with the methods
     elpa_set(3) and **elpa_setup**(3). The global matrix has to be
     symmetric, this is not checked by the routine. In case of a GPU
     build **a** can be a pointer to the device memory.

float complex * **b**;   OR   std::complex<float> ***b**;
     The local matrix **b** defining the generalized eigenvalue problem.
     The dimensions of the matrix **b** must be the same as matrix a. The
     global matrix has to be symmetric, this is not checked by the
     routine. In case of a GPU build **a** can be a pointer to the device
     memory.

float ***ev**;
     The array where the eigenvalues λ will be stored in *ascending*
     order. Note that the eigenvalues of complex hermitian matrices are
     also real. In case of a GPU build **a** can be a pointer to the device
     memory.

int **is_already_decomposed**;
     Has to be set to 0 for the first call with a given **b** and 1 for
     each subsequent call with the same **b**, since **b** then already
     contains decomposition and thus the decomposing step is skipped.

int ***error**;
     The error code of the function. Should be "ELPA_OK". The error
     codes can be queried with **elpa_strerr**(3)

DESCRIPTION
     Computes the generalized eigenvalues for a complex hermitian generalized
     eigenproblem. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3),
     and **elpa_setup**(3) must be called *BEFORE*
     elpa_generalized_eigenvalues_float_complex can be called. Unlike in the
     case of ordinary eigenvalue problem, the generalized problem calls some
     external ScaLAPACK routines. The user is responsible for initialization
     of the BLACS context, which then has to be passed to elpa by **elpa_set**(3)
     BEFORE **elpa_generalized_eigenvalues_float_complex** can be called.

SEE ALSO
      elpa_init(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3)
     elpa_eigenvalues(3) **elpa_eigenvectors**(3) **elpa_cholesky**(3)
     elpa_invert_triangular(3) **elpa_solve_tridiagonal**(3)
     elpa_hermitian_multiply(3) **elpa_uninit**(3) **elpa_deallocate**(3)

D.31 `elpa_generalized_eigenvectors`

NAME

elpa_generalized_eigenvectors – computes all eigenvalues and (part of) eigenvectors of a generalized eigenvalue problem, $A*Q = \lambda*B*Q$, for real symmetric or complex hermitian matrices A, B.

There are also variations of this routine that can accept not only host but also device pointers as input/output. Names of these routines explicitly contain the corresponding datatypes: elpa_generalized_eigenvectors_double, elpa_generalized_eigenvectors_float, elpa_generalized_eigenvectors_double_complex, elpa_generalized_eigenvectors_float_complex.

SYNOPSIS

  FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa_handle

call elpa_handle%**generalized_eigenvectors** (a, b, ev, q, is_already_decomposed, error)

With the definitions of the input and output variables:

class(elpa_t) :: **elpa_handle**
     An instance of the ELPA object.

datatype :: **a**
     The local matrix **a** for which the eigenvalues and eigenvalues should be computed. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The **datatype** of the matrix can be one of "real(kind=c_double)", "real(kind=c_float)", "complex(kind=c_double)", or "complex(kind=c_float)".

datatype :: **b**
     The local matrix **b** defining the generalized eigenvalue problem. The dimensions and datatype of the matrix **b** has to be the same as for matrix **a**.

datatype :: **ev**
     The array where the eigenvalues λ will be stored in *ascending* order. The **datatype_real** of **ev** can be either "real(kind=c_double)" or "real(kind=c_float)", depending of the **datatype** of the matrix. Note that complex hermitian matrices also have real eigenvalues.

datatype :: **q**
     The storage space for the computed eigenvectors Q. The number of requested eigenvectors, nev, must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The **datatype** of the matrix can be one of "real(kind=c_double)", "real(kind=c_float)", "complex(kind=c_double)", or "complex(kind=c_float)".

logical :: **is_already_decomposed**
     Has to be set to .false. for the first call with a given **b** and to .true. for each subsequent call with the same **b**, since **b** then already contains decomposition and thus the decomposing step is skipped.

```
integer, optional :: error
      The return error code of the function. Should be "ELPA_OK". The
      error code can be queried with the function elpa_strerr(3)
```

C/C++ INTERFACE
```
#include <elpa/elpa.h>
elpa_t elpa_handle;
```

void **elpa_generalized_eigenvectors**(**elpa_t** elpa_handle, **datatype** *a,
datatype *b, **datatype** *ev, **datatype** *q, **int** is_already_decomposed, **int**
*error);

With the definitions of the input and output variables:

elpa_t **elpa_handle**;
      The handle to the ELPA object

datatype ***a**;
      The local matrix **a** for which all eigenvalues and (part of)
      eigenvectors should be computed. The dimensions of the matrix must
      be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The
      global matrix has to be symmetric or hermitian, this is not
      checked by the routine. The **datatype** can be one of "double",
      "float", "double complex"/"std::complex<double>", "float
      complex"/"std::complex<float>" for C/C++.

datatype ***b**;
      The local matrix **b** defining the generalized eigenvalue problem.
      The dimensions and the **datatype** of the matrix **b** must be the same
      as matrix **a**. The global matrix has to be symmetric or hermitian,
      this is not checked by the routine. The **datatype** can be one of
      "double", "float", "double complex"/"std::complex<double>", "float
      complex"/"std::complex<float>" for C/C++.

datatype_real ***ev**;
      The array where the eigenvalues λ will be stored in *ascending*
      order. Eigenvalues will be stored in *ascending* order. The
      datatype_real can be either "double" or "float". Note that the
      eigenvalues of complex hermitian matrices are also real.

datatype ***q**;
      The storage space for the computed eigenvectors Q. The number of
      requested eigenvectors must be set *BEFORE* with the methods
      elpa_set(3) and **elpa_setup**(3). The **datatype** can be one of
      "double", "float", "double complex"/"std::complex<double>", "float
      complex"/"std::complex<float>" for C/C++.

int **is_already_decomposed**;
      Has to be set to 0 for the first call with a given **b** and 1 for
      each subsequent call with the same **b**, since **b** then already
      contains decomposition and thus the decomposing step is skipped.

int ***error**;
      The error code of the function. Should be "ELPA_OK". The error
      codes can be queried with **elpa_strerr**(3).

DESCRIPTION
>   Computes the generalized eigenvalues and (part of) the generalized
>   eigenvectors for a real symmetric or complex hermitian generalized
>   eigenproblem. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3),
>   and **elpa_setup**(3) must be called *BEFORE* **elpa_generalized_eigenvalues** can
>   be called. In particular, the number of eigenvectors to be computed,
>   nev, can be set with **elpa_set**(3). Unlike in the case of ordinary
>   eigenvalue problem, the generalized problem calls some external
>   ScaLAPACK routines. The user is responsible for initialization of the
>   BLACS context, which then has to be passed to ELPA by **elpa_set**(3) *BEFORE*
>   **elpa_generalized_eigenvalues** can be called.

SEE ALSO
>   **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3)
>   **elpa_eigenvalues**(3) **elpa_eigenvectors**(3) **elpa_cholesky**(3)
>   **elpa_invert_triangular**(3) **elpa_solve_tridiagonal**(3)
>   **elpa_hermitian_multiply**(3) **elpa_uninit**(3) **elpa_deallocate**(3)

D.32 `elpa_generalized_eigenvectors_double`

NAME
      elpa_generalized_eigenvectors_double – computes all eigenvalues and
      (part of) eigenvectors of a generalized eigenvalue problem, A*Q = λ*B*Q,
      for real symmetric matrices A, B.

SYNOPSIS
  FORTRAN INTERFACE
      use elpa
      class(elpa_t), pointer :: elpa_handle

      call elpa_handle%**generalized_eigenvectors** (a, b, ev, q,
      is_already_decomposed, error)

      With the definitions of the input and output variables:

      class(elpa_t) :: **elpa_handle**
          An instance of the ELPA object.

      real(kind=c_double) :: **a**   OR   type(c_ptr) :: **a**
          The local part of the host/device matrix A for which the
          eigenvalues and eigenvalues should be computed. The dimensions of
          matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and
          elpa_setup(3). The global matrix has to be symmetric, this is not
          checked by the routine. In case of a GPU build **a** can be a pointer
          to the device memory.

      real(kind=c_double) :: **b**   OR   type(c_ptr) :: **b**
          The host/device local matrix **b** defining the generalized eigenvalue
          problem. The dimensions of the matrix **b** has to be the same as for
          matrix **a**. The global matrix has to be symmetric, this is not
          checked by the routine. In case of a GPU build **b** can be a pointer
          to the device memory.

      real(kind=c_double) :: **ev**   OR   type(c_ptr) :: **ev**
          The host/device vector **ev** where the eigenvalues will be stored in
          ascending order. In case of a GPU build **ev** can be a device pointer
          to a vector **ev** in the device memory.

      real(kind=c_double) :: **q**   OR   type(c_ptr) :: **q**
          The storage space for the computed eigenvectors Q. The number of
          requested eigenvectors, nev, must be set *BEFORE* with the methods
          elpa_set(3) and **elpa_setup**(3). In case of a GPU build **q** can be a
          pointer to the device memory.

      logical :: **is_already_decomposed**
          Has to be set to .false. for the first call with a given **b** and to
          .true. for each subsequent call with the same **b**, since **b** then
          already contains decomposition and thus the decomposing step is
          skipped.

      integer, optional :: **error**
          The return error code of the function. Should be "ELPA_OK". The
          error code can be queried with the function **elpa_strerr**(3).

C/C++ INTERFACE
```
#include <elpa/elpa.h>
elpa_t elpa_handle;
```

void **elpa_generalized_eigenvectors_double**(**elpa_t** elpa_handle, **double** *a,
double *b, **double** *ev, **double** *q, **int** is_already_decomposed, **int**
*error);

With the definitions of the input and output variables:

elpa_t   **elpa_handle**;
> The handle to the ELPA object

double   ***a**;
> The local matrix **a** for which all eigenvalues and (part of)
> eigenvectors should be computed. The dimensions of the matrix must
> be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The
> global matrix has to be symmetric, this is not checked by the
> routine. In case of a GPU build **a** can be a pointer to the device
> memory.

double   ***b**;
> The local matrix **b** defining the generalized eigenvalue problem.
> The dimensions of the matrix **b** must be the same as matrix **a**. The
> global matrix has to be symmetric, this is not checked by the
> routine. In case of a GPU build **a** can be a pointer to the device
> memory.

double   ***ev**;
> The array where the eigenvalues λ will be stored in *ascending*
> order. Eigenvalues will be stored in *ascending* order. In case of a
> GPU build **a** can be a pointer to the device memory.

double   ***q**;
> The storage space for the computed eigenvectors Q. The number of
> requested eigenvectors, nev, must be set *BEFORE* with the methods
> **elpa_set**(3) and **elpa_setup**(3). In case of a GPU build **a** can be a
> pointer to the device memory.

int   **is_already_decomposed**;
> Has to be set to 0 for the first call with a given **b** and 1 for
> each subsequent call with the same **b**, since **b** then already
> contains decomposition and thus the decomposing step is skipped.

int   ***error**;
> The error code of the function. Should be "ELPA_OK". The error
> codes can be queried with **elpa_strerr**(3)

DESCRIPTION
> Computes the generalized eigenvalues and (part of) the generalized
> eigenvectors for a real symmetric generalized eigenproblem. The
> functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3)
> must be called *BEFORE* **elpa_generalized_eigenvalues_double** can be called.
> In particular, the number of eigenvectors to be computed, nev, can be
> set with **elpa_set**(3). Unlike in the case of ordinary eigenvalue problem,
> the generalized problem calls some external ScaLAPACK routines. The user
> is responsible for initialization of the BLACS context, which then has

to be passed to ELPA by **elpa_set**(3) *BEFORE*
elpa_generalized_eigenvalues_double can be called.

SEE ALSO
  elpa_init(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3)
  elpa_eigenvalues(3) **elpa_eigenvectors**(3) **elpa_cholesky**(3)
  elpa_invert_triangular(3) **elpa_solve_tridiagonal**(3)
  elpa_hermitian_multiply(3) **elpa_uninit**(3) **elpa_deallocate**(3)

D.33 `elpa_generalized_eigenvectors_double_complex`

NAME

      elpa_generalized_eigenvectors_double_complex – computes all eigenvalues
      and (part of) eigenvectors of a generalized eigenvalue problem, A*Q =
      λ*B*Q,

SYNOPSIS

  FORTRAN INTERFACE

      use elpa
      class(elpa_t), pointer :: elpa_handle

      call elpa_handle%**generalized_eigenvectors** (a, b, ev, q,
      is_already_decomposed, error)

      With the definitions of the input and output variables:

      class(elpa_t) :: **elpa_handle**
            An instance of the ELPA object.

      real(kind=c_double_complex) :: **a**   OR   type(c_ptr) :: **a**
            The local part of the host/device matrix A for which the
            eigenvalues and eigenvalues should be computed. The dimensions of
            matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and
            elpa_setup(3). The global matrix has to be symmetric, this is not
            checked by the routine. In case of a GPU build **a** can be a pointer
            to the device memory.

      real(kind=c_double_complex) :: **b**   OR   type(c_ptr) :: **b**
            The host/device local matrix **b** defining the generalized eigenvalue
            problem. The dimensions of the matrix **b** has to be the same as for
            matrix **a**. The global matrix has to be symmetric, this is not
            checked by the routine. In case of a GPU build **b** can be a pointer
            to the device memory.

      real(kind=c_double) :: **ev**   OR   type(c_ptr) :: **ev**
            The host/device vector **ev** where the eigenvalues will be stored in
            ascending order. Note that the eigenvalues of complex hermitian
            matrices are also real. In case of a GPU build **ev** can be a device
            pointer to a vector **ev** in the device memory.

      real(kind=c_double_complex) :: **q**   OR   type(c_ptr) :: **q**
            The storage space for the computed eigenvectors Q. The number of
            requested eigenvectors, nev, must be set *BEFORE* with the methods
            elpa_set(3) and **elpa_setup**(3). In case of a GPU build **q** can be a
            pointer to the device memory.

      logical :: **is_already_decomposed**
            Has to be set to .false. for the first call with a given **b** and to
            .true. for each subsequent call with the same **b**, since **b** then
            already contains decomposition and thus the decomposing step is
            skipped.

      integer, optional :: **error**
            The return error code of the function. Should be "ELPA_OK". The
            error code can be queried with the function **elpa_strerr**(3).

C/C++ INTERFACE
```
#include <elpa/elpa.h>
elpa_t elpa_handle;

// C:
void elpa_generalized_eigenvectors_double_complex(elpa_t elpa_handle,
double complex *a, double complex *b, double complex *ev, double complex
*q,  int is_already_decomposed, int *error);

// C++:
void elpa_generalized_eigenvectors_double_complex(elpa_t elpa_handle,
std::complex<double>  *a, std::complex<double>  *b, std::complex<double>
*ev, std::complex<double>  *q,  int is_already_decomposed, int *error);
```

With the definitions of the input and output variables:

elpa_t **elpa_handle**;
    The handle to the ELPA object

double complex ***a**;   OR   std::complex<double> ***a**;
    The local matrix **a** for which all eigenvalues and (part of)
    eigenvectors should be computed. The dimensions of the matrix must
    be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The
    global matrix has to be symmetric, this is not checked by the
    routine. In case of a GPU build **a** can be a pointer to the device
    memory.

double complex ***b**;   OR   std::complex<double> ***b**;
    The local matrix **b** defining the generalized eigenvalue problem.
    The dimensions of the matrix **b** must be the same as matrix **a**. The
    global matrix has to be symmetric, this is not checked by the
    routine. In case of a GPU build **a** can be a pointer to the device
    memory.

double ***ev**;
    The array where the eigenvalues λ will be stored in *ascending*
    order. Eigenvalues will be stored in *ascending* order. Note that
    the eigenvalues of complex hermitian matrices are also real. In
    case of a GPU build **a** can be a pointer to the device memory.

double complex ***q**;   OR   std::complex<double> ***q**;
    The storage space for the computed eigenvectors Q. The number of
    requested eigenvectors, nev, must be set *BEFORE* with the methods
    elpa_set(3) and **elpa_setup**(3). In case of a GPU build **a** can be a
    pointer to the device memory.

int **is_already_decomposed**;
    Has to be set to 0 for the first call with a given **b** and 1 for
    each subsequent call with the same **b**, since **b** then already
    contains decomposition and thus the decomposing step is skipped.

int ***error**;
    The error code of the function. Should be "ELPA_OK". The error
    codes can be queried with **elpa_strerr**(3)

DESCRIPTION

      Computes the generalized eigenvalues and (part of) the eigenvector
      spectrum of a complex hermitian matrix. The functions **elpa_init**(3),
      elpa_allocate(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE*
      elpa_generalized_eigenvalues_double_complex can be called. In
      particular, the number of eigenvectors to be computed, nev, can be set
      with **elpa_set**(3). Unlike in the case of ordinary eigenvalue problem, the
      generalized problem calls some external ScaLAPACK routines. The user is
      responsible for initialization of the BLACS context, which then has to
      be passed to ELPA by **elpa_set**(3) *BEFORE*
      elpa_generalized_eigenvalues_double_complex can be called.

SEE ALSO

       elpa_init(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3)
      elpa_eigenvalues(3) **elpa_eigenvectors**(3) **elpa_cholesky**(3)
      elpa_invert_triangular(3) **elpa_solve_tridiagonal**(3)
      elpa_hermitian_multiply(3) **elpa_uninit**(3) **elpa_deallocate**(3)

D.34 `elpa_generalized_eigenvectors_float`

NAME

elpa_generalized_eigenvectors_float – computes all eigenvalues and (part of) eigenvectors of a generalized eigenvalue problem, $A*Q = \lambda*B*Q$, for real symmetric matrices A, B.

SYNOPSIS

FORTRAN INTERFACE

```
use elpa
class(elpa_t), pointer :: elpa_handle

call elpa_handle%generalized_eigenvectors (a, b, ev, q,
is_already_decomposed, error)
```

With the definitions of the input and output variables:

class(elpa_t) :: **elpa_handle**
> An instance of the ELPA object.

real(kind=c_float) :: **a**   OR   type(c_ptr) :: **a**
> The local part of the host/device matrix A for which the eigenvalues and eigenvalues should be computed. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The global matrix has to be symmetric, this is not checked by the routine. In case of a GPU build **a** can be a pointer to the device memory.

real(kind=c_float) :: **b**   OR   type(c_ptr) :: **b**
> The host/device local matrix **b** defining the generalized eigenvalue problem. The dimensions of the matrix **b** has to be the same as for matrix **a**. The global matrix has to be symmetric, this is not checked by the routine. In case of a GPU build **b** can be a pointer to the device memory.

real(kind=c_float) :: **ev**   OR   type(c_ptr) :: **ev**
> The host/device vector **ev** where the eigenvalues will be stored in *ascending* order. In case of a GPU build **ev** can be a device pointer to a vector **ev** in the device memory.

real(kind=c_float) :: **q**   OR   type(c_ptr) :: **q**
> The storage space for the computed eigenvectors Q. The number of requested eigenvectors, nev, must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). In case of a GPU build **q** can be a pointer to the device memory.

logical :: **is_already_decomposed**
> Has to be set to .false. for the first call with a given **b** and to .true. for each subsequent call with the same **b**, since **b** then already contains decomposition and thus the decomposing step is skipped.

integer, optional :: **error**
> The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3).

C/C++ INTERFACE
```
#include <elpa/elpa.h>
elpa_t elpa_handle;
```

void **elpa_generalized_eigenvectors_float**(**elpa_t** elpa_handle, **float** *a,
float *b, **float** *ev, **float** *q, **int** is_already_decomposed, **int** *error);

With the definitions of the input and output variables:

elpa_t **elpa_handle**;
> The handle to the ELPA object

float ***a**;
> The local matrix **a** for which all eigenvalues and (part of)
> eigenvectors should be computed. The dimensions of the matrix must
> be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The
> global matrix has to be symmetric, this is not checked by the
> routine. In case of a GPU build **a** can be a pointer to the device
> memory.

float ***b**;
> The local matrix **b** defining the generalized eigenvalue problem.
> The dimensions of the matrix **b** must be the same as matrix **a**. The
> global matrix has to be symmetric, this is not checked by the
> routine. In case of a GPU build **a** can be a pointer to the device
> memory.

float ***ev**;
> The array where the eigenvalues λ will be stored in *ascending*
> order. Eigenvalues will be stored in *ascending* order. In case of a
> GPU build **a** can be a pointer to the device memory.

float ***q**;
> The storage space for the computed eigenvectors Q. The number of
> requested eigenvectors, nev, must be set *BEFORE* with the methods
> **elpa_set**(3) and **elpa_setup**(3). In case of a GPU build **a** can be a
> pointer to the device memory.

int **is_already_decomposed**;
> Has to be set to 0 for the first call with a given **b** and 1 for
> each subsequent call with the same **b**, since **b** then already
> contains decomposition and thus the decomposing step is skipped.

int ***error**;
> The error code of the function. Should be "ELPA_OK". The error
> codes can be queried with **elpa_strerr**(3)

DESCRIPTION
> Computes the generalized eigenvalues and (part of) the eigenvectors for
> a real symmetric generalized eigenproblem. The functions **elpa_init**(3),
> **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE*
> **elpa_generalized_eigenvalues_float** can be called. In particular, the
> number of eigenvectors to be computed, nev, can be set with **elpa_set**(3).
> Unlike in the case of ordinary eigenvalue problem, the generalized
> problem calls some external ScaLAPACK routines. The user is responsible
> for initialization of the BLACS context, which then has to be passed to
> ELPA by **elpa_set**(3) *BEFORE* **elpa_generalized_eigenvalues_float** can be

called.

SEE ALSO
     elpa_init(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3)
     elpa_eigenvalues(3) **elpa_eigenvectors**(3) **elpa_cholesky**(3)
     elpa_invert_triangular(3) **elpa_solve_tridiagonal**(3)
     elpa_hermitian_multiply(3) **elpa_uninit**(3) **elpa_deallocate**(3)

D.35 `elpa_generalized_eigenvectors_float_complex`

NAME

elpa_generalized_eigenvectors_float_complex – computes all eigenvalues and (part of) eigenvectors of a generalized eigenvalue problem, A*Q = λ*B*Q, for complex hermitian matrices A, B.

SYNOPSIS

FORTRAN INTERFACE

```
use elpa
class(elpa_t), pointer :: elpa_handle

call elpa_handle%generalized_eigenvectors (a, b, ev, q,
is_already_decomposed, error)
```

With the definitions of the input and output variables:

class(elpa_t) :: **elpa_handle**
An instance of the ELPA object.

real(kind=c_float_complex) :: **a**   OR   type(c_ptr) :: **a**
The local part of the host/device matrix A for which the eigenvalues and eigenvalues should be computed. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The global matrix has to be symmetric, this is not checked by the routine. In case of a GPU build **a** can be a pointer to the device memory.

real(kind=c_float_complex) :: **b**   OR   type(c_ptr) :: **b**
The host/device local matrix **b** defining the generalized eigenvalue problem. The dimensions of the matrix **b** has to be the same as for matrix **a**. The global matrix has to be symmetric, this is not checked by the routine. In case of a GPU build **b** can be a pointer to the device memory.

real(kind=c_float) :: **ev**   OR   type(c_ptr) :: **ev**
The host/device vector **ev** where the eigenvalues will be stored in *ascending* order. Note that the eigenvalues of complex hermitian matrices are also real. In case of a GPU build **ev** can be a device pointer to a vector **ev** in the device memory.

real(kind=c_float_complex) :: **q**   OR   type(c_ptr) :: **q**
The storage space for the computed eigenvectors Q. The number of requested eigenvectors, nev, must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). In case of a GPU build **q** can be a pointer to the device memory.

logical :: **is_already_decomposed**
Has to be set to .false. for the first call with a given **b** and to .true. for each subsequent call with the same **b**, since **b** then already contains decomposition and thus the decomposing step is skipped.

integer, optional :: **error**
The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3).

C/C++ INTERFACE
```
#include <elpa/elpa.h>
elpa_t elpa_handle;

// C:
void elpa_generalized_eigenvectors_float_complex(elpa_t elpa_handle,
float complex *a, float complex *b, float complex *ev, float complex *q,
int is_already_decomposed, int *error);

// C++:
void elpa_generalized_eigenvectors_float_complex(elpa_t elpa_handle,
std::complex<float>  *a, std::complex<float>  *b, std::complex<float>
*ev, std::complex<float>  *q,  int is_already_decomposed, int *error);
```

With the definitions of the input and output variables:

elpa_t **elpa_handle**;
    The handle to the ELPA object

float complex ***a**;   OR   std::complex<float> ***a**;
    The local matrix **a** for which all eigenvalues and (part of)
    eigenvectors should be computed. The dimensions of the matrix must
    be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The
    global matrix has to be symmetric, this is not checked by the
    routine. In case of a GPU build **a** can be a pointer to the device
    memory.

float complex ***b**;   OR   std::complex<float> ***b**;
    The local matrix **b** defining the generalized eigenvalue problem.
    The dimensions of the matrix **b** must be the same as matrix **a**. The
    global matrix has to be symmetric, this is not checked by the
    routine. In case of a GPU build **a** can be a pointer to the device
    memory.

float ***ev**;
    The array where the eigenvalues λ will be stored in *ascending*
    order. Eigenvalues will be stored in *ascending* order. Note that
    the eigenvalues of complex hermitian matrices are also real. In
    case of a GPU build **a** can be a pointer to the device memory.

float complex ***q**;   OR   std::complex<float> ***q**;
    The storage space for the computed eigenvectors Q. The number of
    requested eigenvectors, nev, must be set *BEFORE* with the methods
    elpa_set(3) and **elpa_setup**(3). In case of a GPU build **a** can be a
    pointer to the device memory.

int **is_already_decomposed**;
    Has to be set to 0 for the first call with a given **b** and 1 for
    each subsequent call with the same **b**, since **b** then already
    contains decomposition and thus the decomposing step is skipped.

int ***error**;
    The error code of the function. Should be "ELPA_OK". The error
    codes can be queried with **elpa_strerr**(3)

DESCRIPTION

    Computes the generalized eigenvalues and (part of) the eigenvectors for
    a complex hermitian generalized eigenproblem. The functions
    elpa_init(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be
    called *BEFORE* **elpa_generalized_eigenvalues_float_complex** can be called.
    In particular, the number of eigenvectors to be computed, nev, can be
    set with **elpa_set**(3). Unlike in the case of ordinary eigenvalue problem,
    the generalized problem calls some external ScaLAPACK routines. The user
    is responsible for initialization of the BLACS context, which then has
    to be passed to ELPA by **elpa_set**(3) *BEFORE*
    elpa_generalized_eigenvalues_float_complex can be called.

SEE ALSO

     elpa_init(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3)
    elpa_eigenvalues(3) **elpa_eigenvectors**(3) **elpa_cholesky**(3)
    elpa_invert_triangular(3) **elpa_solve_tridiagonal**(3)
    elpa_hermitian_multiply(3) **elpa_uninit**(3) **elpa_deallocate**(3)

D.36 `elpa_get_communicators`

NAME
      elpa_get_communicators – splits the global MPI communicator
      mpi_comm_global communicator into rows and column communicators
      mpi_comm_rows and mpi_comm_cols

SYNOPSIS
  FORTRAN INTERFACE
      use elpa1

      status = **elpa_get_communicators** (mpi_comm_global, my_prow, my_pcol,
      mpi_comm_rows, mpi_comm_cols)

      integer, intent(in)   :: **mpi_comm_global**
          Global communicator for the calculation

      integer, intent(in)   :: **my_prow**
          Row coordinate of the calling process in the process grid

      integer, intent(in)   :: **my_pcol**
          Column coordinate of the calling process in the process grid

      integer, intent(out) :: **mpi_comm_rows**
          Communicator for communication within rows of processes

      integer, intent(out) :: **mpi_comm_cols**
          Communicator for communication within columns of processes

      integer   :: **status**
          Return value indicating success or failure of the underlying
          MPI_COMM_SPLIT function

  C/C++ INTERFACE
      #include "elpa_generated.h

      status = **elpa_get_communicators** (int mpi_comm_world, int my_prow, int
      my_pcol, int *mpi_comm_rows, int *mpi_comm_cols);

      int **mpi_comm_global**;
          Global communicator for the calculation

      int **my_prow**;
          Row coordinate of the calling process in the process grid

      int **my_pcol**;
          Column coordinate of the calling process in the process grid

      int ***mpi_comm_rows**;
          Pointer to the communicator for communication within rows of
          processes

      int ***mpi_comm_cols**;
          Pointer to the communicator for communication within columns of
          processes

      int   **status**;
          Return value indicating success or failure of the underlying
          MPI_COMM_SPLIT function

DESCRIPTION

      All ELPA routines need MPI communicators for communicating within rows
      or columns of processes. These communicators are created from the
      mpi_comm_global communicator. It is assumed that the matrix used in ELPA
      is distributed with **my_prow** rows and **my_pcol** columns on the calling
      process. This function has to be invoked by all involved processes
      before any other calls to ELPA routines.

SEE ALSO

      elpa_get_communicators(3) **elpa_solve_evp_real**(3)
      elpa_solve_evp_complex(3) **elpa2_print_kernels**(1)

D.37 `elpa_hermitian_multiply`

NAME
    elpa_hermitian_multiply – performs a "hermitian" multiplication of
    matrices: C = A^T * B for real matrices and C = A^H * B for complex
    matrices

    There are also variations of this routine that can accept not only host
    but also device pointers as input/output. Names of these routines
    explicitly contain the corresponding datatypes:
    elpa_hermitian_multiply_double, elpa_hermitian_multiply_float,
    elpa_hermitian_multiply_double_complex,
    elpa_hermitian_multiply_float_complex.

SYNOPSIS
  FORTRAN INTERFACE
    use elpa
    class(elpa_t), pointer :: elpa_handle

    call elpa_handle%**hermitian_multiply** (uplo_a, uplo_c, ncb, a, b, nrows_b,
    ncols_b, c, nrows_c, ncols_c, error)

    With the definitions of the input and output variables:

    class(elpa_t) :: **elpa_handle**
        An instance of the ELPA object.

    character*1   :: **uplo_a**
        Should be set to 'U' if A is upper triangular, to 'L' if A is
        lower triangular or to anything else if A is a full matrix.

    character*1   :: **uplo_c**
        Should be set to 'U' if only the upper diagonal part of C is
        needed, to 'L' if only the upper diagonal part of C is needed, or
        to anything else if the full matrix C is needed.

    integer   :: **ncb**
        The number of columns of the global matrices **b** and **c**.

    datatype   :: **a**
        The matrix **a**. The dimensions of matrix **a** must be set *BEFORE* with
        the methods **elpa_set**(3) and **elpa_setup**(3). The **datatype** of the
        matrix can be one of "real(kind=c_double)", "real(kind=c_float)",
        "complex(kind=c_double)", or "complex(kind=c_float)".

    datatype   :: **b**
        The matrix **b**. The dimensions of the matrix are specified by the
        parameters **nrows_b** and **ncols_b**. The **datatype** of the matrix can be
        one of "real(kind=c_double)", "real(kind=c_float)",
        "complex(kind=c_double)", or "complex(kind=c_float)".

    integer   :: **nrows_b**
        The number of rows of matrix **b**.

    integer   :: **ncols_b**
        The number of columns of matrix **b**.

    datatype   :: **c**
        The matrix **c**. The dimensions of the matrix are specified by the
        parameters **nrows_c** and **ncols_c**. The **datatype** of the matrix can be
        one of "real(kind=c_double)", "real(kind=c_float)",

```
                    "complex(kind=c_double)", or "complex(kind=c_float)".
       integer         :: nrows_c
             The number of rows of matrix c.

       integer          :: ncols_c
             The number of columns of matrix c.

       integer, optional :: error
             The return error code of the function. Should be "ELPA_OK". The
             error code can be queried with the function elpa_strerr(3)
```

   C/C++ INTERFACE
```
       #include <elpa/elpa.h>
       elpa_t elpa_handle;

       void elpa_hermitian_multiply(elpa_t elpa_handle, char uplo_a, char
       uplo_c, int ncb, datatype *a, datatype *b, int nrows_b, int ncols_b,
       datatype *c, int nrows_c, int ncols_c, int *error);

       With the definitions of the input and output variables:


       elpa_t  elpa_handle;
             The handle to the ELPA object

       char  uplo_a;
             Should be set to 'U' if A is upper triangular, to 'L' if A is
             lower triangular or anything else if A is a full matrix.

       char  uplo_c;
             Should be set to 'U' if only the upper diagonal part of C is
             needed, to 'L' if only the upper diagonal part of C is needed, or
             to anything else if the full matrix C is needed.

       int  ncb;
             The number of columns of the global matrices b and  c.

       datatype  *a;
             The matrix a. The dimensions of matrix a must be set BEFORE with
             the methods elpa_set(3) and elpa_setup(3). The datatype can be one
             of "double", "float", "double complex"/"std::complex<double>",
             "float complex"/"std::complex<float>" for C/C++.

       datatype  *b;
             The matrix b. The dimensions of the matrix are specified by the
             parameters nrows_b and ncols_b. The datatype can be one of
             "double", "float", "double complex"/"std::complex<double>", "float
             complex"/"std::complex<float>" for C/C++.

       int  nrows_b;
             The number of rows of matrix b.

       int ncols_b;
             The number of columns of matrix b.

       datatype  *c;
             The matrix c. The dimensions of the matrix are specified by the
             parametersn rows_c and ncols_c. The datatype can be one of
             "double", "float", "double complex"/"std::complex<double>", "float
             complex"/"std::complex<float>" for C/C++.
```

```
      int  nrows_c;
            The number of rows of matrix c.

      int  ncols_c;
            The number of columns of matrix c.

      int  *error;
            The return error code of the function. Should be "ELPA_OK". The
            error code can be queried with the function elpa_strerr(3)
```

DESCRIPTION
      Performs a "hermitian" multiplication: C = A^T * B for real matrices and
      C=A^H * B for complex matrices. The functions **elpa_init**(3),
      elpa_allocate(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE*
      elpa_hermitian_multiply can be called.

SEE ALSO
      elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3)
      elpa_setup(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3)
      elpa_solve_tridiagonal(3) **elpa_uninit**(3) **elpa_deallocate**(3)

D.38 `elpa_hermitian_multiply_double`

NAME
    elpa_hermitian_multiply_double – performs a "hermitian" multiplication
    of real double-precision matrices: C = A^T * B

SYNOPSIS
  FORTRAN INTERFACE
    use elpa
    class(elpa_t), pointer :: elpa_handle

    call elpa_handle%**hermitian_multiply_double** (uplo_a, uplo_c, ncb, a, b,
    nrows_b, ncols_b, c, nrows_c, ncols_c, error)

    With the definitions of the input and output variables:

    class(elpa_t) :: **elpa_handle**
        An instance of the ELPA object.

    character*1   :: **uplo_a**
        Should be set to 'U' if A is upper triangular, to 'L' if A is
        lower triangular or to anything else if A is a full matrix.

    character*1   :: **uplo_c**
        Should be set to 'U' if only the upper diagonal part of C is
        needed, to 'L' if only the upper diagonal part of C is needed, or
        to anything else if the full matrix C is needed.

    integer   :: **ncb**
        The number of columns of the global matrices **b** and **c**.

    real(kind=c_double) :: **a**   OR   type(c_ptr) :: **a**
        The local part of the host/device matrix A. The dimensions of
        matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and
        elpa_setup(3). In case of a GPU build **a** can be a pointer to the
        device memory.

    real(kind=c_double) :: **b**   OR   type(c_ptr) :: **b**
        The local part of the host/device matrix B. The dimensions of the
        matrix are specified by the parameters **nrows_b** and **ncols_b**. In
        case of a GPU build **b** can be a pointer to the device memory.

    integer   :: **nrows_b**
        The number of rows of matrix **b**.

    integer   :: **ncols_b**
        The number of columns of matrix **b**.

    real(kind=c_double) :: **c**   OR   type(c_ptr) :: **c**
        The local part of the host/device matrix C. The dimensions of the
        matrix are specified by the parameters **nrows_c** and **ncols_c**. In
        case of a GPU build **c** can be a pointer to the device memory.

    integer   :: **nrows_c**
        The number of rows of matrix **c**.

    integer   :: **ncols_c**
        The number of columns of matrix **c**.

    integer, optional :: **error**
        The return error code of the function. Should be "ELPA_OK". The
        error code can be queried with the function **elpa_strerr**(3)

C/C++ INTERFACE
```
#include <elpa/elpa.h>
elpa_t elpa_handle;
```

void **elpa_hermitian_multiply_double**(**elpa_t** elpa_handle, **char** uplo_a,
char uplo_c, **int** ncb, **double** *a, **double** *b, **int** nrows_b, **int** ncols_b,
double *c, **int** nrows_c, **int** ncols_c, **int** *error);

With the definitions of the input and output variables:

elpa_t   **elpa_handle**;
>   The handle to the ELPA object

char   **uplo_a**;
>   Should be set to 'U' if A is upper triangular, to 'L' if A is
>   lower triangular or anything else if A is a full matrix.

char   **uplo_c**;
>   Should be set to 'U' if only the upper diagonal part of C is
>   needed, to 'L' if only the upper diagonal part of C is needed, or
>   to anything else if the full matrix C is needed.

int   **ncb**;
>   The number of columns of the global matrices **b** and **c**.

double   ***a**;
>   The local part of the host/device matrix A. The dimensions of
>   matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and
>   **elpa_setup**(3). In case of a GPU build **a** can be a pointer to the
>   device memory.

double   ***b**;
>   The local part of the host/device matrix B. The dimensions of the
>   matrix are specified by the parameters **nrows_b** and **ncols_b**. In
>   case of a GPU build **b** can be a pointer to the device memory.

int   **nrows_b**;
>   The number of rows of matrix **b**.

int   **ncols_b**;
>   The number of columns of matrix **b**.

double   ***c**;
>   The local part of the host/device matrix C. The dimensions of the
>   matrix are specified by the parameters **n rows_c** and **ncols_c**. In
>   case of a GPU build **c** can be a pointer to the device memory.

int   **nrows_c**;
>   The number of rows of matrix **c**.

int   **ncols_c**;
>   The number of columns of matrix **c**.

int   ***error**;
>   The return error code of the function. Should be "ELPA_OK". The
>   error code can be queried with the function **elpa_strerr**(3)

DESCRIPTION
>   Performs a "hermitian" multiplication C=A^T * B for real double-
>   precision matrices. The functions **elpa_init**(3), **elpa_allocate**(3),

elpa_set(3), and **elpa_setup**(3) must be called *BEFORE*
elpa_hermitian_multiply_double can be called.

SEE ALSO
elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3)
elpa_setup(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3)
elpa_solve_tridiagonal(3) **elpa_uninit**(3) **elpa_deallocate**(3)

D.39 `elpa_hermitian_multiply_double_complex`

NAME
        elpa_hermitian_multiply_double_complex – performs a "hermitian"
        multiplication of complex double-precision matrices: C = A^H * B

SYNOPSIS
  FORTRAN INTERFACE
        use elpa
        class(elpa_t), pointer :: elpa_handle

        call elpa_handle%**hermitian_multiply_double_complex** (uplo_a, uplo_c, ncb,
        a, b, nrows_b, ncols_b, c, nrows_c, ncols_c, error)

        With the definitions of the input and output variables:

        class(elpa_t) :: **elpa_handle**
                An instance of the ELPA object.

        character*1   :: **uplo_a**
                Should be set to 'U' if A is upper triangular, to 'L' if A is
                lower triangular or to anything else if A is a full matrix.

        character*1   :: **uplo_c**
                Should be set to 'U' if only the upper diagonal part of C is
                needed, to 'L' if only the upper diagonal part of C is needed, or
                to anything else if the full matrix C is needed.

        integer   :: **ncb**
                The number of columns of the global matrices **b** and **c**.

        real(kind=c_double_complex) :: **a**   OR   type(c_ptr) :: **a**
                The local part of the host/device matrix A. The dimensions of
                matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and
                elpa_setup(3). In case of a GPU build **a** can be a pointer to the
                device memory.

        real(kind=c_double_complex) :: **b**   OR   type(c_ptr) :: **b**
                The local part of the host/device matrix B. The dimensions of the
                matrix are specified by the parameters **nrows_b** and **ncols_b**. In
                case of a GPU build **b** can be a pointer to the device memory.

        integer   :: **nrows_b**
                The number of rows of matrix **b**.

        integer   :: **ncols_b**
                The number of columns of matrix **b**.

        real(kind=c_double_complex) :: **c**   OR   type(c_ptr) :: **c**
                The local part of the host/device matrix C. The dimensions of the
                matrix are specified by the parameters **nrows_c** and **ncols_c**. In
                case of a GPU build **c** can be a pointer to the device memory.

        integer   :: **nrows_c**
                The number of rows of matrix **c**.

        integer   :: **ncols_c**
                The number of columns of matrix **c**.

        integer, optional :: **error**
                The return error code of the function. Should be "ELPA_OK". The
                error code can be queried with the function **elpa_strerr**(3)

C/C++ INTERFACE
```
#include <elpa/elpa.h>
elpa_t elpa_handle;

// C:
void elpa_hermitian_multiply_double_complex(elpa_t elpa_handle, char
uplo_a, char uplo_c, int ncb, double complex *a, double complex *b, int
nrows_b, int ncols_b, double complex *c, int nrows_c, int ncols_c, int
*error);

// C++:
void elpa_hermitian_multiply_double_complex(elpa_t elpa_handle, char
uplo_a, char uplo_c, int ncb, std::complex<double> *a,
std::complex<double> *b, int nrows_b, int ncols_b, std::complex<double>
*c, int nrows_c, int ncols_c, int *error);
```

With the definitions of the input and output variables:

elpa_t **elpa_handle**;
> The handle to the ELPA object

char **uplo_a**;
> Should be set to 'U' if A is upper triangular, to 'L' if A is
> lower triangular or anything else if A is a full matrix.

char **uplo_c**;
> Should be set to 'U' if only the upper diagonal part of C is
> needed, to 'L' if only the upper diagonal part of C is needed, or
> to anything else if the full matrix C is needed.

int **ncb**;
> The number of columns of the global matrices **b** and **c**.

double complex ***a**;   OR   std::complex<double> ***a**;
> The local part of the host/device matrix A. The dimensions of
> matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and
> **elpa_setup**(3). In case of a GPU build **a** can be a pointer to the
> device memory.

double complex ***b**;   OR   std::complex<double> ***b**;
> The local part of the host/device matrix B. The dimensions of the
> matrix are specified by the parameters **nrows_b** and **ncols_b**. In
> case of a GPU build **b** can be a pointer to the device memory.

int **nrows_b**;
> The number of rows of matrix **b**.

int **ncols_b**;
> The number of columns of matrix **b**.

double complex ***c**;   OR   std::complex<double> ***c**;
> The local part of the host/device matrix C. The dimensions of the
> matrix are specified by the parameters**n rows_c** and **ncols_c**. In
> case of a GPU build **c** can be a pointer to the device memory.

int **nrows_c**;
> The number of rows of matrix **c**.

```
int  ncols_c;
      The number of columns of matrix c.

int  *error;
      The return error code of the function. Should be "ELPA_OK". The
      error code can be queried with the function elpa_strerr(3)
```

DESCRIPTION
Performs a "hermitian" multiplication C=A^H * B for complex double-precision matrices. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE* **elpa_hermitian_multiply_double_complex** can be called.

SEE ALSO
elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3) **elpa_solve_tridiagonal**(3) **elpa_uninit**(3) **elpa_deallocate**(3)

D.40 `elpa_hermitian_multiply_float`

NAME

elpa_hermitian_multiply_float – performs a "hermitian" multiplication of
real float-precision matrices: C = A^T * B

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa_handle

call elpa_handle%**hermitian_multiply_float** (uplo_a, uplo_c, ncb, a, b,
nrows_b, ncols_b, c, nrows_c, ncols_c, error)

With the definitions of the input and output variables:

class(elpa_t) :: **elpa_handle**
     An instance of the ELPA object.

character*1   :: **uplo_a**
     Should be set to 'U' if A is upper triangular, to 'L' if A is
     lower triangular or to anything else if A is a full matrix.

character*1   :: **uplo_c**
     Should be set to 'U' if only the upper diagonal part of C is
     needed, to 'L' if only the upper diagonal part of C is needed, or
     to anything else if the full matrix C is needed.

integer   :: **ncb**
     The number of columns of the global matrices **b** and **c**.

real(kind=c_float) :: **a**   OR   type(c_ptr) :: **a**
     The local part of the host/device matrix A. The dimensions of
     matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and
     elpa_setup(3). In case of a GPU build **a** can be a pointer to the
     device memory.

real(kind=c_float) :: **b**   OR   type(c_ptr) :: **b**
     The local part of the host/device matrix B. The dimensions of the
     matrix are specified by the parameters **nrows_b** and **ncols_b**. In
     case of a GPU build **b** can be a pointer to the device memory.

integer   :: **nrows_b**
     The number of rows of matrix **b**.

integer   :: **ncols_b**
     The number of columns of matrix **b**.

real(kind=c_float) :: **c**   OR   type(c_ptr) :: **c**
     The local part of the host/device matrix C. The dimensions of the
     matrix are specified by the parameters **nrows_c** and **ncols_c**. In
     case of a GPU build **c** can be a pointer to the device memory.

integer   :: **nrows_c**
     The number of rows of matrix **c**.

integer   :: **ncols_c**
     The number of columns of matrix **c**.

integer, optional :: **error**
     The return error code of the function. Should be "ELPA_OK". The
     error code can be queried with the function **elpa_strerr**(3)

C/C++ INTERFACE
```
#include <elpa/elpa.h>
elpa_t elpa_handle;

void elpa_hermitian_multiply_float(elpa_t elpa_handle, char uplo_a, char
uplo_c, int ncb, float *a, float *b, int nrows_b, int ncols_b, float *c,
int nrows_c, int ncols_c, int *error);
```

With the definitions of the input and output variables:

elpa_t **elpa_handle**;
> The handle to the ELPA object

char **uplo_a**;
> Should be set to 'U' if A is upper triangular, to 'L' if A is
> lower triangular or anything else if A is a full matrix.

char **uplo_c**;
> Should be set to 'U' if only the upper diagonal part of C is
> needed, to 'L' if only the upper diagonal part of C is needed, or
> to anything else if the full matrix C is needed.

int **ncb**;
> The number of columns of the global matrices **b** and **c.**

float ***a**;
> The local part of the host/device matrix A. The dimensions of
> matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and
> **elpa_setup**(3). In case of a GPU build **a** can be a pointer to the
> device memory.

float ***b**;
> The local part of the host/device matrix B. The dimensions of the
> matrix are specified by the parameters **nrows_b** and **ncols_b**. In
> case of a GPU build **b** can be a pointer to the device memory.

int **nrows_b**;
> The number of rows of matrix **b.**

int **ncols_b**;
> The number of columns of matrix **b.**

float ***c**;
> The local part of the host/device matrix C. The dimensions of the
> matrix are specified by the parameters **n rows_c** and **ncols_c**. In
> case of a GPU build **c** can be a pointer to the device memory.

int **nrows_c**;
> The number of rows of matrix **c.**

int **ncols_c**;
> The number of columns of matrix **c.**

int ***error**;
> The return error code of the function. Should be "ELPA_OK". The
> error code can be queried with the function **elpa_strerr**(3)

DESCRIPTION
> Performs a "hermitian" multiplication C=A^T * B for real float-precision
> matrices. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and

elpa_setup(3) must be called *BEFORE* **elpa_hermitian_multiply_float** can be called.

SEE ALSO
elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3)
elpa_setup(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3)
elpa_solve_tridiagonal(3) **elpa_uninit**(3) **elpa_deallocate**(3)

D.41 `elpa_hermitian_multiply_float_complex`

NAME

elpa_hermitian_multiply_float_complex – performs a "hermitian"
multiplication of complex float-precision matrices: C = A^H * B

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa_handle

call elpa_handle%**hermitian_multiply_float_complex** (uplo_a, uplo_c, ncb,
a, b, nrows_b, ncols_b, c, nrows_c, ncols_c, error)

With the definitions of the input and output variables:

class(elpa_t) :: **elpa_handle**
    An instance of the ELPA object.

character*1   :: **uplo_a**
    Should be set to 'U' if A is upper triangular, to 'L' if A is
    lower triangular or to anything else if A is a full matrix.

character*1   :: **uplo_c**
    Should be set to 'U' if only the upper diagonal part of C is
    needed, to 'L' if only the upper diagonal part of C is needed, or
    to anything else if the full matrix C is needed.

integer   :: **ncb**
    The number of columns of the global matrices **b** and **c**.

real(kind=c_float_complex) :: **a**   OR   type(c_ptr) :: **a**
    The local part of the host/device matrix A. The dimensions of
    matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and
    elpa_setup(3). In case of a GPU build **a** can be a pointer to the
    device memory.

real(kind=c_float_complex) :: **b**   OR   type(c_ptr) :: **b**
    The local part of the host/device matrix B. The dimensions of the
    matrix are specified by the parameters **nrows_b** and **ncols_b**. In
    case of a GPU build **b** can be a pointer to the device memory.

integer   :: **nrows_b**
    The number of rows of matrix **b**.

integer   :: **ncols_b**
    The number of columns of matrix **b**.

real(kind=c_float_complex) :: **c**   OR   type(c_ptr) :: **c**
    The local part of the host/device matrix C. The dimensions of the
    matrix are specified by the parameters **nrows_c** and **ncols_c**. In
    case of a GPU build **c** can be a pointer to the device memory.

integer   :: **nrows_c**
    The number of rows of matrix **c**.

integer   :: **ncols_c**
    The number of columns of matrix **c**.

integer, optional :: **error**
    The return error code of the function. Should be "ELPA_OK". The
    error code can be queried with the function **elpa_strerr**(3)

C/C++ INTERFACE
```
#include <elpa/elpa.h>
elpa_t elpa_handle;

// C:
void elpa_hermitian_multiply_float_complex(elpa_t elpa_handle, char
uplo_a, char uplo_c, int ncb, float complex *a, float complex *b, int
nrows_b, int ncols_b, float complex *c, int nrows_c, int ncols_c, int
*error);

// C++:
void elpa_hermitian_multiply_float_complex(elpa_t elpa_handle, char
uplo_a, char uplo_c, int ncb, std::complex<float> *a,
std::complex<float> *b, int nrows_b, int ncols_b, std::complex<float>
*c, int nrows_c, int ncols_c, int *error);
```

With the definitions of the input and output variables:

elpa_t **elpa_handle;**
>   The handle to the ELPA object

char **uplo_a;**
>   Should be set to 'U' if A is upper triangular, to 'L' if A is
>   lower triangular or anything else if A is a full matrix.

char **uplo_c;**
>   Should be set to 'U' if only the upper diagonal part of C is
>   needed, to 'L' if only the upper diagonal part of C is needed, or
>   to anything else if the full matrix C is needed.

int **ncb;**
>   The number of columns of the global matrices **b** and **c.**

float complex ***a;**   OR   std::complex<float> ***a;**
>   The local part of the host/device matrix A. The dimensions of
>   matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and
>   **elpa_setup**(3). In case of a GPU build **a** can be a pointer to the
>   device memory.

float complex ***b;**   OR   std::complex<float> ***b;**
>   The local part of the host/device matrix B. The dimensions of the
>   matrix are specified by the parameters **nrows_b** and **ncols_b**. In
>   case of a GPU build **b** can be a pointer to the device memory.

int **nrows_b;**
>   The number of rows of matrix **b**.

int **ncols_b;**
>   The number of columns of matrix **b**.

float complex ***c;**   OR   std::complex<float> ***c;**
>   The local part of the host/device matrix C. The dimensions of the
>   matrix are specified by the parameters**n rows_c** and **ncols_c**. In
>   case of a GPU build **c** can be a pointer to the device memory.

int **nrows_c;**
>   The number of rows of matrix **c**.

>   int **ncols_c;**
>   The number of columns of matrix **c**.
>
>   int ***error;**
>   The return error code of the function. Should be "ELPA_OK". The
>   error code can be queried with the function **elpa_strerr**(3)

DESCRIPTION

>   Performs a "hermitian" multiplication C=AˆH * B for complex float-
>   precision matrices. The functions **elpa_init**(3), **elpa_allocate**(3),
>   **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE*
>   **elpa_hermitian_multiply_float_complex** can be called.

SEE ALSO

>   **elpa2_print_kernels**(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3)
>   **elpa_setup**(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3)
>   **elpa_solve_tridiagonal**(3) **elpa_uninit**(3) **elpa_deallocate**(3)

D.42 `elpa_init`

NAME
      elpa_init – initializes the ELPA library

SYNOPSIS
  FORTRAN INTERFACE
      use elpa

      error = **elpa_init** (api_version)

      With the definitions of the input and output variables:

      integer, intent(in) :: **api_version**
            The api version that you want to initialize, currently the version
            is 20171201

      integer   :: **error**
            The return code. If the function returns without an error, the
            error code will be ELPA_OK.

  C/C++ INTERFACE
      #include <elpa/elpa.h>

      int error = **elpa_init** (**int** api_version);

      With the definitions of the input and output variables:

      int **api_version**;
            The api version that you want to initialize currently the version
            is 20171201

      int **error**;
            The return code. If the function returns without an error, the
            error code will be ELPA_OK.

DESCRIPTION
      Initializes the ELPA library for usage. The return code should be
      ELPA_OK. The return code can be queried with the **elpa_strerr**(3)
      function.

SEE ALSO
      elpa2_print_kernels(1) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3)
      elpa_strerr(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3) **elpa_choleksy**(3)
      elpa_invert_triangular(3) **elpa_solve_tridiagonal**(3)
      elpa_hermitian_multiply(3) **elpa_uninit**(3) **elpa_deallocate**(3)

D.43 `elpa_invert_triangular`

NAME

elpa_invert_triangular – inverts an upper triangular matrix.

There are also variations of this routine that can accept not only host but also device pointers as input/output. Names of these routines explicitly contain the corresponding datatypes: elpa_invert_triangular_double, elpa_invert_triangular_float, elpa_invert_triangular_double_complex, elpa_invert_triangular_float_complex.

SYNOPSIS

  FORTRAN INTERFACE

    use elpa
    class(elpa_t), pointer :: elpa_handle

    call elpa_handle%**invert_triangular** (a, error)

    With the definitions of the input and output variables:

    datatype :: **a**
        The local part of the host/device upper triangular matrix A that should be inverted. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The **datatype** of the matrix can be one of "real(kind=c_double)", "real(kind=c_float)", "complex(kind=c_double)", or "complex(kind=c_float)".

    integer, optional :: **error**
        The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3).

  C/C++ INTERFACE

    #include <elpa/elpa.h>
    elpa_t elpa_handle;

    void **elpa_invert_triangular**(**elpa_t** elpa_handle, **datatype** *a, **int** *error);

    With the definitions of the input and output variables:

    elpa_t **elpa_handle**;
        The handle to the ELPA object

    datatype ***a**;
        The local part of the host/device upper triangular matrix A that should be inverted. The dimensions of the matrix must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The **datatype** can be one of "double", "float", "double complex"/"std::complex<double>", "float complex"/"std::complex<float>" for C/C++.

    int ***error**;
        The error code of the function. Should be "ELPA_OK". The error codes can be queried with **elpa_strerr**(3).

DESCRIPTION
> Inverts an upper triangular real or complex matrix. The functions
> **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be
> called *BEFORE* **elpa_invert_triangular** can be called.

SEE ALSO
> **elpa2_print_kernels**(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3)
> **elpa_setup**(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3)
> **elpa_choleksy**(3) **elpa_solve_tridiagonal**(3) **elpa_hermitian_multiply**(3)
> **elpa_uninit**(3) **elpa_deallocate**(3)

D.44 `elpa_invert_triangular_double`

NAME

elpa_invert_triangular – inverts an upper triangular real double-precision matrix

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa_handle

call elpa_handle%**invert_triangular_double** (a, error)

With the definitions of the input and output variables:

real(kind=c_double) :: **a**　　　OR　　　type(c_ptr) :: **a**
The local part of the host/device upper triangular matrix A that should be inverted. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). In case of a GPU build **a** can be a pointer to the device memory.

integer, optional :: **error**
The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3).

C/C++ INTERFACE

#include <elpa/elpa.h>
elpa_t elpa_handle;

void **elpa_invert_triangular_double**(**elpa_t** elpa_handle, **datatype** *a, **int** *error);

With the definitions of the input and output variables:

elpa_t　**elpa_handle**;
The handle to the ELPA object

double　***a**;
The host/device local matrix that should be inverted. The dimensions of the matrix must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). In case of a GPU build **a** can be a pointer to the device memory.

int　***error**;
The error code of the function. Should be "ELPA_OK". The error codes can be queried with **elpa_strerr**(3).

DESCRIPTION

Inverts an upper triangular real double-precision matrix. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE* **elpa_invert_triangular_double** can be called.

SEE ALSO

elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3) **elpa_choleksy**(3) **elpa_solve_tridiagonal**(3) **elpa_hermitian_multiply**(3) **elpa_uninit**(3) **elpa_deallocate**(3)

D.45 `elpa_invert_triangular_double_complex`

NAME

    elpa_invert_triangular – inverts an upper triangular complex double-precision matrix

SYNOPSIS

  FORTRAN INTERFACE

    use elpa

    class(elpa_t), pointer :: elpa_handle

    call elpa_handle%**invert_triangular_double_complex** (a, error)

    With the definitions of the input and output variables:

    real(kind=c_double_complex) :: **a**   OR   type(c_ptr) :: **a**
        The local part of the host/device upper triangular matrix A that should be inverted. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). In case of a GPU build **a** can be a pointer to the device memory.

    integer, optional :: **error**
        The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3).

  C/C++ INTERFACE

    #include <elpa/elpa.h>

    elpa_t elpa_handle;

    // C:

    void **elpa_invert_triangular_double_complex**(**elpa_t** elpa_handle, **double complex** *a, **int** *error);

    // C++:

    void **elpa_invert_triangular_double_complex**(**elpa_t** elpa_handle, **std::complex<double>** *a, **int** *error);

    With the definitions of the input and output variables:

    elpa_t **elpa_handle**;
        The handle to the ELPA object

    double complex ***a**;   OR   std::complex<double> ***a**;
        The host/device local matrix that should be inverted. The dimensions of the matrix must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). In case of a GPU build **a** can be a pointer to the device memory.

    int ***error**;
        The error code of the function. Should be "ELPA_OK". The error codes can be queried with **elpa_strerr**(3).

DESCRIPTION

    Inverts an upper triangular complex double-precision matrix. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE* **elpa_invert_triangular_double_complex** can be called.

SEE ALSO
       elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3)
       elpa_setup(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3)
       elpa_choleksy(3) **elpa_solve_tridiagonal**(3) **elpa_hermitian_multiply**(3)
       elpa_uninit(3) **elpa_deallocate**(3)

D.46 `elpa_invert_triangular_float`

NAME
　　　　elpa_invert_triangular – inverts an upper triangular real float-
　　　　precision matrix

SYNOPSIS
　　FORTRAN INTERFACE
　　　　use elpa
　　　　class(elpa_t), pointer :: elpa_handle

　　　　call elpa_handle%**invert_triangular_float** (a, error)

　　　　With the definitions of the input and output variables:

　　　　real(kind=c_float) :: **a**　　　OR　　type(c_ptr) :: **a**
　　　　　　The local part of the host/device upper triangular matrix A that
　　　　　　should be inverted. The dimensions of matrix **a** must be set *BEFORE*
　　　　　　with the methods **elpa_set**(3) and **elpa_setup**(3). In case of a GPU
　　　　　　build **a** can be a pointer to the device memory.

　　　　integer, optional :: **error**
　　　　　　The return error code of the function. Should be "ELPA_OK". The
　　　　　　error code can be queried with the function **elpa_strerr**(3).

　　C/C++ INTERFACE
　　　　#include <elpa/elpa.h>
　　　　elpa_t elpa_handle;

　　　　void **elpa_invert_triangular_float**(**elpa_t** elpa_handle, **datatype** *a, **int**
　　　　*error);

　　　　With the definitions of the input and output variables:

　　　　elpa_t　**elpa_handle**;
　　　　　　The handle to the ELPA object

　　　　float　***a**;
　　　　　　The host/device local matrix that should be inverted. The
　　　　　　dimensions of the matrix must be set *BEFORE* with the methods
　　　　　　elpa_set(3) and **elpa_setup**(3). In case of a GPU build **a** can be a
　　　　　　pointer to the device memory.

　　　　int　***error**;
　　　　　　The error code of the function. Should be "ELPA_OK". The error
　　　　　　codes can be queried with **elpa_strerr**(3).

DESCRIPTION
　　　　Inverts an upper triangular real float-precision matrix. The functions
　　　　elpa_init(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be
　　　　called *BEFORE* **elpa_invert_triangular_float** can be called.

SEE ALSO
　　　　elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3)
　　　　elpa_setup(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3)
　　　　elpa_choleksy(3) **elpa_solve_tridiagonal**(3) **elpa_hermitian_multiply**(3)
　　　　elpa_uninit(3) **elpa_deallocate**(3)

D.47 `elpa_invert_triangular_float_complex`

NAME
      elpa_invert_triangular – inverts an upper triangular complex float-
      precision matrix

SYNOPSIS
  FORTRAN INTERFACE
      use elpa
      class(elpa_t), pointer :: elpa_handle

      call elpa_handle%**invert_triangular_float_complex** (a, error)

      With the definitions of the input and output variables:

      real(kind=c_float_complex) :: **a**   OR   type(c_ptr) :: **a**
          The local part of the host/device upper triangular matrix A that
          should be inverted. The dimensions of matrix **a** must be set *BEFORE*
          with the methods **elpa_set**(3) and **elpa_setup**(3). In case of a GPU
          build **a** can be a pointer to the device memory.

      integer, optional :: **error**
          The return error code of the function. Should be "ELPA_OK". The
          error code can be queried with the function **elpa_strerr**(3).

  C/C++ INTERFACE
      #include <elpa/elpa.h>
      elpa_t elpa_handle;

      // C:
      void **elpa_invert_triangular_float_complex**(**elpa_t** elpa_handle, **float
      complex** *a, **int** *error);

      // C++:
      void **elpa_invert_triangular_float_complex**(**elpa_t** elpa_handle,
      std::complex<float> *a, **int** *error);

      With the definitions of the input and output variables:

      elpa_t **elpa_handle**;
          The handle to the ELPA object

      float complex ***a**;   OR   std::complex<float> ***a**;
          The host/device local matrix that should be inverted. The
          dimensions of the matrix must be set *BEFORE* with the methods
          elpa_set(3) and **elpa_setup**(3). In case of a GPU build **a** can be a
          pointer to the device memory.

      int ***error**;
          The error code of the function. Should be "ELPA_OK". The error
          codes can be queried with **elpa_strerr**(3).

DESCRIPTION
      Inverts an upper triangular complex float-precision matrix. The
      functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3)
      must be called *BEFORE* **elpa_invert_triangular_float_complex** can be
      called.

SEE ALSO
elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3)
elpa_setup(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3)
elpa_choleksy(3) **elpa_solve_tridiagonal**(3) **elpa_hermitian_multiply**(3)
elpa_uninit(3) **elpa_deallocate**(3)

D.48 `elpa_load_settings`

NAME
      elpa_load_settings – loads the setting of an elpa object

SYNOPSIS
  FORTRAN INTERFACE
      use elpa
      class(elpa_t), pointer :: elpa_handle

      call elpa_handle%**load_settings** (filename, error)

      With the definitions of the input and output variables:

      class(elpa_t)   :: **elpa**
           An instance of the ELPA object

      character(*)   :: **filename**
           The file from where to load the settings

      integer, optional :: **error**
           An error return code

  C/C++ INTERFACE
      #include <elpa/elpa.h>
      elpa_t elpa_handle;

      void **elpa_load_settings**(**elpa_t** elpa_handle, **const char** *filename, **int**
      *error);

      With the definitions of the input and output variables:

      elpa_t **elpa_handle**;
           The handle to the ELPA object

      const char ***filename**;
           The filename to load the settings

      int ***error**;
           The error return code

DESCRIPTION
      Loads all the settings of an previously stored ELPA object from a file
      specified via the **filename** parameter.

SEE ALSO
      elpa_store_setting(3)

D.49 `elpa_print_settings`

NAME
　　　　elpa_print_settings – prints the setting of an elpa object

SYNOPSIS
　　FORTRAN INTERFACE
　　　　use elpa
　　　　class(elpa_t), pointer :: elpa_handle

　　　　call elpa_handle%**print_settings** (error)

　　　　With the definitions of the input and output variables:

　　　　class(elpa_t)　　:: **elpa_handle**
　　　　　　An instance of the ELPA object

　　　　integer, optional :: **error**
　　　　　　An error return code

　　C/C++ INTERFACE
　　　　#include <elpa/elpa.h>
　　　　elpa_t elpa_handle;

　　　　void **elpa_print_settings**(**elpa_t** elpa_handle, **int** *error);

　　　　With the definitions of the input and output variables:

　　　　elpa_t　**elpa_handle**;
　　　　　　The handle to the ELPA object

　　　　int　***error**;
　　　　　　The error return code

DESCRIPTION
　　　　Prints all the settings of an ELPA object. The settings can be stored,
　　　　or loaded with **elpa_store_settings**.3 or **elpa_load_settings**.3

SEE ALSO
　　　　elpa_store_setting(3) **elpa_load_settings**.(3)

D.50 `elpa_print_times`

NAME
    elpa_print_times – prints the timings of individual ELPA solution steps.

SYNOPSIS
  FORTRAN INTERFACE
    use elpa
    class(elpa_t), pointer :: elpa_handle

    call elpa_handle%**print_times** (name)

    With the definitions of the input and output variables:

    class(elpa_t) :: **elpa_handle**
        An instance of the ELPA object

    character(*)   :: **name**
        The name of the ELPA procedure for which the timings should be
        printed.

  C/C++ INTERFACE
    #include <elpa/elpa.h>
    elpa_t elpa_handle;

    void **elpa_print_times**(**elpa_t** elpa_handle, **char** *name);

    With the definitions of the input and output variables:

    elpa_t **handle**;
        The handle to the ELPA object

    char ***name**;
        The name of the ELPA procedure for which the timings should be
        printed.

DESCRIPTION
    Prints the timings of individual ELPA solution steps. Can be invoked
    after the calls to **elpa_timer_start** and **elpa_timer_stop** with the same
    name argument. In order timings were printed, the **timings** parameter
    should be set to **1** by **elpa_set**.

SEE ALSO
    elpa_timer_start(3) **elpa_timer_stop**(3)

D.51 `elpa_pxgemm_multiply`

NAME

elpa_pxgemm_multiply – performs multiplication of two block-cyclic distributed matrices C = op(A)*op(B)

THIS IS AN EXPERIMENTAL ROUTINE. FOR NOW ONLY SQUARE MATRICES **(ncb=elpa_handle%na)**, HAVING THE SAME BLOCK-CYCLIC DISTRIBUTION **(nrows_b=nrows_c=elpa_handle%local_nrows, ncols_b=ncols_c=elpa_handle%local_ncols)** ARE SUPPORTED. USE WITH CARE.

There are also variations of this routine that can accept not only host but also device pointers as input/output. Names of these routines explicitly contain the corresponding datatypes: elpa_pxgemm_multiply_double, elpa_pxgemm_multiply_float, elpa_pxgemm_multiply_double_complex, elpa_pxgemm_multiply_float_complex.

SYNOPSIS

  FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa_handle

call elpa_handle%**pxgemm_multiply** (trans_a, trans_b, ncb, a, b, nrows_b, ncols_b, c, nrows_c, ncols_c, error)

With the definitions of the input and output variables:

class(elpa_t) :: **elpa_handle**
    An instance of the ELPA object.

character*1   :: **trans_a**
    Should be set to 'N' if A is non-transposed: op(A) = A; to 'T' if A is transposed: op(A) = AˆT; to 'H' if A is conjugate-transposed: op(A) = AˆH.

character*1   :: **trans_b**
    Should be set to 'N' if B is non-transposed: op(B) = B; to 'T' if B is transposed: op(B) = BˆT; to 'H' if B is conjugate-transposed op(B): = BˆH.

integer   :: **ncb**
    The number of columns of the global matrices **B** and **C**.

datatype   :: **a**
    The local part of matrix A. The local dimensions of matrix **a** (elpa_handle%local_nrows, elpa_handle%local_ncols) as well as the corresponding global dimensions (elpa_handle%na, elpa_handle%na) must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The **datatype** of the matrix can be one of "real(kind=c_double)", "real(kind=c_float)", "complex(kind=c_double)", or "complex(kind=c_float)".

datatype   :: **b**
    The local part of matrix B. The local dimensions of the matrix are specified by the parameters **nrows_b** and **ncols_b**. The **datatype** of the matrix can be one of "real(kind=c_double)", "real(kind=c_float)", "complex(kind=c_double)", or "complex(kind=c_float)".

```
integer        :: nrows_b
      The local number of rows of matrix b.
```

```
integer        :: ncols_b
      The local number of columns of matrix b.
```

```
datatype       :: c
      The local part of matrix C. The local dimensions of the matrix are
      specified by the parameters nrows_c and ncols_c. The datatype of
      the matrix can be one of "real(kind=c_double)",
      "real(kind=c_float)", "complex(kind=c_double)", or
      "complex(kind=c_float)".
```

```
integer        :: nrows_c
      The local number of rows of matrix c.
```

```
integer        :: ncols_c
      The local number of columns of matrix c.
```

```
integer, optional :: error
      The return error code of the function. Should be "ELPA_OK". The
      error code can be queried with the function elpa_strerr(3)
```

C/C++ INTERFACE

```
#include <elpa/elpa.h>
elpa_t elpa_handle;
```

```
void elpa_pxgemm_multiply(elpa_t elpa_handle, char trans_a, char
trans_b, int ncb, datatype *a, datatype *b, int nrows_b, int ncols_b,
datatype *c, int nrows_c, int ncols_c, int *error);
```

With the definitions of the input and output variables:

```
elpa_t  elpa_handle;
      The handle to the ELPA object
```

```
char  trans_a;
      Should be set to 'N' if A is non-transposed: op(A) = A; to 'T' if
      A is transposed: op(A) = A^T; to 'H' if A is conjugate-transposed:
      op(A) = A^H.
```

```
char  trans_b;
      Should be set to 'N' if B is non-transposed: op(B) = B; to 'T' if
      B is transposed: op(B) = B^T; to 'H' if B is conjugate-transposed:
      op(B) = B^H.
```

```
int  ncb;
      The number of columns of the global matrices B and  C.
```

```
datatype  *a;
      The local matrix a. The local dimensions of matrix a (local_nrows,
      local_ncols) as well as the corresponding global dimensions (na,
      na) must be set BEFORE with the methods elpa_set(3) and
      elpa_setup(3). The datatype can be one of "double", "float",
      "double complex"/"std::complex<double>", "float
      complex"/"std::complex<float>" for C/C++.
```

```
datatype  *b;
      The local matrix b. The local dimensions of the matrix are
      specified by the parameters nrows_b and ncols_b. The datatype can
```

be one of "double", "float", "double
complex"/"std::complex<double>", "float
complex"/"std::complex<float>" for C/C++.

int **nrows_b**;
The local number of rows of matrix **b**.

int **ncols_b**;
The local number of columns of matrix **b**.

datatype ***c**;
The local matrix **c**. The local dimensions of the matrix are
specified by the parameters **nrows_c** and **ncols_c**. The **datatype** can
be one of "double", "float", "double
complex"/"std::complex<double>", "float
complex"/"std::complex<float>" for C/C++.

int **nrows_c**;
The number of rows of matrix **c**.

int **ncols_c**;
The number of columns of matrix **c**.

int ***error**;
The return error code of the function. Should be "ELPA_OK". The
error code can be queried with the function **elpa_strerr**(3)

DESCRIPTION

Performs a matrix multiplication: C = op(A)* op(B) where op(A)=A or
op(A)=A^T/H for real/complex matrices. The functions **elpa_init**(3),
elpa_allocate(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE*
elpa_pxgemm_multiply can be called.

SEE ALSO

 elpa_init(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3)
elpa_eigenvalues(3) **elpa_eigenvectors**(3) **elpa_solve_tridiagonal**(3)
elpa_uninit(3) **elpa_deallocate**(3)

D.52 `elpa_pxgemm_multiply_double`

NAME

elpa_pxgemm_multiply_double – performs multiplication of two block-cyclic distributed matrices C = op(A)*op(B)

THIS IS AN EXPERIMENTAL ROUTINE. FOR NOW ONLY SQUARE MATRICES **(ncb=elpa_handle%na)**, HAVING THE SAME BLOCK-CYCLIC DISTRIBUTION **(nrows_b=nrows_c=elpa_handle%local_nrows, ncols_b=ncols_c=elpa_handle%local_ncols)** ARE SUPPORTED. USE WITH CARE.

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa_handle

call elpa_handle%**pxgemm_multiply_double** (trans_a, trans_b, ncb, a, b, nrows_b, ncols_b, c, nrows_c, ncols_c, error)

With the definitions of the input and output variables:

class(elpa_t) :: **elpa_handle**
        An instance of the ELPA object.

character*1   :: **trans_a**
        Should be set to 'N' if A is non-transposed: op(A) = A; to 'T' if A is transposed: op(A) = A^T.

character*1   :: **trans_b**
        Should be set to 'N' if B is non-transposed: op(B) = B; to 'T' if B is transposed: op(B) = B^T.

integer   :: **ncb**
        The number of columns of the global matrices **B** and **C**.

real(kind=c_double) :: **a**   OR   type(c_ptr) :: **a**
        The local part of the host/device matrix A. The local dimensions of matrix **a** (elpa_handle%local_nrows, elpa_handle%local_ncols) as well as the corresponding global dimensions (elpa_handle%na, elpa_handle%na) must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). In case of a GPU build **a** can be a pointer to the device memory.

real(kind=c_double) :: **b**   OR   type(c_ptr) :: **b**
        The local part of the host/device matrix B. The local matrix **b**. The local dimensions of the matrix are specified by the parameters **nrows_b** and **ncols_b**. In case of a GPU build **b** can be a pointer to the device memory.

integer   :: **nrows_b**
        The local number of rows of matrix **b**.

integer   :: **ncols_b**
        The local number of columns of matrix **b**.

real(kind=c_double) :: **c**   OR   type(c_ptr) :: **c**
        The local part of the host/device matrix C. The local dimensions of the matrix are specified by the parameters **nrows_c** and **ncols_c**. In case of a GPU build **c** can be a pointer to the device memory.

```
integer        :: nrows_c
     The local number of rows of matrix c.

integer        :: ncols_c
     The local number of columns of matrix c.

integer, optional :: error
     The return error code of the function. Should be "ELPA_OK". The
     error code can be queried with the function elpa_strerr(3)
```

C/C++ INTERFACE

```
#include <elpa/elpa.h>
elpa_t elpa_handle;

void elpa_pxgemm_multiply_double(elpa_t elpa_handle, char trans_a, char
trans_b, int ncb, double *a, double *b, int nrows_b, int ncols_b, double
*c, int nrows_c, int ncols_c, int *error);
```

With the definitions of the input and output variables:

```
elpa_t elpa_handle;
     The handle to the ELPA object

char   trans_a;
     Should be set to 'N' if A is non-transposed: op(A) = A; to 'T' if
     A is transposed: op(A) = A^T.

char   trans_b;
     Should be set to 'N' if B is non-transposed: op(B) = B; to 'T' if
     B is transposed: op(B) = B^T.

int    ncb;
     The number of columns of the global matrices B and  C.

double *a;
     The local part of the host/device matrix A. The local dimensions
     of matrix a (local_nrows, local_ncols) as well as the
     corresponding global dimensions (na, na) must be set BEFORE with
     the methods elpa_set(3) and elpa_setup(3). In case of a GPU build
     a can be a pointer to the device memory.

double *b;
     The local part of the host/device matrix B. The local dimensions
     of the matrix are specified by the parameters nrows_b and ncols_b.
     In case of a GPU build b can be a pointer to the device memory.

int    nrows_b;
     The local number of rows of matrix b.

int    ncols_b;
     The local number of columns of matrix b.

double *c;
     The local part of the host/device matrix C. The local dimensions
     of the matrix are specified by the parameters nrows_c and ncols_c.
     In case of a GPU build c can be a pointer to the device memory.

int    nrows_c;
     The local number of rows of matrix c.
```

      int   **ncols_c;**
          The local number of columns of matrix **c**.

      int   ***error;**
          The return error code of the function. Should be "ELPA_OK". The
          error code can be queried with the function **elpa_strerr**(3)

DESCRIPTION
      Performs a matrix multiplication: C = op(A)* op(B) where op(A)=A or
      op(A)=A^T for double real matrices. Can use either host or device
      pointers for the matrices **a**, **b**, and **c**. The functions **elpa_init**(3),
      elpa_allocate(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE*
      elpa_pxgemm_multiply_double can be called.

SEE ALSO
      elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3)
      elpa_setup(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3)
      elpa_solve_tridiagonal(3) **elpa_uninit**(3) **elpa_deallocate**(3)

D.53 `elpa_pxgemm_multiply_double_complex`

NAME

elpa_pxgemm_multiply_double_complex – performs multiplication of two
block-cyclic distributed matrices C = op(A)*op(B)

THIS IS AN EXPERIMENTAL ROUTINE. FOR NOW ONLY SQUARE MATRICES
(ncb=elpa_handle%na), HAVING THE SAME BLOCK-CYCLIC DISTRIBUTION
(nrows_b=nrows_c=elpa_handle%local_nrows,
ncols_b=ncols_c=elpa_handle%local_ncols) ARE SUPPORTED. USE WITH CARE.

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa_handle

call elpa_handle%**pxgemm_multiply_double_complex** (trans_a, trans_b, ncb,
a, b, nrows_b, ncols_b, c, nrows_c, ncols_c, error)

With the definitions of the input and output variables:

class(elpa_t) :: **elpa_handle**
      An instance of the ELPA object.

character*1   :: **trans_a**
      Should be set to 'N' if A is non-transposed: op(A) = A; to 'H' if
      A is conjugate-transposed: op(A) = A^H.

character*1   :: **trans_b**
      Should be set to 'N' if B is non-transposed: op(B) = B; to 'H' if
      B is conjugate-transposed: op(B) = B^H.

integer   :: **ncb**
      The number of columns of the global matrices **B** and **C**.

real(kind=c_double_complex) :: **a**   OR   type(c_ptr) :: **a**
      The local part of the host/device matrix A. The local dimensions
      of matrix **a** (elpa_handle%local_nrows, elpa_handle%local_ncols) as
      well as the corresponding global dimensions (elpa_handle%na,
      elpa_handle%na) must be set *BEFORE* with the methods **elpa_set**(3)
      and **elpa_setup**(3). In case of a GPU build **a** can be a pointer to
      the device memory.

real(kind=c_double_complex) :: **b**   OR   type(c_ptr) :: **b**
      The local part of the host/device matrix B. The local matrix **b**.
      The local dimensions of the matrix are specified by the parameters
      nrows_b and **ncols_b**. In case of a GPU build **b** can be a pointer to
      the device memory.

integer   :: **nrows_b**
      The local number of rows of matrix **b**.

integer   :: **ncols_b**
      The local number of columns of matrix **b**.

real(kind=c_double_complex) :: **c**   OR   type(c_ptr) :: **c**
      The local part of the host/device matrix C. The local dimensions
      of the matrix are specified by the parameters **nrows_c** and **ncols_c**.
      In case of a GPU build **c** can be a pointer to the device memory.

```
integer          :: nrows_c
       The local number of rows of matrix c.

integer          :: ncols_c
       The local number of columns of matrix c.

integer, optional :: error
       The return error code of the function. Should be "ELPA_OK". The
       error code can be queried with the function elpa_strerr(3)
```

C/C++ INTERFACE
```
#include <elpa/elpa.h>
elpa_t elpa_handle;

// C:
void elpa_pxgemm_multiply_double_complex(elpa_t elpa_handle, char
trans_a, char trans_b, int ncb, double complex *a, double complex *b,
int nrows_b, int ncols_b, double complex *c, int nrows_c, int ncols_c,
int *error);

// C++:
void elpa_pxgemm_multiply_double_complex(elpa_t elpa_handle, char
trans_a, char trans_b, int ncb, std::complex<double> *a,
std::complex<double> *b, int nrows_b, int ncols_b, std::complex<double>
*c, int nrows_c, int ncols_c, int *error);
```

With the definitions of the input and output variables:

elpa_t **elpa_handle**;
       The handle to the ELPA object

char   **trans_a**;
       Should be set to 'N' if A is non-transposed: op(A) = A; to 'H' if
       A is conjugate-transposed: op(A) = A^H.

char   **trans_b**;
       Should be set to 'N' if B is non-transposed: op(B) = B; to 'H' if
       B is conjugate-transposed: op(B) = B^H.

int   **ncb**;
       The number of columns of the global matrices **B** and **C**.

double complex ***a**;   OR   std::complex<double> ***a**;
       The local part of the host/device matrix A. The local dimensions
       of matrix **a** (local_nrows, local_ncols) as well as the
       corresponding global dimensions (na, na) must be set *BEFORE* with
       the methods **elpa_set**(3) and **elpa_setup**(3). In case of a GPU build
       a can be a pointer to the device memory.

double complex ***b**;   OR   std::complex<double> ***b**;
       The local part of the host/device matrix B. The local dimensions
       of the matrix are specified by the parameters **nrows_b** and **ncols_b**.
       In case of a GPU build **b** can be a pointer to the device memory.

int   **nrows_b**;
       The local number of rows of matrix **b**.

```
int    ncols_b;
       The local number of columns of matrix b.
```

```
double complex *c;      OR    std::complex<double> *c;
       The local part of the host/device matrix C. The local dimensions
       of the matrix are specified by the parameters nrows_c and ncols_c.
       In case of a GPU build c can be a pointer to the device memory.
```

```
int    nrows_c;
       The local number of rows of matrix c.
```

```
int    ncols_c;
       The local number of columns of matrix c.
```

```
int    *error;
       The return error code of the function. Should be "ELPA_OK". The
       error code can be queried with the function elpa_strerr(3)
```

DESCRIPTION
     Performs a matrix multiplication: C = op(A)* op(B) where op(A)=A or
     op(A)=A^H for double complex matrices. Can use either host or device
     pointers for the matrices **a**, **b**, and **c**. The functions **elpa_init**(3),
     elpa_allocate(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE*
     elpa_pxgemm_multiply_double_complex can be called.

SEE ALSO
     elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3)
     elpa_setup(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3)
     elpa_solve_tridiagonal(3) **elpa_uninit**(3) **elpa_deallocate**(3)

D.54 `elpa_pxgemm_multiply_float`

NAME

elpa_pxgemm_multiply_float – performs multiplication of two block-cyclic
distributed matrices C = op(A)*op(B)

THIS IS AN EXPERIMENTAL ROUTINE. FOR NOW ONLY SQUARE MATRICES
(ncb=elpa_handle%na), HAVING THE SAME BLOCK-CYCLIC DISTRIBUTION
(nrows_b=nrows_c=elpa_handle%local_nrows,
ncols_b=ncols_c=elpa_handle%local_ncols) ARE SUPPORTED. USE WITH CARE.

SYNOPSIS

   FORTRAN INTERFACE

   use elpa
   class(elpa_t), pointer :: elpa_handle

   call elpa_handle%**pxgemm_multiply_float** (trans_a, trans_b, ncb, a, b,
   nrows_b, ncols_b, c, nrows_c, ncols_c, error)

   With the definitions of the input and output variables:

   class(elpa_t) :: **elpa_handle**
         An instance of the ELPA object.

   character*1   :: **trans_a**
         Should be set to 'N' if A is non-transposed: op(A) = A; to 'T' if
         A is transposed: op(A) = A^T.

   character*1   :: **trans_b**
         Should be set to 'N' if B is non-transposed: op(B) = B; to 'T' if
         B is transposed: op(B) = B^T.

   integer   :: **ncb**
         The number of columns of the global matrices **B** and **C**.

   real(kind=c_float) :: **a**   OR   type(c_ptr) :: **a**
         The local part of the host/device matrix A. The local dimensions
         of matrix **a** (elpa_handle%local_nrows, elpa_handle%local_ncols) as
         well as the corresponding global dimensions (elpa_handle%na,
         elpa_handle%na) must be set *BEFORE* with the methods **elpa_set**(3)
         and **elpa_setup**(3). In case of a GPU build **a** can be a pointer to
         the device memory.

   real(kind=c_float) :: **b**   OR   type(c_ptr) :: **b**
         The local part of the host/device matrix B. The local matrix **b**.
         The local dimensions of the matrix are specified by the parameters
         nrows_b and **ncols_b**. In case of a GPU build **b** can be a pointer to
         the device memory.

   integer   :: **nrows_b**
         The local number of rows of matrix **b**.

   integer   :: **ncols_b**
         The local number of columns of matrix **b**.

   real(kind=c_float) :: **c**   OR   type(c_ptr) :: **c**
         The local part of the host/device matrix C. The local dimensions
         of the matrix are specified by the parameters **nrows_c** and **ncols_c**.
         In case of a GPU build **c** can be a pointer to the device memory.

```
integer        :: nrows_c
      The local number of rows of matrix c.
```

```
integer        :: ncols_c
      The local number of columns of matrix c.
```

```
integer, optional :: error
      The return error code of the function. Should be "ELPA_OK". The
      error code can be queried with the function elpa_strerr(3)
```

C/C++ INTERFACE
```
#include <elpa/elpa.h>
elpa_t elpa_handle;
```

void **elpa_pxgemm_multiply_float**(**elpa_t** elpa_handle, **char** trans_a, **char** trans_b, **int** ncb, **float** *a, **float** *b, **int** nrows_b, **int** ncols_b, **float** *c, **int** nrows_c, **int** ncols_c, **int** *error);

With the definitions of the input and output variables:

```
elpa_t elpa_handle;
      The handle to the ELPA object
```

```
char    trans_a;
      Should be set to 'N' if A is non-transposed: op(A) = A; to 'T' if
      A is transposed: op(A) = AˆT.
```

```
char    trans_b;
      Should be set to 'N' if B is non-transposed: op(B) = B; to 'T' if
      B is transposed: op(B) = BˆT.
```

```
int    ncb;
      The number of columns of the global matrices B and  C.
```

```
float  *a;
      The local part of the host/device matrix A. The local dimensions
      of matrix a (local_nrows, local_ncols) as well as the
      corresponding global dimensions (na, na) must be set BEFORE with
      the methods elpa_set(3) and elpa_setup(3). In case of a GPU build
      a can be a pointer to the device memory.
```

```
float  *b;
      The local part of the host/device matrix B. The local dimensions
      of the matrix are specified by the parameters nrows_b and ncols_b.
      In case of a GPU build b can be a pointer to the device memory.
```

```
int    nrows_b;
      The local number of rows of matrix b.
```

```
int    ncols_b;
      The local number of columns of matrix b.
```

```
float  *c;
      The local part of the host/device matrix C. The local dimensions
      of the matrix are specified by the parameters nrows_c and ncols_c.
      In case of a GPU build c can be a pointer to the device memory.
```

```
int    nrows_c;
      The local number of rows of matrix c.
```

```
       int     ncols_c;
              The local number of columns of matrix c.

       int     *error;
              The return error code of the function. Should be "ELPA_OK". The
              error code can be queried with the function elpa_strerr(3)
```

DESCRIPTION

      Performs a matrix multiplication: C = op(A)* op(B) where op(A)=A or
op(A)=A^T for float real matrices. Can use either host or device
pointers for the matrices **a**, **b**, and **c**. The functions **elpa_init**(3),
elpa_allocate(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE*
elpa_pxgemm_multiply_float can be called.

SEE ALSO

      elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3)
elpa_setup(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3)
elpa_solve_tridiagonal(3) **elpa_uninit**(3) **elpa_deallocate**(3)

D.55 `elpa_pxgemm_multiply_float_complex`

NAME

elpa_pxgemm_multiply_float_complex – performs multiplication of two
block-cyclic distributed matrices C = op(A)*op(B)

THIS IS AN EXPERIMENTAL ROUTINE. FOR NOW ONLY SQUARE MATRICES
(ncb=elpa_handle%na), HAVING THE SAME BLOCK-CYCLIC DISTRIBUTION
**(nrows_b=nrows_c=elpa_handle%local_nrows,
ncols_b=ncols_c=elpa_handle%local_ncols)** ARE SUPPORTED. USE WITH CARE.

SYNOPSIS

  FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa_handle

call elpa_handle%**pxgemm_multiply_float_complex** (trans_a, trans_b, ncb,
a, b, nrows_b, ncols_b, c, nrows_c, ncols_c, error)

With the definitions of the input and output variables:

class(elpa_t) :: **elpa_handle**
      An instance of the ELPA object.

character*1   :: **trans_a**
      Should be set to 'N' if A is non-transposed: op(A) = A; to 'H' if
      A is conjugate-transposed: op(A) = A^H.

character*1   :: **trans_b**
      Should be set to 'N' if B is non-transposed: op(B) = B; to 'H' if
      B is conjugate-transposed: op(B) = B^H.

integer   :: **ncb**
      The number of columns of the global matrices **B** and **C**.

real(kind=c_float_complex) :: **a**   OR   type(c_ptr) :: **a**
      The local part of the host/device matrix A. The local dimensions
      of matrix **a** (elpa_handle%local_nrows, elpa_handle%local_ncols) as
      well as the corresponding global dimensions (elpa_handle%na,
      elpa_handle%na) must be set *BEFORE* with the methods **elpa_set**(3)
      and **elpa_setup**(3). In case of a GPU build **a** can be a pointer to
      the device memory.

real(kind=c_float_complex) :: **b**   OR   type(c_ptr) :: **b**
      The local part of the host/device matrix B. The local matrix **b**.
      The local dimensions of the matrix are specified by the parameters
      nrows_b and **ncols_b**. In case of a GPU build **b** can be a pointer to
      the device memory.

integer   :: **nrows_b**
      The local number of rows of matrix **b**.

integer   :: **ncols_b**
      The local number of columns of matrix **b**.

real(kind=c_float_complex) :: **c**   OR   type(c_ptr) :: **c**
      The local part of the host/device matrix C. The local dimensions
      of the matrix are specified by the parameters **nrows_c** and **ncols_c**.
      In case of a GPU build **c** can be a pointer to the device memory.

```
integer        :: nrows_c
      The local number of rows of matrix c.
```

```
integer        :: ncols_c
      The local number of columns of matrix c.
```

```
integer, optional :: error
      The return error code of the function. Should be "ELPA_OK". The
      error code can be queried with the function elpa_strerr(3)
```

C/C++ INTERFACE
```
#include <elpa/elpa.h>
elpa_t elpa_handle;
```

```
// C:
void elpa_pxgemm_multiply_float_complex(elpa_t elpa_handle, char
trans_a, char trans_b, int ncb, float complex *a, float complex *b, int
nrows_b, int ncols_b, float complex *c, int nrows_c, int ncols_c, int
*error);
```

```
// C++:
void elpa_pxgemm_multiply_float_complex(elpa_t elpa_handle, char
trans_a, char trans_b, int ncb, std::complex<float> *a,
std::complex<float> *b, int nrows_b, int ncols_b, std::complex<float>
*c, int nrows_c, int ncols_c, int *error);
```

```
With the definitions of the input and output variables:
```

```
elpa_t elpa_handle;
      The handle to the ELPA object
```

```
char   trans_a;
      Should be set to 'N' if A is non-transposed: op(A) = A; to 'H' if
      A is conjugate-transposed: op(A) = A^H.
```

```
char   trans_b;
      Should be set to 'N' if B is non-transposed: op(B) = B; to 'H' if
      B is conjugate-transposed: op(B) = B^H.
```

```
int    ncb;
      The number of columns of the global matrices B and  C.
```

```
float complex *a;      OR   std::complex<float> *a;
      The local part of the host/device matrix A. The local dimensions
      of matrix a (local_nrows, local_ncols) as well as the
      corresponding global dimensions (na, na) must be set BEFORE with
      the methods elpa_set(3) and elpa_setup(3). In case of a GPU build
      a can be a pointer to the device memory.
```

```
float complex *b;      OR   std::complex<float> *b;
      The local part of the host/device matrix B. The local dimensions
      of the matrix are specified by the parameters nrows_b and ncols_b.
      In case of a GPU build b can be a pointer to the device memory.
```

```
int    nrows_b;
      The local number of rows of matrix b.
```

```
int    ncols_b;
       The local number of columns of matrix b.

float complex *c;       OR    std::complex<float> *c;
       The local part of the host/device matrix C. The local dimensions
       of the matrix are specified by the parameters nrows_c and ncols_c.
       In case of a GPU build c can be a pointer to the device memory.

int    nrows_c;
       The local number of rows of matrix c.

int    ncols_c;
       The local number of columns of matrix c.

int    *error;
       The return error code of the function. Should be "ELPA_OK". The
       error code can be queried with the function elpa_strerr(3)
```

DESCRIPTION

Performs a matrix multiplication: C = op(A)* op(B) where op(A)=A or
op(A)=A^H for float complex matrices. Can use either host or device
pointers for the matrices **a**, **b**, and **c**. The functions **elpa_init**(3),
elpa_allocate(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE*
elpa_pxgemm_multiply_float_complex can be called.

SEE ALSO

elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3)
elpa_setup(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3)
elpa_solve_tridiagonal(3) **elpa_uninit**(3) **elpa_deallocate**(3)

D.56 `elpa_set`

NAME

elpa_set – set a parameter for the ELPA object.

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa_handle

call elpa_handle%**set** (name, value, error)

With the definitions of the input and output variables:

class(elpa_t)   :: **elpa**
  An instance of the ELPA object.

character(*)   :: **name**
  The name of the option to be set.

datatype   :: **value**
  the value which should be assigned to the option **name**. The
  datatype can be **integer** or **real(kind=c_double)**.

integer, optional :: **error**
  The returned error code. On success it is ELPA_OK, otherwise an
  error. he error code can be queried with **elpa_strerr**(3)

C/C++ INTERFACE

#include <elpa/elpa.h>
elpa_t elpa_handle;

void elpa_set (**elpa_t** elpa_handle, **const char** *name, **datatype** value, **int**
*error);

With the definitions of the input and output variables:

elpa_t **elpa_handle**;
  The handle of an ELPA object, obtained before with
  elpa_allocate(3)

const char ***name**;
  The name of the option to be set.

datatype **value**;
  The value which should be assigned to the option **name**. The
  datatype can be either **int** or **double**.

int ***error**;
  The error code of the function. Should be "ELPA_OK". The error
  codes can be queried with **elpa_strerr**(3)

DESCRIPTION

The **elpa_set** function is used to set **mandatory parameters** and **runtime
options** of the ELPA library. It returns an error code which can be
queried with **elpa_strerr**(3).

  Mandatory parameters:

Mandatory parameters of an ELPA instance have to be set *BEFORE* the ELPA instance is set up with the function **elpa_setup**(3).

At the moment the following mandatory parameters are supported:

"na": integer parameter. The global matrix has size is (na * na)

"nev":
> integer parameter. The number of eigenvectors to be computed in a call to **elpa_eigenvectors**(3). Must satisfy 1 <= nev <= na.

"local_nrows":
> integer parameter. Number of matrix rows stored on this MPI process.

"local_ncols":
> integer parameter. Number of matrix columns stored on this MPI process.

"process_row":
> integer parameter. Process row number in the 2D domain decomposition.

"process_col":
> integer parameter. Process column number in the 2D domain decomposition.

"mpi_comm_parent":
> integer parameter. The parent MPI communicator which includes all MPI process which are used in the 2D domain decomposition.

"bandwidth":
> integer parameter. Some ELPA computational steps can be accelerated if the input matrix is already in banded form. If set, ELPA assumes that the matrix has the provided bandwidth.

"BLACS_context":
> integer parameter. The generalized eigenvalue solver **elpa_generalized_eigenvectors**(3) uses internal calls to some of the ScaLAPACK routines. Thus before calling it, the user has to provide properly initialized BLACS context.

"timings":
> integer parameter. Choose whether time measurements should be done in the ELPA routines (1) or not (0).

 Runtime options:

Runtime options of an ELPA option can be set at *any time*.

Here are some suuported runtime options:

"solver":
> Choose which solver should be used in the compute steps **elpa_eigenvalues**(3) or **elpa_eigenvectors**(3). At the moment allowed option are **"ELPA_SOLVER_1STAGE"** or **"ELPA_SOLVER_2STAGE"**.

"real_kernel":
> Choose which real kernel should be used in the **elpa_eigenvalues**(3) or **elpa_eigenvectors**(3) compute steps, if solver is set to **"ELPA_SOLVER_2STAGE"**. The available kernels can be queried with

        elpa2_print_kernels(1).

    "complex_kernel":
        Choose which complex kernel should be used in the
        elpa_eigenvalues(3) or **elpa_eigenvectors**(3) compute steps, if
        solver is set to **"ELPA_SOLVER_2STAGE"**. The available kernels can
        be queried with **elpa2_print_kernels**(1).

    "qr": Choose whether a QR decomposition should be used for the real case
        computations in **elpa_eigenvalues**(3) or **elpa_eigenvectors**(3)
        computational steps, if solver was set to **"ELPA_SOLVER_2STAGE"**.

    "debug":
        Choose whether, in case of an error, more debug information should
        be provided.

        The full list of supported options can found in the ELPA
        documentation.

SEE ALSO
    elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_setup**(3)
    elpa_strerr(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3) **elpa_cholesky**(3)
    elpa_invert_triangular(3) **elpa_solve_tridiagonal**(3)
    elpa_hermitian_multiply(3) **elpa_deallocate**(3) **elpa_uninit**(3)

D.57 `elpa_setup`

NAME
      elpa_setup – setup an instance of the ELPA library

SYNOPSIS
  FORTRAN INTERFACE
     use elpa
     class(elpa_t), pointer :: elpa_handle

     status = elpa_handle%**setup**()

     With the definitions of the input and output variables:

     class(elpa_t) :: **elpa_handle**
         An instance of the ELPA object.

     integer :: **status**
         The returned error code. Should normally be ELPA_OK. Can be
         queried with **elpa_strerr**(3)

  C/C++ INTERFACE
     #include <elpa/elpa.h>
     elpa_t elpa_handle;

     int status = **elpa_setup** (**elpa_t** elpa_handle);

     With the definitions of the input and output variables:

     elpa_t **elpa_handle**;
         The handle of an ELPA object, obtained before with
         elpa_allocate(3)

     int **status**;
         The returned error code. Should normally be ELPA_OK. Can be
         queried with **elpa_strerr**(3)

DESCRIPTION
     Finalizes setting of the mandatory parameters and setups an ELPA object.
     Prior to calling the setup, the functions **elpa_init**(3),
     elpa_allocate(3) *must have been called* and the mandatory parameters must
     have been set with **elpa_set**(3).

SEE ALSO
     elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3)
     elpa_strerr(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3) **elpa_cholesky**(3)
     elpa_invert_triangular(3) **elpa_solve_tridiagonal**(3)
     elpa_hermitian_multiply(3) **elpa_deallocate**(3) **elpa_uninit**(3)

D.58 `elpa_setup_gpu`

NAME
   elpa_setup_gpu – finalize the setup of GPU in ELPA

SYNOPSIS
   FORTRAN INTERFACE
   use elpa
   class(elpa_t), pointer :: elpa_handle

   status = elpa_handle%**setup_gpu**()

   With the definitions of the input and output variables:

   class(elpa_t) :: **elpa_handle**
      An instance of the ELPA object.

   integer :: **status**
      The returned error code. Should normally be ELPA_OK. Can be
      queried with **elpa_strerr**(3)

   C/C++ INTERFACE
   #include <elpa/elpa.h>
   elpa_t elpa_handle;

   int status = **elpa_setup_gpu** (**elpa_t** elpa_handle);

   With the definitions of the input and output variables:

   elpa_t **elpa_handle**;
      The handle of an ELPA object, obtained before with
      elpa_allocate(3)

   int **status**;
      The returned error code. Should normally be ELPA_OK. Can be
      queried with **elpa_strerr**(3)

DESCRIPTION
   Finalizes the setup of GPU runtime options in ELPA. **elpa_setup_gpu** has
   to be called after the runtime option for GPU usage has been set with
   elpa_set(3), e.g. by 'call elpa_handle%set("nvidia-gpu", 1, status)' in
   Fortran or 'elpa_set(handle, "nvidia-gpu", 1, &status)' in C/C++.
      elpa_setup_gpu will check if the GPU is available and if the GPU is
   supported by the ELPA library. If the GPU is not available or not
   supported, the function will return an error code. If the GPU is
   available and supported, the function will finalize the setup of the GPU
   and return ELPA_OK.

SEE ALSO
   elpa_set(3) **elpa_strerr**(3) **elpa_setup**(3)

D.59 `elpa_skew_eigenvalues`

NAME

elpa_skew_eigenvalues – computes all eigenvalues λ for a real skew-symmetric eigenproblem: A*Q = λ*Q

There are also variations of this routine that can accept not only host but also device pointers as input/output. Names of these routines explicitly contain the corresponding datatypes: elpa_skew_eigenvalues_double, elpa_skew_eigenvalues_float.

SYNOPSIS

FORTRAN INTERFACE

    use elpa
    class(elpa_t), pointer :: elpa_handle

    call elpa_handle%**skew_eigenvalues** (a, ev, error)

    With the definitions of the input and output variables:

    class(elpa_t) :: **elpa_handle**
          An instance of the ELPA object.

    datatype :: **a**
          The local part of matrix A for which the eigenvalues should be
          computed. The dimensions of matrix **a** must be set *BEFORE* with the
          methods **elpa_set**(3) and **elpa_setup**(3). The **datatype_real** of the
          matrix can be one of "real(kind=c_double)" or
          "real(kind=c_float)". The global matrix has to be skew-symmetric,
          this is not checked by the routine.

    datatype_real :: **ev**
          The array where the eigenvalues λ will be stored in *ascending*
          order. The **datatype_real** of **ev** can be either "real(kind=c_double)"
          or "real(kind=c_float)", depending of the **datatype** of the matrix.

    integer, optional :: **error**
          The return error code of the function. Should be "ELPA_OK". The
          error code can be queried with the function **elpa_strerr**(3).

C/C++ INTERFACE

    #include <elpa/elpa.h>
    elpa_t elpa_handle;

    void **elpa_skew_eigenvalues**(**elpa_t** elpa_handle, **datatype_real** *a,
    datatype_real *ev, **int** *error);

    With the definitions of the input and output variables:

    elpa_t **handle**;
          The handle to the ELPA object

    datatype_real ***a**;
          The local part of matrix A for which the eigenvalues should be
          computed. The dimensions of the matrix must be set *BEFORE* with the
          methods **elpa_set**(3) and **elpa_setup**(3). The **datatype** can be one of

"double" or "float". The global matrix has to be skew-symmetric, this is not checked by the routine.

datatype_real ***ev**;
>   The array where the eigenvalues λ will be stored in *ascending* order. Eigenvalues will be stored in *ascending* order. The **datatype_real** can be either "double" or "float".

int ***error**;
>   The error code of the function. Should be "ELPA_OK". The error codes can be queried with **elpa_strerr**(3).

DESCRIPTION
>   Computes the eigenvalues of a real skew-symmetric matrix. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE* **elpa_skew_eigenvalues** can be called.

SEE ALSO
>   **elpa2_print_kernels**(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3)
>   **elpa_setup**(3) **elpa_strerr**(3) **elpa_eigenvectors**(3)
>   **elpa_skew_eigenvectors**(3) **elpa_eigenvalues**(3) **elpa_cholesky**(3)
>   **elpa_invert_triangular**(3) **elpa_solve_tridiagonal**(3) **elpa_eigenvalues**(3)
>   **elpa_uninit**(3) **elpa_deallocate**(3)

D.60 `elpa_skew_eigenvalues_double`

NAME

      elpa_skew_eigenvalues_double – computes all eigenvalues λ for a real skew-symmetric eigenproblem: A*Q = λ*Q

SYNOPSIS

  FORTRAN INTERFACE

      use elpa
      class(elpa_t), pointer :: elpa_handle

      call elpa_handle%**skew_eigenvalues** (a, ev, error)

      With the definitions of the input and output variables:

      class(elpa_t) :: **elpa_handle**
          An instance of the ELPA object.

      real(kind=c_double) :: **a**   OR   type(c_ptr) :: **a**
          The local part of matrix A for which the eigenvalues should be computed. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The global matrix has to be skew-symmetric, this is not checked by the routine. In case of a GPU build **a** can be a pointer to the device memory.

      real(kind=c_double) :: **ev**   OR   type(c_ptr) :: **ev**
          The array where the eigenvalues λ will be stored in *ascending* order. In case of a GPU build **ev** can be a pointer to the device memory.

      integer, optional :: **error**
          The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3).

  C/C++ INTERFACE

      #include <elpa/elpa.h>
      elpa_t elpa_handle;

      void **elpa_skew_eigenvalues**(**elpa_t** elpa_handle, **double** *a, **double** *ev, **int** *error);

      With the definitions of the input and output variables:

      elpa_t **handle**;
          The handle to the ELPA object

      double ***a**;
          The local part of matrix A for which the eigenvalues should be computed. The dimensions of the matrix must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The global matrix has to be skew-symmetric, this is not checked by the routine. In case of a GPU build **a** can be a pointer to the device memory.

      double ***ev**;
          The array where the eigenvalues λ will be stored in *ascending* order. Eigenvalues will be stored in *ascending* order. In case of a GPU build **ev** can be a pointer to the device memory.

```
    int  *error;
         The error code of the function. Should be "ELPA_OK". The error
         codes can be queried with elpa_strerr(3).
```

DESCRIPTION

     Computes the eigenvalues of a real skew-symmetric matrix. The functions
     elpa_init(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be
     called *BEFORE* **elpa_skew_eigenvalues** can be called.

SEE ALSO

     elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3)
     elpa_setup(3) **elpa_strerr**(3) **elpa_eigenvectors**(3)
     elpa_skew_eigenvectors(3) **elpa_eigenvalues**(3) **elpa_cholesky**(3)
     elpa_invert_triangular(3) **elpa_solve_tridiagonal**(3) **elpa_eigenvalues**(3)
     elpa_uninit(3) **elpa_deallocate**(3)

D.61 `elpa_skew_eigenvalues_float`

NAME

      elpa_skew_eigenvalues_float – computes all eigenvalues λ for a real
      skew-symmetric eigenproblem: A*Q = λ*Q

SYNOPSIS

  FORTRAN INTERFACE
      use elpa
      class(elpa_t), pointer :: elpa_handle

      call elpa_handle%**skew_eigenvalues** (a, ev, error)

      With the definitions of the input and output variables:

      class(elpa_t) :: **elpa_handle**
          An instance of the ELPA object.

      real(kind=c_float) :: **a**   OR   type(c_ptr) :: **a**
          The local part of matrix A for which the eigenvalues should be
          computed. The dimensions of matrix **a** must be set *BEFORE* with the
          methods **elpa_set**(3) and **elpa_setup**(3). The global matrix has to be
          skew-symmetric, this is not checked by the routine. In case of a
          GPU build **a** can be a pointer to the device memory.

      real(kind=c_float) :: **ev**   OR   type(c_ptr) :: **ev**
          The array where the eigenvalues λ will be stored in *ascending*
          order. In case of a GPU build **ev** can be a pointer to the device
          memory.

      integer, optional :: **error**
          The return error code of the function. Should be "ELPA_OK". The
          error code can be queried with the function **elpa_strerr**(3).

  C/C++ INTERFACE
      #include <elpa/elpa.h>
      elpa_t elpa_handle;

      void **elpa_skew_eigenvalues**(**elpa_t** elpa_handle, **float** *a, **float** *ev, **int**
      *error);

      With the definitions of the input and output variables:

      elpa_t **handle**;
          The handle to the ELPA object

      float ***a**;
          The local part of matrix A for which the eigenvalues should be
          computed. The dimensions of the matrix must be set *BEFORE* with the
          methods **elpa_set**(3) and **elpa_setup**(3). The global matrix has to be
          skew-symmetric, this is not checked by the routine. In case of a
          GPU build **a** can be a pointer to the device memory.

      float ***ev**;
          The array where the eigenvalues λ will be stored in *ascending*
          order. Eigenvalues will be stored in *ascending* order. In case of a
          GPU build **ev** can be a pointer to the device memory.

```
    int  *error;
          The error code of the function. Should be "ELPA_OK". The error
          codes can be queried with elpa_strerr(3).
```

DESCRIPTION
     Computes the eigenvalues of a real skew-symmetric matrix. The functions
     elpa_init(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be
     called *BEFORE* **elpa_skew_eigenvalues** can be called.

SEE ALSO
     elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3)
     elpa_setup(3) **elpa_strerr**(3) **elpa_eigenvectors**(3)
     elpa_skew_eigenvectors(3) **elpa_eigenvalues**(3) **elpa_cholesky**(3)
     elpa_invert_triangular(3) **elpa_solve_tridiagonal**(3) **elpa_eigenvalues**(3)
     elpa_uninit(3) **elpa_deallocate**(3)

D.62 `elpa_skew_eigenvectors`

NAME
  elpa_skew_eigenvectors – computes all eigenvalues λ and (part of) the
  eigenvectors for a real skew-symmetric eigenproblem: A*Q = λ*Q

SYNOPSIS
  FORTRAN INTERFACE
  use elpa
  class(elpa_t), pointer :: elpa_handle

  call elpa_handle%**skew_eigenvectors** (a, ev, q, error)

  With the definitions of the input and output variables:

  class(elpa_t) :: **elpa**
      An instance of the ELPA object

  datatype_real :: **a**
      The local part of matrix A for which the eigenvalues and
      eigenvectors should be computed. The dimensions of matrix **a** must
      be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The
      datatype_real of the matrix can be one of "real(kind=c_double)" or
      "real(kind=c_float)". The global matrix has to be skew-symmetric,
      this is not checked by the routine.

  datatype_real :: **ev**
      The array where the eigenvalues λ will be stored in *ascending*
      order. The **datatype_real** of **ev** can be either "real(kind=c_double)"
      or "real(kind=c_float)", depending of the **datatype** of the matrix.

  datatype_real :: **q**
      The storage space for the computed eigenvectors Q. The number of
      requested eigenpairs must be set *BEFORE* with the methods
      elpa_set(3) and **elpa_setup**(3). The number of requested
      eigenvectors must be set *BEFORE* with the methods **elpa_set**(3) and
      elpa_setup(3). The **datatype_real** can be one of
      "complex(kind=c_double)" or "complex(kind=c_float)". Note, that
      for a skew-symmetric matrix the eigenvectors are complex. The
      routine returns separately the real and imaginary parts of the
      complex eigenvectors. Thus, the storage space has to be of
      dimension q(#number_of_rows, 2*#number_of_columns).

  integer, optional :: **error**
      The return error code of the function. Should be "ELPA_OK". The
      error code can be queried with the function **elpa_strerr**(3)

  C/C++ INTERFACE
  #include <elpa/elpa.h>
  elpa_t elpa_handle;

  void **elpa_eigenvalues**(**elpa_t** elpa_handle, **datatype_real** *a,
  datatype_real *ev, **datatype_real** *q, **int** *error);

  With the definitions of the input and output variables:

      `elpa_t` **`elpa_handle`**`;`
          The handle to the ELPA object

      `datatype_real` **`*a`**`;`
          The local part of matrix A for which the eigenvalues and
          eigenvectors should be computed. The dimensions of the matrix must
          be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The
          datatype_real can be "double" or "float". The global matrix has to
          be symmetric or hermitian, this is not checked by the routine.

      `datatype_real` **`*ev`**`;`
          The array where the eigenvalues λ will be stored in *ascending*
          order. Eigenvalues will be stored in *ascending* order. The
          datatype_real can be either "double" or "float".

      `datatype_real` **`*q`**`;`
          The storage space for the computed eigenvectors Q. The number of
          requested eigenvectors must be set *BEFORE* with the methods
          elpa_set(3) and **elpa_setup**(3). The **datatype_real** can "double
          complex" or "float complex". Note, that for a skew-symmetric
          matrix the eigenvectors are complex. The routine returns
          separately the real and imaginary parts of the complex
          eigenvectors. Thus, the storage space has to be of dimension
          q(#number_of_rows, 2*#number_of_columns).

      `int` **`*error`**`;`
          The error code of the function. Should be "ELPA_OK". The error
          codes can be queried with **elpa_strerr**(3).

DESCRIPTION
      Computes the eigenvalues and (part of) the eigenvector spectrum of a
      real skew-symmetric matrix. The functions **elpa_init**(3),
      elpa_allocate(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE*
      elpa_skew_eigenvectors can be called. In particular, the number of the
      requested eigenpairs, "nev", must be set with **elpa_set**(3).

SEE ALSO
      elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3)
      elpa_setup(3) **elpa_strerr**(3) **elpa_eigenvalues**(3)
      elpa_skew_eigenvalues(3) **elpa_eigenvectors**(3) **elpa_cholesky**(3)
      elpa_invert_triangular(3) **elpa_solve_tridiagonal**(3)
      elpa_hermitian_multiply(3) **elpa_uninit**(3) **elpa_deallocate**(3)

D.63 `elpa_skew_eigenvectors_double`

NAME

elpa_skew_eigenvectors_double – computes all eigenvalues λ and (part of) the eigenvectors for a real skew-symmetric eigenproblem: A*Q = λ*Q

SYNOPSIS

FORTRAN INTERFACE

```
use elpa
class(elpa_t), pointer :: elpa_handle

call elpa_handle%skew_eigenvectors_double (a, ev, q, error)
```

With the definitions of the input and output variables:

class(elpa_t) :: **elpa**
> An instance of the ELPA object

real(kind=c_double) :: **a**   OR   type(c_ptr) :: **a**
> The local part of matrix A for which the eigenvalues and eigenvectors should be computed. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The global matrix has to be skew-symmetric, this is not checked by the routine. In case of a GPU build **a** can be a pointer to the device memory.

real(kind=c_double) :: **ev**   OR   type(c_ptr) :: **ev**
> The array where the eigenvalues λ will be stored in *ascending* order. In case of a GPU build **ev** can be a pointer to the device memory.

real(kind=c_double) :: **q**   OR   type(c_ptr) :: **q**
> The storage space for the computed eigenvectors Q. The number of requested eigenpairs must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The number of requested eigenvectors must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). Note, that for a skew-symmetric matrix the eigenvectors are complex. The routine returns separately the real and imaginary parts of the complex eigenvectors. Thus, the storage space has to be of dimension q(#number_of_rows, 2*#number_of_columns). In case of a GPU build **q** can be a pointer to the device memory.

integer, optional :: **error**
> The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3)

C/C++ INTERFACE

```
#include <elpa/elpa.h>
elpa_t elpa_handle;

void elpa_eigenvalues(elpa_t elpa_handle, double *a, double *ev, double
*q, int *error);
```

With the definitions of the input and output variables:

elpa_t **elpa_handle**;
>   The handle to the ELPA object

double ***a**;
>   The local part of matrix A for which the eigenvalues and
>   eigenvectors should be computed. The dimensions of the matrix must
>   be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The
>   global matrix has to be symmetric or hermitian, this is not
>   checked by the routine. In case of a GPU build **a** can be a pointer
>   to the device memory.

double ***ev**;
>   The array where the eigenvalues λ will be stored in *ascending*
>   order. In case of a GPU build **ev** can be a pointer to the device
>   memory.

double ***q**;
>   The storage space for the computed eigenvectors Q. The number of
>   requested eigenvectors must be set *BEFORE* with the methods
>   **elpa_set**(3) and **elpa_setup**(3). Note, that for a skew-symmetric
>   matrix the eigenvectors are complex. The routine returns
>   separately the real and imaginary parts of the complex
>   eigenvectors. Thus, the storage space has to be of dimension
>   q(#number_of_rows, 2*#number_of_columns). In case of a GPU build **q**
>   can be a pointer to the device memory.

int ***error**;
>   The error code of the function. Should be "ELPA_OK". The error
>   codes can be queried with **elpa_strerr**(3).

DESCRIPTION
Computes the eigenvalues and (part of) the eigenvector spectrum of a
real skew-symmetric matrix. The functions **elpa_init**(3),
elpa_allocate(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE*
elpa_skew_eigenvectors_double can be called. In particular, the number
of the requested eigenpairs, "nev", must be set with **elpa_set**(3).

SEE ALSO
elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3)
elpa_setup(3) **elpa_strerr**(3) **elpa_eigenvalues**(3)
elpa_skew_eigenvalues(3) **elpa_eigenvectors**(3) **elpa_cholesky**(3)
elpa_invert_triangular(3) **elpa_solve_tridiagonal**(3)
elpa_hermitian_multiply(3) **elpa_uninit**(3) **elpa_deallocate**(3)

D.64 `elpa_skew_eigenvectors_float`

NAME

elpa_skew_eigenvectors_float – computes all eigenvalues λ and (part of) the eigenvectors for a real skew-symmetric eigenproblem: A*Q = λ*Q

SYNOPSIS

FORTRAN INTERFACE

    use elpa
    class(elpa_t), pointer :: elpa_handle

    call elpa_handle%**skew_eigenvectors_float** (a, ev, q, error)

With the definitions of the input and output variables:

class(elpa_t) :: **elpa**
    An instance of the ELPA object

real(kind=c_float) :: **a**   OR   type(c_ptr) :: **a**
    The local part of matrix A for which the eigenvalues and eigenvectors should be computed. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The global matrix has to be skew-symmetric, this is not checked by the routine. In case of a GPU build **a** can be a pointer to the device memory.

real(kind=c_float) :: **ev**   OR   type(c_ptr) :: **ev**
    The array where the eigenvalues λ will be stored in *ascending* order. In case of a GPU build **ev** can be a pointer to the device memory.

real(kind=c_float) :: **q**   OR   type(c_ptr) :: **q**
    The storage space for the computed eigenvectors Q. The number of requested eigenpairs must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The number of requested eigenvectors must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). Note, that for a skew-symmetric matrix the eigenvectors are complex. The routine returns separately the real and imaginary parts of the complex eigenvectors. Thus, the storage space has to be of dimension q(#number_of_rows, 2*#number_of_columns). In case of a GPU build **q** can be a pointer to the device memory.

integer, optional :: **error**
    The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3)

C/C++ INTERFACE

    #include <elpa/elpa.h>
    elpa_t elpa_handle;

    void **elpa_eigenvalues**(elpa_t elpa_handle, **float** *a, **float** *ev, **float** *q,
    int *error);

With the definitions of the input and output variables:

elpa_t **elpa_handle**;
>   The handle to the ELPA object

float ***a**;
>   The local part of matrix A for which the eigenvalues and
>   eigenvectors should be computed. The dimensions of the matrix must
>   be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The
>   global matrix has to be symmetric or hermitian, this is not
>   checked by the routine. In case of a GPU build **a** can be a pointer
>   to the device memory.

float ***ev**;
>   The array where the eigenvalues λ will be stored in *ascending*
>   order. In case of a GPU build **ev** can be a pointer to the device
>   memory.

float ***q**;
>   The storage space for the computed eigenvectors Q. The number of
>   requested eigenvectors must be set *BEFORE* with the methods
>   **elpa_set**(3) and **elpa_setup**(3). Note, that for a skew-symmetric
>   matrix the eigenvectors are complex. The routine returns
>   separately the real and imaginary parts of the complex
>   eigenvectors. Thus, the storage space has to be of dimension
>   q(#number_of_rows, 2*#number_of_columns). In case of a GPU build **q**
>   can be a pointer to the device memory.

int ***error**;
>   The error code of the function. Should be "ELPA_OK". The error
>   codes can be queried with **elpa_strerr**(3).

DESCRIPTION
Computes the eigenvalues and (part of) the eigenvector spectrum of a
real skew-symmetric matrix. The functions **elpa_init**(3),
elpa_allocate(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE*
elpa_skew_eigenvectors_float can be called. In particular, the number of
the requested eigenpairs, "nev", must be set with **elpa_set**(3).

SEE ALSO
elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3)
elpa_setup(3) **elpa_strerr**(3) **elpa_eigenvalues**(3)
elpa_skew_eigenvalues(3) **elpa_eigenvectors**(3) **elpa_cholesky**(3)
elpa_invert_triangular(3) **elpa_solve_tridiagonal**(3)
elpa_hermitian_multiply(3) **elpa_uninit**(3) **elpa_deallocate**(3)

D.65 `elpa_solve_tridiagonal`

NAME
>   elpa_solve_tridiagonal – computes the eigenvalue problem for real
>   symmetric tridiagonal matrix computes all eigenvalues λ and all
>   eigenvectors for a real symmetric tridiagonal matrix: T*Q = λ*Q

SYNOPSIS
>   **FORTRAN INTERFACE**
>   use elpa
>   class(elpa_t), pointer :: elpa_handle
>
>   call elpa_handle%**solve_tridiagonal** (d, e, q, error)
>
>   With the definitions of the input and output variables:
>
>   class(elpa_t) :: **elpa_handle**
>   An instance of the ELPA object.
>
>   datatype_real :: **d**
>   The diagonal elements of matrix T.
>   **d** is a gloabal array of size elpa_handle%na, replicated on all
>   MPI ranks. The dimensions of the matrix must be set *BEFORE* with
>   **elpa_setup**(3). On exit the eigenvalues are stored in **d**. The
>   **datatype_real** of the diagonal elements can either be
>   "real(kind=c_double)" or "real(kind=c_float)".
>
>   datatype_real :: **e**
>   The offdiagonal elements of matrix T.
>   **e** is a gloabal array of size elpa_handle%na-1, replicated on all
>   MPI ranks. The **datatype_real** of the diagonal elements can either
>   be "real(kind=c_double)" or "real(kind=c_float)".
>
>   datatype_real :: **q**
>   The storage space for the computed eigenvectors Q. The
>   **datatype_real** of the matrix can be either "real(kind=c_double)" or
>   "real(kind=c_float)".
>
>   integer, optional :: **error**
>   The return error code of the function. Should be "ELPA_OK". The
>   error code can be queried with the function **elpa_strerr**(3)
>
>   **C/C++ INTERFACE**
>   #include <elpa/elpa.h>
>   elpa_t elpa_handle;
>
>   void **elpa_solve_tridiagonal**(**elpa_t** elpa_handle, **datatype_real** *d,
>   **datatype_real** *e, **datatype_real** *q, **int** *error);
>
>   With the definitions of the input and output variables:
>
>   elpa_t **elpa_handle**;
>   The handle to the ELPA object
>
>   datatype_real ***d**;
>   The diagonal elements of matrix T.
>   **d** is a gloabal array of size **na**, replicated on all MPI ranks. The
>   dimensions of the matrix must be set *BEFORE* with **elpa_setup**(3). On

exit the eigenvalues are stored in **d**. The **datatype_real** can be one
of "double" or "float".

datatype_real ***e**;
  The offdiagonal elements of matrix T.
  ev is a gloabal array of size **na-1**, replicated on all MPI ranks.
  The **datatype_real** can be one of "double" or "float".

datatype_real ***q**;
  The storage space for the computed eigenvectors Q. The
  datatype_real can be one of "double" or "float".

int ***error**;
  The error code of the function. Should be "ELPA_OK". The error
  codes can be queried with **elpa_strerr**(3)

DESCRIPTION
  Computes the eigenvalue problem of a real symmetric tridiagonal matrix.
  The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and
  elpa_setup(3) must be called *BEFORE* **elpa_solve_tridiagonal** can be
  called.

SEE ALSO
  elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3)
  elpa_setup(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_cholesky**(3)
  elpa_invert_triangular(3) **elpa_hermitian_multiply**(3) **elpa_uninit**(3)
  elpa_deallocate(3)

D.66 `elpa_store_settings`

NAME
      elpa_store_settings – stores the setting of an ELPA object

SYNOPSIS
  FORTRAN INTERFACE
      use elpa
      class(elpa_t), pointer :: elpa_handle

      call elpa_handle%**store_settings** (filename, error)

      With the definitions of the input and output variables:

      class(elpa_t)   :: **elpa_handle**
          An instance of the ELPA object

      character(*)   :: **filename**
          The filename to be used for storing the settings

      integer, optional :: **error**
          An error return code

  C/C++ INTERFACE
      #include <elpa/elpa.h>
      elpa_t elpa_handle;

      void **elpa_store_settings**(**elpa_t** elpa_handle, **const char** *filename, **int**
      *error);

      With the definitions of the input and output variables:

      elpa_t **elpa_handle**;
          The handle to the ELPA object

      const char ***filename**;
          The filename to store the settings

      int ***error**;
          The error return code

DESCRIPTION
      Stores all the settings of an ELPA object in a human readable form to a
      file specified via the **filename** parameter. The settings can later be
      restored with the **elpa_load_settings**(3) method.

SEE ALSO
      elpa_load_setting(3)

D.67 `elpa_timer_start`

NAME
>   elpa_timer_start – start the timer for the individual ELPA solution
>   steps.

SYNOPSIS
>   **FORTRAN INTERFACE**
>   use elpa
>   class(elpa_t), pointer :: elpa_handle
>
>   call elpa_handle%**timer_start** (name)
>
>   With the definitions of the input and output variables:
>
>   class(elpa_t) :: **elpa**
>   An instance of the ELPA object
>
>   character(*)   :: **name**
>   The name of the ELPA procedure for which the timings should be
>   recorded.
>
>   **C/C++ INTERFACE**
>   #include <elpa/elpa.h>
>   elpa_t elpa_handle;
>
>   void **elpa_timer_start**(**elpa_t** elpa_handle, **char** *name);
>
>   With the definitions of the input and output variables:
>
>   elpa_t **handle**;
>   The handle to the ELPA object
>
>   char ***name**;
>   The name of the ELPA procedure for which the timings should be
>   recorded.

DESCRIPTION
>   Starts the timer for the individual ELPA solution steps.

SEE ALSO
>   **elpa_timer_stop**(3) **elpa_print_times**(3)

D.68 `elpa_timer_stop`

NAME
>   elpa_timer_stop – stop the timer for the individual ELPA solution steps.

SYNOPSIS
>   **FORTRAN INTERFACE**
>   use elpa
>   class(elpa_t), pointer :: elpa_handle
>
>   call elpa_handle%**timer_start** (name)
>
>   With the definitions of the input and output variables:
>
>   class(elpa_t) :: **elpa_handle**
>   An instance of the ELPA object
>
>   character(*)   :: **name**
>   The name of the ELPA procedure for which the timings should be
>   recorded.
>
>   **C/C++ INTERFACE**
>   #include <elpa/elpa.h>
>   elpa_t elpa_handle;
>
>   void **elpa_timer_start**(**elpa_t** elpa_handle, **char** *name);
>
>   With the definitions of the input and output variables:
>
>   elpa_t **handle**;
>   The handle to the ELPA object
>
>   char   ***name**;
>   The name of the ELPA procedure for which the timings should be
>   recorded.

DESCRIPTION
>   Stops the timer for the individual ELPA solution steps.

SEE ALSO
>   **elpa_timer_start**(3) **elpa_print_times**(3)

D.69 `elpa_uninit`

NAME
       elpa_uninit – uninitializes the ELPA library

SYNOPSIS
  FORTRAN INTERFACE
       use elpa

       call **elpa_uninit** (error)

       With the definitions of the input and output variables:

       integer, optional :: error
              The error code

  C/C++ INTERFACE
       #include <elpa/elpa.h>

       void elpa_uninit (int *error);

       With the definitions of the input and output variables:

       int **error*;**
              The error code

DESCRIPTION
       Uninitializes the ELPA library after usage. The function **elpa_init**(3)
       must have been called *BEFORE* elpa_uninit can be called.

SEE ALSO
       elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3)
       elpa_strerr(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3) **elpa_cholesky**(3)
       elpa_invert_triangular(3) **elpa_solve_tridiagonal**(3)
       elpa_hermitian_multiply(3) **elpa_setup**(3) **elpa_deallocate**(3)