PRIMME

large-scale eigenvalue and
singular value solver

Documentation

Release 3.2

Andreas Stathopoulos
Eloy Romero Alcalde
Lingfei Wu
James R. McCombs

Jan 30, 2021

CONTENTS

1 PRIMME: PReconditioned Iterative MultiMethod Eigensolver **1**

1.1 Incompatibilities . . . 2
1.2 Changelog . . . 3
1.3 License Information . . . 6
1.4 Citing the code . . . 7
1.5 Contact Information . . . 8
1.6 Support . . . 9
1.7 Directory Structure . . . 10
1.8 Making and Linking . . . 11
1.9 Tested Systems . . . 13
1.10 Main Contributors . . . 14

2 Eigenvalue Problems **15**

2.1 C Library Interface . . . 15
2.2 FORTRAN 77 Library Interface . . . 25
2.3 FORTRAN 90 Library Interface . . . 37
2.4 Python Interface . . . 44
2.5 MATLAB Interface . . . 48
2.6 Parameters Description . . . 52
2.7 Preset Methods . . . 74
2.8 Error Codes . . . 77

3 Singular Value Problems **79**

3.1 C Library Interface . . . 79
3.2 FORTRAN Library Interface . . . 87
3.3 FORTRAN 90 Library Interface . . . 97
3.4 Python Interface . . . 103
3.5 MATLAB Interface . . . 106
3.6 Parameter Description . . . 109
3.7 Preset Methods . . . 125
3.8 Error Codes . . . 126

4 Indices **127**

Bibliography **129**

Index **131**

CHAPTER

ONE

PRIMME: PRECONDITIONED ITERATIVE MULTIMETHOD EIGENSOLVER

PRIMME, pronounced as *prime*, is a high-performance library for computing a few eigenvalues/eigenvectors, and singular values/vectors. PRIMME is especially optimized for large, difficult problems. Real symmetric and complex Hermitian problems, standard $Ax = \lambda x$ and generalized $Ax = \lambda Bx$, are supported. Besides standard eigenvalue problems with a normal matrix are supported. It can find largest, smallest, or interior singular/eigenvalues, and can use preconditioning to accelerate convergence. PRIMME is written in C99, but complete interfaces are provided for Fortran, MATLAB, Python, and R.

1.1 Incompatibilities

From PRIMME 2.2 to 3.0:

- Removed constants `primme_thick` and `primme_dtr`, and the member `scheme` from `restarting_params`.
- Added members *numBroadcast*, *volumeBroadcast*, `flopsDense`, *timeBroadcast*, `timeDense`, `estimateBNorm`, `estimateInvBNorm`, and *lockingIssue* to `primme_stats`.
- Added members *matrixMatvec_type*, *applyPreconditioner_type*, *massMatrixMatvec_type*, *globalSumReal_type*, *broadcastReal*, *broadcastReal_type*, *BNorm*, *invBNorm*, *orth*, *internalPrecision*, *massMatrix*, *convTestFun_type*, *monitorFun_type*, *queue*, and `profile` to *primme_params*.
- Added members *matrixMatvec_type*, *applyPreconditioner_type*, `globalSumReal_type`, *broadcastReal*, *broadcastReal_type*, *internalPrecision*, *convTestFun_type*, *monitorFun_type*, *queue*, and `profile` to *primme_svds_params*.
- Changed callbacks *monitorFun* and *monitorFun*.
- Changed the value of all constants; see `primme_get_member_f77()`, `primme_set_member_f77()`, `primme_svds_get_member_f77()`, and `primme_svds_set_member_f77()`.
- Removed `intWorkSize`, `realWorkSize`, `intWork`, `realWork` from *primme_params* and *primme_svds_params*.

From PRIMME 2.0 to 2.1:

- Added members *monitorFun* and `monitor` to *primme_params*.
- Added members *monitorFun* and `monitor` to *primme_svds_params*.
- Renamed `PRIMME_SUBSPACE_ITERATION` as *PRIMME_STEEPEST_DESCENT*.

From PRIMME 1.x to 2.0:

- Prototype of callbacks has changed: *matrixMatvec*, *applyPreconditioner*, *massMatrixMatvec* and *globalSumReal*.
- The next parameters are *PRIMME_INT*: *n*, *nLocal*, *maxMatvecs*, *iseed*, *numOuterIterations*, *numRestarts*, *numMatvecs* and *numMatvecs*; use the macro `PRIMME_INT_P` to print the values.
- Rename the values of the enum *primme_preset_method*.
- Rename `primme_Free` to *primme_free()*.
- Integer parameters in Fortran functions are of the same size as *PRIMME_INT*, which is `integer*8` by default.
- Extra parameter in many Fortran functions to return the error code.
- Removed `primme_display_stats_f77`.

1.2 Changelog

Changes in PRIMME 3.2 (released on Jan 29, 2021):

- Fixed Intel 2021 compiler error `"Unsupported combination of types for <tgmath.h>."`
- Fixed compiling issues with PGI compiler also about `tgmath.h`.
- Fixed `dprimme()` and other variants not returning error code *PRIMME_MAIN_ITER_FAILURE* when it should do in some corner cases.
- Fixed warnings from gcc/clang undefined behavior sanitizers.
- Matlab: renamed *disp* to *reportLevel*.
- Matlab: add flag *returnUnconverged* to return unconverged pairs optionally.
- Matlab: return primme_params/primme_svds_params.

Changes in PRIMME 3.1 (released on May 2, 2020):

- Fixed compilation issues in F90 interface and examples.
- Fixed bug in block orthogonalization.
- Updated Python interface to Python version 3.8.

Changes in PRIMME 3.0 (released on December 14, 2019):

- Added support for the generalized Hermitian eigenvalue problem (see `massMatrixMatvec`) and the standard normal eigenvalue problem (see `zprimme_normal()`).
- Added support for GPU (see `magma_dprimme()`, `magma_zprimme_normal()`, and `magma_dprimme_svds()`).
- Added support for half precision (see `hprimme()` and `kprimme()`, and other variants for normal eigenproblems and singular value problems).
- Added block orthogonalization (see `orth`).
- Resolution of all linear system of equations simultaneously in Jacobi-Davidson.
- Added interface for Fortran 90.
- Added an optional callback for broadcasting (see `broadcastReal` and `broadcastReal`).
- The callbacks can work with different precision than the main call (see for instance `matrixMatvec_type` and `globalSumReal_type`).
- Added new counters: `numGlobalSum`, `volumeGlobalSum`, `numBroadcast`, `volumeGlobalSum`, `timeOrtho`, `timeGlobalSum`, `timeBroadcast`.
- Added `primme_params_create()`, `primme_params_destroy()`, `primme_svds_params_create()`, and `primme_svds_params_destroy()`.

Changes in PRIMME 2.2 (released on October 26, 2018):

- Improved stability for single precision.
- Improved support for the shared library.
- Updated PETSc examples in Fortran; added new examples in single precision.
- Improved support for FreeBSD and MacOS.
- New install and uninstall actions.

- MATLAB interface support for user-defined stopping criterion (see *convTestFun* and *convTestFun*) and stopping with ctr+c.
- Optional suffix on BLAS/LAPACK function names (for OpenBLAS, see *PRIMME_BLAS_SUFFIX*).
- Replaced XHEGV by XHEGVX (to support ESSL).
- Fixed bugs in the library and in the Matlab interface.

Changes in PRIMME 2.1 (released on April 4, 2017):

- Improved robustness by broadcasting the result of critical LAPACK operations instead of replicating them on every process; this is useful when using a threaded BLAS/LAPACK or when some parallel processes may run on different architectures or libraries.
- New stopping criteria in QMR that improve performance for interior problems.
- MATLAB interface reimplementation with support for singular value problems, `primme_svds()`, with double and single precision, and compatible with Octave.
- R interface
- Proper reporting of convergence history for singular value solvers.

Changes in PRIMME 2.0 (released on September 19, 2016):

- Changed license to BSD 3-clause.
- New support for singular value problems; see *dprimme_svds()*.
- New support for `float` and `complex float` arithmetic.
- Support for problem dimensions larger than 2^31, without requiring BLAS and LAPACK compiled with 64-bits integers.
- Improve robustness and performance for interior problems; implemented advanced refined and harmonic-Ritz extractions.
- Python interface compatible with NumPy and SciPy Library.
- Added parameter to indicate the leading dimension of the input/output matrices and to return an error code in callbacks *matrixMatvec*, *applyPreconditioner*, *massMatrixMatvec* and *globalSumReal*.
- Changed to type *PRIMME_INT* the options *n*, *nLocal*, *maxMatvecs* and *iseed*, and the stats counters *numOuterIterations*, *numRestarts*, *numMatvecs*, *numPreconds*. Also changed `realWorkSize` to `size_t`. Fortran interface functions will expect an `interger` of size compatible with *PRIMME_INT* for all parameters with integer type: `int`, *PRIMME_INT* and `size_t`; see also parameter `value` in functions `primmetop_set_member_f77()`, `primmetop_get_member_f77()`, `primme_set_member_f77()` and `primme_get_member_f77()`.
- Added parameter to return an error code in Fortran interface functions: `primmetop_set_member_f77()`, `primmetop_get_member_f77()`, `primme_set_member_f77()` and `primme_get_member_f77()`.
- Added leading dimension for `evecs` *ldevecs* and preferred leading dimension for the operators *ldOPs*, such as *matrixMatvec*.
- Optional user-defined convergence function, *convTestFun*.
- Prefixed methods with `PRIMME_`. Rename Fortran constants from `PRIMMEF77_` to `PRIMME_`.
- Removed `primme_display_stats_f77`.

Changes in PRIMME 1.2.2 (released on October 13, 2015):

- Fixed wrong symbols in `libdprimme.a` and `libzprimme.a`.

- *primme_set_method()* sets *PRIMME_JDQMR* instead of *PRIMME_JDQMR_ETol* for preset methods *PRIMME_DEFAULT_MIN_TIME* and *PRIMME_DYNAMIC* when seeking interior values.
- Fixed compilation of driver with a PETSc installation without HYPRE.
- Included the content of the environment variable `INCLUDE` for compiling the driver.

Changes in PRIMME 1.2.1 (released on September 7, 2015):

- Added MATLAB interface to full PRIMME functionality.
- Support for BLAS/LAPACK with 64bits integers (`-DPRIMME_BLASINT_SIZE=64`).
- Simplified configuration of Make_flags and Make_links (removed `TOP` variable and replaced defines `NUM_SUM` and `NUM_IBM` by `F77UNDERSCORE`).
- Replaced directories `DTEST` and `ZTEST` by `TEST`, that has:
  - `driver.c`: read matrices in MatrixMarket format and PETSc binary and call PRIMME with the parameters specified in a file; support complex arithmetic and MPI and can use PETSc preconditioners.
  - `ex*.c` and `ex*.f`: small, didactic examples of usage in C and Fortran and in parallel (with PETSc).
- Fixed a few minor bugs and improved documentation (especially the F77 interface).
- Using Sphinx to manage documentation.

Changes in PRIMME 1.2 (released on December 21, 2014):

- A Fortran compiler is no longer required for building the PRIMME library. Fortran programs can still be linked to PRIMME's F77 interface.
- Fixed some uncommon issues with the F77 interface.
- PRIMME can be called now multiple times from the same program.
- Performance improvements in the QMR inner solver, especially for complex arithmetic.
- Fixed a couple of bugs with the locking functionality.
  - In certain extreme cases where all eigenvalues of a matrix were needed.
  - The order of selecting interior eigenvalues.

  The above fixes have improved robustness and performance.
- PRIMME now assigns unique random seeds per parallel process for up to 4096^3 (140 trillion) processes.
- For the *PRIMME_DYNAMIC* method, fixed issues with initialization and synchronization decisions across multiple processes.
- Fixed uncommon library interface bugs, coordinated better setting the method and the user setting of parameters, and improved the interface in the sample programs and makefiles.
- Other performance and documentation improvements.

1.3 License Information

PRIMME is licensed under the 3-clause license BSD. Python and MATLAB interfaces have BSD-compatible licenses. Source code under `tests` is compatible with LGPLv3. Details can be taken from `COPYING.txt`:

1.4 Citing the code

Please cite [r1] and [r6]. Find the BibTeX in the following and also in `doc/primme.bib`:

```
@Article{PRIMME,
  author =       {Andreas Stathopoulos and James R. McCombs},
  title =        {{PRIMME}: {PR}econditioned {I}terative {M}ulti{M}ethod
                {E}igensolver: Methods and software description},
  journal =      {ACM Transactions on Mathematical Software},
  volume =     {37},
  number =     {2},
  year =         {2010},
  pages =       {21:1--21:30},
}

@Article{svds_software,
  author     = {Lingfei Wu and Eloy Romero and Andreas Stathopoulos},
  title      = {PRIMME{\_}SVDS: {A} High-Performance Preconditioned {SVD} Solver for
               Accurate Large-Scale Computations},
  journal    = {SIAM Journal on Scientific Computing},
  volume     = {39},
  number     = {5},
  pages      = {S248--S271},
  year       = {2017},
  doi        = {10.1137/16M1082214},
  url        = { https://doi.org/10.1137/16M1082214 },
}
```

More information on the algorithms and research that led to this software can be found in the rest of the papers [r2], [r3], [r4], [r5], [r7]. The work has been supported by a number of grants from the National Science Foundation.

1.5 Contact Information

For reporting bugs or questions about functionality contact Andreas Stathopoulos by email, *andreas* at *cs.wm.edu*. See further information in the webpage http://www.cs.wm.edu/~andreas/software and on github.

1.6 Support

- National Science Foundation through grants CCF 1218349, ACI SI2-SSE 1440700, and NSCI 1835821
- Department of Energy through grant Exascale Computing Project 17-SC-20-SC

1.7 Directory Structure

The next directories and files should be available:

- `COPYING.txt`, license;
- `Make_flags`, flags to be used by makefiles to compile library and tests;
- `Link_flags`, flags needed in making and linking the test programs;
- `include/`, directory with headers files;
- `src/`, directory with the source code for `libprimme`:
  - `include/`, common headers;
  - `eigs/`, eigenvalue interface and implementation;
  - `svds/`, singular value interface and implementation;
  - `tools/`, tools used to generated some headers;
- `Matlab/`, MATLAB interface;
- `Python/`, Python interface;
- `examples/`, sample programs in C, C++ and F77, both sequential and parallel;
- `tests/`, drivers for testing purpose and test cases;
- `lib/libprimme.a`, the PRIMME library (to be made);
- `makefile` main make file;
- `readme.txt` text version of the documentation;
- `doc/` directory with the HTML and PDF versions of the documentation.

1.8 Making and Linking

`Make_flags` has the flags and compilers used to make `libprimme.a`:

- *CC*, compiler program such as `gcc`, `clang` or `icc`.
- ***CFLAGS*, compiler options such as `-g` or `-O3` and macro definitions** like the ones described next.

Compiler flags for the BLAS and LAPACK libraries:

- `-DF77UNDERSCORE`, if Fortran appends an underscore to function names (usually it does).
- `-DPRIMME_BLASINT_SIZE=64`, if the library integers are 64-bit integer (`kind=8`) type, aka ILP64 interface; usually integers are 32-bits even in 64-bit architectures (aka LP64 interface).
- `-DPRIMME_BLAS_SUFFIX=<suffix>`, set a suffix to BLAS/LAPACK function names; for instance, OpenBlas compiled with ILP64 may append `_64` to the function names.

By default PRIMME sets the integer type for matrix dimensions and counters (*`PRIMME_INT`*) to 64 bits integer `int64_t`. This can be changed by setting the macro `PRIMME_INT_SIZE` to one of the following values:

- `0`: use the regular `int` of your compiler.
- `32`: use C99 `int32_t`.
- `64`: use C99 `int64_t`.

Note: When `-DPRIMME_BLASINT_SIZE=64` is set the code uses the type `int64_t` supported by the C99 standard. In case the compiler doesn't honor the standard, you can set the corresponding type name supported, for instance `-DPRIMME_BLASINT_SIZE=__int64`.

After customizing `Make_flags`, type this to generate `libprimme.a`:

```
make lib
```

Making can be also done at the command line:

```
make lib CC=clang CFLAGS='-O3'
```

`Link_flags` has the flags for linking with external libraries and making the executables located in `examples` and `tests`:

- *LDFLAGS*, linker flags such as `-framework Accelerate`.
- *LIBS*, flags to link with libraries (BLAS and LAPACK are required), such as `-lprimme -llapack -lblas -lgfortran -lm`.

After that, type this to compile and execute a simple test:

```
$ make test
...
Test passed!
...
Test passed!
```

In case of linking problems check flags in *LDFLAGS* and *LIBS* and consider to add/remove `-DF77UNDERSCORE` from *CFLAGS*. If the execution fails consider to add/remove `-DPRIMME_BLASINT_SIZE=64` from *CFLAGS*.

Full description of actions that *make* can take:

- *make lib*, builds the static library `libprimme.a`.

- *make solib*, builds the shared library `libprimme.so`.
- *make install*, installs header files and the static and dynamic libraries.
- *make uninstall*, uninstalls header files and the static and dynamic libraries.
- *make matlab*, builds *libprimme.a* compatible with MATLAB and the MATLAB module.
- *make octave*, builds *libprimme.a* and the Octave module.
- *make python*, builds *libprimme.a* and the Python module.
- *make python_install*, install the Python module.
- *make R_install*, builds and installs the R package.
- *make test*, build and execute simple examples.
- *make clean*, removes all `*.o`, `a.out`, and core files from `src`.

1.8.1 Considerations using an IDE

PRIMME can be built in other environments such as Anjuta, Eclipse, KDevelop, Qt Creator, Visual Studio and XCode. To build the PRIMME library do the following:

1. Create a new project and include the source files under the directory `src`.
2. Add the directories `include` and `src/include` as include directories.

To build an example code using PRIMME make sure:

- to add a reference for PRIMME, BLAS and LAPACK libraries;
- to add the directory `include` as an include directory.

1.9 Tested Systems

PRIMME is primary developed with GNU gcc, g++ and gfortran (versions 4.8 and later). Many users have reported builds on several other platforms/compilers:

- SUSE 13.1 & 13.2
- CentOS 6.6
- Ubuntu 18.04
- MacOS X 10.9 & 10.10
- Cygwin & MinGW
- Cray XC30
- FreeBSD 11.2

1.10 Main Contributors

- James R. McCombs
- Eloy Romero Alcalde
- Andreas Stathopoulos
- Lingfei Wu

CHAPTER

TWO

EIGENVALUE PROBLEMS

2.1 C Library Interface

The PRIMME interface is composed of the following functions. To solve real symmetric and complex Hermitian problems, standard $Ax = \lambda x$ and generalized $Ax = \lambda Bx$, call:

```
int dprimme (double *evals, double *evecs, double *resNorms,
                        primme_params *primme)
int zprimme (double *evals, PRIMME_COMPLEX_DOUBLE *evecs,
                 double *resNorms, primme_params *primme)
```

There are more versions for matrix problems working in other precisions:

Precision	Real	Complex
half	*hprimme()* *hsprimme()*	*kprimme()* *ksprimme()*
single	*sprimme()*	*cprimme()*
double	*dprimme()*	*zprimme()*

To solve standard eigenproblems with normal but not necessarily Hermitian matrices call:

```
int zprimme_normal (PRIMME_COMPLEX_DOUBLE *evals,
                 PRIMME_COMPLEX_DOUBLE *evecs,
                 double *resNorms, primme_params *primme)
```

There are more versions for matrix problems working in other precisions:

Precision	Complex
half	*kprimme_normal()* *kcprimme_normal()*
single	*cprimme_normal()*
double	*zprimme_normal()*

Other useful functions:

```
void primme_initialize (primme_params *primme)
int primme_set_method (primme_preset_method method,
                                                      primme_params *params)
void primme_display_params (primme_params primme)
void primme_free (primme_params *primme)
```

PRIMME stores its data on the structure *primme_params*. See *Parameters Guide* for an introduction about its fields.

2.1.1 Running

To use PRIMME, follow these basic steps.

1. Include:

   ```
   #include "primme.h"   /* header file is required to run primme */
   ```

2. Initialize a PRIMME parameters structure for default settings:

   ```
   primme_params primme;
   primme_initialize (&primme);
   ```

3. Set problem parameters (see also *Parameters Guide*), and, optionally, set one of the `preset methods`:

   ```
   primme.matrixMatvec = LaplacianMatrixMatvec; /* MV product */
   primme.n = 100;                         /* set problem dimension */
   primme.numEvals = 10;       /* Number of wanted eigenpairs */
   ret = primme_set_method (method, &primme);
   ...
   ```

4. Then call the solver:

   ```
   ret = dprimme (evals, evecs, resNorms, &primme);
   ```

   The call arguments are:

   - *evals*, array to return the found eigenvalues;
   - *evecs*, array to return the found eigenvectors;
   - *resNorms*, array to return the residual norms of the found eigenpairs; and
   - *ret*, returned error code.

5. To free the work arrays in PRIMME:

   ```
   primme_free (&primme);
   ```

See usage examples at the directory *examples*.

2.1.2 Parameters Guide

PRIMME stores the data on the structure `primme_params`, which has the next fields:

Basic
`PRIMME_INT n`, matrix dimension.
`void (* matrixMatvec )(...)`, matrix-vector product.
`void (* massMatrixMatvec )(...)`, mass matrix-vector product (null for standard problems).
`int numEvals`, how many eigenpairs to find.
`primme_target target`, which eigenvalues to find.
`int numTargetShifts`, for targeting interior eigenpairs.
`double * targetShifts`
`double eps`, tolerance of the residual norm of converged eigenpairs.

For parallel programs
`int numProcs`, number of processes
`int procID`, rank of this process

`PRIMME_INT` *`nLocal`*, number of rows stored in this process

`void (* globalSumReal )(...)`, sum reduction among processes

`void (* broadcastReal )(...)`, broadcast array among processes

Accelerate the convergence

`void (* applyPreconditioner )(...)`, preconditioner-vector product.

`int initSize`, initial vectors as approximate solutions.

`int maxBasisSize`

`int minRestartSize`

`int maxBlockSize`

User data

`void * commInfo`

`void * matrix`

`void * massMatrix`

`void * preconditioner`

`void * convtest`

`void * monitor`

Advanced options

`PRIMME_INT ldevecs`, leading dimension of the evecs.

`int numOrthoConst`, orthogonal constrains to the eigenvectors.

`int dynamicMethodSwitch`

`int locking`

`PRIMME_INT maxMatvecs`

`PRIMME_INT maxOuterIterations`

`PRIMME_INT iseed [4]`

`double aNorm`

`double BNorm`

`double invBNorm`

`int printLevel`

`FILE * outputFile`

`double * ShiftsForPreconditioner`

`primme_init initBasisMode`

`struct projection_params projectionParams`

`struct restarting_params restartingParams`

`struct correction_params correctionParams`

`struct primme_stats stats`

`void (* convTestFun )(...)`, custom convergence criterion.

`PRIMME_INT ldOPs`, leading dimension to use in `matrixMatvec`.

`void (* monitorFun )(...)`, custom convergence history.

`primme_op_datatype matrixMatvec_type`

`primme_op_datatype massMatrixMatvec_type`

`primme_op_datatype applyPreconditioner_type`

`primme_op_datatype globalSumReal_type`

`primme_op_datatype broadcastReal_type`

`primme_op_datatype internalPrecision`

`primme_orth` *orth*

PRIMME requires the user to set at least the dimension of the matrix (*n*) and the matrix-vector product (*matrixMatvec*), as they define the problem to be solved. For parallel programs, *nLocal*, *procID* and *globalSumReal* are also required.

In addition, most users would want to specify how many eigenpairs to find, *numEvals*, and provide a preconditioner *applyPreconditioner* (if available).

It is useful to have set all these before calling *primme_set_method()*. Also, if users have a preference on *maxBasisSize*, *maxBlockSize*, etc, they should also provide them into *primme_params* prior to the *primme_set_method()* call. This helps *primme_set_method()* make the right choice on other parameters. It is sometimes useful to check the actual parameters that PRIMME is going to use (before calling it) or used (on return) by printing them with *primme_display_params()*.

2.1.3 Interface Description

The next enumerations and functions are declared in `primme.h`.

?primme

int **hprimme** (*PRIMME_HALF* **evals*, *PRIMME_HALF* **evecs*, *PRIMME_HALF* **resNorms*, *primme_params* **primme*)

int **hsprimme** (float **evals*, *PRIMME_HALF* **evecs*, float **resNorms*, *primme_params* **primme*)

int **kprimme** (*PRIMME_HALF* **evals*, *PRIMME_COMPLEX_HALF* **evecs*, *PRIMME_HALF* **resNorms*, *primme_params* **primme*)

int **ksprimme** (float **evals*, *PRIMME_COMPLEX_HALF* **evecs*, float **resNorms*, *primme_params* **primme*)

New in version 3.0.

int **sprimme** (float **evals*, float **evecs*, float **resNorms*, *primme_params* **primme*)

int **cprimme** (float **evals*, *PRIMME_COMPLEX_FLOAT* **evecs*, float **resNorms*, *primme_params* **primme*)

New in version 2.0.

int **dprimme** (double **evals*, double **evecs*, double **resNorms*, *primme_params* **primme*)

int **zprimme** (double **evals*, *PRIMME_COMPLEX_DOUBLE* **evecs*, double **resNorms*, *primme_params* **primme*)

Solve a real symmetric/Hermitian standard or generalized eigenproblem.

All arrays should be hosted on CPU. The computations are performed on CPU (see *magma_dprimme()* for using GPUs).

Parameters

- **evals** – array at least of size *numEvals* to store the computed eigenvalues; all processes in a parallel run return this local array with the same values.
- **evecs** – array at least of size *nLocal* times (*numOrthoConst* + *numEvals*) with leading dimension *ldevecs* to store column-wise the (local part for this process of the) computed eigenvectors.
- **resNorms** – array at least of size *numEvals* to store the residual norms of the computed eigenpairs; all processes in parallel run return this local array with the same values.

- **`primme`** – parameters structure.

Returns error indicator; see *Error Codes*.

On input, `evecs` should start with the content of the `numOrthoConst` vectors, followed by the `initSize` vectors.

On return, the i-th eigenvector starts at evecs[(`numOrthoConst` + i)* `ldevecs`], with value *evals[i]* and associated residual 2-norm *resNorms[i]*. The first vector has index i=0. The number of eigenpairs marked as converged (see `eps`) is returned on `initSize`. Since version 4.0, if the returned error code is *PRIMME_MAIN_ITER_FAILURE*, PRIMME may return also unconverged eigenpairs and its residual norms in *evecs*, *evals*, and *resNorms* starting at i=|initSize| and going up to either `numEvals`-1 or the last *resNorms[i]* with non-negative value.

All internal operations are performed at the same precision than `evecs` unless the user sets `internalPrecision` otherwise. The functions `hsprimme()` and `ksprimme()` perform all computations in half precision by default and report the eigenvalues and the residual norms in single precision. These functions may help in applications that may be not built with a compiler supporting half precision.

The type and precision of the callbacks is also the same as *evecs*. Although this can be changed. See details for `matrixMatvec`, `massMatrixMatvec`, `applyPreconditioner`, `globalSumReal`, `broadcastReal`, and `convTestFun`.

magma_?primme

int **`magma_hprimme`** (*PRIMME_HALF *evals*, *PRIMME_HALF *evecs*, *PRIMME_HALF *resNorms*, *primme_params *primme*)

int **`magma_hsprimme`** (float **evals*, *PRIMME_HALF *evecs*, float **resNorms*, *primme_params *primme*)

int **`magma_kprimme`** (*PRIMME_HALF *evals*, *PRIMME_COMPLEX_HALF *evecs*, *PRIMME_HALF *resNorms*, *primme_params *primme*)

int **`magma_sprimme`** (float **evals*, float **evecs*, float **resNorms*, *primme_params *primme*)

int **`magma_ksprimme`** (float **evals*, *PRIMME_COMPLEX_HALF *evecs*, float **resNorms*, *primme_params *primme*)

int **`magma_cprimme`** (float **evals*, *PRIMME_COMPLEX_FLOAT *evecs*, float **resNorms*, *primme_params *primme*)

int **`magma_dprimme`** (double **evals*, double **evecs*, double **resNorms*, *primme_params *primme*)

int **`magma_zprimme`** (double **evals*, *PRIMME_COMPLEX_DOUBLE *evecs*, double **resNorms*, *primme_params *primme*)

Solve a real symmetric/Hermitian standard or generalized eigenproblem.

Most of the computations are performed on GPU (see `dprimme()` for using only the CPU).

Parameters

- **`evals`** – CPU array at least of size `numEvals` to store the computed eigenvalues; all processes in a parallel run return this local array with the same values.
- **`evecs`** – GPU array at least of size `nLocal` times (`numOrthoConst` + `numEvals`) with leading dimension `ldevecs` to store column-wise the (local part for this process of the) computed eigenvectors.
- **`resNorms`** – CPU array at least of size `numEvals` to store the residual norms of the computed eigenpairs; all processes in parallel run return this local array with the same values.
- **`primme`** – parameters structure.

Returns error indicator; see *Error Codes*.

On input, `evecs` should start with the content of the `numOrthoConst` vectors, followed by the `initSize` vectors.

On return, the i-th eigenvector starts at evecs[(`numOrthoConst` + i)* `ldevecs`], with value *evals[i]* and associated residual 2-norm *resNorms[i]*. The first vector has index i=0. The number of eigenpairs marked as converged (see `eps`) is returned on `initSize`. Since version 4.0, if the returned error code is *PRIMME_MAIN_ITER_FAILURE*, PRIMME may return also unconverged eigenpairs and its residual norms in *evecs*, *evals*, and *resNorms* starting at i=|initSize| and going up to either `numEvals`-1 or the last *resNorms[i]* with non-negative value.

All internal operations are performed at the same precision than `evecs` unless the user sets `internalPrecision` otherwise. The functions `hsprimme()` and `ksprimme()` perform all computations in half precision by default and report the eigenvalues and the residual norms in single precision. These functions may help in applications that may be not built with a compiler supporting half precision.

The type and precision of the callbacks is also the same as *evecs*. Although this can be changed. See details for `matrixMatvec`, `massMatrixMatvec`, `applyPreconditioner`, `globalSumReal`, `broadcastReal`, and `convTestFun`.

New in version 3.0.

?primme_normal

int **kprimme_normal**(*PRIMME_COMPLEX_HALF *evals*, *PRIMME_COMPLEX_HALF *evecs*, *PRIMME_HALF *resNorms*, *primme_params *primme*)

int **kcprimme_normal**(*PRIMME_COMPLEX_FLOAT *evals*, *PRIMME_COMPLEX_HALF *evecs*, float **resNorms*, *primme_params *primme*)

int **cprimme_normal**(*PRIMME_COMPLEX_FLOAT *evals*, *PRIMME_COMPLEX_FLOAT *evecs*, float **resNorms*, *primme_params *primme*)

int **zprimme_normal**(*PRIMME_COMPLEX_DOUBLE *evals*, *PRIMME_COMPLEX_DOUBLE *evecs*, double **resNorms*, *primme_params *primme*)

Solve a normal standard eigenproblem, which may not be Hermitian.

All arrays should be hosted on CPU. The computations are performed on CPU (see `magma_zprimme_normal()` for using GPUs).

Parameters

- **evals** – array at least of size `numEvals` to store the computed eigenvalues; all processes in a parallel run return this local array with the same values.
- **evecs** – array at least of size `nLocal` times (`numOrthoConst` + `numEvals`) with leading dimension `ldevecs` to store column-wise the (local part for this process of the) computed eigenvectors.
- **resNorms** – array at least of size `numEvals` to store the residual norms of the computed eigenpairs; all processes in parallel run return this local array with the same values.
- **primme** – parameters structure.

Returns error indicator; see *Error Codes*.

On input, `evecs` should start with the content of the `numOrthoConst` vectors, followed by the `initSize` vectors.

On return, the i-th eigenvector starts at evecs[(`numOrthoConst` + i)* `ldevecs`], with value *evals[i]* and associated residual 2-norm *resNorms[i]*. The first vector has index i=0. The number of eigenpairs marked as converged (see `eps`) is returned on `initSize`. Since version 4.0, if the returned error code is *PRIMME_MAIN_ITER_FAILURE*, PRIMME may return also unconverged eigenpairs and its residual norms in *evecs*, *evals*, and *resNorms* starting at i=|initSize| and going up to either `numEvals`-1 or the last *resNorms[i]* with non-negative value.

All internal operations are performed at the same precision than `evecs` unless the user sets `internalPrecision` otherwise. The functions `hsprimme()` and `ksprimme()` perform all computations in half precision by default and report the eigenvalues and the residual norms in single precision. These functions may help in applications that may be not built with a compiler supporting half precision.

The type and precision of the callbacks is also the same as *evecs*. Although this can be changed. See details for `matrixMatvec`, `applyPreconditioner`, `globalSumReal`, `broadcastReal`, and `convTestFun`.

New in version 3.0.

magma_?primme_normal

int **magma_kprimme_normal**(*PRIMME_COMPLEX_HALF* **evals*, *PRIMME_COMPLEX_HALF* **evecs*, *PRIMME_HALF* **resNorms*, *primme_params* **primme*)

int **magma_kcprimme_normal**(*PRIMME_COMPLEX_FLOAT* **evals*, *PRIMME_COMPLEX_HALF* **evecs*, float **resNorms*, *primme_params* **primme*)

int **magma_cprimme_normal**(*PRIMME_COMPLEX_FLOAT* **evals*, *PRIMME_COMPLEX_FLOAT* **evecs*, float **resNorms*, *primme_params* **primme*)

int **magma_zprimme_normal**(*PRIMME_COMPLEX_DOUBLE* **evals*, *PRIMME_COMPLEX_DOUBLE* **evecs*, double **resNorms*, *primme_params* **primme*)

Solve a normal standard eigenproblem, which may not be Hermitian.

Most of the computations are performed on GPU (see `zprimme_normal()` for using only the CPU).

Parameters

- **evals** – CPU array at least of size `numEvals` to store the computed eigenvalues; all processes in a parallel run return this local array with the same values.
- **evecs** – GPU array at least of size `nLocal` times (`numOrthoConst` + `numEvals`) with leading dimension `ldevecs` to store column-wise the (local part for this process of the) computed eigenvectors.
- **resNorms** – CPU array at least of size `numEvals` to store the residual norms of the computed eigenpairs; all processes in parallel run return this local array with the same values.
- **primme** – parameters structure.

Returns error indicator; see *Error Codes*.

On input, `evecs` should start with the content of the `numOrthoConst` vectors, followed by the `initSize` vectors.

On return, the i-th eigenvector starts at evecs[(`numOrthoConst` + i)* `ldevecs`], with value *evals[i]* and associated residual 2-norm *resNorms[i]*. The first vector has index i=0. The number of eigenpairs marked as converged (see `eps`) is returned on `initSize`. Since version 4.0, if the returned error code is *PRIMME_MAIN_ITER_FAILURE*, PRIMME may return also unconverged eigenpairs and its residual norms in *evecs*, *evals*, and *resNorms* starting at i=|initSize| and going up to either `numEvals`-1 or the last *resNorms[i]* with non-negative value.

All internal operations are performed at the same precision than `evecs` unless the user sets `internalPrecision` otherwise. The functions `hsprimme()` and `ksprimme()` perform all computations in half precision by default and report the eigenvalues and the residual norms in single precision. These functions may help in applications that may be not built with a compiler supporting half precision.

The type and precision of the callbacks is also the same as *evecs*. Although this can be changed. See details for `matrixMatvec`, `applyPreconditioner`, `globalSumReal`, `broadcastReal`, and `convTestFun`.

New in version 3.0.

primme_initialize

void **`primme_initialize`**(*primme_params *primme*)

Initialize a PRIMME parameters structure to the default values.

After calling `dprimme()` (or a variant), call `primme_free()` to release allocated resources by PRIMME.

Parameters

- **`primme`** – parameters structure.

Example:

```
primme_params primme;
primme_initialize(&primme);

primme.n = 100;
...
dprimme(evals, evecs, rnorms, &primme);
...

primme_free(&primme);
```

See the alternative function `primme_params_create()` that also allocates the *primme_params* structure.

primme_params_create

primme_params ***`primme_params_create`**(void)

Allocate and initialize a parameters structure to the default values.

After calling `dprimme()` (or a variant), call `primme_params_destroy()` to release allocated resources by PRIMME.

Returns pointer to a parameters structure.

Example:

```
primme_params *primme = primme_params_create();

primme->n = 100;
...
dprimme(evals, evecs, rnorms, primme);
...

primme_params_destroy(primme);
```

See the alternative function `primme_initialize()` that only initializes the structure.

New in version 3.0.

primme_set_method

int **primme_set_method**(*primme_preset_method method*, *primme_params *primme*)

Set PRIMME parameters to one of the preset configurations.

Parameters

- **method** – preset configuration; one of

  PRIMME_DYNAMIC
  PRIMME_DEFAULT_MIN_TIME
  PRIMME_DEFAULT_MIN_MATVECS
  PRIMME_Arnoldi
  PRIMME_GD
  PRIMME_GD_plusK
  PRIMME_GD_Olsen_plusK
  PRIMME_JD_Olsen_plusK
  PRIMME_RQI
  PRIMME_JDQR
  PRIMME_JDQMR
  PRIMME_JDQMR_ETol
  PRIMME_STEEPEST_DESCENT
  PRIMME_LOBPCG_OrthoBasis
  PRIMME_LOBPCG_OrthoBasis_Window

- **primme** – parameters structure.

See also *Preset Methods*.

primme_display_params

void **primme_display_params**(*primme_params primme*)

Display all printable settings of `primme` into the file descriptor *outputFile*.

Parameters

- **primme** – parameters structure.

primme_free

void **primme_free**(*primme_params *primme*)

Free memory allocated by PRIMME.

Parameters

- **primme** – parameters structure.

primme_params_destroy

int **`primme_params_destroy`**(*primme_params *primme*)

Free memory allocated by PRIMME associated to a parameters structure created with *`primme_params_create()`*.

Parameters

- **`primme`** – parameters structure.

Returns nonzero value if the call is not successful.

New in version 3.0.

2.2 FORTRAN 77 Library Interface

The next enumerations and functions are declared in `primme_f77.h`.

`type ptr`

Fortran datatype with the same size as a pointer. Use `integer*4` when compiling in 32 bits and `integer*8` in 64 bits.

2.2.1 primme_initialize_f77

`subroutine primme_initialize_f77`(*primme*)

Allocate and initialize a PRIMME parameters structure to the default values.

After calling *`dprimme_f77()`* (or a variant), call *`primme_free_f77()`* to release allocated resources by PRIMME.

Parameters **primme** *[ptr]* :: (output) parameters structure.

2.2.2 primme_set_method_f77

`subroutine primme_set_method_f77`(*method*, *primme*, *ierr*)

Set PRIMME parameters to one of the preset configurations.

Parameters

- **method** *[integer]* :: (input) preset configuration. One of:

  ```
  PRIMME_DYNAMIC
  PRIMME_DEFAULT_MIN_TIME
  PRIMME_DEFAULT_MIN_MATVECS
  PRIMME_Arnoldi
  PRIMME_GD
  PRIMME_GD_plusK
  PRIMME_GD_Olsen_plusK
  PRIMME_JD_Olsen_plusK
  PRIMME_RQI
  PRIMME_JDQR
  PRIMME_JDQMR
  PRIMME_JDQMR_ETol
  PRIMME_STEEPEST_DESCENT
  PRIMME_LOBPCG_OrthoBasis
  PRIMME_LOBPCG_OrthoBasis_Window
  ```

  See *`primme_preset_method`*.

- **primme** *[ptr]* :: (input) parameters structure.
- **ierr** *[integer]* :: (output) if 0, successful; if negative, something went wrong.

2.2.3 primme_free_f77

subroutine primme_free_f77(*primme*)

Free memory allocated by PRIMME and delete all values set.

Parameters **primme** *[ptr]* :: (input/output) parameters structure.

2.2.4 sprimme_f77

subroutine sprimme_f77(*evals*, *evecs*, *resNorms*, *primme*, *ierr*)

Solve a real symmetric standard or generalized eigenproblem.

All arrays should be hosted on CPU. The computations are performed on CPU (see `magma_sprimme_f77()` for using GPUs).

Parameters

- **evals** (*) *[real]* :: (output) array at least of size `numEvals` to store the computed eigenvalues; all parallel calls return the same value in this array.
- **evecs** (*) *[real]* :: (input/output) array at least of size `nLocal` times (`numOrthoConst` + `numEvals`) with leading dimension `ldevecs` to store column-wise the (local part for this process of the) computed eigenvectors.
- **resNorms** (*) *[real]* :: (output) array at least of size `numEvals` to store the residual norms of the computed eigenpairs; all parallel calls return the same value in this array.
- **primme** *[ptr]* :: parameters structure.
- **ierr** *[integer]* :: (output) error indicator; see *Error Codes*.

Further descriptions of *evals*, *evecs*, and *resNorms* on notes in subroutine `dprimme_f77()`.

New in version 2.0.

2.2.5 cprimme_f77

subroutine cprimme_f77(*evals*, *evecs*, *resNorms*, *primme*, *ierr*)

Solve a Hermitian standard or generalized eigenproblem.

All arrays should be hosted on CPU. The computations are performed on CPU (see `magma_cprimme_f77()` for using GPUs).

Parameters

- **evals** (*) *[real]* :: (output) array at least of size `numEvals` to store the computed eigenvalues; all parallel calls return the same value in this array.
- **evecs** (*) *[complex real]* :: (input/output) array at least of size `nLocal` times (`numOrthoConst` + `numEvals`) with leading dimension `ldevecs` to store column-wise the (local part for this process of the) computed eigenvectors.
- **resNorms** (*) *[real]* :: (output) array at least of size `numEvals` to store the residual norms of the computed eigenpairs; all parallel calls return the same value in this array.
- **primme** *[ptr]* :: (input) parameters structure.
- **ierr** *[integer]* :: (output) error indicator; see *Error Codes*.

Further descriptions of *evals*, *evecs*, and *resNorms* on notes in subroutine `dprimme_f77()`.

New in version 2.0.

2.2.6 dprimme_f77

subroutine dprimme_f77(*evals*, *evecs*, *resNorms*, *primme*, *ierr*)

Solve a real symmetric standard or generalized eigenproblem.

All arrays should be hosted on CPU. The computations are performed on CPU (see `magma_dprimme_f77()` for using GPUs).

Parameters

- **evals** (*) *[double precision]* :: (output) array at least of size `numEvals` to store the computed eigenvalues; all parallel calls return the same value in this array.
- **evecs** (*) *[double precision]* :: (input/output) array at least of size `nLocal` times (`numOrthoConst` + `numEvals`) with leading dimension `ldevecs` to store columnwise the (local part for this process of the) computed eigenvectors.
- **resNorms** (*) *[double precision]* :: (output) array at least of size `numEvals` to store the residual norms of the computed eigenpairs; all parallel calls return the same value in this array.
- **primme** *[ptr]* :: parameters structure.
- **ierr** *[integer]* :: (output) error indicator; see *Error Codes*.

On input, `evecs` should start with the content of the `numOrthoConst` vectors, followed by the `initSize` vectors.

On return, the i-th eigenvector starts at evecs((`numOrthoConst` + i - 1)* `ldevecs` + 1), with value *evals(i)* and associated residual 2-norm *resNorms(i)*. The first vector has index i=1. The number of eigenpairs marked as converged (see `eps`) is returned on `initSize`. Since version 4.0, if the returned error code is *PRIMME_MAIN_ITER_FAILURE*, PRIMME may return also unconverged eigenpairs and its residual norms in *evecs*, *evals*, and *resNorms* starting at i = `initSize` + 1 and going up to either `numEvals` or the last *resNorms(i)* with non-negative value.

All internal operations are performed at the same precision than `evecs` unless the user sets `internalPrecision` otherwise.

The type and precision of the callbacks is also the same as *evecs*. Although this can be changed. See details for `matrixMatvec`, `massMatrixMatvec`, `applyPreconditioner`, `globalSumReal`, `broadcastReal`, and `convTestFun`.

2.2.7 zprimme_f77

subroutine zprimme_f77(*evals*, *evecs*, *resNorms*, *primme*, *ierr*)

Solve a Hermitian standard or generalized eigenproblem.

All arrays should be hosted on CPU. The computations are performed on CPU (see `magma_zprimme_f77()` for using GPUs).

Parameters

- **evals** (*) *[double precision]* :: (output) array at least of size `numEvals` to store the computed eigenvalues; all parallel calls return the same value in this array.
- **evecs** (*) *[complex double precision]* :: (input/output) array at least of size `nLocal` times (`numOrthoConst` + `numEvals`) with leading dimension `ldevecs` to store columnwise the (local part for this process of the) computed eigenvectors.

- **resNorms** (*) *[double precision]* :: (output) array at least of size `numEvals` to store the residual norms of the computed eigenpairs; all parallel calls return the same value in this array.
- **primme** *[*ptr*]* :: parameters structure.
- **ierr** *[integer]* :: (output) error indicator; see *Error Codes*.

Further descriptions of *evals*, *evecs*, and *resNorms* on notes in subroutine `dprimme_f77()`.

2.2.8 magma_sprimme_f77

subroutine magma_sprimme_f77(*evals*, *evecs*, *resNorms*, *primme*, *ierr*)

Solve a real symmetric standard or generalized eigenproblem.

Most of the computations are performed on GPU (see `sprimme_f77()` for using only the CPU).

Parameters

- **evals** (*) *[real]* :: (output) CPU array at least of size `numEvals` to store the computed eigenvalues; all parallel calls return the same value in this array.
- **evecs** (*) *[real]* :: (input/output) GPU array at least of size `nLocal` times (`numOrthoConst` + `numEvals`) with leading dimension `ldevecs` to store column-wise the (local part for this process of the) computed eigenvectors.
- **resNorms** (*) *[real]* :: (output) CPU array at least of size `numEvals` to store the residual norms of the computed eigenpairs; all parallel calls return the same value in this array.
- **primme** *[*ptr*]* :: parameters structure.
- **ierr** *[integer]* :: (output) error indicator; see *Error Codes*.

Further descriptions of *evals*, *evecs*, and *resNorms* on notes in subroutine `dprimme_f77()`.

New in version 3.0.

2.2.9 magma_cprimme_f77

subroutine magma_cprimme_f77(*evals*, *evecs*, *resNorms*, *primme*, *ierr*)

Solve a Hermitian standard or generalized eigenproblem.

Most of the computations are performed on GPU (see `cprimme_f77()` for using only the CPU).

Parameters

- **evals** (*) *[real]* :: (output) CPU array at least of size `numEvals` to store the computed eigenvalues; all parallel calls return the same value in this array.
- **evecs** (*) *[complex real]* :: (input/output) GPU array at least of size `nLocal` times (`numOrthoConst` + `numEvals`) with leading dimension `ldevecs` to store column-wise the (local part for this process of the) computed eigenvectors.
- **resNorms** (*) *[real]* :: (output) CPU array at least of size `numEvals` to store the residual norms of the computed eigenpairs; all parallel calls return the same value in this array.
- **primme** *[*ptr*]* :: (input) parameters structure.
- **ierr** *[integer]* :: (output) error indicator; see *Error Codes*.

Further descriptions of *evals*, *evecs*, and *resNorms* on notes in subroutine `dprimme_f77()`.

New in version 3.0.

2.2.10 magma_dprimme_f77

subroutine magma_dprimme_f77(*evals*, *evecs*, *resNorms*, *primme*, *ierr*)

Solve a real symmetric standard or generalized eigenproblem.

Most of the computations are performed on GPU (see *dprimme_f77()* for using only the CPU).

Parameters

- **evals** (*) *[double precision]* :: (output) CPU array at least of size *numEvals* to store the computed eigenvalues; all parallel calls return the same value in this array.
- **evecs** (*) *[double precision]* :: (input/output) GPU array at least of size *nLocal* times (*numOrthoConst* + *numEvals*) with leading dimension *ldevecs* to store column-wise the (local part for this process of the) computed eigenvectors.
- **resNorms** (*) *[double precision]* :: (output) CPU array at least of size *numEvals* to store the residual norms of the computed eigenpairs; all parallel calls return the same value in this array.
- **primme** *[ptr]* :: parameters structure.
- **ierr** *[integer]* :: (output) error indicator; see *Error Codes*.

Further descriptions of *evals*, *evecs*, and *resNorms* on notes in subroutine *dprimme_f77()*.

New in version 3.0.

2.2.11 magma_zprimme_f77

subroutine magma_zprimme_f77(*evals*, *evecs*, *resNorms*, *primme*, *ierr*)

Solve a Hermitian standard or generalized eigenproblem.

Most of the computations are performed on GPU (see *zprimme_f77()* for using only the CPU).

Parameters

- **evals** (*) *[double precision]* :: (output) CPU array at least of size *numEvals* to store the computed eigenvalues; all parallel calls return the same value in this array.
- **evecs** (*) *[complex double precision]* :: (input/output) GPU array at least of size *nLocal* times (*numOrthoConst* + *numEvals*) with leading dimension *ldevecs* to store column-wise the (local part for this process of the) computed eigenvectors.
- **resNorms** (*) *[double precision]* :: (output) CPU array at least of size *numEvals* to store the residual norms of the computed eigenpairs; all parallel calls return the same value in this array.
- **primme** *[ptr]* :: (input) parameters structure.
- **ierr** *[integer]* :: (output) error indicator; see *Error Codes*.

Further descriptions of *evals*, *evecs*, and *resNorms* on notes in subroutine *dprimme_f77()*.

New in version 3.0.

2.2.12 cprimme_normal_f77

subroutine cprimme_normal_f77 (*evals*, *evecs*, *resNorms*, *primme*, *ierr*)

Solve a normal standard eigenproblem, which may not be Hermitian.

All arrays should be hosted on CPU. The computations are performed on CPU (see `magma_cprimme_normal_f77()` for using GPUs).

Parameters

- **evals** (*) *[real]* :: (output) array at least of size `numEvals` to store the computed eigenvalues; all parallel calls return the same value in this array.
- **evecs** (*) *[complex real]* :: (input/output) array at least of size `nLocal` times (`numOrthoConst` + `numEvals`) with leading dimension `ldevecs` to store column-wise the (local part for this process of the) computed eigenvectors.
- **resNorms** (*) *[real]* :: (output) array at least of size `numEvals` to store the residual norms of the computed eigenpairs; all parallel calls return the same value in this array.
- **primme** *[ptr]* :: (input) parameters structure.
- **ierr** *[integer]* :: (output) error indicator; see *Error Codes*.

Further descriptions of *evals*, *evecs*, and *resNorms* on notes in subroutine `dprimme_f77()`.

New in version 3.0.

2.2.13 zprimme_normal_f77

subroutine zprimme_normal_f77 (*evals*, *evecs*, *resNorms*, *primme*, *ierr*)

Solve a normal standard eigenproblem, which may not be Hermitian.

All arrays should be hosted on CPU. The computations are performed on CPU (see `magma_zprimme_normal_f77()` for using GPUs).

Parameters

- **evals** (*) *[double precision]* :: (output) array at least of size `numEvals` to store the computed eigenvalues; all parallel calls return the same value in this array.
- **evecs** (*) *[complex double precision]* :: (input/output) array at least of size `nLocal` times (`numOrthoConst` + `numEvals`) with leading dimension `ldevecs` to store column-wise the (local part for this process of the) computed eigenvectors.
- **resNorms** (*) *[double precision]* :: (output) array at least of size `numEvals` to store the residual norms of the computed eigenpairs; all parallel calls return the same value in this array.
- **primme** *[ptr]* :: parameters structure.
- **ierr** *[integer]* :: (output) error indicator; see *Error Codes*.

Further descriptions of *evals*, *evecs*, and *resNorms* on notes in subroutine `dprimme_f77()`.

New in version 3.0.

2.2.14 magma_cprimme_normal_f77

subroutine magma_cprimme_normal_f77 (*evals*, *evecs*, *resNorms*, *primme*, *ierr*)

Solve a normal standard eigenproblem, which may not be Hermitian.

Most of the arrays are stored on GPU, and also most of the computations are done on GPU (see `cprimme_normal_f77()` for using only the CPU).

Parameters

- **evals** (*) *[real]* :: (output) CPU array at least of size `numEvals` to store the computed eigenvalues; all parallel calls return the same value in this array.
- **evecs** (*) *[complex real]* :: (input/output) GPU array at least of size `nLocal` times (`numOrthoConst` + `numEvals`) with leading dimension `ldevecs` to store column-wise the (local part for this process of the) computed eigenvectors.
- **resNorms** (*) *[real]* :: (output) CPU array at least of size `numEvals` to store the residual norms of the computed eigenpairs; all parallel calls return the same value in this array.
- **primme** *[ptr]* :: (input) parameters structure.
- **ierr** *[integer]* :: (output) error indicator; see *Error Codes*.

Further descriptions of *evals*, *evecs*, and *resNorms* on notes in subroutine `dprimme_f77()`.

New in version 3.0.

2.2.15 magma_zprimme_normal_f77

subroutine magma_zprimme_normal_f77 (*evals*, *evecs*, *resNorms*, *primme*, *ierr*)

Solve a normal standard eigenproblem, which may not be Hermitian.

Most of the arrays are stored on GPU, and also most of the computations are done on GPU (see `zprimme_normal_f77()` for using only the CPU).

Parameters

- **evals** (*) *[double precision]* :: (output) CPU array at least of size `numEvals` to store the computed eigenvalues; all parallel calls return the same value in this array.
- **evecs** (*) *[complex double precision]* :: (input/output) GPU array at least of size `nLocal` times (`numOrthoConst` + `numEvals`) with leading dimension `ldevecs` to store column-wise the (local part for this process of the) computed eigenvectors.
- **resNorms** (*) *[double precision]* :: (output) CPU array at least of size `numEvals` to store the residual norms of the computed eigenpairs; all parallel calls return the same value in this array.
- **primme** *[ptr]* :: (input) parameters structure.
- **ierr** *[integer]* :: (output) error indicator; see *Error Codes*.

Further descriptions of *evals*, *evecs*, and *resNorms* on notes in subroutine `dprimme_f77()`.

New in version 3.0.

2.2.16 primme_set_member_f77

subroutine primme_set_member_f77 (*primme*, *label*, *value*)

Set a value in some field of the parameter structure.

Parameters

- **primme** *[ptr]* :: (input) parameters structure.
- **label** *[integer]* :: field where to set value. One of:

PRIMME_n
PRIMME_matrixMatvec
PRIMME_matrixMatvec_type
PRIMME_applyPreconditioner
PRIMME_applyPreconditioner_type
PRIMME_massMatrixMatvec
PRIMME_massMatrixMatvec_type
PRIMME_numProcs
PRIMME_procID
PRIMME_commInfo
PRIMME_nLocal
PRIMME_globalSumReal
PRIMME_globalSumReal_type
PRIMME_broadcastReal
PRIMME_broadcastReal_type
PRIMME_numEvals
PRIMME_target
PRIMME_numTargetShifts
PRIMME_targetShifts
PRIMME_locking
PRIMME_initSize
PRIMME_numOrthoConst
PRIMME_maxBasisSize
PRIMME_minRestartSize
PRIMME_maxBlockSize
PRIMME_maxMatvecs
PRIMME_maxOuterIterations
PRIMME_iseed
PRIMME_aNorm
PRIMME_BNorm
PRIMME_invBNorm
PRIMME_eps
PRIMME_orth
PRIMME_internalPrecision
PRIMME_printLevel
PRIMME_outputFile
PRIMME_matrix
PRIMME_massMatrix

PRIMME_preconditioner
PRIMME_initBasisMode
PRIMME_projectionParams_projection
PRIMME_restartingParams_maxPrevRetain
PRIMME_correctionParams_precondition
PRIMME_correctionParams_robustShifts
PRIMME_correctionParams_maxInnerIterations
PRIMME_correctionParams_projectors_LeftQ
PRIMME_correctionParams_projectors_LeftX
PRIMME_correctionParams_projectors_RightQ
PRIMME_correctionParams_projectors_RightX
PRIMME_correctionParams_projectors_SkewQ
PRIMME_correctionParams_projectors_SkewX
PRIMME_correctionParams_convTest
PRIMME_correctionParams_relTolBase
PRIMME_stats_numOuterIterations
PRIMME_stats_numRestarts
PRIMME_stats_numMatvecs
PRIMME_stats_numPreconds
PRIMME_stats_numGlobalSum
PRIMME_stats_volumeGlobalSum
PRIMME_stats_numBroadcast
PRIMME_stats_volumeBroadcast
PRIMME_stats_flopsDense
PRIMME_stats_numOrthoInnerProds
PRIMME_stats_elapsedTime
PRIMME_stats_timeMatvec
PRIMME_stats_timePrecond
PRIMME_stats_timeOrtho
PRIMME_stats_timeGlobalSum
PRIMME_stats_timeBroadcast
PRIMME_stats_timeDense
PRIMME_stats_estimateMinEVal
PRIMME_stats_estimateMaxEVal
PRIMME_stats_estimateLargestSVal
PRIMME_stats_estimateBNorm
PRIMME_stats_estimateInvBNorm
PRIMME_stats_maxConvTol
PRIMME_stats_lockingIssue
PRIMME_dynamicMethodSwitch
PRIMME_convTestFun
PRIMME_convTestFun_type
PRIMME_convtest
PRIMME_ldevecs
PRIMME_ldOPs
PRIMME_monitorFun
PRIMME_monitorFun_type

PRIMME_monitor
PRIMME_queue

- **value** :: (input) value to set.

  If the type of the option is integer (`int`, *PRIMME_INT*, `size_t`), the type of `value` should be as long as *PRIMME_INT*, which is `integer*8` by default.

Note: Don't use this subroutine inside PRIMME's callback functions, e.g., *matrixMatvec* or *applyPreconditioner*, or in functions called by these functions.

2.2.17 primmetop_get_member_f77

subroutine primmetop_get_member_f77 (*primme*, *label*, *value*)
Get the value in some field of the parameter structure.

Parameters

- **primme** *[ptr]* :: (input) parameters structure.
- **label** *[integer]* :: (input) field where to get value. One of the detailed in subroutine `primmetop_set_member_f77()`.
- **value** :: (output) value of the field.

  If the type of the option is integer (`int`, *PRIMME_INT*, `size_t`), the type of `value` should be as long as *PRIMME_INT*, which is `integer*8` by default.

Note: Don't use this subroutine inside PRIMME's callback functions, e.g., *matrixMatvec* or *applyPreconditioner*, or in functions called by these functions. In those cases use *primme_get_member_f77()*.

Note: When `label` is one of `PRIMME_matrixMatvec`, `PRIMME_applyPreconditioner`, `PRIMME_commInfo`, `PRIMME_matrix` and `PRIMME_preconditioner`, the returned `value` is a C pointer (`void*`). Use Fortran pointer or other extensions to deal with it. For instance:

```
use iso_c_binding
MPI_Comm comm

comm = MPI_COMM_WORLD
call primme_set_member_f77(primme, PRIMME_commInfo, comm)
...
subroutine par_GlobalSumDouble(x,y,k,primme)
use iso_c_binding
implicit none
...
MPI_Comm, pointer :: comm
type(c_ptr) :: pcomm

call primme_get_member_f77(primme, PRIMME_commInfo, pcomm)
call c_f_pointer(pcomm, comm)
call MPI_Allreduce(x,y,k,MPI_DOUBLE,MPI_SUM,comm,ierr)
```

Most users would not need to retrieve these pointers in their programs.

2.2.18 primmetop_get_prec_shift_f77

subroutine primmetop_get_prec_shift_f77(*primme*, *index*, *value*)

Get the value in some position of the array *ShiftsForPreconditioner*.

Parameters

- **primme** *[ptr]* :: (input) parameters structure.
- **index** *[integer]* :: (input) position of the array; the first position is 1.
- **value** :: (output) value of the array at that position.

2.2.19 primme_get_member_f77

subroutine primme_get_member_f77(*primme*, *label*, *value*)

Get the value in some field of the parameter structure.

Parameters

- **primme** *[ptr]* :: (input) parameters structure.
- **label** *[integer]* :: (input) field where to get value. One of the detailed in subroutine `primmetop_set_member_f77()`.
- **value** :: (output) value of the field.

  If the type of the option is integer (`int`, *PRIMME_INT*, `size_t`), the type of `value` should be as long as *PRIMME_INT*, which is `integer*8` by default.

Note: Use this subroutine exclusively inside PRIMME's callback functions, e.g., *matrixMatvec* or *applyPreconditioner*, or in functions called by these functions. Otherwise, e.g., from the main program, use the subroutine *primmetop_get_member_f77()*.

Note: When `label` is one of `PRIMME_matrixMatvec`, `PRIMME_applyPreconditioner`, `PRIMME_commInfo`, `PRIMME_matrix` and `PRIMME_preconditioner`, the returned `value` is a C pointer (`void*`). Use Fortran pointer or other extensions to deal with it. For instance:

```
use iso_c_binding
MPI_Comm comm

comm = MPI_COMM_WORLD
call primme_set_member_f77(primme, PRIMME_commInfo, comm)
...
subroutine par_GlobalSumDouble(x,y,k,primme)
use iso_c_binding
implicit none
...
MPI_Comm, pointer :: comm
type(c_ptr) :: pcomm

call primme_get_member_f77(primme, PRIMME_commInfo, pcomm)
```

(continues on next page)

(continued from previous page)

```
call c_f_pointer(pcomm, comm)
call MPI_Allreduce(x,y,k,MPI_DOUBLE,MPI_SUM,comm,ierr)
```

Most users would not need to retrieve these pointers in their programs.

2.2.20 primme_get_prec_shift_f77

subroutine primme_get_prec_shift_f77 (*primme*, *index*, *value*)

Get the value in some position of the array *ShiftsForPreconditioner*.

Parameters

- **primme** *[ptr]* :: (input) parameters structure.
- **index** *[integer]* :: (input) position of the array; the first position is 1.
- **value** :: (output) value of the array at that position.

Note: Use this subroutine exclusively inside the subroutine *matrixMatvec*, *massMatrixMatvec*, or *applyPreconditioner*. Otherwise use the subroutine *primmetop_get_prec_shift_f77()*.

2.3 FORTRAN 90 Library Interface

New in version 3.0.

The next enumerations and functions are declared in `primme_f90.inc`.

`type c_ptr`
Fortran datatype for C pointers in module *iso_c_binding*.

`subroutine primme_eigs_matvec`(*x*, *ldx*, *y*, *ldy*, *blockSize*, *primme*, *ierr*)
Abstract interface for the callbacks *`matrixMatvec`*, *`massMatrixMatvec`*, and *`applyPreconditioner`*.

Parameters

- **x** (ldx,*) *[type(*),in]* :: matrix of size *`nLocal`* x `blockSize` in column-major order with leading dimension `ldx`.
- **ldx** *[c_int64_t]* :: the leading dimension of the array `x`.
- **y** (ldy,*) *[type(*),out]* :: matrix of size *`nLocal`* x `blockSize` in column-major order with leading dimension `ldy`.
- **ldy** *[c_int64_t]* :: the leading dimension of the array `y`.
- **blockSize** *[c_int,in]* :: number of columns in `x` and `y`.
- **primme** *[c_ptr,in]* :: parameters structure created by *`primme_params_create()`*.
- **ierr** *[c_int,out]* :: output error code; if it is set to non-zero, the current call to PRIMME will stop.

See more details about the precision and type for *x* and *y* in the documentation of the callbacks.

2.3.1 primme_params_create

`function primme_params_create`()
Allocate and initialize a parameters structure to the default values.

After calling *`xprimme()`* (or a variant), call *`primme_params_destroy()`* to release allocated resources by PRIMME.

Return **primme_params_create** *[c_ptr]* :: pointer to a parameters structure.

2.3.2 primme_set_method

`function primme_set_method`(*method*, *primme*)
Set PRIMME parameters to one of the preset configurations.

Parameters

- **method** *[c_int,in]* :: preset configuration. One of:

```
PRIMME_DYNAMIC
PRIMME_DEFAULT_MIN_TIME
PRIMME_DEFAULT_MIN_MATVECS
PRIMME_Arnoldi
PRIMME_GD
```

```
PRIMME_GD_plusK
PRIMME_GD_Olsen_plusK
PRIMME_JD_Olsen_plusK
PRIMME_RQI
PRIMME_JDQR
PRIMME_JDQMR
PRIMME_JDQMR_ETol
PRIMME_STEEPEST_DESCENT
PRIMME_LOBPCG_OrthoBasis
PRIMME_LOBPCG_OrthoBasis_Window
```

See *primme_preset_method*.

- **primme** *[c_ptr,in]* :: parameters structure created by *primme_params_create()*.

Return **primme_set_method** *[c_int]* :: nonzero value if the call is not successful.

2.3.3 primme_params_destroy

function primme_params_destroy (*primme*)

Free memory allocated by PRIMME associated to a parameters structure created with *primme_params_create()*.

Parameters **primme** *[c_ptr]* :: parameters structure.

Return **primme_params_destroy** :: nonzero value if the call is not successful.

2.3.4 xprimme

function xprimme (*evals*, *evecs*, *resNorms*, *primme*)

Solve a real symmetric/Hermitian standard or generalized eigenproblem.

All arrays should be hosted on CPU. The computations are performed on CPU (see *magma_xprimme()* for using GPUs).

Parameters

- **evals** (*) *[real(kind),out]* :: array at least of size *numEvals* to store the computed eigenvalues; all processes in a parallel run return this local array with the same values.
- **evecs** (*) *[real(kind) or complex(kind)]* :: array at least of size *nLocal* times (*numOrthoConst* + *numEvals*) with leading dimension *ldevecs* to store columnwise the (local part for this process of the) computed eigenvectors.
- **resNorms** (*) *[real(kind),out]* :: array at least of size *numEvals* to store the residual norms of the computed eigenpairs; all processes in parallel run return this local array with the same values.
- **primme** *[c_ptr,in]* :: parameters structure created by *primme_params_create()*.

Return **xprimme** *[c_int]* :: error indicator; see *Error Codes*.

The arrays `evals`, `evecs`, and `resNorms` should have the same kind.

On input, `evecs` should start with the content of the *numOrthoConst* vectors, followed by the *initSize* vectors.

On return, the i-th eigenvector starts at evecs((`numOrthoConst` + i - 1)* `ldevecs` + 1), with value *evals(i)* and associated residual 2-norm *resNorms(i)*. The first vector has index i=1. The number of eigenpairs marked as converged (see `eps`) is returned on `initSize`. Since version 4.0, if the returned error code is *PRIMME_MAIN_ITER_FAILURE*, PRIMME may return also unconverged eigenpairs and its residual norms in *evecs*, *evals*, and *resNorms* starting at i = `initSize` + 1 and going up to either `numEvals` or the last *resNorms(i)* with non-negative value.

All internal operations are performed at the same precision than `evecs` unless the user sets `internalPrecision` otherwise.

The type and precision of the callbacks is also the same as *evecs*. Although this can be changed. See details for `matrixMatvec`, `massMatrixMatvec`, `applyPreconditioner`, `globalSumReal`, `broadcastReal`, and `convTestFun`.

2.3.5 magma_xprimme

function magma_xprimme (*evals*, *evecs*, *resNorms*, *primme*)

Solve a real symmetric/Hermitian standard or generalized eigenproblem.

Most of the computations are performed on GPU (see `xprimme()` for using only the CPU).

Parameters

- **evals** (*) *[real(kind),out]* :: CPU array at least of size `numEvals` to store the computed eigenvalues; all processes in a parallel run return this local array with the same values.
- **evecs** :: GPU array at least of size `nLocal` times (`numOrthoConst` + `numEvals`) with leading dimension `ldevecs` to store column-wise the (local part for this process of the) computed eigenvectors.
- **resNorms** (*) *[real(kind),out]* :: CPU array at least of size `numEvals` to store the residual norms of the computed eigenpairs; all processes in parallel run return this local array with the same values.
- **primme** *[*c_ptr*,in]* :: parameters structure created by `primme_params_create()`.

Return **magma_xprimme** *[c_int]* :: error indicator; see *Error Codes*.

The arrays `evals`, `evecs`, and `resNorms` should have the same kind.

Further descriptions of *evals*, *evecs*, and *resNorms* on notes in function `xprimme()`.

2.3.6 xprimme_normal

function xprimme_normal (*evals*, *evecs*, *resNorms*, *primme*)

Solve a normal standard eigenproblem, which may not be Hermitian.

All arrays should be hosted on CPU. The computations are performed on CPU (see `magma_xprimme_normal()` for using GPUs).

Parameters

- **evals** (*) *[complex(kind),out]* :: array at least of size `numEvals` to store the computed eigenvalues; all processes in a parallel run return this local array with the same values.
- **evecs** :: array at least of size `nLocal` times (`numOrthoConst` + `numEvals`) with leading dimension `ldevecs` to store column-wise the (local part for this process of the) computed eigenvectors.

- **resNorms** (*) *[real(kind),out]* :: array at least of size `numEvals` to store the residual norms of the computed eigenpairs; all processes in parallel run return this local array with the same values.
- **primme** *[*c_ptr*,in]* :: parameters structure created by `primme_params_create()`.

Return **xprimme_normal** *[c_int]* :: error indicator; see *Error Codes*.

The arrays `evals`, `evecs`, and `resNorms` should have the same kind.

Further descriptions of *evals*, *evecs*, and *resNorms* on notes in function `xprimme()`.

2.3.7 magma_xprimme_normal

function magma_xprimme_normal (*evals*, *evecs*, *resNorms*, *primme*)

Solve a normal standard eigenproblem, which may not be Hermitian.

Most of the arrays are stored on GPU, and also most of the computations are done on GPU (see `xprimme()` for using only the CPU).

Parameters

- **evals** (*) *[complex(kind),out]* :: CPU array at least of size `numEvals` to store the computed eigenvalues; all processes in a parallel run return this local array with the same values.
- **evecs** :: GPU array at least of size `nLocal` times (`numOrthoConst` + `numEvals`) with leading dimension `ldevecs` to store column-wise the (local part for this process of the) computed eigenvectors.
- **resNorms** (*) *[real(kind),out]* :: CPU array at least of size `numEvals` to store the residual norms of the computed eigenpairs; all processes in parallel run return this local array with the same values.
- **primme** *[*c_ptr*,in]* :: parameters structure created by `primme_params_create()`.

Return **magma_xprimme_normal** *[c_int]* :: error indicator; see *Error Codes*.

The arrays `evals`, `evecs`, and `resNorms` should have the same kind.

Further descriptions of *evals*, *evecs*, and *resNorms* on notes in function `xprimme()`.

2.3.8 primme_set_member

function primme_set_member (*primme*, *label*, *value*)

Set a value in some field of the parameter structure.

Parameters

- **primme** *[*c_ptr*,in]* :: parameters structure created by `primme_params_create()`.
- **label** *[c_int,in]* :: field where to set value. One of:

```
PRIMME_n
PRIMME_matrixMatvec
PRIMME_matrixMatvec_type
PRIMME_applyPreconditioner
PRIMME_applyPreconditioner_type
PRIMME_massMatrixMatvec
```

PRIMME_massMatrixMatvec_type
PRIMME_numProcs
PRIMME_procID
PRIMME_commInfo
PRIMME_nLocal
PRIMME_globalSumReal
PRIMME_globalSumReal_type
PRIMME_broadcastReal
PRIMME_broadcastReal_type
PRIMME_numEvals
PRIMME_target
PRIMME_numTargetShifts
PRIMME_targetShifts
PRIMME_locking
PRIMME_initSize
PRIMME_numOrthoConst
PRIMME_maxBasisSize
PRIMME_minRestartSize
PRIMME_maxBlockSize
PRIMME_maxMatvecs
PRIMME_maxOuterIterations
PRIMME_iseed
PRIMME_aNorm
PRIMME_BNorm
PRIMME_invBNorm
PRIMME_eps
PRIMME_orth
PRIMME_internalPrecision
PRIMME_printLevel
PRIMME_outputFile
PRIMME_matrix
PRIMME_massMatrix
PRIMME_preconditioner
PRIMME_initBasisMode
PRIMME_projectionParams_projection
PRIMME_restartingParams_maxPrevRetain
PRIMME_correctionParams_precondition
PRIMME_correctionParams_robustShifts
PRIMME_correctionParams_maxInnerIterations
PRIMME_correctionParams_projectors_LeftQ
PRIMME_correctionParams_projectors_LeftX
PRIMME_correctionParams_projectors_RightQ
PRIMME_correctionParams_projectors_RightX
PRIMME_correctionParams_projectors_SkewQ
PRIMME_correctionParams_projectors_SkewX
PRIMME_correctionParams_convTest
PRIMME_correctionParams_relTolBase

PRIMME_stats_numOuterIterations
PRIMME_stats_numRestarts
PRIMME_stats_numMatvecs
PRIMME_stats_numPreconds
PRIMME_stats_numGlobalSum
PRIMME_stats_volumeGlobalSum
PRIMME_stats_numBroadcast
PRIMME_stats_volumeBroadcast
PRIMME_stats_flopsDense
PRIMME_stats_numOrthoInnerProds
PRIMME_stats_elapsedTime
PRIMME_stats_timeMatvec
PRIMME_stats_timePrecond
PRIMME_stats_timeOrtho
PRIMME_stats_timeGlobalSum
PRIMME_stats_timeBroadcast
PRIMME_stats_timeDense
PRIMME_stats_estimateMinEVal
PRIMME_stats_estimateMaxEVal
PRIMME_stats_estimateLargestSVal
PRIMME_stats_estimateBNorm
PRIMME_stats_estimateInvBNorm
PRIMME_stats_maxConvTol
PRIMME_stats_lockingIssue
PRIMME_dynamicMethodSwitch
PRIMME_convTestFun
PRIMME_convTestFun_type
PRIMME_convtest
PRIMME_ldevecs
PRIMME_ldOPs
PRIMME_monitorFun
PRIMME_monitorFun_type
PRIMME_monitor
PRIMME_queue

- **value** *[in]* :: value to set. The allowed types are *c_int64*, *c_double*, *c_ptr*, *c_funptr* and *procedure(primme_eigs_matvec)*

Return primme_set_member *[c_int]* :: nonzero value if the call is not successful.

Examples:

```
type(c_ptr) :: primme
integer :: ierr
...

integer(c_int64_t) :: n              = 100
ierr = primme_set_member(primme, PRIMME_n, n)
```

(continues on next page)

(continued from previous page)

```
ierr = primme_set_member(primme, PRIMME_correctionParams_precondition,
                         1_c_int64_t)

real(c_double) :: tol             = 1.0D-12
ierr = primme_set_member(primme, PRIMME_eps, tol)

integer(c_int64_t), parameter :: numTargetShifts = 2
real(c_double) :: TargetShifts(numTargetShifts) = (/3.0D0, 5.1D0/)
ierr = primme_set_member(primme, PRIMME_numTargetShifts,␣
↪numTargetShifts)
ierr = primme_set_member(primme, PRIMME_targetShifts, TargetShifts)

ierr = primme_set_member(primme, PRIMME_target, primme_closest_abs)

procedure(primme_eigs_matvec) :: MV, ApplyPrecon
ierr = primme_set_member(primme, PRIMME_matrixMatvec, MV)

ierr = primme_set_member(primme, PRIMME_applyPreconditioner,
                         c_funloc(ApplyPrecon))
```

2.3.9 primme_get_member

function primme_get_member(*primme*, *label*, *value*)

Get the value in some field of the parameter structure.

Parameters

- **primme** *[c_ptr,in]* :: parameters structure created by `primme_params_create()`.
- **label** *[integer,in]* :: field where to get value. One of the detailed in function `primme_set_member()`.
- **value** *[out]* :: value of the field. The allowed types are *c_int64*, *c_double*, and *c_ptr*.

Return **primme_get_member** *[c_int]* :: nonzero value if the call is not successful.

Examples:

```
type(c_ptr) :: primme
integer :: ierr
...

integer(c_int64_t) :: n
ierr = primme_get_member(primme, PRIMME_n, n)

real(c_double) :: aNorm
ierr = primme_get_member(primme, PRIMME_aNorm, aNorm)

real(c_double), pointer :: shifts(:)
type(c_ptr) :: pshifts
ierr = primme_get_member(primme, PRIMME_ShiftsForPreconditioner, pshifts)
call c_f_pointer(pshifts, shifts, shape=[k])
```

2.4 Python Interface

`primme.eigsh`(*A*, *k=6*, *M=None*, *sigma=None*, *which='LM'*, *v0=None*, *ncv=None*, *maxiter=None*, *tol=0*, *return_eigenvectors=True*, *Minv=None*, *OPinv=None*, *mode='normal'*, *lock=None*, *return_stats=False*, *maxBlockSize=0*, *minRestartSize=0*, *maxPrevRetain=0*, *method=None*, *return_history=False*, *convtest=None*, ***kargs*)

Find k eigenvalues and eigenvectors of the real symmetric square matrix or complex Hermitian matrix A.

Solves `A * x[i] = w[i] * x[i]`, the standard eigenvalue problem for w[i] eigenvalues with corresponding eigenvectors x[i].

If M is specified, solves `A * x[i] = w[i] * M * x[i]`, the generalized eigenvalue problem for w[i] eigenvalues with corresponding eigenvectors x[i]

Parameters

- **A** (*An N x N matrix, array, sparse matrix, or LinearOperator*) – the operation A * x, where A is a real symmetric matrix or complex Hermitian.
- **k** (*int, optional*) – The number of eigenvalues and eigenvectors to be computed. Must be 1 <= k < min(A.shape).
- **M** (*An N x N matrix, array, sparse matrix, or LinearOperator*) – the operation M * x for the generalized eigenvalue problem

  A * x = w * M * x.

  M must represent a real, symmetric matrix if A is real, and must represent a complex, Hermitian matrix if A is complex. For best results, the data type of M should be the same as that of A.
- **sigma** (*real, optional*) – Find eigenvalues near sigma.
- **v0** (*N x i, ndarray, optional*) – Initial guesses to the eigenvectors.
- **ncv** (*int, optional*) – The maximum size of the basis
- **which** (*str ['LM' | 'SM' | 'LA' | 'SA' | number]*) – Which *k* eigenvectors and eigenvalues to find:

  'LM' : Largest in magnitude eigenvalues; the farthest from sigma

  'SM' : Smallest in magnitude eigenvalues; the closest to sigma

  'LA' : Largest algebraic eigenvalues

  'SA' : Smallest algebraic eigenvalues

  'CLT' : closest but left to sigma

  'CGT' : closest but greater than sigma

  number : the closest to which

  When sigma == None, 'LM', 'SM', 'CLT', and 'CGT' treat sigma as zero.
- **maxiter** (*int, optional*) – Maximum number of iterations.
- **tol** (*float*) – Tolerance for eigenpairs (stopping criterion). The default value is sqrt of machine precision.

  An eigenpair `(lamba,v)` is marked as converged when ||A*v - lambda*B*v|| < max(**|eig(A,B)|**)*tol.

  The value is ignored if convtest is provided.

- **Minv** (*(not supported yet)*) – The inverse of M in the generalized eigenproblem.
- **OPinv** (*N x N matrix, array, sparse matrix, or LinearOperator, optional*) – Preconditioner to accelerate the convergence. Usually it is an approximation of the inverse of (A - sigma*M).
- **return_eigenvectors** (*bool, optional*) – Return eigenvectors (True) in addition to eigenvalues
- **mode** (*string ['normal' | 'buckling' | 'cayley']*) – Only 'normal' mode is supported.
- **lock** (*N x i, ndarray, optional*) – Seek the eigenvectors orthogonal to these ones. The provided vectors *should* be orthonormal. Useful to avoid converging to previously computed solutions.
- **maxBlockSize** (*int, optional*) – Maximum number of vectors added at every iteration.
- **minRestartSize** (*int, optional*) – Number of approximate eigenvectors kept during restart.
- **maxPrevRetain** (*int, optional*) – Number of approximate eigenvectors kept from previous iteration in restart. Also referred as +k vectors in GD+k.
- **method** (*int, optional*) – Preset method, one of:
  - DEFAULT_MIN_TIME : a variant of JDQMR,
  - DEFAULT_MIN_MATVECS : GD+k
  - DYNAMIC : choose dynamically between these previous methods.

  See a detailed description of the methods and other possible values in[2].
- **convtest** (*callable*) – User-defined function to mark an approximate eigenpair as converged.

  The function is called as convtest(eval, evec, resNorm) and returns True if the eigenpair with value *eval*, vector *evec* and residual norm *resNorm* is considered converged.
- **return_stats** (*bool, optional*) – If True, the function returns extra information (see stats in Returns).
- **return_history** (*bool, optional*) – If True, the function returns performance information at every iteration (see hist in Returns).

Returns

- **w** (*array*) – Array of k eigenvalues ordered to best satisfy "which".
- **v** (*array*) – An array representing the *k* eigenvectors. The column `v[:, i]` is the eigenvector corresponding to the eigenvalue `w[i]`.
- **stats** (*dict, optional (if return_stats)*) – Extra information reported by PRIMME:
  - "numOuterIterations": number of outer iterations
  - "numRestarts": number of restarts
  - "numMatvecs": number of A*v
  - "numPreconds": number of OPinv*v
  - "elapsedTime": time that took

[2] Preset Methods, http://www.cs.wm.edu/~andreas/software/doc/readme.html#preset-methods

  - "estimateMinEVal": the leftmost Ritz value seen
  - "estimateMaxEVal": the rightmost Ritz value seen
  - "estimateLargestSVal": the largest singular value seen
  - "rnorms" : ||A*x[i] - x[i]*w[i]||
  - "hist" : (if return_history) report at every outer iteration of:
    * "elapsedTime": time spent up to now
    * "numMatvecs": number of A*v spent up to now
    * "nconv": number of converged pair
    * "eval": eigenvalue of the first unconverged pair
    * "resNorm": residual norm of the first unconverged pair

Raises `PrimmeError` – When the requested convergence is not obtained.

The PRIMME error code can be found as `err` attribute of the exception object.

See also:

`scipy.sparse.linalg.eigs()` eigenvalues and eigenvectors for a general (nonsymmetric) matrix A

`primme.svds()` singular value decomposition for a matrix A

Notes

This function is a wrapper to PRIMME functions to find the eigenvalues and eigenvectors[1].

References

Examples

```
>>> import primme, scipy.sparse
>>> A = scipy.sparse.spdiags(range(100), [0], 100, 100) # sparse diag. matrix
>>> evals, evecs = primme.eigsh(A, 3, tol=1e-6, which='LA')
>>> evals # the three largest eigenvalues of A
array([99., 98., 97.])
>>> new_evals, new_evecs = primme.eigsh(A, 3, tol=1e-6, which='LA', lock=evecs)
>>> new_evals # the next three largest eigenvalues
array([96., 95., 94.])
>>> evals, evecs = primme.eigsh(A, 3, tol=1e-6, which=50.1)
>>> evals # the three closest eigenvalues to 50.1
array([50.,  51.,  49.])
>>> M = scipy.sparse.spdiags(np.asarray(range(99,-1,-1)), [0], 100, 100)
>>> # the smallest eigenvalues of the eigenproblem (A,M)
>>> evals, evecs = primme.eigsh(A, 3, M=M, tol=1e-6, which='SA')
>>> evals
array([1.0035e-07, 1.0204e-02, 2.0618e-02])
```

[1] PRIMME Software, https://github.com/primme/primme

```
>>> # Giving the matvec as a function
>>> import primme, scipy.sparse, numpy as np
>>> Adiag = np.arange(0, 100).reshape((100,1))
>>> def Amatmat(x):
...     if len(x.shape) == 1: x = x.reshape((100,1))
...     return Adiag * x   # equivalent to diag(Adiag).dot(x)
...
>>> A = scipy.sparse.linalg.LinearOperator((100,100), matvec=Amatmat,␣
↪matmat=Amatmat)
>>> evals, evecs = primme.eigsh(A, 3, tol=1e-6, which='LA')
>>> evals
array([99., 98., 97.])
```

2.5 MATLAB Interface

function [varargout] = primme_eigs(varargin)

`primme_eigs()` finds a few eigenvalues and their corresponding eigenvectors of a symmetric/Hermitian matrix, `A`, or of a generalized problem `(A,B)`, by calling PRIMME.

`D = primme_eigs(A)` returns a vector of `A`'s 6 largest magnitude eigenvalues.

`D = primme_eigs(A,B)` returns a vector of the 6 largest magnitude eigenvalues of `(A,B)`

`D = primme_eigs(Afun,Bfun,dim)` `D = primme_eigs(Afun,dim)` accepts a functions `Afun` and `Bfun` instead of matrices. `Afun` and `Bfun` are function handles. `Afun(x)` and `Bfun(x)` return the matrix-vector product `A*x` and `B*x`.

`D = primme_eigs(...,k)` finds the `k` largest magnitude eigenvalues. `k` must be less than the dimension of the matrix `A`.

`D = primme_eigs(...,k,target)` returns `k` eigenvalues such that: If `target` is a real number, it finds the closest eigenvalues to `target`. If `target` is

- `'LA'` or `'SA'`, eigenvalues with the largest or smallest algebraic value.
- `'LM'` or `'SM'`, eigenvalues with the largest or smallest magnitude if `OPTS.targetShifts` is empty. If `target` is a real or complex scalar including 0, `primme_eigs()` finds the eigenvalues closest to `target`.

  In addition, if some values are provided in `OPTS.targetShifts`, it finds eigenvalues that are farthest (`'LM'`) or closest (`'SM'`) in absolute value from the given values.

  Examples:

  `k=1, 'LM', OPTS.targetShifts=[]` returns the largest magnitude `eig(A)`. `k=1, 'SM', OPTS.targetShifts=[]` returns the smallest magnitude `eig(A)`. `k=3, 'SM', OPTS.targetShifts=[2, 5]` returns the closest eigenvalue in absolute sense to 2, and the two closest eigenvalues to 5.
- `'CLT'` or `'CGT'`, find eigenvalues closest to but less or greater than the given values in `OPTS.targetShifts`.

`D = primme_eigs(...,k,target,OPTS)` specifies extra solver parameters. Some default values are indicated in brackets {}:

- *`aNorm`*: the estimated 2-norm of A {0.0 (estimate the norm internally)}
- `tol`: convergence tolerance: `NORM(A*X(:,i)-X(:,i)*D(i,i)) < tol*NORM(A)` (see *`eps`*) {10^4 times the machine precision}
- *`maxBlockSize`*: maximum block size (useful for high multiplicities) {1}
- `reportLevel`: reporting level (0-3) (see HIST) {no reporting 0}
- `display`: whether displaying reporting on screen (see HIST) {0 if HIST provided}
- `isreal`: whether A represented by `Afun` is real or complex {false}
- `isdouble`: whether the class of in/out vectors in `Afun` are double or single {false}
- `isgpu`: whether the class of in/out vectors in `Afun` are `gpuArray` {false}
- `ishermitian`: whether `A` is Hermitian; otherwise it is considered normal {true}
- *`targetShifts`*: shifts for interior eigenvalues (see `target`) {[]}
- `v0`: any number of initial guesses to the eigenvectors (see *`initSize`* {[]}

- orthoConst: external orthogonalization constraints (see *numOrthoConst* {[]}
- *locking*: 1, hard locking; 0, soft locking
- p: maximum size of the search subspace (see *maxBasisSize*)
- *minRestartSize*: minimum Ritz vectors to keep in restarting
- *maxMatvecs*: maximum number of matrix vector multiplications {Inf}
- maxit: maximum number of outer iterations (see *maxOuterIterations*) {Inf}
- *maxPrevRetain*: number of Ritz vectors from previous iteration that are kept after restart {typically >0}
- *robustShifts*: setting to true may avoid stagnation or misconvergence
- *maxInnerIterations*: maximum number of inner solver iterations
- *LeftQ*: use the locked vectors in the left projector
- *LeftX*: use the approx. eigenvector in the left projector
- *RightQ*: use the locked vectors in the right projector
- *RightX*: use the approx. eigenvector in the right projector
- *SkewQ*: use the preconditioned locked vectors in the right projector
- *SkewX*: use the preconditioned approx. eigenvector in the right projector
- *relTolBase*: a legacy from classical JDQR (not recommended)
- *convTest*: how to stop the inner QMR Method
- *convTestFun*: function handler with an alternative convergence criterion. If FUN(EVAL,EVEC, RNORM) returns a nonzero value, the pair (EVAL,EVEC) with residual norm RNORM is considered converged
- *iseed*: random seed
- returnUnconverged: whether to return unconverged pairs if maximum iterations or matvecs is reached.

D = primme_eigs(A,k,target,OPTS,METHOD) specifies the eigensolver method. METHOD can be one of the next strings:

- '*PRIMME_DYNAMIC*', (default) switches dynamically to the best method
- '*PRIMME_DEFAULT_MIN_TIME*', best method for low-cost matrix-vector product
- '*PRIMME_DEFAULT_MIN_MATVECS*', best method for heavy matvec/preconditioner
- '*PRIMME_Arnoldi*', Arnoldi not implemented efficiently
- '*PRIMME_GD*', classical block Generalized Davidson
- '*PRIMME_GD_plusK*', GD+k block GD with recurrence restarting
- '*PRIMME_GD_Olsen_plusK*', GD+k with approximate Olsen precond.
- '*PRIMME_JD_Olsen_plusK*', GD+k, exact Olsen (two precond per step)
- '*PRIMME_RQI*', Rayleigh Quotient Iteration. Also INVIT, but for INVIT provide OPTS.targetShifts
- '*PRIMME_JDQR*', Original block, Jacobi Davidson
- '*PRIMME_JDQMR*', Our block JDQMR method (similar to JDCG)
- '*PRIMME_JDQMR_ETol*', Slight, but efficient JDQMR modification

- '*PRIMME_STEEPEST_DESCENT*', equivalent to GD(block,2*block)
- '*PRIMME_LOBPCG_OrthoBasis*', equivalent to GD(nev,3*nev)+nev
- '*PRIMME_LOBPCG_OrthoBasis_Window*' equivalent to GD(block,3*block)+block nev>block

`D = primme_eigs(A,k,target,OPTS,METHOD,P)`

`D = primme_eigs(A,k,target,OPTS,METHOD,P1,P2)` uses preconditioner `P` or `P = P1*P2` to accelerate convergence of the method. Applying `P\x` should approximate `(A-sigma*eye(N))\x`, for `sigma` near the wanted eigenvalue(s). If `P` is `[]` then a preconditioner is not applied. `P` may be a function handle `PFUN` such that `PFUN(x)` returns `P\x`.

`[X,D] = primme_eigs(...)` returns a diagonal matrix `D` with the eigenvalues and a matrix `X` whose columns are the corresponding eigenvectors.

`[X,D,R] = primme_eigs(...)` also returns an array of the residual norms of the computed eigenpairs.

`[X,D,R,STATS] = primme_eigs(...)` returns a `struct` to report statistical information about number of matvecs, elapsed time, and estimates for the largest and smallest algebraic eigenvalues of `A`.

`[X,D,R,STATS,HIST] = primme_eigs(...)` it returns the convergence history, instead of printing it. Every row is a record, and the columns report:

- `HIST(:,1)`: number of matvecs
- `HIST(:,2)`: time
- `HIST(:,3)`: number of converged/locked pairs
- `HIST(:,4)`: block index
- `HIST(:,5)`: approximate eigenvalue
- `HIST(:,6)`: residual norm
- `HIST(:,7)`: QMR residual norm

`OPTS.reportLevel` controls the granularity of the record. If `OPTS.reportLevel == 1`, `HIST` has one row per converged eigenpair and only the first three columns together with the fifth and the sixth are reported. If `OPTS.reportLevel == 2`, `HIST` has one row per outer iteration and converged value, and only the first six columns are reported. Otherwise `HIST` has one row per QMR iteration, outer iteration and converged value, and all columns are reported.

The convergence history is displayed if `OPTS.reportLevel > 0` and either `HIST` is not returned or `OPTS.display == 1`.

Examples:

```
A = diag(1:100);

d = primme_eigs(A,10) % the 10 largest magnitude eigenvalues

d = primme_eigs(A,10,'SM') % the 10 smallest magnitude eigenvalues

d = primme_eigs(A,10,25.0) % the 10 closest eigenvalues to 25.0

opts.targetShifts = [2 20];
d = primme_eigs(A,10,'SM',opts) % 1 eigenvalue closest to 2 and
                                % 9 eigenvalues closest to 20

B = diag(100:-1:1);
d = primme_eigs(A,B,10,'SM') % the 10 smallest magnitude eigenvalues
```

(continues on next page)

(continued from previous page)

```
opts = struct();
opts.tol = 1e-4; % set tolerance
opts.maxBlockSize = 2; % set block size
[x,d] = primme_eigs(A,10,'SA',opts,'DEFAULT_MIN_TIME')

opts.orthoConst = x;
[d,rnorms] = primme_eigs(A,10,'SA',opts) % find another 10 with the default method

% Compute the 6 eigenvalues closest to 30.5 using ILU(0) as a preconditioner
% by passing the matrices L and U.
A = sparse(diag(1:50) + diag(ones(49,1), 1) + diag(ones(49,1), -1));
[L,U] = ilu(A, struct('type', 'nofill'));
d = primme_eigs(A, k, 30.5, [], [], L, U);

% Compute the 6 eigenvalues closest to 30.5 using Jacobi preconditioner
% by passing a function.
Pfun = @(x)(diag(A) - 30.5)\x;
d = primme_eigs(A,6,30.5,[],[],Pfun) % find the closest 5 to 30.5
```

See also: MATLAB eigs, `primme_svds()`

2.6 Parameters Description

2.6.1 Types

The following data types are macros used in PRIMME as followed.

type PRIMME_INT

Integer type used in matrix dimensions (such as *n* and *nLocal*) and counters (such as *numMatvecs*).

The integer size is controlled by the compilation flag `PRIMME_INT_SIZE`, see *Making and Linking*.

type PRIMME_HALF

Macro that is `__fp16` if half precision is supported by the compiler. Otherwise it is a struct with the same size as `int short`.

New in version 3.0.

type PRIMME_COMPLEX_HALF

Macro that is a struct with fields `r` and `i` with type `PRIMME_HALF`.

New in version 3.0.

type PRIMME_COMPLEX_FLOAT

Macro that is `complex float` in C and `std::complex<float>` in C++.

New in version 2.0.

type PRIMME_COMPLEX_DOUBLE

Macro that is `complex double` in C and `std::complex<double>` in C++.

New in version 2.0.

2.6.2 Other macros

PRIMME_VERSION_MAJOR

Constant `int` with the major version number.

For instance, the value of the macro is `3` for version 3.0.

New in version 3.0.

PRIMME_VERSION_MINOR

Constant `int` with the minor version number.

For instance, the value of the macro is `0` for version 3.0.

New in version 3.0.

2.6.3 primme_params

type primme_params

Structure to set the problem matrices and eigensolver options.

PRIMME_INT **n**

Dimension of the matrix.

Input/output:

primme_initialize() sets this field to 0;
this field is read by *dprimme()*.

void (***matrixMatvec**)(void *x, *PRIMME_INT* *ldx, void *y, *PRIMME_INT* *ldy, int *blockSize, *primme_params* *primme, int *ierr)

Block matrix-multivector multiplication, $y = Ax$ in solving $Ax = \lambda x$ or $Ax = \lambda Bx$.

Parameters

- **x** – matrix of size `nLocal` x `blockSize` in column-major order with leading dimension `ldx`.
- **ldx** – the leading dimension of the array `x`.
- **y** – matrix of size `nLocal` x `blockSize` in column-major order with leading dimension `ldy`.
- **ldy** – the leading dimension of the array `y`.
- **blockSize** – number of columns in `x` and `y`.
- **primme** – parameters structure.
- **ierr** – output error code; if it is set to non-zero, the current call to PRIMME will stop.

The actual type of `x` and `y` matches the type of `evecs` of the calling `dprimme()` (or a variant), unless the user sets `matrixMatvec_type` to another precision.

Input/output:

`primme_initialize()` sets this field to NULL;
this field is read by `dprimme()`.

Note: If you have performance issues with leading dimension different from `nLocal`, set `ldOPs` to `nLocal`.

Changed in version 2.0.

primme_op_datatype **matrixMatvec_type**

Precision of the vectors `x` and `y` passed to `matrixMatvec`.

If it is `primme_op_default`, the vectors' type matches the calling `dprimme()` (or a variant). Otherwise, the user can force the precision of the vectors `x` and `y` to be a particular precision regardless of the calling `dprimme()` (or a variant) function: half, single, or double, if `matrixMatvec_type` is `primme_half`, `primme_float` or `primme_double` respectively.

It is not recommended to set a lower precision than the one required to converge. An example of this is calling `dprimme()` setting `eps` to 1e-10 and `matrixMatvec_type` to `primme_op_half`.

Input/output:

`primme_initialize()` sets this field to `primme_op_default`;
this field is read by `dprimme()`, and if it is `primme_op_default` it is set to the value that matches the precision of calling function.

New in version 3.0.

void (***applyPreconditioner**)(void *x, *PRIMME_INT* *ldx, void *y, *PRIMME_INT* *ldy, int *blockSize, *primme_params* *primme, int *ierr)

Block preconditioner-multivector application, $y = M^{-1}x$ where M is usually an approximation of $A - \sigma I$ or $A - \sigma B$ for finding eigenvalues close to σ.

Parameters

- **x** – matrix of size `nLocal` x `blockSize` in column-major order with leading dimension `ldx`.

- **ldx** – the leading dimension of the array x.
- **y** – matrix of size *nLocal* x blockSize in column-major order with leading dimension ldy.
- **ldy** – the leading dimension of the array y.
- **blockSize** – number of columns in x and y.
- **primme** – parameters structure.
- **ierr** – output error code; if it is set to non-zero, the current call to PRIMME will stop.

The actual type of x and y matches the type of evecs of the calling *dprimme()* (or a variant), unless the user sets *applyPreconditioner_type* to another precision.

Input/output:

primme_initialize() sets this field to NULL;
this field is read by *dprimme()*.

Changed in version 2.0.

primme_op_datatype **applyPreconditioner_type**

Precision of the vectors x and y passed to *applyPreconditioner*.

If it is primme_op_default, the vectors' type matches the calling *dprimme()* (or a variant). Otherwise, the user can force the precision of the vectors x and y to be a particular precision regardless of the calling *dprimme()* (or a variant) function: half, single, or double, if *matrixMatvec_type* is primme_half, primme_float or primme_double respectively.

Input/output:

primme_initialize() sets this field to primme_op_default;
this field is read by *dprimme()*, and if it is primme_op_default it is set to the value that matches the precision of calling function.

New in version 3.0.

void (***massMatrixMatvec**)(void *x, *PRIMME_INT* *ldx, void *y, *PRIMME_INT* *ldy, int *blockSize, *primme_params* *primme, int *ierr)

Block matrix-multivector multiplication, $y = Bx$ in solving $Ax = \lambda Bx$. If it is NULL, the standard eigenvalue problem $Ax = \lambda x$ is solved.

Parameters

- **x** – matrix of size *nLocal* x blockSize in column-major order with leading dimension ldx.
- **ldx** – the leading dimension of the array x.
- **y** – matrix of size *nLocal* x blockSize in column-major order with leading dimension ldy.
- **ldy** – the leading dimension of the array y.
- **blockSize** – number of columns in x and y.
- **primme** – parameters structure.
- **ierr** – output error code; if it is set to non-zero, the current call to PRIMME will stop.

The actual type of x and y matches the type of evecs of the calling *dprimme()* (or a variant), unless the user sets *matrixMatvec_type* to another precision.

Input/output:

primme_initialize() sets this field to NULL;

this field is read by *dprimme()*.

Changed in version 2.0.

primme_op_datatype **`massMatrixMatvec_type`**

Precision of the vectors `x` and `y` passed to *massMatrixMatvec*.

If it is `primme_op_default`, the vectors' type matches the calling *dprimme()* (or a variant). Otherwise, the user can force the precision of the vectors `x` and `y` to be a particular precision regardless of the calling *dprimme()* (or a variant) function: half, single, or double, if *massMatrixMatvec_type* is `primme_half`, `primme_float` or `primme_double` respectively.

It is not recommended to set a lower precision than the one required to converge. An example of this is calling *dprimme()* setting *eps* to 1e-10 and *massMatrixMatvec_type* to `primme_op_half`.

Input/output:

primme_initialize() sets this field to `primme_op_default`;

this field is read by *dprimme()*, and if it is `primme_op_default` it is set to the value that matches the precision of calling function.

New in version 3.0.

int **`numProcs`**

Number of processes calling *dprimme()* or *zprimme()* in parallel.

Input/output:

primme_initialize() sets this field to 1;

this field is read by *dprimme()*.

int **`procID`**

The identity of the local process within a parallel execution calling *dprimme()* or *zprimme()*. Only the process with id 0 prints information.

Input/output:

primme_initialize() sets this field to 0;

dprimme() sets this field to 0 if *numProcs* is 1;

this field is read by *dprimme()*.

int **`nLocal`**

Number of local rows on this process.

Input/output:

primme_initialize() sets this field to -1;

dprimme() sets this field to *n* if *numProcs* is 1;

this field is read by *dprimme()*.

void ***`commInfo`**

A pointer to whatever parallel environment structures needed. For example, with MPI, it could be a pointer to the MPI communicator. PRIMME does not use this. It is available for possible use in user functions defined in *matrixMatvec*, *applyPreconditioner*, *massMatrixMatvec*, *globalSumReal*, and *broadcastReal*.

Input/output:

primme_initialize() sets this field to NULL;

void (***globalSumReal**)(void *sendBuf, void *recvBuf, int *count, *primme_params* *primme, int *ierr)

Global sum reduction function. No need to set for sequential programs.

Parameters

- **sendBuf** – array of size `count` with the local input values.
- **recvBuf** – array of size `count` with the global output values so that the i-th element of recvBuf is the sum over all processes of the i-th element of `sendBuf`.
- **count** – array size of `sendBuf` and `recvBuf`.
- **primme** – parameters structure.
- **ierr** – output error code; if it is set to non-zero, the current call to PRIMME will stop.

The actual type of `sendBuf` and `recvBuf` matches the type of `evecs` of the calling `dprimme()` (or a variant), unless the user sets `globalSumReal_type` to another precision. See the recomendation about precision in `globalSumReal_type`.

Note that `count` is the number of values of the real type.

Input/output:

`primme_initialize()` sets this field to an internal function;

`dprimme()` sets this field to an internal function if `numProcs` is 1 and `globalSumReal` is NULL;

this field is read by `dprimme()`.

When MPI is used, this can be a simply wrapper to MPI_Allreduce() as shown below:

```
void par_GlobalSumForDouble(void *sendBuf, void *recvBuf, int *count,
                         primme_params *primme, int *ierr) {
   MPI_Comm communicator = *(MPI_Comm *) primme->commInfo;
   if (sendBuf == recvBuf) {
     *ierr = MPI_Allreduce(MPI_IN_PLACE, recvBuf, *count, MPIU_REAL, MPI_SUM,␣
↪communicator) != MPI_SUCCESS;
   } else {
     *ierr = MPI_Allreduce(sendBuf, recvBuf, *count, MPIU_REAL, MPI_SUM,␣
↪communicator) != MPI_SUCCESS;
   }
}
```

When calling `sprimme()` and `cprimme()` replace `MPI_DOUBLE` by ``MPI_FLOAT`.

Changed in version 2.0.

primme_op_datatype **globalSumReal_type**

Precision of the vectors `sendBuf` and `recvBuf` passed to `globalSumReal`.

If it is `primme_op_default`, the vectors' type matches the calling `dprimme()` (or a variant). Otherwise, the user can force the precision of the vectors `sendBuf` and `recvBuf` to be a particular precision regardless of the calling `dprimme()` (or a variant) function: half, single, or double, if `globalSumReal_type` is `primme_half`, `primme_float` or `primme_double` respectively.

It is recommended to set a precision so that the machine precision times log2(`numProcs`) is smaller than the precision required to converge. An example of this is calling `hprimme()` setting `eps` to 0.01 and `globalSumReal_type` to `primme_op_single` for 1000 processes.

Input/output:

`primme_initialize()` sets this field to `primme_op_default`;

this field is read by `dprimme()`, and if it is `primme_op_default` it is set to the value that matches the precision of calling function.

New in version 3.0.

void (***broadcastReal**)(void *buffer, int *count, *primme_params* *primme, int *ierr)
Broadcast function from process with ID zero. It is optional in parallel executions, and not needed for sequential programs.

Parameters

- **buffer** – array of size `count` with the local input values.
- **count** – array size of `sendBuf` and `recvBuf`.
- **primme** – parameters structure.
- **ierr** – output error code; if it is set to non-zero, the current call to PRIMME will stop.

The actual type of `buffer` matches the type of `evecs` of the calling `dprimme()` (or a variant), unless the user sets `broadcastReal_type` to another precision.

If `broadcastReal` is not provided, PRIMME uses `globalSumReal` for broadcasting, which is usually a bit more expensive.

Input/output:

`primme_initialize()` sets this field to NULL;
this field is read by `dprimme()`.

When MPI is used, this can be a simply wrapper to MPI_Bcast() as shown below:

```
void broadcastForDouble(void *buffer, int *count,
                        primme_params *primme, int *ierr) {
   MPI_Comm communicator = *(MPI_Comm *) primme->commInfo;
   if(MPI_Bcast(buffer, *count, MPI_DOUBLE, 0 /* root */,
                communicator) == MPI_SUCCESS) {
      *ierr = 0;
   } else {
      *ierr = 1;
   }
}
```

When calling `sprimme()` and `cprimme()` replace `MPI_DOUBLE` by ``MPI_FLOAT`.

New in version 3.0.

primme_op_datatype **broadcastReal_type**
Precision of the vector `buffer`` passed to `broadcastReal`.

If it is `primme_op_default`, the vectors' type matches the calling `dprimme()` (or a variant). Otherwise, the user can force the precision of the vectors `x` and `y` to be a particular precision regardless of the calling `dprimme()` (or a variant) function: half, single, or double, if `matrixMatvec_type` is `primme_half`, `primme_float` or `primme_double` respectively.

It is not recommended to set a lower precision than the one required to converge. An example of this is calling `dprimme()` setting `eps` to 1e-10 and `broadcastReal_type` to `primme_op_half`.

Input/output:

`primme_initialize()` sets this field to `primme_op_default`;
this field is read by `dprimme()`, and if it is `primme_op_default` it is set to the value that matches the precision of calling function.

New in version 3.0.

primme_op_datatype **internalPrecision**

Internal working precision.

If it is `primme_op_default`, most of the vectors are stored with the same precision as the calling *dprimme()* (or a variant), and most of the computations are done in that precision too. Otherwise, the working precision is changed to half, single, or double, if the user sets *internalPrecision* to `primme_half`, `primme_float` or `primme_double` respectively.

Input/output:

primme_initialize() sets this field to `primme_op_default`;
this field is read by *dprimme()*.

New in version 3.0.

int **numEvals**

Number of eigenvalues wanted.

Input/output:

primme_initialize() sets this field to 1;
this field is read by *primme_set_method()* (see *Preset Methods*) and *dprimme()*.

primme_target **target**

Which eigenpairs to find:

primme_smallest Smallest algebraic eigenvalues; *targetShifts* is ignored.

primme_largest Largest algebraic eigenvalues; *targetShifts* is ignored.

primme_closest_geq Closest to, but greater or equal than the shifts in *targetShifts*.

primme_closest_leq Closest to, but less or equal than the shifts in *targetShifts*.

primme_closest_abs Closest in absolute value to the shifts in *targetShifts*.

primme_largest_abs Furthest in absolute value to the shifts in *targetShifts*.

Input/output:

primme_initialize() sets this field to *primme_smallest*;
this field is read by *dprimme()*.

int **numTargetShifts**

Size of the array *targetShifts*. Used only when *target* is *primme_closest_geq*, *primme_closest_leq*, *primme_closest_abs* or *primme_largest_abs*. The default values is 0.

Input/output:

primme_initialize() sets this field to 0;
this field is read by *dprimme()*.

double ***targetShifts**

Array of shifts, at least of size *numTargetShifts*. Used only when *target* is *primme_closest_geq*, *primme_closest_leq*, *primme_closest_abs* or *primme_largest_abs*.

Eigenvalues are computed in order so that the i-th eigenvalue is the closest (or closest but left or closest but right, see *target*) to the i-th shift. If *numTargetShifts* < *numEvals*, the last shift given is used for all the remaining i's.

Input/output:

primme_initialize() sets this field to NULL;
this field is read by *dprimme()*.

Note: Considerations for interior problems:

- PRIMME will try to compute the eigenvalues in the order given in the *targetShifts*. However, for code efficiency and robustness, the shifts should be ordered. Order them in ascending (descending) order for shifts closer to the lower (higher) end of the spectrum.
- If some shift is close to the lower (higher) end of the spectrum, use either *primme_closest_geq* (*primme_closest_leq*) or *primme_closest_abs*.
- *primme_closest_leq* and *primme_closest_geq* are more efficient than *primme_closest_abs*.
- For interior eigenvalues larger *maxBasisSize* is usually more robust.
- To find the largest magnitude eigenvalues set *target* to *primme_largest_abs*, *numTargetShifts* to 1 and *targetShifts* to an array with a zero value.

int **printLevel**

The level of message reporting from the code. All output is written in *outputFile*.

One of:

- 0: silent.
- 1: print some error messages when these occur.
- 2: as in 1, and info about targeted eigenpairs when they are marked as converged:

  ```
  #Converged $1 eval[ $2 ]= $3 norm $4 Mvecs $5 Time $7
  ```

  or locked:

  ```
  #Lock epair[ $1 ]= $3 norm $4 Mvecs $5 Time $7
  ```

- 3: in as 2, and info about targeted eigenpairs every outer iteration:

  ```
  OUT $6 conv $1 blk $8 MV $5 Sec $7 EV $3 |r| $4
  ```

  Also, if it is used the dynamic method, show JDQMR/GDk performance ratio and the current method in use.

- 4: in as 3, and info about targeted eigenpairs every inner iteration:

  ```
  INN MV $5 Sec $7 Eval $3 Lin|r| $9 EV|r| $4
  ```

- 5: in as 4, and verbose info about certain choices of the algorithm.

Output key:

$1: Number of converged pairs up to now.
$2: The index of the pair currently converged.
$3: The eigenvalue.
$4: Its residual norm.
$5: The current number of matrix-vector products.
$6: The current number of outer iterations.

$7: The current elapsed time.

$8: Index within the block of the targeted pair .

$9: QMR norm of the linear system residual.

In parallel programs, output is produced in call with `procID` 0 when `printLevel` is from 0 to 4. If `printLevel` is 5 output can be produced in any of the parallel calls.

Input/output:

> `primme_initialize()` sets this field to 1;
> this field is read by `dprimme()`.

Note: Convergence history for plotting may be produced simply by:

```
grep OUT outputfile | awk '{print $8" "$14}' > out
grep INN outputfile | awk '{print $3" "$11}' > inn
```

Then in gnuplot:

```
plot 'out' w lp, 'inn' w lp
```

double **aNorm**

An estimate of the norm of A, which is used in the default convergence criterion (see `eps`).

If `aNorm` is less than or equal to 0, the code uses the largest absolute Ritz value seen divided by `invBNorm`. On return, `aNorm` is then replaced with that value.

Input/output:

> `primme_initialize()` sets this field to 0.0;
> this field is read and written by `dprimme()`.

double **BNorm**

An estimate of the norm of B, which is used to estimate the conditioning number of the matrix B.

If `BNorm` is less than or equal to 0, the code uses the largest inner-product with B seen. On return, `BNorm` is then replaced with that value.

Input/output:

> `primme_initialize()` sets this field to 0.0;
> this field is read and written by `dprimme()`.

New in version 3.0.

double **invBNorm**

An estimate of the norm of the inverse of B, which is used in the default convergence criterion (see `eps`).

If `invBNorm` is less than or equal to 0, the code uses the inverse of the smallest inner-product with B seen. On return, `invBNorm` is then replaced with that value.

Input/output:

> `primme_initialize()` sets this field to 0.0;
> this field is read and written by `dprimme()`.

New in version 3.0.

primme_orth **orth**

Selects the orthogonalization method used by PRIMME.

If the value is `primme_orth_implicit_I`, the bases are orthogonalized with classical Gram-Schmidt with reorthogonalization stopping when the new vector's norm is not reduced more than $1/sqrt2$ (Daniel's test) from the previous iteration. If several vectors are going to be orthogonalized, the algorithm is applied vector by vector.

If the value is `primme_orth_explicit_I`, the bases are orthogonalized with iterative Cholesky QR (or SVQB if Cholesky factorization fails), stopping when the conditioning of the basis is around $sqrt(3)$. That deviation of the orthogonality level is taken into account in the Galerkin method.

The option `primme_orth_explicit_I` is usually more expensive in FLOPS, but it may be faster in time than `primme_orth_implicit_I` when *maxBlockSize* is large.

`primme_orth_implicit_I` is set by default if the precision is higher than single precision and *maxBlockSize* is 1. Otherwise, `primme_orth_explicit_I` is set by default.

Input/output:

primme_initialize() sets this field to `primme_orth_default`;
this field is read and written by *dprimme()*.

New in version 3.0.

double **eps**

If *convTestFun* is NULL, an eigenpairs is marked as converged when the 2-norm of the residual vector is less than *eps* * *aNorm* * *invBNorm*. The residual vector is $Ax - \lambda x$ or $Ax - \lambda Bx$.

The default value is machine precision times 10^4.

Input/output:

primme_initialize() sets this field to 0.0;
this field is read and written by *dprimme()*.

FILE ***outputFile**

Opened file to write down the output.

Input/output:

primme_initialize() sets this field to the standard output;
this field is read by *dprimme()* and *primme_display_params()*.

int **dynamicMethodSwitch**

If this value is 1, it alternates dynamically between *PRIMME_DEFAULT_MIN_TIME* and *PRIMME_DEFAULT_MIN_MATVECS*, trying to identify the fastest method.

On exit, it holds a recommended method for future runs on this problem:

-1: use *PRIMME_DEFAULT_MIN_MATVECS* next time.
-2: use *PRIMME_DEFAULT_MIN_TIME* next time.
-3: close call, use *PRIMME_DYNAMIC* next time again.

Input/output:

primme_initialize() sets this field to 0;
written by *primme_set_method()* (see *Preset Methods*);
this field is read by *dprimme()*.

Note: Even for expert users we do not recommend setting *dynamicMethodSwitch* directly, but through *primme_set_method()*.

Note: The code obtains timings by the `gettimeofday` Unix utility. If a cheaper, more accurate timer is available, modify the `PRIMMESRC/COMMONSRC/wtime.c`

int **locking**

If set to 1, hard locking will be used (locking converged eigenvectors out of the search basis). If set to 0, the code will try to use soft locking (à la ARPACK), when large enough *minRestartSize* is available.

Input/output:

primme_initialize() sets this field to -1;
written by *primme_set_method()* (see *Preset Methods*);
this field is read by *dprimme()*.

int **initSize**

On input, the number of initial vector guesses provided in `evecs` argument in *dprimme()* or *zprimme()*.

On output, *initSize* holds the number of converged eigenpairs. Without *locking* all *numEvals* approximations are in `evecs` but only the *initSize* ones are converged.

During execution, it holds the current number of converged eigenpairs. In addition, if locking is used, these are accessible in `evals` and `evecs`.

Input/output:

primme_initialize() sets this field to 0;
this field is read and written by *dprimme()*.

PRIMME_INT **ldevecs**

The leading dimension of `evecs`. The default is *nLocal*.

Input/output:

primme_initialize() sets this field to -1;
this field is read by *dprimme()*.

New in version 2.0.

int **numOrthoConst**

Number of vectors to be used as external orthogonalization constraints. These vectors are provided in the first *numOrthoConst* positions of the `evecs` argument in *dprimme()* or *zprimme()* and must be orthonormal.

PRIMME finds new eigenvectors orthogonal to these constraints (equivalent to solving the problem with $(I - YY^*)A(I - YY^*)$ and $(I - YY^*)B(I - YY^*)$ matrices where Y are the given constraint vectors). This is a handy feature if some eigenvectors are already known, or for finding more eigenvalues after a call to *dprimme()* or *zprimme()*, possibly with different parameters (see an example in `TEST/ex_zseq.c`).

Input/output:

primme_initialize() sets this field to 0;
this field is read by *dprimme()*.

int **`maxBasisSize`**

The maximum basis size allowed in the main iteration. This has memory implications.

Input/output:

`primme_initialize()` sets this field to 0;
this field is read and written by *`primme_set_method()`* (see *Preset Methods*);
this field is read by *`dprimme()`*.

int **`minRestartSize`**

Maximum Ritz vectors kept after restarting the basis.

Input/output:

`primme_initialize()` sets this field to 0;
this field is read and written by *`primme_set_method()`* (see *Preset Methods*);
this field is read by *`dprimme()`*.

int **`maxBlockSize`**

The maximum block size the code will try to use.

The user should set this based on the architecture specifics of the target computer, as well as any a priori knowledge of multiplicities. The code does *not* require that *`maxBlockSize`* > 1 to find multiple eigenvalues. For some methods, keeping to 1 yields the best overall performance.

Input/output:

`primme_initialize()` sets this field to 1;
this field is read and written by *`primme_set_method()`* (see *Preset Methods*);
this field is read by *`dprimme()`*.

Note: Inner iterations of QMR are not performed in a block fashion. Every correction equation from a block is solved independently.

PRIMME_INT **`maxMatvecs`**

Maximum number of matrix vector multiplications (approximately equal to the number of preconditioning operations) that the code is allowed to perform before it exits.

Input/output:

`primme_initialize()` sets this field to `INT_MAX`;
this field is read by *`dprimme()`*.

PRIMME_INT **`maxOuterIterations`**

Maximum number of outer iterations that the code is allowed to perform before it exits.

Input/output:

`primme_initialize()` sets this field to `INT_MAX`;
this field is read by *`dprimme()`*.

PRIMME_INT **`iseed`**

The `PRIMME_INT iseed[4]` is an array with the seeds needed by the LAPACK dlarnv and zlarnv.

The default value is an array with values -1, -1, -1 and -1. In that case, `iseed` is set based on the value of *`procID`* to avoid every parallel process generating the same sequence of pseudorandom numbers.

Input/output:

`primme_initialize()` sets this field to `[-1, -1, -1, -1]`;
this field is read and written by *`dprimme()`*.

void ***matrix**

This field may be used to pass any required information in the matrix-vector product `matrixMatvec`.

Input/output:

> `primme_initialize()` sets this field to NULL;

void ***massMatrix**

This field may be used to pass any required information in the matrix-vector product `massMatrixMatvec`.

Input/output:

> `primme_initialize()` sets this field to NULL;

New in version 3.0.

void ***preconditioner**

This field may be used to pass any required information in the preconditioner function `applyPreconditioner`.

Input/output:

> `primme_initialize()` sets this field to NULL;

double ***ShiftsForPreconditioner**

Array of size `blockSize` provided during execution of `dprimme()` and `zprimme()` holding the shifts to be used (if needed) in the preconditioning operation.

For example if the block size is 3, there will be an array of three shifts in `ShiftsForPreconditioner`. Then the user can invert a shifted preconditioner for each of the block vectors $(M - ShiftsForPreconditioner_i)^{-1}x_i$. Classical Davidson (diagonal) preconditioning is an example of this.

this field is read and written by `dprimme()`.

primme_init **initBasisMode**

Select how the search subspace basis is initialized up to `minRestartSize` vectors if not enough initial vectors are provided (see `initSize`):

- `primme_init_krylov`, with a block Krylov subspace generated by the matrix problem and the last initial vectors if given or a random vector otherwise; the size of the block is `maxBlockSize`.
- `primme_init_random`, with random vectors.
- `primme_init_user`, the initial basis will have only initial vectors if given, or a single random vector.

Input/output:

> `primme_initialize()` sets this field to `primme_init_krylov`;
> this field is read by `dprimme()`.

New in version 2.0.

primme_projection `projectionParams`**.projection**

Select the extraction technique, i.e., how the approximate eigenvectors x_i and eigenvalues λ_i are computed from the search subspace $\mathcal{V}$:

- `primme_proj_RR`, Rayleigh-Ritz, $Ax_i - Bx_i\lambda_i \perp \mathcal{V}$.

- `primme_proj_harmonic`, Harmonic Rayleigh-Ritz, $Ax_i - Bx_i\lambda_i \perp (A - \tau B)\mathcal{V}$, where τ is the current target shift (see *targetShifts*).
- `primme_proj_refined`, refined extraction, compute x_i with $||x_i|| = 1$ that minimizes $||(A - \tau B)x_i||$; the eigenvalues are computed as the Rayleigh quotients, $\lambda_i = \frac{x_i^* A x_i}{x_i^* B x_i}$.

Input/output:

primme_initialize() sets this field to *primme_proj_default*;
primme_set_method() and *dprimme()* sets it to *primme_proj_RR* if it is *primme_proj_default*.

int `restartingParams.`**`maxPrevRetain`**

Number of approximations from previous iteration to be retained after restart (this is the locally optimal restarting, see [r2]). The restart size is *minRestartSize* plus *maxPrevRetain*.

Input/output:

primme_initialize() sets this field to 0;
this field is read and written by *primme_set_method()* (see *Preset Methods*);
this field is read by *dprimme()*.

int `correctionParams.`**`precondition`**

Set to 1 to use preconditioning. Make sure *applyPreconditioner* is not NULL then!

Input/output:

primme_initialize() sets this field to 0;
this field is read and written by *primme_set_method()* (see *Preset Methods*);
this field is read by *dprimme()*.

int `correctionParams.`**`robustShifts`**

Set to 1 to use robust shifting. It tries to avoid stagnation and misconvergence by providing as shifts in *ShiftsForPreconditioner* the Ritz values displaced by an approximation of the eigenvalue error.

Input/output:

primme_initialize() sets this field to 0;
written by *primme_set_method()* (see *Preset Methods*);
this field is read by *dprimme()*.

int `correctionParams.`**`maxInnerIterations`**

Control the maximum number of inner QMR iterations:

- 0: no inner iterations;
- >0: perform at most that number of inner iterations per outer step;
- <0: perform at most the rest of the remaining matrix-vector products up to reach *maxMatvecs*.

Input/output:

primme_initialize() sets this field to 0;
this field is read and written by *primme_set_method()* (see *Preset Methods*);
this field is read by *dprimme()*.

See also *convTest*.

double `correctionParams.`**`relTolBase`**

Parameter used when *convTest* is *primme_decreasing_LTolerance*.

Input/output:

primme_initialize() sets this field to 0;
written by *primme_set_method()* (see *Preset Methods*);
this field is read by *dprimme()*.

primme_convergencetest `correctionParams.`**`convTest`**

Set how to stop the inner QMR method:

- `primme_full_LTolerance`: stop by iterations only;
- `primme_decreasing_LTolerance`, stop when $\text{relTolBase}^{-\text{outIts}}$ where outIts is the number of outer iterations and retTolBase is set in *relTolBase*; This is a legacy option from classical JDQR and we recommend **strongly** against its use.
- `primme_adaptive`, stop when the estimated eigenvalue residual has reached the required tolerance (based on Notay's JDCG).
- `primme_adaptive_ETolerance`, as *primme_adaptive* but also stopping when the estimated eigenvalue residual has reduced 10 times.

Input/output:

primme_initialize() sets this field to `primme_adaptive_ETolerance`;
written by *primme_set_method()* (see *Preset Methods*);
this field is read by *dprimme()*.

Note: Avoid to set *maxInnerIterations* to -1 and *convTest* to *primme_full_LTolerance*.

See also *maxInnerIterations*.

int `correctionParams.projectors.`**`LeftQ`**

int `correctionParams.projectors.`**`LeftX`**

int `correctionParams.projectors.`**`RightQ`**

int `correctionParams.projectors.`**`RightX`**

int `correctionParams.projectors.`**`SkewQ`**

int `correctionParams.projectors.`**`SkewX`**

Control the projectors involved in the computation of the correction appended to the basis every (outer) iteration.

Consider the current selected Ritz value Λ and vectors X, the residual associated vectors $R = AX - X\Lambda$, the previous locked vectors Q, and the preconditioner M^{-1}.

When *maxInnerIterations* is 0, the correction D appended to the basis in GD is:

RightX	SkewX	D
0	0	$M^{-1}R$ (Classic GD)
1	0	$M^{-1}(R - \Delta X)$ (cheap Olsen's Method)
1	1	$(I - M^{-1}X(X^*M^{-1}X)^{-1}X^*)M^{-1}R$ (Olsen's Method)
0	1	error

Where Δ is a diagonal matrix that $\Delta_{i,i}$ holds an estimation of the error of the approximate eigenvalue $\Lambda_{i,i}$.

The values of `RightQ`, `SkewQ`, `LeftX` and `LeftQ` are ignored.

When *maxInnerIterations* is not 0, the correction D in Jacobi-Davidson results from solving:

$$P_Q^l P_X^l (A - \sigma I) P_X^r P_Q^r M^{-1} D' = -R, \quad D = P_X^r P_Q^l M^{-1} D'.$$

For `LeftQ`:

0: $P_Q^l = I$;
1: $P_Q^l = I - QQ^*$.

For `LeftX`:

0: $P_X^l = I$;
1: $P_X^l = I - XX^*$.

For `RightQ` and `SkewQ`:

RightQ	SkewQ	P_Q^r
0	0	I
1	0	$I - QQ^*$
1	1	$I - KQ(Q^*KQ)^{-1}Q^*$
0	1	error

For `RightX` and `SkewX`:

RightX	SkewX	P_X^r
0	0	I
1	0	$I - XX^*$
1	1	$I - KX(X^*KX)^{-1}X^*$
0	1	error

Input/output:

`primme_initialize()` sets all of them to 0;
this field is written by *`primme_set_method()`* (see *Preset Methods*);
this field is read by *`dprimme()`*.

See [r3] for a study about different projector configurations in JD.

PRIMME_INT **`ldOPs`**

Recommended leading dimension to be used in *`matrixMatvec`*, *`applyPreconditioner`* and *`massMatrixMatvec`*. The default value is zero, which means no user recommendation. In that case, PRIMME computes ldOPs internally to get better memory performance.

Input/output:

`primme_initialize()` sets this field to -1;
this field is read by *`dprimme()`*.

New in version 2.0.

void (***`monitorFun`**)(void *basisEvals, int *basisSize, int *basisFlags, int *iblock, int *blockSize, void *basisNorms, int *numConverged, void *lockedEvals, int *numLocked, int *lockedFlags, void *lockedNorms, int *inner_its, void *LSRes, **`const`** char *msg, double *time, primme_event *event, **`struct`** *primme_params* *primme, int *ierr)

Convergence monitor. Used to customize how to report solver information during execution (iteration number, matvecs, time, unconverged and converged eigenvalues, residual norms, targets, etc).

Parameters

- **`basisEvals`** – array with approximate eigenvalues of the basis.
- **`basisSize`** – size of the arrays, `basisEvals`, `basisFlags` and `basisNorms`.

- **basisFlags** – state of every approximate pair in the basis.
- **iblock** – indices of the approximate pairs in the block targeted during current iteration.
- **blockSize** – size of array `iblock`.
- **basisNorms** – array with residual norms of the pairs in the basis.
- **numConverged** – number of pairs converged in the basis plus the number of the locked pairs (note that this value isn't monotonic).
- **lockedEvals** – array with the locked eigenvalues.
- **numLocked** – size of the arrays `lockedEvals`, `lockedFlags` and `lockedNorms`.
- **lockedFlags** – state of each locked eigenpair.
- **lockedNorms** – array with the residual norms of the locked pairs.
- **inner_its** – number of performed QMR iterations in the current correction equation. It resets for each block vector.
- **LSRes** – residual norm of the linear system at the current QMR iteration.
- **msg** – output message or function name.
- **time** – time duration.
- **event** – event reported.
- **primme** – parameters structure; the counter in `stats` are updated with the current number of matrix-vector products, iterations, elapsed time, etc., since start.
- **ierr** – output error code; if it is set to non-zero, the current call to PRIMME will stop.

This function is called at the following events:

- `*event == primme_event_outer_iteration`: every outer iterations.

  For this event the following inputs are provided: `basisEvals`, `basisNorms`, `basisSize`, `basisFlags`, `iblock` and `blockSize`.

  `basisNorms[iblock[i]]` has the residual norm for the selected pair in the block. PRIMME avoids computing the residual of soft-locked pairs, `basisNorms[i]` for `i<iblock[0]`. So those values may correspond to previous iterations. The values `basisNorms[i]` for `i>iblock[blockSize-1]` are not valid.

  If *`locking`* is enabled, `lockedEvals`, `numLocked`, `lockedFlags` and `lockedNorms` are also provided.

  `inner_its` and `LSRes` are not provided.

- `*event == primme_event_inner_iteration`: every QMR iteration.

  `basisEvals[0]` and `basisNorms[0]` provides the approximate eigenvalue and the residual norm of the pair which is improved in the current correction equation. If *`convTest`* is *`primme_adaptive`* or *`primme_adaptive_ETolerance`*, `basisEvals[0]` and `basisNorms[0]` are updated every QMR iteration.

  `inner_its` and `LSRes` are also provided.

  `lockedEvals`, `numLocked`, `lockedFlags` and `lockedNorms` may not be provided.

- `*event == primme_event_converged`: a new eigenpair in the basis passed the convergence criterion.

iblock[0] is the index of the newly converged pair in the basis which will be locked or soft-locked. The following are provided: basisEvals, basisNorms, basisSize, basisFlags and blockSize[0]==1.

lockedEvals, numLocked, lockedFlags and lockedNorms may not be provided.

inner_its and LSRes are not provided.

- *event == primme_event_locked: new pair was added to the locked eigenvectors.

  lockedEvals, numLocked, lockedFlags and lockedNorms are provided. The last element of lockedEvals, lockedFlags and lockedNorms corresponds to the recent locked pair.

  basisEvals, numConverged, basisFlags and basisNorms may not be provided.

  inner_its and LSRes are not provided.

- *event == primme_event_message: output message

  msg is the message to print.

  The rest of the arguments are not provided.

The values of basisFlags and lockedFlags are:

- 0: unconverged.
- 1: internal use; only in basisFlags.
- 2: passed convergence test *convTestFun*.
- 3: *practically converged* because the solver may not be able to reduce the residual norm further without recombining the locked eigenvectors.

The actual type of basisEvals, basisNorms, lockedEvals, lockedNorms and LSRes matches the type of evecs of the calling *dprimme()* (or a variant), unless the user sets *monitorFun_type* to another precision.

Input/output:

primme_initialize() sets this field to NULL;
dprimme() sets this field to an internal function if it is NULL;
this field is read by *dprimme()*.

Changed in version 3.0.

primme_op_datatype **monitorFun_type**

Precision of the vectors basisEvals, basisNorms, lockedEvals, lockedNorms and LSRes passed to *monitorFun*.

If it is primme_op_default, the vectors' type matches the calling *dprimme()* (or a variant). Otherwise, the precision is half, single, or double, if *monitorFun_type* is primme_half, primme_float or primme_double respectively.

Input/output:

primme_initialize() sets this field to primme_op_default;
this field is read by *dprimme()*, and if it is primme_op_default it is set to the value that matches the precision of calling function.

New in version 3.0.

void ***monitor**

This field may be used to pass any required information to the function *monitorFun*.

Input/output:

primme_initialize() sets this field to NULL;

New in version 2.0.

PRIMME_INT stats.**numOuterIterations**

Hold the number of outer iterations. The value is available during execution and at the end.

Input/output:

primme_initialize() sets this field to 0;
written by *dprimme()*.

PRIMME_INT stats.**numRestarts**

Hold the number of restarts during execution and at the end.

Input/output:

primme_initialize() sets this field to 0;
written by *dprimme()*.

PRIMME_INT stats.**numMatvecs**

Hold how many vectors the operator in *matrixMatvec* has been applied on. The value is available during execution and at the end.

Input/output:

primme_initialize() sets this field to 0;
written by *dprimme()*.

PRIMME_INT stats.**numPreconds**

Hold how many vectors the operator in *applyPreconditioner* has been applied on. The value is available during execution and at the end.

Input/output:

primme_initialize() sets this field to 0;
written by *dprimme()*.

PRIMME_INT stats.**numGlobalSum**

Hold how many times *globalSumReal* has been called. The value is available during execution and at the end.

Input/output:

primme_initialize() sets this field to 0;
written by *dprimme()*.

double stats.**volumeGlobalSum**

Hold how many `REAL` have been reduced by *globalSumReal*. The value is available during execution and at the end.

Input/output:

primme_initialize() sets this field to 0;
written by *dprimme()*.

PRIMME_INT stats.**numBroadcast**

Hold how many times *broadcastReal* has been called. The value is available during execution and at the end.

Input/output:

primme_initialize() sets this field to 0;
written by *dprimme()*.

New in version 3.0.

double `stats.`**`volumeBroadcast`**

Hold how many `REAL` have been broadcast by *`broadcastReal`*. The value is available during execution and at the end.

Input/output:

`primme_initialize()` sets this field to 0;
written by *`dprimme()`*.

New in version 3.0.

PRIMME_INT `stats.`**`numOrthoInnerProds`**

Hold how many inner products with vectors of length *`nLocal`* have been computed during orthogonalization. The value is available during execution and at the end.

Input/output:

`primme_initialize()` sets this field to 0;
written by *`dprimme()`*.

New in version 3.0.

double `stats.`**`elapsedTime`**

Hold the wall clock time spent by the call to *`dprimme()`* or *`zprimme()`*. The value is available at the end of the execution.

Input/output:

`primme_initialize()` sets this field to 0;
written by *`dprimme()`*.

double `stats.`**`timeMatvec`**

Hold the wall clock time spent by *`matrixMatvec`*. The value is available at the end of the execution.

Input/output:

`primme_initialize()` sets this field to 0;
written by *`dprimme()`*.

double `stats.`**`timePrecond`**

Hold the wall clock time spent by *`applyPreconditioner`*. The value is available at the end of the execution.

Input/output:

`primme_initialize()` sets this field to 0;
written by *`dprimme()`*.

double `stats.`**`timeOrtho`**

Hold the wall clock time spent by orthogonalization. The value is available at the end of the execution.

Input/output:

`primme_initialize()` sets this field to 0;
written by *`dprimme()`*.

double `stats.`**`timeGlobalSum`**

Hold the wall clock time spent by *`globalSumReal`*. The value is available at the end of the execution.

Input/output:

`primme_initialize()` sets this field to 0;

written by `dprimme()`.

double `stats.`**`timeBroadcast`**

Hold the wall clock time spent by `broadcastReal`. The value is available at the end of the execution.

Input/output:

`primme_initialize()` sets this field to 0;
written by `dprimme()`.

New in version 3.0.

double `stats.`**`estimateMinEVal`**

Hold the estimation of the smallest eigenvalue for the current eigenproblem. The value is available during execution and at the end.

Input/output:

`primme_initialize()` sets this field to 0;
written by `dprimme()`.

double `stats.`**`estimateMaxEVal`**

Hold the estimation of the largest eigenvalue for the current eigenproblem. The value is available during execution and at the end.

Input/output:

`primme_initialize()` sets this field to 0;
written by `dprimme()`.

double `stats.`**`estimateLargestSVal`**

Hold the estimation of the largest singular value (i.e., the absolute value of the eigenvalue with largest absolute value) for the current eigenproblem. The value is available during execution and at the end.

Input/output:

`primme_initialize()` sets this field to 0;
written by `dprimme()`.

double `stats.`**`maxConvTol`**

Hold the maximum residual norm of the converged eigenvectors. The value is available during execution and at the end.

Input/output:

`primme_initialize()` sets this field to 0;
written by `dprimme()`.

PRIMME_INT `stats.`**`lockingIssue`**

It is set to a nonzero value if some of the returned eigenpairs do not pass the convergence criterion. See `convTestFun` and `eps`.

Input/output:

`primme_initialize()` sets this field to 0;
written by `dprimme()`.

New in version 3.0.

void (***`convTestFun`**) (double *eval, void *evec, double *resNorm, int *isconv, *primme_params* *primme, int *ierr)

Function that evaluates if the approximate eigenpair has converged. If NULL, it is used the default convergence criteria (see `eps`).

Parameters

- **eval** – the approximate value to evaluate.
- **evec** – one dimensional array of size *nLocal* containing the approximate vector; it can be NULL.
- **resNorm** – the norm of the residual vector.
- **isconv** – (output) the function sets zero if the pair is not converged and non zero otherwise.
- **primme** – parameters structure.
- **ierr** – output error code; if it is set to non-zero, the current call to PRIMME will stop.

The actual type of `evec` matches the type of `evecs` of the calling *dprimme()* (or a variant), unless the user sets *convTestFun_type* to another precision.

Input/output:

primme_initialize() sets this field to NULL;
this field is read by *dprimme()*.

New in version 2.0.

primme_op_datatype **convTestFun_type**

Precision of the vectors `evec` passed to *convTestFun*.

If it is `primme_op_default`, `evec`'s type matches the calling *dprimme()* (or a variant). Otherwise, the precision is half, single, or double, if *convTestFun_type* is `primme_half`, `primme_float` or `primme_double` respectively.

Input/output:

primme_initialize() sets this field to `primme_op_default`;
this field is read by *dprimme()*, and if it is `primme_op_default` it is set to the value that matches the precision of calling function.

New in version 3.0.

void ***convtest**

This field may be used to pass any required information to the function *convTestFun*.

Input/output:

primme_initialize() sets this field to NULL;

New in version 2.0.

void ***queue**

Pointer to the accelerator's data structure.

If the main call is `dprimme_magma()` or a variant, this field should have the pointer to an initialized `magma_queue_t`.

See example `examples/ex_eigs_dmagma.c`.

Input/output:

primme_initialize() sets this field to NULL;
this field is read by `dprimme_magma()`.

New in version 3.0.

2.7 Preset Methods

enum primme_preset_method

enumerator PRIMME_DEFAULT_MIN_TIME
Set as *PRIMME_JDQMR_ETol* when *target* is either `primme_smallest` or `primme_largest`, and as *PRIMME_JDQMR* otherwise. This method is usually the fastest if the cost of the matrix vector product is inexpensive.

enumerator PRIMME_DEFAULT_MIN_MATVECS
Currently set as *PRIMME_GD_Olsen_plusK*; this method usually performs fewer matrix vector products than other methods, so it's a good choice when this operation is expensive.

enumerator PRIMME_DYNAMIC
Switches to the best method dynamically; currently, between methods *PRIMME_DEFAULT_MIN_TIME* and *PRIMME_DEFAULT_MIN_MATVECS*.

With *PRIMME_DYNAMIC* *primme_set_method()* sets *dynamicMethodSwitch* = 1 and makes the same changes as for method *PRIMME_DEFAULT_MIN_TIME*.

enumerator PRIMME_Arnoldi
Arnoldi implemented à la Generalized Davidson.

With *PRIMME_Arnoldi* *primme_set_method()* sets:

- *locking* = 0;
- *maxPrevRetain* = 0;
- *precondition* = 0;
- *maxInnerIterations* = 0.

enumerator PRIMME_GD
Generalized Davidson.

With *PRIMME_GD* *primme_set_method()* sets:

- *locking* = 0;
- *maxPrevRetain* = 0;
- *robustShifts* = 1;
- *maxInnerIterations* = 0;
- *RightX* = 0;
- *SkewX* = 0.

enumerator PRIMME_GD_plusK
GD with locally optimal restarting.

With *PRIMME_GD_plusK* *primme_set_method()* sets *maxPrevRetain* = 2 if *maxBlockSize* is 1 and *numEvals* > 1; otherwise it sets *maxPrevRetain* to *maxBlockSize*. Also:

- *locking* = 0;
- *maxInnerIterations* = 0;
- *RightX* = 0;
- *SkewX* = 0.

enumerator PRIMME_GD_Olsen_plusK
GD+k and the cheap Olsen's Method.

With *PRIMME_GD_Olsen_plusK* *primme_set_method()* makes the same changes as for method *PRIMME_GD_plusK* and sets *RightX* = 1.

enumerator PRIMME_JD_Olsen_plusK

GD+k and Olsen's Method.

With `PRIMME_JD_Olsen_plusK` `primme_set_method()` makes the same changes as for method `PRIMME_GD_plusK` and also sets `robustShifts` = 1, `RightX` to 1, and `SkewX` to 1.

enumerator PRIMME_RQI

(Accelerated) Rayleigh Quotient Iteration.

With `PRIMME_RQI` `primme_set_method()` sets:

- `locking` = 1;
- `maxPrevRetain` = 0;
- `robustShifts` = 1;
- `maxInnerIterations` = -1;
- `LeftQ` = 1;
- `LeftX` = 1;
- `RightQ` = 0;
- `RightX` = 1;
- `SkewQ` = 0;
- `SkewX` = 0;
- `convTest` = `primme_full_LTolerance`.

Note: If `numTargetShifts` > 0 and `targetShifts` are provided, the interior problem solved uses these shifts in the correction equation. Therefore RQI becomes INVIT (inverse iteration) in that case.

enumerator PRIMME_JDQR

Jacobi-Davidson with fixed number of inner steps.

With `PRIMME_JDQR` `primme_set_method()` sets:

- `locking` = 1;
- `maxPrevRetain` = 1;
- `robustShifts` = 0;
- `maxInnerIterations` = 10 if it is 0;
- `LeftQ` = 0;
- `LeftX` = 1;
- `RightQ` = 1;
- `RightX` = 1;
- `SkewQ` = 1;
- `SkewX` = 1;
- `relTolBase` = 1.5;
- `convTest` = `primme_full_LTolerance`.

enumerator PRIMME_JDQMR

Jacobi-Davidson with adaptive stopping criterion for inner Quasi Minimum Residual (QMR).

With `PRIMME_JDQMR` `primme_set_method()` sets:

- `locking` = 0;
- `maxPrevRetain` = 1 if it is 0
- `maxInnerIterations` = -1;
- `LeftQ` = `precondition`;
- `LeftX` = 1;
- `RightQ` = 0;
- `RightX` = 0;
- `SkewQ` = 0;

- *SkewX* = 1;
- *convTest* = *primme_adaptive*.

enumerator PRIMME_JDQMR_ETol

JDQMR but QMR stops after residual norm reduces by a 0.1 factor.

With *PRIMME_JDQMR_ETol* *primme_set_method()* makes the same changes as for the method *PRIMME_JDQMR* and sets *convTest* = *primme_adaptive_ETolerance*.

enumerator PRIMME_STEEPEST_DESCENT

Steepest descent.

With *PRIMME_STEEPEST_DESCENT* *primme_set_method()* sets:

- *locking* = 1;
- *maxBasisSize* = *numEvals* * 2;
- *minRestartSize* = *numEvals*;
- *maxBlockSize* = *numEvals*;
- *maxPrevRetain* = 0;
- *robustShifts* = 0;
- *maxInnerIterations* = 0;
- *RightX* = 1;
- *SkewX* = 0.

enumerator PRIMME_LOBPCG_OrthoBasis

LOBPCG with orthogonal basis.

With *PRIMME_LOBPCG_OrthoBasis* *primme_set_method()* sets:

- *locking* = 0;
- *maxBasisSize* = *numEvals* * 3;
- *minRestartSize* = *numEvals*;
- *maxBlockSize* = *numEvals*;
- *maxPrevRetain* = *numEvals*;
- *robustShifts* = 0;
- *maxInnerIterations* = 0;
- *RightX* = 1;
- *SkewX* = 0.

enumerator PRIMME_LOBPCG_OrthoBasis_Window

LOBPCG with sliding window of *maxBlockSize* < 3 * *numEvals*.

With *PRIMME_LOBPCG_OrthoBasis_Window* *primme_set_method()* sets:

- *locking* = 0;
- *maxBasisSize* = *maxBlockSize* * 3;
- *minRestartSize* = *maxBlockSize*;
- *maxPrevRetain* = *maxBlockSize*;
- *robustShifts* = 0;
- *maxInnerIterations* = 0;
- *RightX* = 1;
- *SkewX* = 0.

2.8 Error Codes

The functions `dprimme()` and `zprimme()` return one of the following error codes. Some of the error codes have a macro associated which is indicated in brackets.

- 0: success; usually all requested eigenpairs have converged.
- -1: (`PRIMME_UNEXPECTED_FAILURE`) unexpected internal error; please consider to set `printLevel` to a value larger than 0 to see the call stack and to report these errors because they may be bugs.
- -2: (`PRIMME_MALLOC_FAILURE`) failure in allocating memory; it can be either CPU or GPU.
- -3: (`PRIMME_MAIN_ITER_FAILURE`) maximum number of outer iterations `maxOuterIterations` or matvecs `maxMatvecs` reached.
- -4: if argument `primme` is NULL.
- -5: if `n` < 0 or `nLocal` < 0 or `nLocal` > `n`.
- -6: if `numProcs` < 1.
- -7: if `matrixMatvec` is NULL.
- -8: if `applyPreconditioner` is NULL and `precondition` > 0.
- -10: if `numEvals` > `n`.
- -11: if `numEvals` < 0.
- -12: if `convTestFun` is not NULL and `eps` > 0 and `eps` < machine precision given by `internalPrecision` and the precision of PRIMME call (`sprimme()`, `dprimme()`...).
- -13: if `target` is not properly defined.
- -14: if `target` is one of `primme_closest_geq`, `primme_closest_leq`, `primme_closest_abs` or `primme_largest_abs` but `numTargetShifts` <= 0 (no shifts).
- -15: if `target` is one of `primme_closest_geq`, `primme_closest_leq`, `primme_closest_abs` or `primme_largest_abs` but `targetShifts` is NULL (no shifts array).
- -16: if `numOrthoConst` < 0 or `numOrthoConst` > `n`. (no free dimensions left).
- -17: if `maxBasisSize` < 2 and `n` > 2.
- -18: if `minRestartSize` < 0, or `minRestartSize` is zero but `n` > 2 and `numEvals` > 0.
- -19: if `maxBlockSize` < 0, or `maxBlockSize` is zero but `numEvals` > 0.
- -20: if `maxPrevRetain` < 0.
- -22: if `initSize` < 0.
- -23: if `locking` == 0 and `initSize` > `maxBasisSize`.
- -24: if `locking` and `initSize` > `numEvals`.
- -25: if `maxPrevRetain` + `minRestartSize` >= `maxBasisSize`, and `n` > `maxBasisSize`.
- -26: if `minRestartSize` >= `n`, and `n` > 2.
- -27: if `printLevel` < 0 or `printLevel` > 5.
- -28: if `convTest` is not one of `primme_full_LTolerance`, `primme_decreasing_LTolerance`, `primme_adaptive_ETolerance` or `primme_adaptive`.
- -29: if `convTest` == `primme_decreasing_LTolerance` and `relTolBase` <= 1.
- -30: if `evals` is NULL.

- -31: if `evecs` is NULL, or is not a GPU pointer when calling a GPU variant (for instance `magma_dprimme`).
- -32: if `resNorms` is NULL.
- -33: if *`locking`* == 0 and *`minRestartSize`* < *`numEvals`* and *`n`* > 2.
- -34: if *`ldevecs`* < *`nLocal`*.
- -35: if *`ldOPs`* is not zero and less than *`nLocal`*.
- -38: if *`locking`* == 0 and *`target`* is *`primme_closest_leq`* or *`primme_closest_geq`*.
- -40: (`PRIMME_LAPACK_FAILURE`) some LAPACK function performing a factorization returned an error code; set *`printLevel`* > 0 to see the error code and the call stack.
- -41: (`PRIMME_USER_FAILURE`) some of the user-defined functions (*`matrixMatvec`*, *`applyPreconditioner`*, ...) returned a non-zero error code; set *`printLevel`* > 0 to see the call stack that produced the error.
- -42: (`PRIMME_ORTHO_CONST_FAILURE`) the provided orthogonal constraints (see *`numOrthoConst`*) are not full rank.
- -43: (`PRIMME_PARALLEL_FAILURE`) some process has a different value in an input option than the process zero, or it is not acting coherently; set *`printLevel`* > 0 to see the call stack that produced the error.
- -44: (`PRIMME_FUNCTION_UNAVAILABLE`) PRIMME was not compiled with support for the requesting precision or for GPUs.

CHAPTER

THREE

SINGULAR VALUE PROBLEMS

3.1 C Library Interface

New in version 2.0.

The PRIMME SVDS interface is composed of the following functions. To solve real and complex singular value problems call respectively:

```
int dprimme_svds (double *svals, double *svecs, double *resNorms,
                        primme_svds_params *primme_svds)
int zprimme_svds (double *svals, PRIMME_COMPLEX_DOUBLE *svecs,
                  double *resNorms, primme_svds_params *primme_svds)
```

There are more versions for matrix problems working in other precisions:

Precision	Real	Complex
half	*hprimme_svds()* *hsprimme_svds()*	*kprimme_svds()* *ksprimme_svds()*
single	*sprimme_svds()*	*cprimme_svds()*
double	*dprimme_svds()*	*zprimme_svds()*

Other useful functions:

```
void primme_svds_initialize (primme_svds_params *primme_svds)
int primme_svds_set_method (primme_svds_preset_method method,
   primme_preset_method methodStage1,
   primme_preset_method methodStage2, primme_svds_params *primme_svds)
void primme_svds_display_params (primme_svds_params primme_svds)
void primme_svds_free (primme_svds_params *primme_svds)
```

PRIMME SVDS stores its data on the structure `primme_svds_params`. See *Parameters Guide* for an introduction about its fields.

3.1.1 Running

To use PRIMME SVDS, follow these basic steps.

1. Include:

```
#include "primme.h"   /* header file is required to run primme */
```

2. Initialize a PRIMME SVDS parameters structure for default settings:

```
primme_svds_params primme_svds;
primme_svds_initialize (&primme_svds);
```

3. Set problem parameters (see also *Parameters Guide*), and, optionally, set one of the `preset methods`:

   ```
   primme_svds.matrixMatvec = matrixMatvec; /* MV product */
   primme_svds.m = 1000;      /* set the matrix dimensions */
   primme_svds.n = 100;
   primme_svds.numSvals = 10; /* Number of singular values */
   primmesvds_set_method (primme_svds_hybrid, PRIMME_DEFAULT_METHOD,
                                PRIMME_DEFAULT_METHOD, &primme_svds);
   ...
   ```

4. Then to solve a real singular value problem call:

   ```
   ret = dprimme_svds (svals, svecs, resNorms, &primme_svds);
   ```

   The previous is the double precision call. There is available calls for complex double, single and complex single; check `zprimme_svds()`, `sprimme_svds()` and `cprimme_svds()`.

   To solve complex singular value problems call:

   ```
   ret = zprimme_svds (svals, svecs, resNorms, &primme_svds);
   ```

   The call arguments are:

   - *svals*, array to return the found singular values;
   - *svecs*, array to return the found left and right singular vectors;
   - *resNorms*, array to return the residual norms of the found triplets; and
   - *ret*, returned error code.

5. To free the work arrays in PRIMME SVDS:

   ```
   primme_svds_free (&primme_svds);
   ```

3.1.2 Parameters Guide

PRIMME SVDS stores the data on the structure `primme_svds_params`, which has the next fields:

Basic
`PRIMME_INT` `m`, number of rows of the matrix.
`PRIMME_INT` `n`, number of columns of the matrix.
`void (* matrixMatvec )(...)`, matrix-vector product.
`int` `numSvals`, how many singular triplets to find.
`primme_svds_target` `target`, which singular values to find.
`double` `eps`, tolerance of the residual norm of converged triplets.

For parallel programs
`int` `numProcs`, number of processes
`int` `procID`, rank of this process
`PRIMME_INT` `mLocal`, number of rows stored in this process
`PRIMME_INT` `nLocal`, number of columns stored in this process
`void (* globalSumReal )(...)`, sum reduction among processes

Accelerate the convergence
`void (* applyPreconditioner )(...)`, preconditioner-vector product.

`int` *initSize*, initial vectors as approximate solutions.
`int` *maxBasisSize*
`int` `minRestartSize`
`int` *maxBlockSize*

User data
`void *` *commInfo*
`void *` *matrix*
`void *` *preconditioner*
`void *` *convtest*
`void *` *monitor*

Advanced options
`int` *numTargetShifts*, for targeting interior singular values.
`double *` *targetShifts*
`int` `numOrthoConst`, orthogonal constrains to the singular vectors.
`int` *locking*
`PRIMME_INT` *maxMatvecs*
`PRIMME_INT` *iseed* `[4]`
`double` *aNorm*
`int` *printLevel*
`FILE *` *outputFile*
`primme_svds_operator` *method*
`primme_svds_operator` *methodStage2*
primme_params *primme*
primme_params *primmeStage2*
`void (*` *convTestFun* `)(...)`, custom convergence criterion.
`void (*` *monitorFun* `)(...)`, custom convergence history.
`primme_op_datatype` *matrixMatvec_type*
`primme_op_datatype` *applyPreconditioner_type*
`primme_op_datatype` `globalSumReal_type`
`primme_op_datatype` *broadcastReal_type*
`primme_op_datatype` *internalPrecision*

PRIMME SVDS requires the user to set at least the matrix dimensions (*m* x *n*) and the matrix-vector product (*matrixMatvec*), as they define the problem to be solved. For parallel programs, *mLocal*, *nLocal*, *procID* and *globalSumReal* are also required.

In addition, most users would want to specify how many singular triplets to find, and provide a preconditioner (if available).

It is useful to have set all these before calling *primme_svds_set_method()*. Also, if users have a preference on *maxBasisSize*, *maxBlockSize*, etc, they should also provide them into *primme_svds_params* prior to the *primme_svds_set_method()* call. This helps *primme_svds_set_method()* make the right choice on other parameters. It is sometimes useful to check the actual parameters that PRIMME SVDS is going to use (before calling it) or used (on return) by printing them with *primme_svds_display_params()*.

3.1.3 Interface Description

The next enumerations and functions are declared in `primme.h`.

?primme_svds

int **hprimme_svds** (*PRIMME_HALF *svals*, *PRIMME_HALF *svecs*, *PRIMME_HALF *resNorms*, *primme_svds_params *primme_svds*)

int **hsprimme_svds** (float **svals*, *PRIMME_HALF *svecs*, float **resNorms*, *primme_svds_params *primme_svds*)

int **kprimme_svds** (*PRIMME_HALF *svals*, *PRIMME_COMPLEX_HALF *svecs*, *PRIMME_HALF *resNorms*, *primme_svds_params *primme_svds*)

int **ksprimme_svds** (float **svals*, *PRIMME_COMPLEX_HALF *svecs*, float **resNorms*, *primme_svds_params *primme_svds*)

New in version 3.0.

int **sprimme_svds** (float **svals*, float **svecs*, float **resNorms*, *primme_svds_params *primme_svds*)

int **cprimme_svds** (float **svals*, *PRIMME_COMPLEX_FLOAT *svecs*, float **resNorms*, *primme_svds_params *primme_svds*)

New in version 2.0.

int **dprimme_svds** (double **svals*, double **svecs*, double **resNorms*, *primme_svds_params *primme_svds*)

int **zprimme_svds** (double **svals*, *PRIMME_COMPLEX_DOUBLE *svecs*, double **resNorms*, *primme_svds_params *primme_svds*)

Solve a real singular value problem.

All arrays should be hosted on CPU. The computations are performed on CPU (see `magma_dprimme_svds()` for using GPUs).

Parameters

- **`svals`** – array at least of size `numSvals` to store the computed singular values; all processes in a parallel run return this local array with the same values.
- **`svecs`** – array at least of size (`mLocal` + `nLocal`) times (`numOrthoConst` + `numSvals`) to store column-wise the (local part for this process of the) computed left singular vectors and the right singular vectors.
- **`resNorms`** – array at least of size `numSvals` to store the residual norms of the computed triplets; all processes in parallel run return this local array with the same values.
- **`primme_svds`** – parameters structure.

Returns error indicator; see *Error Codes*.

On input, `svecs` should start with the content of the `numOrthoConst` left vectors, followed by the `initSize` left vectors, followed by the `numOrthoConst` right vectors, and followed by the `initSize` right vectors.

On return, the i-th left singular vector starts at svecs[(`numOrthoConst` +i)* `mLocal`]. The i-th right singular vector starts at svecs[(`numOrthoConst` + `initSize`)* `mLocal` + (`numOrthoConst` +i)* `nLocal`]. The first vector has i=0.

All internal operations are performed at the same precision than `svecs` unless the user sets `internalPrecision` otherwise. The functions `hsprimme_svds()` and `ksprimme_svds()` perform all computations in half precision by default and report the eigenvalues and the residual norms in single precision. These functions may help in applications that may be not built with a compiler supporting half precision.

The type and precision of the callbacks depends on the type and precision of `svecs`. Although this can be changed. See details for *`matrixMatvec`*, *`applyPreconditioner`*, *`globalSumReal`*, *`broadcastReal`*, and *`convTestFun`*.

magma_?primme_svds

int **`magma_hprimme_svds`** (*PRIMME_HALF *svals*, *PRIMME_HALF *svecs*, *PRIMME_HALF *resNorms*, *primme_svds_params *primme_svds*)

int **`magma_hsprimme_svds`** (float **svals*, *PRIMME_HALF *svecs*, float **resNorms*, *primme_svds_params *primme_svds*)

int **`magma_kprimme_svds`** (*PRIMME_HALF *svals*, *PRIMME_COMPLEX_HALF *svecs*, *PRIMME_HALF *resNorms*, *primme_svds_params *primme_svds*)

int **`magma_ksprimme_svds`** (float **svals*, *PRIMME_COMPLEX_HALF *svecs*, float **resNorms*, *primme_svds_params *primme_svds*)

int **`magma_sprimme_svds`** (float **svals*, float **svecs*, float **resNorms*, *primme_svds_params *primme_svds*)

int **`magma_cprimme_svds`** (float **svals*, *PRIMME_COMPLEX_FLOAT *svecs*, float **resNorms*, *primme_svds_params *primme_svds*)

int **`magma_dprimme_svds`** (double **svals*, double **svecs*, double **resNorms*, *primme_svds_params *primme_svds*)

int **`magma_zprimme_svds`** (double **svals*, *PRIMME_COMPLEX_DOUBLE *svecs*, double **resNorms*, *primme_svds_params *primme_svds*)

Solve a real singular value problem.

Most of the computations are performed on GPU (see *`dprimme_svds()`* for using only the CPU).

Parameters

- **`svals`** – CPU array at least of size *`numSvals`* to store the computed singular values; all processes in a parallel run return this local array with the same values.
- **`svecs`** – GPU array at least of size (*`mLocal`* + *`nLocal`*) times (`numOrthoConst` + *`numSvals`*) to store column-wise the (local part for this process of the) computed left singular vectors and the right singular vectors.
- **`resNorms`** – CPU array at least of size *`numSvals`* to store the residual norms of the computed triplets; all processes in parallel run return this local array with the same values.
- **`primme_svds`** – parameters structure.

Returns error indicator; see *Error Codes*.

On input, `svecs` should start with the content of the `numOrthoConst` left vectors, followed by the *`initSize`* left vectors, followed by the `numOrthoConst` right vectors, and followed by the *`initSize`* right vectors.

On return, the i-th left singular vector starts at svecs[(`numOrthoConst` +i)* *`mLocal`*]. The i-th right singular vector starts at svecs[(`numOrthoConst` + *`initSize`*)* *`mLocal`* + (`numOrthoConst` +i)* *`nLocal`*]. The first vector has i=0.

All internal operations are performed at the same precision than `svecs` unless the user sets *`internalPrecision`* otherwise. The functions *`magma_hsprimme_svds()`* and *`magma_ksprimme_svds()`* perform all computations in half precision by default and report the eigenvalues and the residual norms in single precision. These functions may help in applications that may be not built with a compiler supporting half precision.

The type and precision of the callbacks depends on the type and precision of `svecs`. Although this can be changed. See details for *matrixMatvec*, *applyPreconditioner*, *globalSumReal*, *broadcastReal*, and *convTestFun*.

New in version 3.0.

primme_svds_initialize

void **primme_svds_initialize**(*primme_svds_params* **primme_svds*)

Initialize PRIMME SVDS parameters structure to the default values.

After calling *dprimme_svds()* (or a variant), call *primme_svds_free()* to release allocated resources by PRIMME.

Parameters

- **primme_svds** – parameters structure.

Example:

```
primme_svds_params primme_svds;
primme_svds_initialize(&primme_svds);

primme_svds.n = 100;
...
dprimme_svds(svals, svecs, rnorms, &primme_svds);
...

primme_svds_free(&primme_svds);
```

See the alternative function *primme_svds_params_create()* that also allocates the structure.

primme_svds_create

primme_svds_params ***primme_svds_params_create**(void)

Allocate and initialize a parameters structure to the default values.

After calling *dprimme_svds()* (or a variant), call *primme_svds_params_destroy()* to release allocated resources by PRIMME.

Parameters

- **primme_sv** – parameters structure.

Example:

```
primme_svds_params *primme_svds = primme_svds_params_create();

primme_svds->n = 100;
...
dprimme_svds(svals, svecs, rnorms, primme_svds);
...

primme_svds_params_destroy(primme_svds);
```

See the alternative function *primme_svds_initialize()* that only initializes the structure.

New in version 3.0.

primme_svds_set_method

int **primme_svds_set_method**(*primme_svds_preset_method method*, *primme_preset_method methodStage1*, *primme_preset_method methodStage2*, *primme_svds_params *primme_svds*)

Set PRIMME SVDS parameters to one of the preset configurations.

Parameters

- **method** – preset method to compute the singular triplets; one of
  - `primme_svds_default`, currently set as `primme_svds_hybrid`.
  - `primme_svds_normalequations`, compute the eigenvectors of A^*A or AA^*.
  - `primme_svds_augmented`, compute the eigenvectors of the augmented matrix, $\begin{pmatrix} 0 & A^* \\ A & 0 \end{pmatrix}$.
  - `primme_svds_hybrid`, start with `primme_svds_normalequations`; use the resulting approximate singular vectors as initial vectors for `primme_svds_augmented` if the required accuracy was not achieved.
- **methodStage1** – preset method to compute the eigenpairs at the first stage; see available values at `primme_set_method()`.
- **methodStage2** – preset method to compute the eigenpairs with the second stage of `primme_svds_hybrid`; see available values at `primme_set_method()`.
- **primme_svds** – parameters structure.

See also *Preset Methods*.

primme_svds_display_params

void **primme_svds_display_params**(*primme_svds_params primme_svds*)

Display all printable settings of `primme_svds` into the file descriptor `outputFile`.

Parameters

- **primme_svds** – parameters structure.

primme_svds_free

void **primme_svds_free**(*primme_svds_params *primme_svds*)

Free memory allocated by PRIMME SVDS.

Parameters

- **primme_svds** – parameters structure.

primme_svds_params_destroy

int **`primme_svds_params_destroy`**(*primme_svds_params *primme*)

Free memory allocated by PRIMME associated to a parameters structure created with *`primme_svds_params_create()`*.

Parameters

- **`primme_svds`** – parameters structure.

Returns nonzero value if the call is not successful.

New in version 3.0.

3.2 FORTRAN Library Interface

New in version 2.0.

The next enumerations and functions are declared in `primme_svds_f77.h`.

3.2.1 sprimme_svds_f77

`subroutine sprimme_svds_f77` (*svals*, *svecs*, *resNorms*, *primme_svds*, *ierr*)

Solve a real singular value problem using single precision.

All arrays should be hosted on CPU. The computations are performed on CPU (see *`magma_sprimme_svds_f77()`* for using GPUs).

Parameters

- **svals** (*) *[real]* :: (output) array at least of size *`numSvals`* to store the computed singular values; all processes in a parallel run return this local array with the same values.
- **svecs** (*) *[real]* :: array at least of size (*`mLocal`* + *`nLocal`*) times (`numOrthoConst` + *`numSvals`*) to store column-wise the (local part for this process of the) computed left singular vectors and the right singular vectors.
- **resNorms** (*) *[real]* :: array at least of size *`numSvals`* to store the residual norms of the computed triplets; all processes in parallel run return this local array with the same values.
- **primme_svds** *[*ptr*]* :: parameters structure.
- **ierr** *[integer]* :: (output) error indicator; see *Error Codes*.

On input, `svecs` should start with the content of the `numOrthoConst` left vectors, followed by the *`initSize`* left vectors, followed by the `numOrthoConst` right vectors and followed by the *`initSize`* right vectors.

On return, the i-th left singular vector starts at svecs((`numOrthoConst` + i - 1) * *`mLocal`*). The i-th right singular vector starts at svecs((`numOrthoConst` + *`initSize`*)* *`mLocal`* + (`numOrthoConst` + i - 1)* *`nLocal`*). The first vector has i=1.

All internal operations are performed at the same precision than `svecs` unless the user sets *`internalPrecision`* otherwise.

The type and precision of the callbacks depends on the type and precision of *svecs*. See details for *`matrixMatvec`*, *`applyPreconditioner`*, *`globalSumReal`*, *`broadcastReal`*, and *`convTestFun`*.

3.2.2 cprimme_svds_f77

`subroutine cprimme_svds_f77` (*svals*, *svecs*, *resNorms*, *primme_svds*, *ierr*)

Solve a complex singular value problem using single precision.

All arrays should be hosted on CPU. The computations are performed on CPU (see *`magma_cprimme_svds_f77()`* for using GPUs).

Parameters

- **svals** (*) *[real]* :: (output) array at least of size *`numSvals`* to store the computed singular values; all processes in a parallel run return this local array with the same values.

- **svecs** (*) *[complex]* :: array at least of size (`mLocal` + `nLocal`) times (`numOrthoConst` + `numSvals`) to store column-wise the (local part for this process of the) computed left singular vectors and the right singular vectors.
- **resNorms** (*) *[real]* :: array at least of size `numSvals` to store the residual norms of the computed triplets; all processes in parallel run return this local array with the same values.
- **primme_svds** *[ptr]* :: parameters structure.
- **ierr** *[integer]* :: (output) error indicator; see *Error Codes*.

On input, `svecs` should start with the content of the `numOrthoConst` left vectors, followed by the `initSize` left vectors, followed by the `numOrthoConst` right vectors and followed by the `initSize` right vectors.

On return, the i-th left singular vector starts at svecs((`numOrthoConst` + i - 1) * `mLocal`). The i-th right singular vector starts at svecs((`numOrthoConst` + `initSize`)* `mLocal` + (`numOrthoConst` + i - 1)* `nLocal`). The first vector has i=1.

All internal operations are performed at the same precision than `svecs` unless the user sets `internalPrecision` otherwise.

The type and precision of the callbacks depends on the type and precision of *svecs*. See details for `matrixMatvec`, `applyPreconditioner`, `globalSumReal`, `broadcastReal`, and `convTestFun`.

3.2.3 dprimme_svds_f77

subroutine dprimme_svds_f77 (*svals*, *svecs*, *resNorms*, *primme_svds*, *ierr*)

Solve a real singular value problem using double precision.

All arrays should be hosted on CPU. The computations are performed on CPU (see `magma_dprimme_svds_f77()` for using GPUs).

Parameters

- **svals** (*) *[double precision]* :: (output) array at least of size `numSvals` to store the computed singular values; all processes in a parallel run return this local array with the same values.
- **svecs** (*) *[double precision]* :: array at least of size (`mLocal` + `nLocal`) times (`numOrthoConst` + `numSvals`) to store column-wise the (local part for this process of the) computed left singular vectors and the right singular vectors.
- **resNorms** (*) *[double precision]* :: array at least of size `numSvals` to store the residual norms of the computed triplets; all processes in parallel run return this local array with the same values.
- **primme_svds** *[ptr]* :: parameters structure.
- **ierr** *[integer]* :: (output) error indicator; see *Error Codes*.

On input, `svecs` should start with the content of the `numOrthoConst` left vectors, followed by the `initSize` left vectors, followed by the `numOrthoConst` right vectors and followed by the `initSize` right vectors.

On return, the i-th left singular vector starts at svecs((`numOrthoConst` + i - 1) * `mLocal`). The i-th right singular vector starts at svecs((`numOrthoConst` + `initSize`)* `mLocal` + (`numOrthoConst` + i - 1)* `nLocal`). The first vector has i=1.

All internal operations are performed at the same precision than `svecs` unless the user sets `internalPrecision` otherwise.

The type and precision of the callbacks depends on the type and precision of *svecs*. See details for `matrixMatvec`, `applyPreconditioner`, `globalSumReal`, `broadcastReal`, and `convTestFun`.

3.2.4 zprimme_svds_f77

subroutine zprimme_svds_f77 (*svals*, *svecs*, *resNorms*, *primme_svds*, *ierr*)

Solve a complex singular value problem using double precision.

All arrays should be hosted on CPU. The computations are performed on CPU (see `magma_zprimme_svds_f77()` for using GPUs).

Parameters

- **svals** (*) *[double precision]* :: (output) array at least of size `numSvals` to store the computed singular values; all processes in a parallel run return this local array with the same values.
- **svecs** (*) *[complex double precision]* :: array at least of size (`mLocal` + `nLocal`) times (`numOrthoConst` + `numSvals`) to store column-wise the (local part for this process of the) computed left singular vectors and the right singular vectors.
- **resNorms** (*) *[double precision]* :: array at least of size `numSvals` to store the residual norms of the computed triplets; all processes in parallel run return this local array with the same values.
- **primme_svds** *[ptr]* :: parameters structure.
- **ierr** *[integer]* :: (output) error indicator; see *Error Codes*.

On input, `svecs` should start with the content of the `numOrthoConst` left vectors, followed by the `initSize` left vectors, followed by the `numOrthoConst` right vectors and followed by the `initSize` right vectors.

On return, the i-th left singular vector starts at svecs((`numOrthoConst` + i - 1) * `mLocal`). The i-th right singular vector starts at svecs((`numOrthoConst` + `initSize`)* `mLocal` + (`numOrthoConst` + i - 1)* `nLocal`). The first vector has i=1.

All internal operations are performed at the same precision than `svecs` unless the user sets `internalPrecision` otherwise.

The type and precision of the callbacks depends on the type and precision of *svecs*. See details for `matrixMatvec`, `applyPreconditioner`, `globalSumReal`, `broadcastReal`, and `convTestFun`.

3.2.5 magma_sprimme_svds_f77

subroutine magma_sprimme_svds_f77 (*svals*, *svecs*, *resNorms*, *primme_svds*, *ierr*)

Solve a real singular value problem using single precision.

Most of the computations are performed on GPU (see `sprimme_svds_f77()` for using only the CPU).

Parameters

- **svals** (*) *[real]* :: (output) CPU array at least of size `numSvals` to store the computed singular values; all processes in a parallel run return this local array with the same values.
- **svecs** (*) *[real]* :: GPU array at least of size (`mLocal` + `nLocal`) times (`numOrthoConst` + `numSvals`) to store column-wise the (local part for this process of the) computed left singular vectors and the right singular vectors.

- **resNorms** (*) *[real]* :: CPU array at least of size *numSvals* to store the residual norms of the computed triplets; all processes in parallel run return this local array with the same values.
- **primme_svds** *[ptr]* :: parameters structure.
- **ierr** *[integer]* :: (output) error indicator; see *Error Codes*.

On input, `svecs` should start with the content of the `numOrthoConst` left vectors, followed by the *initSize* left vectors, followed by the `numOrthoConst` right vectors and followed by the *initSize* right vectors.

On return, the i-th left singular vector starts at svecs((`numOrthoConst` + i - 1) * *mLocal*). The i-th right singular vector starts at svecs((`numOrthoConst` + *initSize*)* *mLocal* + (`numOrthoConst` + i - 1)* *nLocal*). The first vector has i=1.

All internal operations are performed at the same precision than `svecs` unless the user sets *internalPrecision* otherwise.

The type and precision of the callbacks depends on the type and precision of *svecs*. See details for *matrixMatvec*, *applyPreconditioner*, *globalSumReal*, *broadcastReal*, and *convTestFun*.

New in version 3.0.

3.2.6 magma_cprimme_svds_f77

subroutine magma_cprimme_svds_f77 (*svals*, *svecs*, *resNorms*, *primme_svds*, *ierr*)

Solve a complex singular value problem using single precision.

Most of the computations are performed on GPU (see *cprimme_svds_f77()* for using only the CPU).

Parameters

- **svals** (*) *[real]* :: (output) CPU array at least of size *numSvals* to store the computed singular values; all processes in a parallel run return this local array with the same values.
- **svecs** (*) *[complex]* :: GPU array at least of size (*mLocal* + *nLocal*) times (`numOrthoConst` + *numSvals*) to store column-wise the (local part for this process of the) computed left singular vectors and the right singular vectors.
- **resNorms** (*) *[real]* :: CPU array at least of size *numSvals* to store the residual norms of the computed triplets; all processes in parallel run return this local array with the same values.
- **primme_svds** *[ptr]* :: parameters structure.
- **ierr** *[integer]* :: (output) error indicator; see *Error Codes*.

On input, `svecs` should start with the content of the `numOrthoConst` left vectors, followed by the *initSize* left vectors, followed by the `numOrthoConst` right vectors and followed by the *initSize* right vectors.

On return, the i-th left singular vector starts at svecs((`numOrthoConst` + i - 1) * *mLocal*). The i-th right singular vector starts at svecs((`numOrthoConst` + *initSize*)* *mLocal* + (`numOrthoConst` + i - 1)* *nLocal*). The first vector has i=1.

All internal operations are performed at the same precision than `svecs` unless the user sets *internalPrecision* otherwise.

The type and precision of the callbacks depends on the type and precision of *svecs*. See details for `matrixMatvec`, `applyPreconditioner`, `globalSumReal`, `broadcastReal`, and `convTestFun`.

New in version 3.0.

3.2.7 magma_dprimme_svds_f77

`subroutine magma_dprimme_svds_f77` (*svals*, *svecs*, *resNorms*, *primme_svds*, *ierr*)

Solve a real singular value problem using double precision.

Most of the computations are performed on GPU (see `dprimme_svds_f77()` for using only the CPU).

Parameters

- **svals** (*) *[double precision]* :: (output) CPU array at least of size `numSvals` to store the computed singular values; all processes in a parallel run return this local array with the same values.
- **svecs** (*) *[double precision]* :: GPU array at least of size (`mLocal` + `nLocal`) times (`numOrthoConst` + `numSvals`) to store column-wise the (local part for this process of the) computed left singular vectors and the right singular vectors.
- **resNorms** (*) *[double precision]* :: CPU array at least of size `numSvals` to store the residual norms of the computed triplets; all processes in parallel run return this local array with the same values.
- **primme_svds** *[ptr]* :: parameters structure.
- **ierr** *[integer]* :: (output) error indicator; see *Error Codes*.

On input, `svecs` should start with the content of the `numOrthoConst` left vectors, followed by the `initSize` left vectors, followed by the `numOrthoConst` right vectors and followed by the `initSize` right vectors.

On return, the i-th left singular vector starts at svecs((`numOrthoConst` + i - 1) * `mLocal`). The i-th right singular vector starts at svecs((`numOrthoConst` + `initSize`)* `mLocal` + (`numOrthoConst` + i - 1)* `nLocal`). The first vector has i=1.

All internal operations are performed at the same precision than `svecs` unless the user sets `internalPrecision` otherwise.

The type and precision of the callbacks depends on the type and precision of *svecs*. See details for `matrixMatvec`, `applyPreconditioner`, `globalSumReal`, `broadcastReal`, and `convTestFun`.

New in version 3.0.

3.2.8 magma_zprimme_svds_f77

`subroutine magma_zprimme_svds_f77` (*svals*, *svecs*, *resNorms*, *primme_svds*, *ierr*)

Solve a complex singular value problem using double precision.

Most of the computations are performed on GPU (see `zprimme_svds_f77()` for using only the CPU).

Parameters

- **svals** (*) *[double precision]* :: (output) CPU array at least of size `numSvals` to store the computed singular values; all processes in a parallel run return this local array with the same values.

- **svecs** (*) *[complex double precision]* :: GPU array at least of size (*mLocal* + *nLocal*) times (numOrthoConst + *numSvals*) to store column-wise the (local part for this process of the) computed left singular vectors and the right singular vectors.
- **resNorms** (*) *[double precision]* :: CPU array at least of size *numSvals* to store the residual norms of the computed triplets; all processes in parallel run return this local array with the same values.
- **primme_svds** *[ptr]* :: parameters structure.
- **ierr** *[integer]* :: (output) error indicator; see *Error Codes*.

On input, svecs should start with the content of the numOrthoConst left vectors, followed by the *initSize* left vectors, followed by the numOrthoConst right vectors and followed by the *initSize* right vectors.

On return, the i-th left singular vector starts at svecs((numOrthoConst + i - 1) * *mLocal*). The i-th right singular vector starts at svecs((numOrthoConst + *initSize*)* *mLocal* + (numOrthoConst + i - 1)* *nLocal*). The first vector has i=1.

All internal operations are performed at the same precision than svecs unless the user sets *internalPrecision* otherwise.

The type and precision of the callbacks depends on the type and precision of *svecs*. See details for *matrixMatvec*, *applyPreconditioner*, *globalSumReal*, *broadcastReal*, and *convTestFun*.

New in version 3.0.

3.2.9 primme_svds_initialize_f77

subroutine primme_svds_initialize_f77 (*primme_svds*, *ierr*)

Set PRIMME SVDS parameters structure to the default values.

After calling *dprimme_svds_f77()* (or a variant), call *primme_svds_free_f77()* to release allocated resources by PRIMME.

Parameters primme_svds *[ptr]* :: (output) parameters structure.

3.2.10 primme_svds_set_method_f77

subroutine primme_svds_set_method_f77 (*method*, *methodStage1*, *methodStage2*, *primme_svds*, *ierr*)

Set PRIMME SVDS parameters to one of the preset configurations.

Parameters

- **method** *[integer]* :: (input) preset configuration to compute the singular triplets; one of
  - *PRIMME_SVDS_default*, currently set as *PRIMME_SVDS_hybrid*.
  - *PRIMME_SVDS_normalequations*, compute the eigenvectors of A^*A or AA^*.
  - *PRIMME_SVDS_augmented*, compute the eigenvectors of the augmented matrix, $\begin{pmatrix} 0 & A^* \\ A & 0 \end{pmatrix}$.
  - *PRIMME_SVDS_hybrid*, start with *PRIMME_SVDS_normalequations*; use the resulting approximate singular vectors as initial vectors for *PRIMME_SVDS_augmented* if the required accuracy was not achieved.

- **methodStage1** *[primme_preset_method]* :: (input) preset method to compute the eigenpairs at the first stage; see available values at `primme_set_method_f77()`.
- **methodStage2** *[primme_preset_method]* :: (input) preset method to compute the eigenpairs with the second stage of `PRIMME_SVDS_hybrid`; see available values at `primme_set_method_f77()`.
- **primme_svds** *[ptr]* :: (input/output) parameters structure.
- **ierr** *[integer]* :: (output) if 0, successful; if negative, something went wrong.

3.2.11 primme_svds_display_params_f77

subroutine primme_svds_display_params_f77 (*primme_svds*)

Display all printable settings of `primme_svds` into the file descriptor `outputFile`.

Parameters **primme_svds** *[ptr]* :: (input) parameters structure.

3.2.12 primme_svds_free_f77

subroutine primme_svds_free_f77 (*primme_svds*, *ierr*)

Free memory allocated by PRIMME SVDS and delete all values set.

Parameters **primme_svds** *[ptr]* :: (input/output) parameters structure.

3.2.13 primme_svds_set_member_f77

subroutine primme_svds_set_member_f77 (*primme_svds*, *label*, *value*, *ierr*)

Set a value in some field of the parameter structure.

Parameters

- **primme_svds** *[ptr]* :: (input) parameters structure.
- **label** *[integer]* :: field where to set value. One of:

```
PRIMME_SVDS_primme
PRIMME_SVDS_primmeStage2
PRIMME_SVDS_m
PRIMME_SVDS_n
PRIMME_SVDS_matrixMatvec
PRIMME_SVDS_matrixMatvec_type
PRIMME_SVDS_applyPreconditioner
PRIMME_SVDS_applyPreconditioner_type
PRIMME_SVDS_numProcs
PRIMME_SVDS_procID
PRIMME_SVDS_mLocal
PRIMME_SVDS_nLocal
PRIMME_SVDS_commInfo
PRIMME_SVDS_globalSumReal
PRIMME_SVDS_globalSumReal_type
PRIMME_SVDS_broadcastReal
```

PRIMME_SVDS_broadcastReal_type
PRIMME_SVDS_numSvals
PRIMME_SVDS_target
PRIMME_SVDS_numTargetShifts
PRIMME_SVDS_targetShifts
PRIMME_SVDS_method
PRIMME_SVDS_methodStage2
PRIMME_SVDS_matrix
PRIMME_SVDS_preconditioner
PRIMME_SVDS_locking
PRIMME_SVDS_numOrthoConst
PRIMME_SVDS_aNorm
PRIMME_SVDS_eps
PRIMME_SVDS_precondition
PRIMME_SVDS_initSize
PRIMME_SVDS_maxBasisSize
PRIMME_SVDS_maxBlockSize
PRIMME_SVDS_maxMatvecs
PRIMME_SVDS_iseed
PRIMME_SVDS_printLevel
PRIMME_SVDS_outputFile
PRIMME_SVDS_internalPrecision
PRIMME_SVDS_convTestFun
PRIMME_SVDS_convTestFun_type
PRIMME_SVDS_convtest
PRIMME_SVDS_monitorFun
PRIMME_SVDS_monitorFun_type
PRIMME_SVDS_monitor
PRIMME_SVDS_queue
PRIMME_SVDS_stats_numOuterIterations
PRIMME_SVDS_stats_numRestarts
PRIMME_SVDS_stats_numMatvecs
PRIMME_SVDS_stats_numPreconds
PRIMME_SVDS_stats_numGlobalSum
PRIMME_SVDS_stats_numBroadcast
PRIMME_SVDS_stats_volumeGlobalSum
PRIMME_SVDS_stats_volumeBroadcast
PRIMME_SVDS_stats_elapsedTime
PRIMME_SVDS_stats_timeMatvec
PRIMME_SVDS_stats_timePrecond
PRIMME_SVDS_stats_timeOrtho
PRIMME_SVDS_stats_timeGlobalSum
PRIMME_SVDS_stats_timeBroadcast
PRIMME_SVDS_stats_lockingIssue

- **value** :: (input) value to set.

Note: **Don't use** this subroutine inside PRIMME SVDS's callback functions, e.g., *`matrixMatvec`* or *`applyPreconditioner`*, or in functions called by these functions.

3.2.14 primme_svdstop_get_member_f77

subroutine primme_svdstop_get_member_f77 (*primme_svds*, *label*, *value*, *ierr*)

Get the value in some field of the parameter structure.

Parameters

- **primme_svds** *[ptr]* :: (input) parameters structure.
- **label** *[integer]* :: (input) field where to get value. One of the detailed in subroutine `primmesvds_top_set_member_f77()`.
- **value** :: (output) value of the field.

Note: **Don't use** this subroutine inside PRIMME SVDS's callback functions, e.g., *`matrixMatvec`* or *`applyPreconditioner`*, or in functions called by these functions. In those cases use *`primme_svds_get_member_f77()`*.

Note: When `label` is one of `PRIMME_SVDS_matrixMatvec`, `PRIMME_SVDS_applyPreconditioner`, `PRIMME_SVDS_commInfo`, `PRIMME_SVDS_intWork`, `PRIMME_SVDS_realWork`, `PRIMME_SVDS_matrix` and `PRIMME_SVDS_preconditioner`, the returned `value` is a C pointer (`void*`). Use Fortran pointer or other extensions to deal with it. For instance:

```
use iso_c_binding
MPI_Comm comm

comm = MPI_COMM_WORLD
call primme_svds_set_member_f77(primme_svds, PRIMME_SVDS_commInfo, comm)
...
subroutine par_GlobalSumDouble(x,y,k,primme_svds)
use iso_c_binding
implicit none
...
MPI_Comm, pointer :: comm
type(c_ptr) :: pcomm

call primme_svds_get_member_f77(primme_svds, PRIMME_SVDS_commInfo, pcomm)
call c_f_pointer(pcomm, comm)
call MPI_Allreduce(x,y,k,MPI_DOUBLE,MPI_SUM,comm,ierr)
```

Most users would not need to retrieve these pointers in their programs.

3.2.15 primme_svds_get_member_f77

subroutine primme_svds_get_member_f77 (*primme_svds*, *label*, *value*, *ierr*)

Get the value in some field of the parameter structure.

Parameters

- **primme_svds** *[ptr]* :: (input) parameters structure.
- **label** *[integer]* :: (input) field where to get value. One of the detailed in subroutine `primme_svdstop_set_member_f77()`.
- **value** :: (output) value of the field.

Note: Use this subroutine exclusively inside PRIMME SVDS's callback functions, e.g., *`matrixMatvec`* or *`applyPreconditioner`*, or in functions called by these functions. Otherwise, e.g., from the main program, use the subroutine *`primme_svdstop_get_member_f77()`*.

Note: When `label` is one of `PRIMME_SVDS_matrixMatvec`, `PRIMME_SVDS_applyPreconditioner`, `PRIMME_SVDS_commInfo`, `PRIMME_SVDS_intWork`, `PRIMME_SVDS_realWork`, `PRIMME_SVDS_matrix` and `PRIMME_SVDS_preconditioner`, the returned `value` is a C pointer (`void*`). Use Fortran pointer or other extensions to deal with it. For instance:

```
use iso_c_binding
MPI_Comm comm

comm = MPI_COMM_WORLD
call primme_svds_set_member_f77(primme_svds, PRIMME_SVDS_commInfo, comm)
...
subroutine par_GlobalSumDouble(x,y,k,primme_svds)
use iso_c_binding
implicit none
...
MPI_Comm, pointer :: comm
type(c_ptr) :: pcomm

call primme_svds_get_member_f77(primme_svds, PRIMME_SVDS_commInfo, pcomm)
call c_f_pointer(pcomm, comm)
call MPI_Allreduce(x,y,k,MPI_DOUBLE,MPI_SUM,comm,ierr)
```

Most users would not need to retrieve these pointers in their programs.

3.3 FORTRAN 90 Library Interface

New in version 3.0.

The next enumerations and functions are declared in `primme_f90.inc`.

subroutine primme_svds_matvec (*x*, *ldx*, *y*, *ldy*, *blockSize*, *mode*, *primme_svds*, *ierr*)

Abstract interface for the callbacks `matrixMatvec` and `applyPreconditioner`.

Parameters

- **x** (ldx,*) *[type(*),in]* :: matrix with `blockSize` columns in column-major order with leading dimension `ldx`.
- **ldx** *[c_int64_t]* :: the leading dimension of the array `x`.
- **y** (ldy,*) *[type(*),out]* :: matrix with `blockSize` columns in column-major order with leading dimension `ldy`.
- **ldy** *[c_int64_t]* :: the leading dimension of the array `y`.
- **blockSize** *[c_int,in]* :: number of columns in `x` and `y`.
- **mode** *[c_int,in]* :: a flag.
- **primme_svds** *[c_ptr,in]* :: parameters structure created by `primme_svds_params_create()`.
- **ierr** *[c_int,out]* :: output error code; if it is set to non-zero, the current call to PRIMME will stop.

See more details about the precision and type and dimension for *x* and *y*, and the meaning of *mode* in the documentation of the callbacks.

3.3.1 primme_svds_params_create

function primme_svds_params_create ()

Allocate and initialize a parameters structure to the default values.

After calling `xprimme_svds()` (or a variant), call `primme_svds_params_destroy()` to release allocated resources by PRIMME.

Return primme_svds_params_create *[c_ptr]* :: pointer to a parameters structure.

3.3.2 primme_svds_set_method

function primme_svds_set_method (*method*, *methodStage1*, *methodStage2*, *primme_svds*)

Set PRIMME SVDS parameters to one of the preset configurations.

Parameters

- **method** *[integer]* :: (input) preset configuration to compute the singular triplets; one of
  - `PRIMME_SVDS_default`, currently set as `PRIMME_SVDS_hybrid`.
  - `PRIMME_SVDS_normalequations`, compute the eigenvectors of A^*A or AA^*.
  - `PRIMME_SVDS_augmented`, compute the eigenvectors of the augmented matrix, $\begin{pmatrix} 0 & A^* \\ A & 0 \end{pmatrix}$.

  - *PRIMME_SVDS_hybrid*, start with *PRIMME_SVDS_normalequations*; use the resulting approximate singular vectors as initial vectors for *PRIMME_SVDS_augmented* if the required accuracy was not achieved.

- **methodStage1** *[primme_preset_method]* :: (input) preset method to compute the eigenpairs at the first stage; see available values at *primme_set_method()*.
- **methodStage2** *[primme_preset_method]* :: (input) preset method to compute the eigenpairs with the second stage of *PRIMME_SVDS_hybrid*; see available values at *primme_set_method()*.
- **primme_svds** *[ptr]* :: (input/output) parameters structure.
- **ierr** *[integer]* :: (output) if 0, successful; if negative, something went wrong.

3.3.3 xprimme_svds

function xprimme_svds (*svals*, *svecs*, *resNorms*, *primme_svds*)

Solve a real or complex singular value problem.

All arrays should be hosted on CPU. The computations are performed on CPU (see *magma_xprimme_svds()* for using GPUs).

Parameters

- **svals** (*) *[out]* :: array at least of size *numSvals* to store the computed singular values; all processes in a parallel run return this local array with the same values.
- **svecs** (*) :: array at least of size (*mLocal* + *nLocal*) times (numOrthoConst + *numSvals*) to store column-wise the (local part for this process of the) computed left singular vectors and the right singular vectors.
- **resNorms** (*) *[out]* :: array at least of size *numSvals* to store the residual norms of the computed triplets; all processes in parallel run return this local array with the same values.
- **primme** *[c_ptr,in]* :: parameters structure created by primme_params_create_svds().

Return xprimme_svds *[c_int]* :: error indicator; see *Error Codes*.

The arrays svals, svecs, and resNorms should have the same kind.

On input, svecs should start with the content of the numOrthoConst left vectors, followed by the *initSize* left vectors, followed by the numOrthoConst right vectors and followed by the *initSize* right vectors.

On return, the i-th left singular vector starts at svecs((numOrthoConst + i - 1) * *mLocal*). The i-th right singular vector starts at svecs((numOrthoConst + *initSize*)* *mLocal* + (numOrthoConst + i - 1)* *nLocal*). The first vector has i=1.

All internal operations are performed at the same precision than svecs unless the user sets *internalPrecision* otherwise.

The type and precision of the callbacks depends on the type and precision of svecs. See details for *matrixMatvec*, *applyPreconditioner*, *globalSumReal*, *broadcastReal*, and *convTestFun*.

3.3.4 magma_xprimme_svds

function magma_xprimme_svds (*svals*, *svecs*, *resNorms*, *primme_svds*)

Solve a real or complex singular value problem.

Most of the computations are performed on GPU (see `xprimme_svds()` for using only the CPU).

Parameters

- **svals** (*) *[out]* :: CPU array at least of size *`numSvals`* to store the computed singular values; all processes in a parallel run return this local array with the same values.
- **svecs** (*) :: GPU array at least of size (*`mLocal`* + *`nLocal`*) times (`numOrthoConst` + *`numSvals`*) to store column-wise the (local part for this process of the) computed left singular vectors and the right singular vectors.
- **resNorms** (*) *[out]* :: CPU array at least of size *`numSvals`* to store the residual norms of the computed triplets; all processes in parallel run return this local array with the same values.
- **primme** *[c_ptr,in]* :: parameters structure created by `primme_params_create_svds()`.

Return magma_xprimme_svds *[c_int]* :: error indicator; see *Error Codes*.

The arrays `svals`, `svecs`, and `resNorms` should have the same kind.

On input, `svecs` should start with the content of the `numOrthoConst` left vectors, followed by the *`initSize`* left vectors, followed by the `numOrthoConst` right vectors and followed by the *`initSize`* right vectors.

On return, the i-th left singular vector starts at svecs((`numOrthoConst` + i - 1) * *`mLocal`*). The i-th right singular vector starts at svecs((`numOrthoConst` + *`initSize`*)* *`mLocal`* + (`numOrthoConst` + i - 1)* *`nLocal`*). The first vector has i=1.

All internal operations are performed at the same precision than `svecs` unless the user sets *`internalPrecision`* otherwise.

The type and precision of the callbacks depends on the type and precision of `svecs`. See details for *`matrixMatvec`*, *`applyPreconditioner`*, *`globalSumReal`*, *`broadcastReal`*, and *`convTestFun`*.

3.3.5 primme_svds_params_destroy

function primme_svds_params_destroy (*primme_svds*)

Free memory allocated by PRIMME associated to a parameters structure created with *`primme_svds_params_create()`*.

Parameters primme_svds *[c_ptr]* :: parameters structure.

Return primme_svds_params_destroy :: nonzero value if the call is not successful.

3.3.6 primme_svds_set_member

function primme_svds_set_member (*primme_svds*, *label*, *value*)

Set a value in some field of the parameter structure.

Parameters

- **primme_svds** *[ptr]* :: (input) parameters structure.
- **label** *[integer]* :: field where to set value. One of:

PRIMME_SVDS_primme
PRIMME_SVDS_primmeStage2
PRIMME_SVDS_m
PRIMME_SVDS_n
PRIMME_SVDS_matrixMatvec
PRIMME_SVDS_matrixMatvec_type
PRIMME_SVDS_applyPreconditioner
PRIMME_SVDS_applyPreconditioner_type
PRIMME_SVDS_numProcs
PRIMME_SVDS_procID
PRIMME_SVDS_mLocal
PRIMME_SVDS_nLocal
PRIMME_SVDS_commInfo
PRIMME_SVDS_globalSumReal
PRIMME_SVDS_globalSumReal_type
PRIMME_SVDS_broadcastReal
PRIMME_SVDS_broadcastReal_type
PRIMME_SVDS_numSvals
PRIMME_SVDS_target
PRIMME_SVDS_numTargetShifts
PRIMME_SVDS_targetShifts
PRIMME_SVDS_method
PRIMME_SVDS_methodStage2
PRIMME_SVDS_matrix
PRIMME_SVDS_preconditioner
PRIMME_SVDS_locking
PRIMME_SVDS_numOrthoConst
PRIMME_SVDS_aNorm
PRIMME_SVDS_eps
PRIMME_SVDS_precondition
PRIMME_SVDS_initSize
PRIMME_SVDS_maxBasisSize
PRIMME_SVDS_maxBlockSize
PRIMME_SVDS_maxMatvecs
PRIMME_SVDS_iseed
PRIMME_SVDS_printLevel
PRIMME_SVDS_outputFile
PRIMME_SVDS_internalPrecision

PRIMME_SVDS_convTestFun
PRIMME_SVDS_convTestFun_type
PRIMME_SVDS_convtest
PRIMME_SVDS_monitorFun
PRIMME_SVDS_monitorFun_type
PRIMME_SVDS_monitor
PRIMME_SVDS_queue
PRIMME_SVDS_stats_numOuterIterations
PRIMME_SVDS_stats_numRestarts
PRIMME_SVDS_stats_numMatvecs
PRIMME_SVDS_stats_numPreconds
PRIMME_SVDS_stats_numGlobalSum
PRIMME_SVDS_stats_numBroadcast
PRIMME_SVDS_stats_volumeGlobalSum
PRIMME_SVDS_stats_volumeBroadcast
PRIMME_SVDS_stats_elapsedTime
PRIMME_SVDS_stats_timeMatvec
PRIMME_SVDS_stats_timePrecond
PRIMME_SVDS_stats_timeOrtho
PRIMME_SVDS_stats_timeGlobalSum
PRIMME_SVDS_stats_timeBroadcast
PRIMME_SVDS_stats_lockingIssue

- **value** :: (input) value to set. The allowed types are *c_int64*, *c_double*, *c_ptr*, *c_funptr* and *procedure(primme_svds_matvec)*

Return primme_svds_set_member *[c_int]* :: nonzero value if the call is not successful.

Examples:

```
type(c_ptr) :: primme_svds
integer :: ierr
...

integer(c_int64_t) :: m                 = 100
ierr = primme_svds_set_member(primme_svds, PRIMME_SVDS_m, m)
ierr = primme_svds_set_member(primme_svds, PRIMME_SVDS_n, m)

real(c_double) :: tol              = 1.0D-12
ierr = primme_svds_set_member(primme, PRIMME_SVDS_eps, tol)

integer(c_int64_t), parameter :: numTargetShifts = 2
real(c_double) :: TargetShifts(numTargetShifts) = (/3.0D0, 5.1D0/)
ierr = primme_svds_set_member(primme_svds, PRIMME_SVDS_
↪numTargetShifts, numTargetShifts)
ierr = primme_svds_set_member(primme_svds, PRIMME_SVDS_targetShifts,␣
↪TargetShifts)

ierr = primme_svds_set_member(primme_svds, PRIMME_SVDS_target, primme_
↪svds_closest_abs)

procedure(primme_svds_matvec) :: MV, ApplyPrecon
```

(continues on next page)

(continued from previous page)

```
ierr = primme_svds_set_member(primme_svds, PRIMME_SVDS_matrixMatvec,␣
↪MV)

ierr = primme_svds_set_member(primme_svds, PRIMME_SVDS_
↪applyPreconditioner,
                               c_funloc(ApplyPrecon))

type(c_ptr) :: primme
ierr = primme_svds_get_member(primme_svds, PRIMME_SVDS_primme, primme)
ierr = primme_set_member(primme, PRIMME_correctionParams_precondition,
                          1_c_int64_t)
```

3.3.7 primme_get_member

function primme_svds_get_member (*primme*, *label*, *value*)

Get the value in some field of the parameter structure.

Parameters

- **primme** *[c_ptr,in]* :: parameters structure created by *primme_svds_params_create()*.
- **label** *[integer,in]* :: field where to get value. One of the detailed in function *primme_svds_set_member()*.
- **value** *[out]* :: value of the field. The allowed types are *c_int64*, *c_double*, and *c_ptr*.

Return **primme_svds_get_member** *[c_int]* :: nonzero value if the call is not successful.

Examples:

```
type(c_ptr) :: primme_svds
integer :: ierr
...

integer(c_int64_t) :: m
ierr = primme_svds_get_member(primme_svds, PRIMME_SVDS_m, m)

real(c_double) :: aNorm
ierr = primme_svds_get_member(primme_svds, PRIMME_SVDS_aNorm, aNorm)

type(c_ptr) :: primme
ierr = primme_svds_get_member(primme_svds, PRIMME_SVDS_primme, primme)
ierr = primme_set_member(primme, PRIMME_correctionParams_precondition,
                          1_c_int64_t)
```

3.4 Python Interface

`primme.svds`(*A*, *k=6*, *ncv=None*, *tol=0*, *which='LM'*, *v0=None*, *maxiter=None*, *return_singular_vectors=True*, *precAHA=None*, *precAAH=None*, *precAug=None*, *u0=None*, *orthou0=None*, *orthov0=None*, *return_stats=False*, *maxBlockSize=0*, *method=None*, *methodStage1=None*, *methodStage2=None*, *return_history=False*, *convtest=None*, ***kargs*)

Compute k singular values and vectors of the matrix A.

Parameters

- **A** (*{sparse matrix, LinearOperator}*) – Array to compute the SVD on, of shape (M, N)
- **k** (*int, optional*) – Number of singular values and vectors to compute. Must be 1 <= k < min(A.shape).
- **ncv** (*int, optional*) – The maximum size of the basis
- **tol** (*float, optional*) – Tolerance for singular values. Zero (default) means 10**4 times the machine precision.

  A triplet `(u,sigma,v)` is marked as converged when (||A*v - sigma*u||**2 + ||A.H*u - sigma*v||**2)**.5 is less than "tol" * ||A||, or close to the minimum tolerance that the method can achieve. See the note.

  The value is ignored if convtest is provided.
- **which** (*str ['LM' | 'SM'] or number, optional*) – Which *k* singular values to find:
  - 'LM' : largest singular values
  - 'SM' : smallest singular values
  - number : closest singular values to (referred as sigma later)
- **u0** (*ndarray, optional*) – Initial guesses for the left singular vectors.

  If only u0 or v0 is provided, the other is computed. If both are provided, u0 and v0 should have the same number of columns.
- **v0** (*ndarray, optional*) – Initial guesses for the right singular vectors.
- **maxiter** (*int, optional*) – Maximum number of matvecs with A and A.H.
- **precAHA** (*{N x N matrix, array, sparse matrix, LinearOperator}, optional*) – Approximate inverse of (A.H*A - sigma**2*I). If provided and M>=N, it usually accelerates the convergence.
- **precAAH** (*{M x M matrix, array, sparse matrix, LinearOperator}, optional*) – Approximate inverse of (A*A.H - sigma**2*I). If provided and M<N, it usually accelerates the convergence.
- **precAug** (*{(M+N) x (M+N) matrix, array, sparse matrix, LinearOperator}, optional*) – Approximate inverse of ([zeros() A.H; zeros() A] - sigma*I).
- **orthou0** (*ndarray, optional*) – Left orthogonal vector constrain.

  Seek singular triplets orthogonal to orthou0 and orthov0. The provided vectors *should* be orthonormal. If only orthou0 or orthov0 is provided, the other is computed. Useful to avoid converging to previously computed solutions.
- **orthov0** (*ndarray, optional*) – Right orthogonal vector constrain. See orthou0.

- **`maxBlockSize`** (*`int, optional`*) – Maximum number of vectors added at every iteration.
- **`convtest`** (*`callable`*) – User-defined function to mark an approximate singular triplet as converged.

  The function is called as convtest(sval, svecleft, svecright, resNorm) and returns True if the triplet with value *sval*, left vector *svecleft*, right vector *svecright*, and residual norm *resNorm* is considered converged.
- **`return_stats`** (*`bool, optional`*) – If True, the function returns extra information (see stats in Returns).
- **`return_history`** (*`bool, optional`*) – If True, the function returns performance information at every iteration

Returns

- **u** (*ndarray, shape=(M, k), optional*) – Unitary matrix having left singular vectors as columns. Returned if *return_singular_vectors* is True.
- **s** (*ndarray, shape=(k,)*) – The singular values.
- **vt** (*ndarray, shape=(k, N), optional*) – Unitary matrix having right singular vectors as rows. Returned if *return_singular_vectors* is True.
- **stats** (*dict, optional (if return_stats)*) – Extra information reported by PRIMME:
  - ”numOuterIterations”: number of outer iterations
  - ”numRestarts”: number of restarts
  - ”numMatvecs”: number of matvecs with A and A.H
  - ”numPreconds”: cumulative number of applications of precAHA, precAAH and precAug
  - ”elapsedTime”: time that took
  - ”rnorms” : (||A*v[:,i] - sigma[i]*u[:,i]||**2 + ||A.H*u[:,i] - sigma[i]*v[:,i]||**2)**.5
  - ”hist” : (if return_history) report at every outer iteration of:
    - ”elapsedTime”: time spent up to now
    - ”numMatvecs”: number of A*v and A.H*v spent up to now
    - ”nconv”: number of converged triplets
    - ”sval”: singular value of the first unconverged triplet
    - ”resNorm”: residual norm of the first unconverged triplet

Notes

The default method used is the hybrid method, which first solves the equivalent eigenvalue problem A.H*A or A*A.H (normal equations) and then refines the solution solving the augmented problem. The minimum tolerance that this method can achieve is ||A||*epsilon, where epsilon is the machine precision. However it may not return triplets with singular values smaller than ||A||*epsilon if “tol” is smaller than ||A||*epsilon/sigma.

This function is a wrapper to PRIMME functions to find singular values and vectors[1].

[1] PRIMME Software, https://github.com/primme/primme

References

See also:

primme.eigsh() eigenvalue decomposition for a sparse symmetrix/complex Hermitian matrix A

scipy.sparse.linalg.eigs() eigenvalues and eigenvectors for a general (nonsymmetric) matrix A

Examples

```
>>> import primme, scipy.sparse
>>> A = scipy.sparse.spdiags(range(1, 11), [0], 100, 10) # sparse diag. rect.␣
↪matrix
>>> svecs_left, svals, svecs_right = primme.svds(A, 3, tol=1e-6, which='LM')
>>> svals # the three largest singular values of A
array([10., 9., 8.])
```

```
>>> import primme, scipy.sparse, numpy as np
>>> A = scipy.sparse.rand(10000, 100, random_state=10)
>>> prec = scipy.sparse.spdiags(np.reciprocal(A.multiply(A).sum(axis=0)),
...           [0], 100, 100) # square diag. preconditioner
>>> # the three smallest singular values of A, using preconditioning
>>> svecs_left, svals, svecs_right = primme.svds(A, 3, which='SM', tol=1e-6,␣
↪precAHA=prec)
>>> ["%.5f" % x for x in svals.flat]
['4.57263', '4.78752', '4.82229']
```

```
>>> # Giving the matvecs as functions
>>> import primme, scipy.sparse, numpy as np
>>> Bdiag = np.arange(0, 100).reshape((100,1))
>>> Bdiagr = np.concatenate((np.arange(0, 100).reshape((100,1)).astype(np.
↪float32), np.zeros((100,1), dtype=np.float32)), axis=None).reshape((200,1))
>>> def Bmatmat(x):
...     if len(x.shape) == 1: x = x.reshape((100,1))
...     return np.vstack((Bdiag * x, np.zeros((100, x.shape[1]), dtype=np.
↪float32)))
...
>>> def Brmatmat(x):
...     if len(x.shape) == 1: x = x.reshape((200,1))
...     return (Bdiagr * x)[0:100,:]
...
>>> B = scipy.sparse.linalg.LinearOperator((200,100), matvec=Bmatmat,␣
↪matmat=Bmatmat, rmatvec=Brmatmat, dtype=np.float32)
>>> svecs_left, svals, svecs_right = primme.svds(B, 5, which='LM', tol=1e-6)
>>> svals
array([99., 98., 97., 96., 95.])
```

3.5 MATLAB Interface

function [varargout] = primme_svds(varargin)

`primme_svds()` finds a few singular values and vectors of a matrix `A` by calling PRIMME. `A` is typically large and sparse.

`S = primme_svds(A)` returns a vector with the 6 largest singular values of `A`.

`S = primme_svds(AFUN,M,N)` accepts the function handle `AFUN` to perform the matrix vector products with an M-by-N matrix `A`. `AFUN(X,'notransp')` returns `A*X` while `AFUN(X,'transp')` returns `A'*X`. In all the following, `A` can be replaced by `AFUN,M,N`.

`S = primme_svds(A,k)` computes the `k` largest singular values of `A`.

`S = primme_svds(A,k,sigma)` computes the `k` singular values closest to the scalar shift `sigma`.

- If `sigma` is a vector, find the singular value `S(i)` closest to each `sigma(i)`, for `i<=k`.
- If `sigma` is `'L'`, it computes the largest singular values.
- if `sigma` is `'S'`, it computes the smallest singular values.

`S = primme_svds(A,k,sigma,OPTIONS)` specifies extra solver parameters. Some default values are indicated in brackets {}:

- *`aNorm`*: estimation of the 2-norm of `A` {0.0 (estimate the norm internally)}
- `tol`: convergence tolerance `NORM([A*V-U*S;A'*U-V*S]) <= tol * NORM(A)` (see *`eps`*) { `1e-10` for double precision and `1e-3` for single precision}
- `maxit`: maximum number of matvecs with `A` and `A'` (see *`maxMatvecs`*) {inf}
- `p`: maximum basis size (see *`maxBasisSize`*)
- `reportLevel`: reporting level (0-3) (see HIST) {no reporting 0}
- `display`: whether displaying reporting on screen (see HIST) {0 if HIST provided}
- `isreal`: if 0, the matrix is complex; else it's real {0: complex}
- `isdouble`: if 0, the matrix is single; else it's double {1: double}
- `method`: which equivalent eigenproblem to solve
  - '*`primme_svds_normalequations`*': `A'*A` or `A*A'`
  - '*`primme_svds_augmented`*': `[0 A';A 0]`
  - '*`primme_svds_hybrid`*': first normal equations and then augmented (default)
- `u0`: initial guesses to the left singular vectors (see *`initSize`*) {[]}
- `v0`: initial guesses to the right singular vectors {[]}
- `orthoConst`: external orthogonalization constraints (see `numOrthoConst`) {[]}
- *`locking`*: 1, hard locking; 0, soft locking
- *`maxBlockSize`*: maximum block size
- *`iseed`*: random seed
- *`primme`*: options for first stage solver
- *`primmeStage2`*: options for second stage solver

- *convTestFun*: function handler with an alternative convergence criterion. If `FUN(SVAL,LSVEC, RSVEC,RNORM)` returns a nonzero value, the triplet `(SVAL,LSVEC,RSVEC)` with residual norm `RNORM` is considered converged.

The available options for `OPTIONS.primme` and `primmeStage2` are the same as `primme_eigs()`, plus the option `'method'`.

`S = primme_svds(A,k,sigma,OPTIONS,P)` applies a preconditioner `P` as follows:

- If `P` is a matrix it applies `P\X` and `P'\X` to approximate `A\X` and `A'\X`.
- If `P` is a function handle, `PFUN`, `PFUN(X,'notransp')` returns `P\X` and `PFUN(X,'transp')` returns `P'\X`, approximating `A\X` and `A'\X` respectively.
- **If `P` is a `struct`, it can have one or more of the following fields:** `P.AHA\X` or `P.AHA(X)` returns an approximation of `(A'*A)\X`, `P.AAH\X` or `P.AAH(X)` returns an approximation of `(A*A')\X`, `P.aug\X` or `P.aug(X)` returns an approximation of `[zeros(N,N) A';A zeros(M,M)]\X`.
- If `P` is `[]` then no preconditioner is applied.

`S = primme_svds(A,k,sigma,OPTIONS,P1,P2)` applies a factorized preconditioner:

- If both `P1` and `P2` are nonempty, apply `(P1*P2)\X` to approximate `A\X`.
- If `P1` is `[]` and `P2` is nonempty, then `(P2'*P2)\X` approximates `A'*A`. `P2` can be the R factor of an (incomplete) QR factorization of `A` or the L factor of an (incomplete) LL' factorization of `A'*A` (RIF).
- If both `P1` and `P2` are `[]` then no preconditioner is applied.

`[U,S,V] = primme_svds(...)` returns also the corresponding singular vectors. If `A` is M-by-N and `k` singular triplets are computed, then `U` is M-by-k with orthonormal columns, `S` is k-by-k diagonal, and `V` is N-by-k with orthonormal columns.

`[S,R] = primme_svds(...)`

`[U,S,V,R] = primme_svds(...)` returns the residual norm of each `k` triplet, `NORM([A*V(:,i)-S(i,i)*U(:,i); A'*U(:,i)-S(i,i)*V(:,i)])`.

`[U,S,V,R,STATS] = primme_svds(...)` returns how many times `A` and `P` were used and elapsed time. The application of `A` is counted independently from the application of `A'`.

`[U,S,V,R,STATS,HIST] = primme_svds(...)` returns the convergence history, instead of printing it. Every row is a record, and the columns report:

- `HIST(:,1)`: number of matvecs
- `HIST(:,2)`: time
- `HIST(:,3)`: number of converged/locked triplets
- `HIST(:,4)`: stage
- `HIST(:,5)`: block index
- `HIST(:,6)`: approximate singular value
- `HIST(:,7)`: residual norm
- `HIST(:,8)`: QMR residual norm

`OPTS.reportLevel` controls the granularity of the record. If `OPTS.reportLevel == 1`, `HIST` has one row per converged eigenpair and only the first three columns together with the fifth and the sixth are reported. If `OPTS.reportLevel == 2`, `HIST` has one row per outer iteration and converged value, and only the first six columns are reported. Otherwise `HIST` has one row per QMR iteration, outer iteration and converged value, and all columns are reported.

The convergence history is displayed if `OPTS.reportLevel > 0` and either `HIST` is not returned or `OPTS.display == 1`.

Examples:

```
A = diag(1:50); A(200,1) = 0; % rectangular matrix of size 200x50

s = primme_svds(A,10) % the 10 largest singular values

s = primme_svds(A,10,'S') % the 10 smallest singular values

s = primme_svds(A,10,25) % the 10 closest singular values to 25

opts = struct();
opts.tol = 1e-4; % set tolerance
opts.method = 'primme_svds_normalequations' % set svd solver method
opts.primme.method = 'DEFAULT_MIN_TIME' % set first stage eigensolver method
opts.primme.maxBlockSize = 2; % set block size for first stage
[u,s,v] = primme_svds(A,10,'S',opts); % find 10 smallest svd triplets

opts.orthoConst = {u,v};
[s,rnorms] = primme_svds(A,10,'S',opts) % find another 10

% Compute the 5 smallest singular values of a rectangular matrix using
% Jacobi preconditioner on (A'*A)
A = sparse(diag(1:50) + diag(ones(49,1), 1));
A(200,50) = 1;  % size(A)=[200 50]
P = diag(sum(abs(A).^2));
precond.AHA = @(x)P\x;
s = primme_svds(A,5,'S',[],precond) % find the 5 smallest values

% Estimation of the smallest singular value
A = diag([1 repmat(2,1,1000) 3:100]);
[~,sval,~,rnorm] = primme_svds(A,1,'S',struct('convTestFun',@(s,u,v,r)r<s*.1));
sval - rnorm % approximate smallest singular value
```

See also: MATLAB svds, `primme_eigs()`

3.6 Parameter Description

3.6.1 primme_svds_params

`type primme_svds_params`

Structure to set the problem matrix and the solver options.

PRIMME_INT **`m`**

Number of rows of the matrix.

Input/output:

`primme_svds_initialize()` sets this field to 0;

this field is read by `dprimme_svds()`.

PRIMME_INT **`n`**

Number of columns of the matrix.

Input/output:

`primme_svds_initialize()` sets this field to 0;

this field is read by `dprimme_svds()`.

void (***`matrixMatvec`**)(void *x, *PRIMME_INT* ldx, void *y, *PRIMME_INT* ldy, int *blockSize, int *transpose, *primme_svds_params* *primme_svds, int *ierr)

Block matrix-multivector multiplication, $y = Ax$ if `transpose` is zero, and $y = A^*x$ otherwise.

Parameters

- **`x`** – input array.
- **`ldx`** – leading dimension of `x`.
- **`y`** – output array.
- **`ldy`** – leading dimension of `y`.
- **`blockSize`** – number of columns in `x` and `y`.
- **`transpose`** – if non-zero, the transpose A should be applied.
- **`primme_svds`** – parameters structure.
- **`ierr`** – output error code; if it is set to non-zero, the current call to PRIMME will stop.

If `transpose` is zero, then `x` and `y` are arrays of dimensions `nLocal` x `blockSize` and `mLocal` x `blockSize` respectively. Elsewhere they have dimensions `mLocal` x `blockSize` and `nLocal` x `blockSize`. Both arrays are in column-major order (elements in the same column with consecutive row indices are consecutive in memory).

The actual type of `x` and `y` matches the type of `evecs` of the calling `dprimme_svds()` (or a variant), unless `matrixMatvec_type` sets another precision.

Input/output:

`primme_svds_initialize()` sets this field to NULL;

this field is read by `dprimme_svds()` and `zprimme_svds()`.

Note: Integer arguments are passed by reference to make easier the interface to other languages (like Fortran).

primme_op_datatype **matrixMatvec_type**

Precision of the vectors x and y passed to *matrixMatvec_type*.

If it is primme_op_default, the vectors' type matches the calling *dprimme_svds()* (or a variant). Otherwise, the precision is half, single, or double, if *matrixMatvec_type* is primme_half, primme_float or primme_double respectively.

Input/output:

primme_svds_initialize() sets this field to primme_op_default;

this field is read by *dprimme_svds()*, and if it is primme_op_default it is set to the value that matches the precision of calling function.

New in version 3.0.

void (***applyPreconditioner**)(void *x, *PRIMME_INT* ldx, void *y, *PRIMME_INT* ldy, int *blockSize, int *mode, *primme_svds_params* *primme_svds, int *ierr)

Block preconditioner-multivector application, $y = M^{-1}x$ for finding singular values close to σ. Depending on mode, M is expected to be an approximation of the following operators:

- primme_svds_op_AtA: $M \approx A^*Ax - \sigma^2 I$,
- primme_svds_op_AAt: $M \approx AA^*x - \sigma^2 I$,
- primme_svds_op_augmented: $M \approx \begin{pmatrix} 0 & A^* \\ A & 0 \end{pmatrix} - \sigma I$.

Parameters

- **x** – input array.
- **ldx** – leading dimension of x.
- **y** – output array.
- **ldy** – leading dimension of y.
- **blockSize** – number of columns in x and y.
- **mode** – one of primme_svds_op_AtA, primme_svds_op_AAt or primme_svds_op_augmented.
- **primme_svds** – parameters structure.
- **ierr** – output error code; if it is set to non-zero, the current call to PRIMME will stop.

If mode is primme_svds_op_AtA, then x and y are arrays of dimensions *nLocal* x blockSize; if mode is primme_svds_op_AAt, they are *mLocal* x blockSize; and otherwise they are (*mLocal* + *nLocal*) x blockSize. Both arrays are in column-major order (elements in the same column with consecutive row indices are consecutive in memory).

The actual type of x and y matches the type of evecs of the calling *dprimme_svds()* (or a variant), unless *matrixMatvec_type* sets another precision.

Input/output:

primme_svds_initialize() sets this field to NULL;

this field is read by *dprimme_svds()* and *zprimme_svds()*.

primme_op_datatype **applyPreconditioner_type**

Precision of the vectors x and y passed to *applyPreconditioner_type*.

If it is `primme_op_default`, the vectors' type matches the calling *`dprimme_svds()`* (or a variant). Otherwise, the precision is half, single, or double, if *`applyPreconditioner_type`* is `primme_half`, `primme_float` or `primme_double` respectively.

Input/output:

> *`primme_svds_initialize()`* sets this field to `primme_op_default`;
>
> this field is read by *`dprimme_svds()`*, and if it is `primme_op_default` it is set to the value that matches the precision of calling function.

New in version 3.0.

int **`numProcs`**

Number of processes calling *`dprimme_svds()`* or *`zprimme_svds()`* in parallel.

Input/output:

> *`primme_svds_initialize()`* sets this field to 1;
>
> this field is read by *`dprimme_svds()`* and *`zprimme_svds()`*.

int **`procID`**

The identity of the local process within a parallel execution calling *`dprimme_svds()`* or *`zprimme_svds()`*. Only the process with id 0 prints information.

Input/output:

> *`primme_svds_initialize()`* sets this field to 0;
>
> *`dprimme_svds()`* sets this field to 0 if *`numProcs`* is 1;
>
> this field is read by *`dprimme_svds()`* and *`zprimme_svds()`*.

PRIMME_INT **`mLocal`**

Number of local rows on this process. The value depends on how the matrix and preconditioner is distributed along the processes.

Input/output:

> *`primme_svds_initialize()`* sets this field to -1;
>
> *`dprimme_svds()`* sets this field to *`m`* if *`numProcs`* is 1;
>
> this field is read by *`dprimme_svds()`* and *`zprimme_svds()`*.

See also: *`matrixMatvec`* and *`applyPreconditioner`*.

PRIMME_INT **`nLocal`**

Number of local columns on this process. The value depends on how the matrix and preconditioner is distributed along the processes.

Input/output:

> *`primme_svds_initialize()`* sets this field to -1;
>
> *`dprimme_svds()`* sets this field to to *`n`* if *`numProcs`* is 1;
>
> this field is read by *`dprimme_svds()`* and *`zprimme_svds()`*.

void ***`commInfo`**

A pointer to whatever parallel environment structures needed. For example, with MPI, it could be a pointer to the MPI communicator. PRIMME does not use this. It is available for possible use in user functions defined in *`matrixMatvec`*, *`applyPreconditioner`*, *`globalSumReal`*, and *`broadcastReal`*.

Input/output:

> *`primme_svds_initialize()`* sets this field to NULL;

void (***globalSumReal**) (double *sendBuf, double *recvBuf, int *count, *primme_svds_params* *primme_svds, int *ierr)

Global sum reduction function. No need to set for sequential programs.

Parameters

- **sendBuf** – array of size `count` with the local input values.
- **recvBuf** – array of size `count` with the global output values so that the i-th element of recvBuf is the sum over all processes of the i-th element of `sendBuf`.
- **count** – array size of `sendBuf` and `recvBuf`.
- **primme_svds** – parameters structure.
- **ierr** – output error code; if it is set to non-zero, the current call to PRIMME will stop.

The actual type of `sendBuf` and `recvBuf` depends on which function is being calling. For `dprimme_svds()` and `zprimme_svds()` it is `double`, and for `sprimme_svds()` and `cprimme_svds()` it is `float`. Note that `count` is the number of values of the actual type.

Input/output:

`primme_svds_initialize()` sets this field to an internal function;

`dprimme_svds()` sets this field to an internal function if `numProcs` is 1 and `globalSumReal` is NULL;

this field is read by `dprimme_svds()` and `zprimme_svds()`.

When MPI is used, this can be a simply wrapper to MPI_Allreduce() as shown below:

```
void par_GlobalSumForDouble(void *sendBuf, void *recvBuf, int *count,
                            primme_svds_params *primme_svds, int *ierr) {
   MPI_Comm communicator = *(MPI_Comm *) primme_svds->commInfo;
   if (sendBuf == recvBuf) {
     *ierr = MPI_Allreduce(MPI_IN_PLACE, recvBuf, *count, MPIU_REAL, MPI_SUM,␣
↪communicator) != MPI_SUCCESS;
   } else {
     *ierr = MPI_Allreduce(sendBuf, recvBuf, *count, MPIU_REAL, MPI_SUM,␣
↪communicator) != MPI_SUCCESS;
   }
}
```

When calling `sprimme_svds()` and `cprimme_svds()` replace `MPI_DOUBLE` by `` `MPI_FLOAT ``.

primme_op_datatype **globalSumReal_type**

Precision of the vectors `sendBuf` and `recvBuf` passed to `globalSumReal`.

If it is `primme_op_default`, the vectors' type matches the calling `dprimme_svds()` (or a variant). Otherwise, the precision is half, single, or double, if `globalSumReal_type` is `primme_half`, `primme_float` or `primme_double` respectively.

Input/output:

`primme_svds_initialize()` sets this field to `primme_op_default`;

this field is read by `dprimme_svds()`, and if it is `primme_op_default` it is set to the value that matches the precision of calling function.

New in version 3.0.

void (***broadcastReal**) (void *buffer, int *count, *primme_svds_params* *primme_svds, int *ierr)

Broadcast function from process with ID zero. It is optional in parallel executions, and not needed for sequential programs.

Parameters

- **buffer** – array of size count with the local input values.
- **count** – array size of sendBuf and recvBuf.
- **primme_svds** – parameters structure.
- **ierr** – output error code; if it is set to non-zero, the current call to PRIMME will stop.

The actual type of buffer matches the type of svecs of the calling *dprimme_svds()* (or a variant), unless globalSumReal_type sets another precision.

Input/output:

primme_svds_initialize() sets this field to NULL;
this field is read by *dprimme_svds()*.

When MPI is used, this can be a simply wrapper to MPI_Bcast() as shown below:

```
void broadcastForDouble(void *buffer, int *count,
                        primme_svds_params *primme_svds, int *ierr) {
   MPI_Comm communicator = *(MPI_Comm *) primme_svds->commInfo;
   if(MPI_Bcast(buffer, *count, MPI_DOUBLE, 0 /* root */,
                communicator) == MPI_SUCCESS) {
      *ierr = 0;
   } else {
      *ierr = 1;
   }
}
```

When calling *sprimme_svds()* and *cprimme_svds()* replace MPI_DOUBLE by `MPI_FLOAT.

New in version 3.0.

int **numSvals**
Number of singular triplets wanted.

Input/output:

primme_svds_initialize() sets this field to 1;
this field is read by *primme_svds_set_method()* (see *Preset Methods*) and *dprimme_svds()*.

primme_op_datatype **broadcastReal_type**
Precision of the vector buffer` passed to *broadcastReal*.

If it is primme_op_default, the vectors' type matches the calling *dprimme_svds()* (or a variant). Otherwise, the precision is half, single, or double, if *broadcastReal_type* is primme_half, primme_float or primme_double respectively.

Input/output:

primme_svds_initialize() sets this field to primme_op_default;
this field is read by *dprimme_svds()*, and if it is primme_op_default it is set to the value that matches the precision of calling function.

New in version 3.0.

primme_op_datatype **internalPrecision**
Internal working precision.

If it is primme_op_default, most of the vectors are stored with the same precision as the calling *dprimme_svds()* (or a variant), and most of the computations are done in that precision too. Otherwise, the working precision is changed to half, single, or double, if *internalPrecision* is `primme_half`, `primme_float` or `primme_double` respectively.

Input/output:

> *primme_svds_initialize()* sets this field to `primme_op_default`;
> this field is read by *dprimme_svds()*.

New in version 3.0.

primme_svds_target **target**

Which singular values to find:

primme_svds_smallest Smallest singular values; *targetShifts* is ignored.

primme_svds_largest Largest singular values; *targetShifts* is ignored.

primme_svds_closest_abs Closest in absolute value to the shifts in *targetShifts*.

Input/output:

> *primme_svds_initialize()* sets this field to *primme_svds_smallest*;
> this field is read by *dprimme_svds()* and *zprimme_svds()*.

int **numTargetShifts**

Size of the array *targetShifts*. Used only when *target* is *primme_svds_closest_abs*. The default values is 0.

Input/output:

> *primme_svds_initialize()* sets this field to 0;
> this field is read by *dprimme_svds()* and *zprimme_svds()*.

double ***targetShifts**

Array of shifts, at least of size *numTargetShifts*. Used only when *target* is *primme_svds_closest_abs*.

Singular values are computed in order so that the i-th singular value is the closest to the i-th shift. If *numTargetShifts* < *numSvals*, the last shift given is used for all the remaining i's.

Input/output:

> *primme_svds_initialize()* sets this field to NULL;
> this field is read by *dprimme_svds()* and *zprimme_svds()*.

Note: Eventually this is used by *dprimme_svds()* and *zprimme_svds()*. Please see considerations of *targetShifts*.

int **printLevel**

The level of message reporting from the code. All output is written in *outputFile*.

One of:

- 0: silent.
- 1: print some error messages when these occur.
- 2: as in 1, and info about targeted singular triplets when they are marked as converged:

```
#Converged $1 sval[ $2 ]= $3 norm $4 Mvecs $5 Time $7 stage $10
```

or locked:

```
#Lock striplet[ $1 ]= $3 norm $4 Mvecs $5 Time $7 stage $10
```

- 3: as in 2, and info about targeted singular triplets every outer iteration:

  ```
  OUT $6 conv $1 blk $8 MV $5 Sec $7 SV $3 |r| $4 stage $10
  ```

  Also, if using *PRIMME_DYNAMIC*, show JDQMR/GD+k performance ratio and the current method in use.

- 4: as in 3, and info about targeted singular triplets every inner iteration:

  ```
  INN MV $5 Sec $7 Sval $3 Lin|r| $9 SV|r| $4 stage $10
  ```

- 5: as in 4, and verbose info about certain choices of the algorithm.

Output key:

$1: Number of converged triplets up to now.
$2: The index of the triplet currently converged.
$3: The singular value.
$4: Its residual norm.
$5: The current number of matrix-vector products.
$6: The current number of outer iterations.
$7: The current elapsed time.
$8: Index within the block of the targeted triplet.
$9: QMR norm of the linear system residual.
$10: stage (1 or 2)

In parallel programs, when *printLevel* is 0 to 4 only *procID* 0 produces output. For *printLevel* 5 output can be produced in any of the parallel calls.

Input/output:

primme_svds_initialize() sets this field to 1;
this field is read by *dprimme_svds()* and *zprimme_svds()*.

Note: Convergence history for plotting may be produced simply by:

```
grep OUT outpufile | awk '{print $8" "$14}' > out
grep INN outpufile | awk '{print $3" "$11}' > inn
```

Or in gnuplot:

```
plot 'out' w lp, 'inn' w lp
```

double **aNorm**

An estimate of the 2-norm of A, which is used in the default convergence criterion (see *eps*).

If *aNorm* is less than or equal to 0, the code uses the largest absolute Ritz value seen. On return, *aNorm* is then replaced with that value.

Input/output:

primme_svds_initialize() sets this field to 0.0;

this field is read and written by `dprimme_svds()` and `zprimme_svds()`.

double **eps**

If `convTestFun` is NULL, a triplet (u, σ, v) is marked as converged when $\sqrt{\|Av - \sigma u\|^2 + \|A^* u - \sigma v\|^2}$ is less than `eps` * `aNorm`, or close to the minimum tolerance that the selected method can achieve in the given machine precision. See *Preset Methods*.

The default value is machine precision times 10^4.

Input/output:

`primme_svds_initialize()` sets this field to 0.0;
this field is read by `dprimme_svds()` and `zprimme_svds()`.

FILE ***outputFile**

Opened file to write down the output.

Input/output:

`primme_svds_initialize()` sets this field to the standard output;
this field is read by `dprimme_svds()`, `zprimme_svds()` and `primme_svds_display_params()`

int **locking**

If set to 1, the underneath eigensolvers will use hard locking. See `locking`.

Input/output:

`primme_svds_initialize()` sets this field to -1;
written by `primme_svds_set_method()` (see *Preset Methods*);
this field is read by `dprimme_svds()` and `zprimme_svds()`.

int **initSize**

On input, the number of initial vector guesses provided in `svecs` argument in `dprimme_svds()` and `zprimme_svds()`.

On output, `initSize` holds the number of converged triplets. Without `locking` all `numSvals` approximations are in `svecs` but only the first `initSize` are converged.

During execution, it holds the current number of converged triplets.

Input/output:

`primme_svds_initialize()` sets this field to 0;
this field is read and written by `dprimme_svds()` and `zprimme_svds()`.

int **numOrthoConst**

Number of vectors to be used as external orthogonalization constraints. The left and the right vector constraints are provided as input of the `svecs` argument in `sprimme_svds()` or other variant, and must be orthonormal.

PRIMME SVDS finds new triplets orthogonal to these constraints (equivalent to solving the problem $(I - UU^*)A(I - VV^*)$ where U and V are the given left and right constraint vectors). This is a handy feature if some singular triplets are already known, or for finding more triplets after a call to `dprimme_svds()` or `zprimme_svds()`, possibly with different parameters (see an example in `TEST/exsvd_zseq.c`).

Input/output:

`primme_svds_initialize()` sets this field to 0;
this field is read by `dprimme_svds()` and `zprimme_svds()`.

int **`maxBasisSize`**

The maximum basis size allowed in the main iteration. This has memory implications.

Input/output:

`primme_svds_initialize()` sets this field to 0;
this field is read and written by *`primme_svds_set_method()`* (see *Preset Methods*);
this field is read by *`dprimme_svds()`* and *`zprimme_svds()`*.

int **`maxBlockSize`**

The maximum block size the code will try to use.

The user should set this based on the architecture specifics of the target computer, as well as any a priori knowledge of multiplicities. The code does *not* require that *`maxBlockSize`* > 1 to find multiple triplets. For some methods, keeping to 1 yields the best overall performance.

Input/output:

`primme_svds_initialize()` sets this field to 1;
this field is read and written by *`primme_svds_set_method()`* (see *Preset Methods*);
this field is read by *`dprimme_svds()`* and *`zprimme_svds()`*.

PRIMME_INT **`maxMatvecs`**

Maximum number of matrix vector multiplications (approximately half the number of preconditioning operations) that the code is allowed to perform before it exits.

Input/output:

`primme_svds_initialize()` sets this field to `INT_MAX`;
this field is read by *`dprimme_svds()`* and *`zprimme_svds()`*.

PRIMME_INT **`iseed`**

The `PRIMME_INT iseed[4]` is an array with the seeds needed by the LAPACK dlarnv and zlarnv.

The default value is an array with values -1, -1, -1 and -1. In that case, `iseed` is set based on the value of *`procID`* to avoid every parallel process generating the same sequence of pseudorandom numbers.

Input/output:

`primme_svds_initialize()` sets this field to `[-1, -1, -1, -1]`;
this field is read and written by *`dprimme_svds()`* and *`zprimme_svds()`*.

void ***`matrix`**

This field may be used to pass any required information in the matrix-vector product *`matrixMatvec`*.

Input/output:

`primme_svds_initialize()` sets this field to NULL;

void ***`preconditioner`**

This field may be used to pass any required information in the preconditioner function *`applyPreconditioner`*.

Input/output:

`primme_svds_initialize()` sets this field to NULL;

int **`precondition`**

Set to 1 to use preconditioning. Make sure *`applyPreconditioner`* is not NULL then!

Input/output:

`primme_svds_initialize()` sets this field to 0;
this field is read and written by *`primme_svds_set_method()`* (see *Preset Methods*);

this field is read by *dprimme_svds()* and *zprimme_svds()*.

primme_svds_op_operator **method**

Select the equivalent eigenvalue problem that will be solved:

- primme_svds_op_AtA: $A^*Ax = \sigma^2 x$,
- primme_svds_op_AAt: $AA^*x = \sigma^2 x$,
- primme_svds_op_augmented: $\left(\begin{array}{cc} 0 & A^* \\ A & 0 \end{array}\right) x = \sigma x$.

The options for this solver are stored in *primme*.

Input/output:

primme_svds_initialize() sets this field to primme_svds_op_none;
this field is read and written by *primme_svds_set_method()* (see *Preset Methods*);
this field is read by *dprimme_svds()* and *zprimme_svds()*.

primme_svds_op_operator **methodStage2**

Select the equivalent eigenvalue problem that will be solved to refine the solution. The allowed options are primme_svds_op_none to not refine the solution and primme_svds_op_augmented to refine the solution by solving the augmented problem with the current solution as the initial vectors. See *method*.

The options for this solver are stored in *primmeStage2*.

Input/output:

primme_svds_initialize() sets this field to primme_svds_op_none;
this field is read and written by *primme_svds_set_method()* (see *Preset Methods*);
this field is read by *dprimme_svds()* and *zprimme_svds()*.

primme_params **primme**

Parameter structure storing the options for underneath eigensolver that will be called at the first stage. See *method*.

Input/output:

primme_svds_initialize() initialize this structure;
this field is read and written by *primme_svds_set_method()* (see *Preset Methods*);
this field is read and written by *dprimme_svds()* and *zprimme_svds()*.

primme_params **primmeStage2**

Parameter structure storing the options for underneath eigensolver that will be called at the second stage. See *methodStage2*.

Input/output:

primme_svds_initialize() initialize this structure;
this field is read and written by *primme_svds_set_method()* (see *Preset Methods*);
this field is read and written by *dprimme_svds()* and *zprimme_svds()*.

void (***convTestFun**)(double *sval, void *leftsvec, void *rightsvec, double *rNorm, int *isconv, *primme_svds_params* *primme_svds, int *ierr)

Function that evaluates if the approximate triplet has converged. If NULL, it is used the default convergence criteria (see *eps*).

Parameters

- **sval** – the approximate singular value to evaluate.

- **leftsvec** – one dimensional array of size *mLocal* containing the approximate left singular vector; it can be NULL.
- **rightsvec** – one dimensional array of size *nLocal* containing the approximate right singular vector; it can be NULL.
- **resNorm** – the norm of the residual vector.
- **isconv** – (output) the function sets zero if the pair is not converged and non zero otherwise.
- **primme_svds** – parameters structure.
- **ierr** – output error code; if it is set to non-zero, the current call to PRIMME will stop.

The actual type of `leftsvec` and `rightsvec` matches the type of `svecs` of the calling *dprimme_svds()* (or a variant), unless *convTestFun_type* sets another precision.

Warning: When solving the augmented problem (for the method *primme_svds_augmented* and at the second stage in the method *primme_svds_hybrid*), the given residual vector norm `resNorm` is an approximation of the actual residual. Also `leftsvec` and `rightsvec` may not have length 1.

Input/output:

`svds_primme_initialize()` sets this field to NULL;

this field is read and written by *dprimme_svds()*.

primme_op_datatype **convTestFun_type**

Precision of the vectors `leftsvec` and `rightsvec` passed to *convTestFun*.

If it is `primme_op_default`, the type matches the calling *dprimme_svds()* (or a variant). Otherwise, the precision is half, single, or double, if *convTestFun_type* is `primme_half`, `primme_float` or `primme_double` respectively.

Input/output:

primme_svds_initialize() sets this field to `primme_op_default`;

this field is read by *dprimme_svds()*, and if it is `primme_op_default` it is set to the value that matches the precision of calling function.

New in version 3.0.

void ***convtest**

This field may be used to pass any required information to the function *convTestFun*.

Input/output:

primme_svds_initialize() sets this field to NULL;

void (***monitorFun**)(void *basisSvals, int *basisSize, int *basisFlags, int *iblock, int *blockSize, void *basisNorms, int *numConverged, void *lockedSvals, int *numLocked, int *lockedFlags, void *lockedNorms, int *inner_its, void *LSRes, **const** char *msg, double *time, primme_event *event, int *stage, *primme_svds_params* *primme_svds, int *ierr)

Convergence monitor. Used to customize how to report solver information during execution (stage, iteration number, matvecs, time, residual norms, targets, etc).

Parameters

- **basisSvals** – array with approximate singular values of the basis.

- **basisSize** – size of the arrays `basisSvals`, `basisFlags` and `basisNorms`.
- **basisFlags** – state of every approximate triplet in the basis.
- **iblock** – indices of the approximate triplet in the block.
- **blockSize** – size of array `iblock`.
- **basisNorms** – array with residual norms of the triplets in the basis.
- **numConverged** – number of triplets converged in the basis plus the number of the locked triplets (note that this value isn't monotonic).
- **lockedSvals** – array with the locked triplets.
- **numLocked** – size of the arrays `lockedSvals`, `lockedFlags` and `lockedNorms`.
- **lockedFlags** – state of each locked triplets.
- **lockedNorms** – array with residual norms of the locked triplets.
- **inner_its** – number of performed QMR iterations in the current correction equation.
- **LSRes** – residual norm of the linear system at the current QMR iteration.
- **msg** – output message or function name.
- **time** – time duration.
- **event** – event reported.
- **stage** – `0` for first stage, `1` for second stage.
- **primme_svds** – parameters structure; the counter in `stats` are updated with the current number of matrix-vector products, iterations, elapsed time, etc., since start.
- **ierr** – output error code; if it is set to non-zero, the current call to PRIMME will stop.

This function is called at the next events:

- `*event == primme_event_outer_iteration`: every outer iterations.

  It is provided `basisSvals`, `basisSize`, `basisFlags`, `iblock` and `blockSize`.

  `basisNorms[iblock[i]]` has the residual norms for the selected triplets in the block. PRIMME avoids computing the residual of soft-locked triplets, `basisNorms[i]` for `i<iblock[0]`. So those values may correspond to previous iterations. The values `basisNorms[i]` for `i>iblock[blockSize-1]` are not valid.

  If *`locking`* is enabled, `lockedSvals`, `numLocked`, `lockedFlags` and `lockedNorms` are also provided.

  `inner_its` and `LSRes` are not provided.

- `*event == primme_event_inner_iteration`: every QMR iteration.

  `basisSvals[0]` and `basisNorms[0]` provides the approximate singular value and the residual norm of the triplet which is improved in the current correction equation. If *`convTest`* is *`primme_adaptive`* or *`primme_adaptive_ETolerance`*, `basisSvals[0]`, and `basisNorms[0]` are updated every QMR iteration.

  `inner_its` and `LSRes` are also provided.

  `lockedSvals`, `numLocked`, `lockedFlags`, and `lockedNorms` may not be provided.

- `*event == primme_event_converged`: a new triplet in the basis passed the convergence criterion iblock[0] is the index of the newly converged triplet in the basis which will be locked or soft locked. The following are provided: basisSvals, basisSize, basisFlags and blockSize[0]==1.

lockedSvals, numLocked, lockedFlags and lockedNorms may not be provided.

inner_its and LSRes are not provided.

- *event == primme_event_locked: a new triplet added to the locked singular vectors.

  lockedSvals, numLocked, lockedFlags and lockedNorms are provided. The last element of lockedSvals, lockedFlags and lockedNorms corresponds to the recent locked triplet.

  basisSvals, numConverged, basisFlags and basisNorms may not be provided.

  inner_its and LSRes are not be provided.

- *event == primme_event_message: output message

  msg is the message to print.

  The rest of the arguments are not provided.

The values of basisFlags and lockedFlags are:

- 0: unconverged.
- 1: internal use; only in basisFlags.
- 2: passed convergence test (see *eps*).
- 3: converged because the solver may not be able to reduce the residual norm further.

The actual type of basisEvals, basisNorms, lockedEvals, lockedNorms and LSRes matches the type of evecs of the calling *dprimme_svds()* (or a variant), unless *monitorFun_type* sets another precision.

Input/output:

primme_svds_initialize() sets this field to NULL;
dprimme_svds() sets this field to an internal function if it is NULL;
this field is read by *dprimme_svds()* and *zprimme_svds()*.

Changed in version 3.0.

primme_op_datatype **monitorFun_type**

Precision of the vectors basisEvals, basisNorms, lockedEvals, lockedNorms and LSRes passed to *monitorFun*.

If it is primme_op_default, the vectors' type matches the calling *dprimme_svds()* (or a variant). Otherwise, the precision is half, single, or double, if *monitorFun_type* is primme_half, primme_float or primme_double respectively.

Input/output:

primme_svds_initialize() sets this field to primme_op_default;
this field is read by *dprimme_svds()*, and if it is primme_op_default it is set to the value that matches the precision of calling function.

New in version 3.0.

void ***monitor**

This field may be used to pass any required information to the function *monitorFun*.

Input/output:

primme_svds_initialize() sets this field to NULL;

PRIMME_INT `stats.`**numOuterIterations**

Hold the number of outer iterations.

Input/output:

primme_svds_initialize() sets this field to 0;
written by *dprimme_svds()* and *zprimme_svds()*.

PRIMME_INT `stats.`**numRestarts**

Hold the number of restarts.

Input/output:

primme_svds_initialize() sets this field to 0;
written by *dprimme_svds()* and *zprimme_svds()*.

PRIMME_INT `stats.`**numMatvecs**

Hold how many vectors the operator in *matrixMatvec* has been applied on.

Input/output:

primme_svds_initialize() sets this field to 0;
written by *dprimme_svds()* and *zprimme_svds()*.

PRIMME_INT `stats.`**numPreconds**

Hold how many vectors the operator in *applyPreconditioner* has been applied on.

Input/output:

primme_svds_initialize() sets this field to 0;
written by *dprimme_svds()* and *zprimme_svds()*.

PRIMME_INT `stats.`**numGlobalSum**

Hold how many times *globalSumReal* has been called. The value is available during execution and at the end.

Input/output:

primme_svds_initialize() sets this field to 0;
written by *dprimme_svds()*.

New in version 3.0.

double `stats.`**volumeGlobalSum**

Hold how many `REAL` have been reduced by *globalSumReal*. The value is available during execution and at the end.

Input/output:

primme_svds_initialize() sets this field to 0;
written by *dprimme_svds()*.

New in version 3.0.

PRIMME_INT `stats.`**numBroadcast**

Hold how many times *broadcastReal* has been called. The value is available during execution and at the end.

Input/output:

primme_svds_initialize() sets this field to 0;
written by *dprimme_svds()*.

New in version 3.0.

double `stats`.**`volumeBroadcast`**

Hold how many `REAL` have been broadcast by *`broadcastReal`*. The value is available during execution and at the end.

Input/output:

`primme_svds_initialize()` sets this field to 0;
written by *`dprimme_svds()`*.

New in version 3.0.

PRIMME_INT `stats`.**`numOrthoInnerProds`**

Hold how many inner products with vectors of length *`mLocal`* and *`nLocal`* have been computed during orthogonalization. The value is available during execution and at the end.

Input/output:

`primme_svds_initialize()` sets this field to 0;
written by *`dprimme_svds()`*.

New in version 3.0.

double `stats`.**`elapsedTime`**

Hold the wall clock time spent by the call to *`dprimme_svds()`* or *`zprimme_svds()`*.

Input/output:

`primme_svds_initialize()` sets this field to 0;
written by *`dprimme_svds()`* and *`zprimme_svds()`*.

double `stats`.**`timeMatvec`**

Hold the wall clock time spent by *`matrixMatvec`*. The value is available at the end of the execution.

Input/output:

`primme_svds_initialize()` sets this field to 0;
written by *`dprimme_svds()`*.

New in version 3.0.

double `stats`.**`timePrecond`**

Hold the wall clock time spent by *`applyPreconditioner`*. The value is available at the end of the execution.

Input/output:

`primme_svds_initialize()` sets this field to 0;
written by *`dprimme_svds()`*.

New in version 3.0.

double `stats`.**`timeOrtho`**

Hold the wall clock time spent by orthogonalization. The value is available at the end of the execution.

Input/output:

`primme_svds_initialize()` sets this field to 0;
written by *`dprimme_svds()`*.

New in version 3.0.

double `stats.`**`timeGlobalSum`**

Hold the wall clock time spent by *`globalSumReal`*. The value is available at the end of the execution.

Input/output:

`primme_svds_initialize()` sets this field to 0;
written by *`dprimme_svds()`*.

New in version 3.0.

double `stats.`**`timeBroadcast`**

Hold the wall clock time spent by *`broadcastReal`*. The value is available at the end of the execution.

Input/output:

`primme_svds_initialize()` sets this field to 0;
written by *`dprimme_svds()`*.

New in version 3.0.

PRIMME_INT `stats.`**`lockingIssue`**

It is set to a nonzero value if some of the returned triplets do not pass the convergence criterion. See *`convTestFun`* and *`eps`*.

Input/output:

`primme_svds_initialize()` sets this field to 0;
written by *`dprimme_svds()`*.

New in version 3.0.

void ***`queue`**

Pointer to the accelerator's data structure.

If the main call is `dprimme_svds_magma()` or a variant, this field should have the pointer to an initialized `magma_queue_t`.

See example `examples/ex_svds_dmagma.c`.

Input/output:

`primme_svds_initialize()` sets this field to NULL;
this field is read by `dprimme_svds_magma()`.

New in version 3.0.

3.7 Preset Methods

enum primme_svds_preset_method

enumerator primme_svds_default

Set as *primme_svds_hybrid*.

enumerator primme_svds_normalequations

Solve the equivalent eigenvalue problem $A^*AV = \Sigma^2 V$ and computes U by normalizing the vectors AV. If *m* is smaller than *n*, AA^* is solved instead.

With *primme_svds_normalequations* *primme_svds_set_method()* sets *method* to `primme_svds_op_AtA` if *m* is larger or equal than *n*, and to `primme_svds_op_AAt` otherwise; and *methodStage2* is set to `primme_svds_op_none`.

The minimum residual norm that this method can achieve is $\|A\|\epsilon\sigma^{-1}$, where ϵ is the machine precision and σ the required singular value.

enumerator primme_svds_augmented

Solve the equivalent eigenvalue problem $\begin{pmatrix} 0 & A^* \\ A & 0 \end{pmatrix} X = \sigma X$ with $X = \begin{pmatrix} V \\ U \end{pmatrix}$.

With *primme_svds_augmented* *primme_svds_set_method()* sets *method* to `primme_svds_op_augmented` and *methodStage2* to `primme_svds_op_none`.

The minimum residual norm that this method can achieve is $\|A\|\epsilon$, where ϵ is the machine precision. However it may not return triplets with singular values smaller than $\|A\|\epsilon$.

enumerator primme_svds_hybrid

First solve the equivalent normal equations (see *primme_svds_normalequations*) and then refine the solution solving the augmented problem (see *primme_svds_augmented*).

With *primme_svds_normalequations* *primme_svds_set_method()* sets *method* to `primme_svds_op_AtA` if *m* is larger or equal than *n*, and to `primme_svds_op_AAt` otherwise; and *methodStage2* is set to `primme_svds_op_augmented`.

The minimum residual norm that this method can achieve is $\|A\|\epsilon$, where ϵ is the machine precision. However it may not return triplets with singular values smaller than $\|A\|\epsilon$ if *eps* is smaller than $\|A\|\epsilon\sigma^{-1}$.

3.8 Error Codes

The functions *dprimme_svds()* and *zprimme_svds()* return one of the following error codes. Some of the error codes have a macro associated which is indicated in brackets.

- 0: success; usually all requested singular triplets have converged.
- -1: (PRIMME_UNEXPECTED_FAILURE) unexpected internal error; please consider to set *printLevel* to a value larger than 0 to see the call stack and to report these errors because they may be bugs.
- -2: (PRIMME_MALLOC_FAILURE) failure in allocating memory; it can be either CPU or GPU.
- -3: (PRIMME_MAIN_ITER_FAILURE) maximum number of matvecs *maxMatvecs* reached.
- -4: primme_svds is NULL.
- -5: Wrong value for *m* or *n* or *mLocal* or *nLocal*.
- -6: Wrong value for *numProcs*.
- -7: *matrixMatvec* is not set.
- -8: *applyPreconditioner* is not set but *precondition* == 1.
- -9: *numProcs* >1 but *globalSumReal* is not set.
- -10: Wrong value for *numSvals*, it's larger than min(*m*, *n*).
- -11: Wrong value for *numSvals*, it's smaller than 1.
- -13: Wrong value for *target*.
- -14: Wrong value for *method*.
- -15: Not supported combination of method and *methodStage2*.
- -16: Wrong value for *printLevel*.
- -17: svals is not set.
- -18: svecs is not set.
- -19: resNorms is not set.
- -40: (PRIMME_LAPACK_FAILURE) some LAPACK function performing a factorization returned an error code; set *printLevel* > 0 to see the error code and the call stack.
- -41: (PRIMME_USER_FAILURE) some of the user-defined functions (*matrixMatvec*, *applyPreconditioner*, ...) returned a non-zero error code; set *printLevel* > 0 to see the call stack that produced the error.
- -42: (PRIMME_ORTHO_CONST_FAILURE) the provided orthogonal constraints (see numOrthoConst) are not full rank.
- -43: (PRIMME_PARALLEL_FAILURE) some process has a different value in an input option than the process zero, or it is not acting coherently; set *printLevel* > 0 to see the call stack that produced the error.
- -44: (PRIMME_FUNCTION_UNAVAILABLE) PRIMME was not compiled with support for the requesting precision or for GPUs.
- -100 up to -199: eigensolver error from first stage; see the value plus 100 in *Error Codes*.
- -200 up to -299: eigensolver error from second stage; see the value plus 200 in *Error Codes*.

CHAPTER FOUR

INDICES

- genindex
- search

BIBLIOGRAPHY

[r1] A. Stathopoulos and J. R. McCombs PRIMME: *PReconditioned Iterative MultiMethod Eigensolver: Methods and software description*, ACM Transaction on Mathematical Software Vol. 37, No. 2, (2010), 21:1-21:30.

[r6] L. Wu, E. Romero and A. Stathopoulos, *PRIMME_SVDS: A High-Performance Preconditioned SVD Solver for Accurate Large-Scale Computations*, J. Sci. Comput., Vol. 39, No. 5, (2017), S248–S271.

[r2] A. Stathopoulos, *Nearly optimal preconditioned methods for Hermitian eigenproblems under limited memory. Part I: Seeking one eigenvalue*, SIAM J. Sci. Comput., Vol. 29, No. 2, (2007), 481–514.

[r3] A. Stathopoulos and J. R. McCombs, *Nearly optimal preconditioned methods for Hermitian eigenproblems under limited memory. Part II: Seeking many eigenvalues*, SIAM J. Sci. Comput., Vol. 29, No. 5, (2007), 2162-2188.

[r4] J. R. McCombs and A. Stathopoulos, *Iterative Validation of Eigensolvers: A Scheme for Improving the Reliability of Hermitian Eigenvalue Solvers*, SIAM J. Sci. Comput., Vol. 28, No. 6, (2006), 2337-2358.

[r5] A. Stathopoulos, *Locking issues for finding a large number of eigenvectors of Hermitian matrices*, Tech Report: WM-CS-2005-03, July, 2005.

[r7] L. Wu and A. Stathopoulos, *A Preconditioned Hybrid SVD Method for Computing Accurately Singular Triplets of Large Matrices*, SIAM J. Sci. Comput. 37-5(2015), pp. S365-S388.

INDEX

C

c_ptr (*fortran type*), **37**
cprimme (*C function*), 18
cprimme_f77() (*fortran subroutine*), **26**
cprimme_normal (*C function*), 20
cprimme_normal_f77() (*fortran subroutine*), **30**
cprimme_svds (*C function*), 82
cprimme_svds_f77() (*fortran subroutine*), **87**

D

dprimme (*C function*), 18
dprimme_f77() (*fortran subroutine*), **27**
dprimme_svds (*C function*), 82
dprimme_svds_f77() (*fortran subroutine*), **88**

E

eigsh() (*in module primme*), 44

H

hprimme (*C function*), 18
hprimme_svds (*C function*), 82
hsprimme (*C function*), 18
hsprimme_svds (*C function*), 82

I

interior problem, 58, 114

K

kcprimme_normal (*C function*), 20
kprimme (*C function*), 18
kprimme_normal (*C function*), 20
kprimme_svds (*C function*), 82
ksprimme (*C function*), 18
ksprimme_svds (*C function*), 82

M

magma_cprimme (*C function*), 19
magma_cprimme_f77() (*fortran subroutine*), **28**
magma_cprimme_normal (*C function*), 21
magma_cprimme_normal_f77() (*fortran subroutine*), **31**
magma_cprimme_svds (*C function*), 83
magma_cprimme_svds_f77() (*fortran subroutine*), **90**
magma_dprimme (*C function*), 19
magma_dprimme_f77() (*fortran subroutine*), **29**
magma_dprimme_svds (*C function*), 83
magma_dprimme_svds_f77() (*fortran subroutine*), **91**
magma_hprimme (*C function*), 19
magma_hprimme_svds (*C function*), 83
magma_hsprimme (*C function*), 19
magma_hsprimme_svds (*C function*), 83
magma_kcprimme_normal (*C function*), 21
magma_kprimme (*C function*), 19
magma_kprimme_normal (*C function*), 21
magma_kprimme_svds (*C function*), 83
magma_ksprimme (*C function*), 19
magma_ksprimme_svds (*C function*), 83
magma_sprimme (*C function*), 19
magma_sprimme_f77() (*fortran subroutine*), **28**
magma_sprimme_svds (*C function*), 83
magma_sprimme_svds_f77() (*fortran subroutine*), **89**
magma_xprimme() (*fortran function*), **39**
magma_xprimme_normal() (*fortran function*), **40**
magma_xprimme_svds() (*fortran function*), **99**
magma_zprimme (*C function*), 19
magma_zprimme_f77() (*fortran subroutine*), **29**
magma_zprimme_normal (*C function*), 21
magma_zprimme_normal_f77() (*fortran subroutine*), **31**
magma_zprimme_svds (*C function*), 83
magma_zprimme_svds_f77() (*fortran subroutine*), **91**

P

PRIMME_COMPLEX_DOUBLE (*C type*), 52
PRIMME_COMPLEX_FLOAT (*C type*), 52
PRIMME_COMPLEX_HALF (*C type*), 52
primme_display_params (*C function*), 23
primme_eigs_matvec() (*fortran subroutine*), **37**
primme_free (*C function*), 23

primme_free_f77() (*fortran subroutine*), **26**
primme_get_member() (*fortran function*), **43**
primme_get_member_f77() (*fortran subroutine*), **35**
primme_get_prec_shift_f77() (*fortran subroutine*), **36**
PRIMME_HALF (*C type*), 52
primme_initialize (*C function*), 22
primme_initialize_f77() (*fortran subroutine*), **25**
PRIMME_INT (*C type*), 52
primme_params (*C type*), 52
primme_params.aNorm (*C member*), 60
primme_params.applyPreconditioner (*C member*), 53
primme_params.applyPreconditioner_type (*C member*), 54
primme_params.BNorm (*C member*), 60
primme_params.broadcastReal (*C member*), 57
primme_params.broadcastReal_type (*C member*), 57
primme_params.commInfo (*C member*), 55
primme_params.convtest (*C member*), 73
primme_params.convTestFun (*C member*), 72
primme_params.convTestFun_type (*C member*), 73
primme_params.correctionParams.convTest (*C member*), 66
primme_params.correctionParams.maxInnerIterations (*C member*), 65
primme_params.correctionParams.precondition (*C member*), 65
primme_params.correctionParams.projectors.LeftQ (*C member*), 66
primme_params.correctionParams.projectors.LeftX (*C member*), 66
primme_params.correctionParams.projectors.RightQ (*C member*), 66
primme_params.correctionParams.projectors.RightX (*C member*), 66
primme_params.correctionParams.projectors.SkewQ (*C member*), 66
primme_params.correctionParams.projectors.SkewX (*C member*), 66
primme_params.correctionParams.relTolBase (*C member*), 65
primme_params.correctionParams.robustShifts (*C member*), 65
primme_params.dynamicMethodSwitch (*C member*), 61
primme_params.eps (*C member*), 61
primme_params.globalSumReal (*C member*), 56
primme_params.globalSumReal_type (*C member*), 56
primme_params.initBasisMode (*C member*), 64
primme_params.initSize (*C member*), 62
primme_params.internalPrecision (*C member*), 58
primme_params.invBNorm (*C member*), 60
primme_params.iseed (*C member*), 63
primme_params.ldevecs (*C member*), 62
primme_params.ldOPs (*C member*), 67
primme_params.locking (*C member*), 62
primme_params.massMatrix (*C member*), 64
primme_params.massMatrixMatvec (*C member*), 54
primme_params.massMatrixMatvec_type (*C member*), 55
primme_params.matrix (*C member*), 64
primme_params.matrixMatvec (*C member*), 52
primme_params.matrixMatvec_type (*C member*), 53
primme_params.maxBasisSize (*C member*), 63
primme_params.maxBlockSize (*C member*), 63
primme_params.maxMatvecs (*C member*), 63
primme_params.maxOuterIterations (*C member*), 63
primme_params.minRestartSize (*C member*), 63
primme_params.monitor (*C member*), 69
primme_params.monitorFun (*C member*), 67
primme_params.monitorFun_type (*C member*), 69
primme_params.n (*C member*), 52
primme_params.nLocal (*C member*), 55
primme_params.numEvals (*C member*), 58
primme_params.numOrthoConst (*C member*), 62
primme_params.numProcs (*C member*), 55
primme_params.numTargetShifts (*C member*), 58
primme_params.orth (*C member*), 60
primme_params.outputFile (*C member*), 61
primme_params.preconditioner (*C member*), 64
primme_params.printLevel (*C member*), 59
primme_params.procID (*C member*), 55
primme_params.projectionParams.projection (*C member*), 64
primme_params.queue (*C member*), 73
primme_params.restartingParams.maxPrevRetain (*C member*), 65
primme_params.ShiftsForPreconditioner (*C member*), 64
primme_params.stats.elapsedTime (*C member*), 71
primme_params.stats.estimateLargestSVal (*C member*), 72

primme_params.stats.estimateMaxEVal (*C member*), 72
primme_params.stats.estimateMinEVal (*C member*), 72
primme_params.stats.lockingIssue (*C member*), 72
primme_params.stats.maxConvTol (*C member*), 72
primme_params.stats.numBroadcast (*C member*), 70
primme_params.stats.numGlobalSum (*C member*), 70
primme_params.stats.numMatvecs (*C member*), 70
primme_params.stats.numOrthoInnerProds (*C member*), 71
primme_params.stats.numOuterIterations (*C member*), 70
primme_params.stats.numPreconds (*C member*), 70
primme_params.stats.numRestarts (*C member*), 70
primme_params.stats.timeBroadcast (*C member*), 72
primme_params.stats.timeGlobalSum (*C member*), 71
primme_params.stats.timeMatvec (*C member*), 71
primme_params.stats.timeOrtho (*C member*), 71
primme_params.stats.timePrecond (*C member*), 71
primme_params.stats.volumeBroadcast (*C member*), 71
primme_params.stats.volumeGlobalSum (*C member*), 70
primme_params.target (*C member*), 58
primme_params.targetShifts (*C member*), 58
primme_params_create (*C function*), 22
primme_params_create() (*fortran function*), **37**
primme_params_destroy (*C function*), 24
primme_params_destroy() (*fortran function*), **38**
primme_preset_method (*C enum*), 74
primme_preset_method.PRIMME_Arnoldi (*C enumerator*), 74
primme_preset_method.PRIMME_DEFAULT_MIN_MATVECS (*C enumerator*), 74
primme_preset_method.PRIMME_DEFAULT_MIN_TIME (*C enumerator*), 74
primme_preset_method.PRIMME_DYNAMIC (*C enumerator*), 74
primme_preset_method.PRIMME_GD (*C enumerator*), 74
primme_preset_method.PRIMME_GD_Olsen_plusK (*C enumerator*), 74
primme_preset_method.PRIMME_GD_plusK (*C enumerator*), 74
primme_preset_method.PRIMME_JD_Olsen_plusK (*C enumerator*), 74
primme_preset_method.PRIMME_JDQMR (*C enumerator*), 75
primme_preset_method.PRIMME_JDQMR_ETol (*C enumerator*), 76
primme_preset_method.PRIMME_JDQR (*C enumerator*), 75
primme_preset_method.PRIMME_LOBPCG_OrthoBasis (*C enumerator*), 76
primme_preset_method.PRIMME_LOBPCG_OrthoBasis_Windo (*C enumerator*), 76
primme_preset_method.PRIMME_RQI (*C enumerator*), 75
primme_preset_method.PRIMME_STEEPEST_DESCENT (*C enumerator*), 76
primme_set_member() (*fortran function*), **40**
primme_set_member_f77() (*fortran subroutine*), **32**
primme_set_method (*C function*), 23
primme_set_method() (*fortran function*), **37**
primme_set_method_f77() (*fortran subroutine*), **25**
primme_svds_display_params (*C function*), 85
primme_svds_display_params_f77() (*fortran subroutine*), **93**
primme_svds_free (*C function*), 85
primme_svds_free_f77() (*fortran subroutine*), **93**
primme_svds_get_member() (*fortran function*), **102**
primme_svds_get_member_f77() (*fortran subroutine*), **96**
primme_svds_initialize (*C function*), 84
primme_svds_initialize_f77() (*fortran subroutine*), **92**
primme_svds_matvec() (*fortran subroutine*), **97**
primme_svds_params (*C type*), 109
primme_svds_params.aNorm (*C member*), 115
primme_svds_params.applyPreconditioner (*C member*), 110
primme_svds_params.applyPreconditioner_type (*C member*), 110
primme_svds_params.broadcastReal (*C member*), 112
primme_svds_params.broadcastReal_type (*C member*), 113
primme_svds_params.commInfo (*C member*), 111
primme_svds_params.convtest (*C member*), 119
primme_svds_params.convTestFun (*C mem-

ber), 118
primme_svds_params.convTestFun_type (*C member*), 119
primme_svds_params.eps (*C member*), 116
primme_svds_params.globalSumReal (*C member*), 112
primme_svds_params.globalSumReal_type (*C member*), 112
primme_svds_params.initSize (*C member*), 116
primme_svds_params.initSize.numOrthoConst (*C member*), 116
primme_svds_params.internalPrecision (*C member*), 113
primme_svds_params.iseed (*C member*), 117
primme_svds_params.locking (*C member*), 116
primme_svds_params.m (*C member*), 109
primme_svds_params.matrix (*C member*), 117
primme_svds_params.matrixMatvec (*C member*), 109
primme_svds_params.matrixMatvec_type (*C member*), 109
primme_svds_params.maxBasisSize (*C member*), 117
primme_svds_params.maxBlockSize (*C member*), 117
primme_svds_params.maxMatvecs (*C member*), 117
primme_svds_params.method (*C member*), 118
primme_svds_params.methodStage2 (*C member*), 118
primme_svds_params.mLocal (*C member*), 111
primme_svds_params.monitor (*C member*), 121
primme_svds_params.monitorFun (*C member*), 119
primme_svds_params.monitorFun_type (*C member*), 121
primme_svds_params.n (*C member*), 109
primme_svds_params.nLocal (*C member*), 111
primme_svds_params.numProcs (*C member*), 111
primme_svds_params.numSvals (*C member*), 113
primme_svds_params.numTargetShifts (*C member*), 114
primme_svds_params.outputFile (*C member*), 116
primme_svds_params.precondition (*C member*), 117
primme_svds_params.preconditioner (*C member*), 117
primme_svds_params.primme (*C member*), 118
primme_svds_params.primmeStage2 (*C member*), 118
primme_svds_params.printLevel (*C member*), 114
primme_svds_params.procID (*C member*), 111
primme_svds_params.queue (*C member*), 124
primme_svds_params.stats.elapsedTime (*C member*), 123
primme_svds_params.stats.lockingIssue (*C member*), 124
primme_svds_params.stats.numBroadcast (*C member*), 122
primme_svds_params.stats.numGlobalSum (*C member*), 122
primme_svds_params.stats.numMatvecs (*C member*), 122
primme_svds_params.stats.numOrthoInnerProds (*C member*), 123
primme_svds_params.stats.numOuterIterations (*C member*), 122
primme_svds_params.stats.numPreconds (*C member*), 122
primme_svds_params.stats.numRestarts (*C member*), 122
primme_svds_params.stats.timeBroadcast (*C member*), 124
primme_svds_params.stats.timeGlobalSum (*C member*), 123
primme_svds_params.stats.timeMatvec (*C member*), 123
primme_svds_params.stats.timeOrtho (*C member*), 123
primme_svds_params.stats.timePrecond (*C member*), 123
primme_svds_params.stats.volumeBroadcast (*C member*), 123
primme_svds_params.stats.volumeGlobalSum (*C member*), 122
primme_svds_params.target (*C member*), 114
primme_svds_params.targetShifts (*C member*), 114
primme_svds_params_create (*C function*), 84
primme_svds_params_create() (*fortran function*), **97**
primme_svds_params_destroy (*C function*), 86
primme_svds_params_destroy() (*fortran function*), **99**
primme_svds_preset_method (*C enum*), 125
primme_svds_preset_method.primme_svds_augmented (*C enumerator*), 125
primme_svds_preset_method.primme_svds_default (*C enumerator*), 125
primme_svds_preset_method.primme_svds_hybrid (*C enumerator*), 125
primme_svds_preset_method.primme_svds_normalequatic (*C enumerator*), 125

primme_svds_set_member() (*fortran function*), **100**
primme_svds_set_member_f77() (*fortran subroutine*), **93**
primme_svds_set_method (*C function*), 85
primme_svds_set_method() (*fortran function*), **97**
primme_svds_set_method_f77() (*fortran subroutine*), **92**
primme_svdstop_get_member_f77() (*fortran subroutine*), **95**
PRIMME_VERSION_MAJOR (*C macro*), 52
PRIMME_VERSION_MINOR (*C macro*), 52
primmetop_get_member_f77() (*fortran subroutine*), **34**
primmetop_get_prec_shift_f77() (*fortran subroutine*), **35**
ptr (*fortran type*), **25**

S

sprimme (*C function*), 18
sprimme_f77() (*fortran subroutine*), **26**
sprimme_svds (*C function*), 82
sprimme_svds_f77() (*fortran subroutine*), **87**
stopping criterion, 63, 117
svds() (*in module primme*), 103

X

xprimme() (*fortran function*), **38**
xprimme_normal() (*fortran function*), **39**
xprimme_svds() (*fortran function*), **98**

Z

zprimme (*C function*), 18
zprimme_f77() (*fortran subroutine*), **27**
zprimme_normal (*C function*), 20
zprimme_normal_f77() (*fortran subroutine*), **30**
zprimme_svds (*C function*), 82
zprimme_svds_f77() (*fortran subroutine*), **89**