# PRIMME Documentation

***Release 1.2.1***

**Andreas Stathopuolos**

October 13, 2015

CHAPTER

ONE

# PRIMME: PRECONDITIONED ITERATIVE MULTIMETHOD EIGENSOLVER

PRIMME, pronounced as *prime*, finds a number of eigenvalues and their corresponding eigenvectors of a real symmetric, or complex hermitian matrix A. Largest, smallest and interior eigenvalues are supported. Preconditioning can be used to accelerate convergence. PRIMME is written in C99, but complete interfaces are provided for Fortran 77 and MATLAB.

## 1.1 Changelog

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

- Fixed wrong symbols in `libdprimme.a` and `libzprimme.a`.
- *`primme_set_method()`* sets *`JDQMR`* instead of *`JDQMR_ETol`* for preset methods *`DEFAULT_MIN_TIME`* and *`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 `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.

## 1.2 Citing this code

Please cite *[r1]*.

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]*. The work has been supported by a number of grants from the National Science Foundation.

## 1.3 License Information



## 1.4 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 .

## 1.5 Directory Structure

The next directories and files should be available:

- `COPYING.txt`, LGPL 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;

- PRIMMESRC/, directory with source code in the following subdirectories:
  - COMMONSRC/, interface and common functions used by all precision versions;
  - DSRC/, the source code for the double precision *dprimme()*;
  - ZSRC/, the source code for the double complex precision *zprimme()*;
- MEX/, MATLAB interface for PRIMME;
- TEST/, sample test programs in C and F77, both sequential and parallel;
- 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.

## 1.6 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. Also include some of the following options if required for the BLAS and LAPACK libraries to be linked:
  - -DF77UNDERSCORE, if Fortran appends an underscore to function names (usually they does).
  - -DPRIMME_BLASINT_SIZE=64, if the library integers are 64-bit integer (kind=8) type (usually they are not).

---

**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, replace the next lines in PRIMMESRC/COMMONSRC/common_numerical.h:

```
#if !defined(PRIMME_BLASINT_SIZE)
#  define PRIMME_BLASINT int
#else
#  include <stdint.h>
#  define GENERIC_INT(N) int ## N ## _t
#  define XGENERIC_INT(N) GENERIC_INT(N)
#  define PRIMME_BLASINT XGENERIC_INT(PRIMME_BLASINT_SIZE)
#endif
```

by the next macro definition with the proper type for an int of 64 bits:

```
#define PRIMME_BLASINT __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 TEST:

- *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!
```

If it worked, try with other examples in `TEST` (see `README` in `TEST` for more information about how to compile the driver and the examples).

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 `libprimme.a`; alternatively:
- *make libd*, if only *dprimme()* is of interest, build `libdprimme.a`:
- *make libz*, if only *zprimme()* is of interest, build `libzprimme.a`;
- *make test*, build and execute a simple example;
- *make clean*, removes all `*.o`, `a.out`, and core files from all directories.

### 1.6.1 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 `PRIMMESRC`.
2. Add the directory `PRIMMESRC/COMMONSRC` as an include directory.

To build an example code using PRIMME make sure:

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

## 1.7 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 14.04
- MacOS X 10.9 & 10.10
- Cygwin & MinGW
- Cray XC30
- SunOS 5.9, quad processor Sun-Fire-280R, and several other UltraSparcs

- AIX 5.2 IBM SP POWER 3+, 16-way SMP, 375 MHz nodes (seaborg at nersc.gov)

CHAPTER

TWO

# C LIBRARY INTERFACE

The PRIMME interface is composed of the following functions. To solve real symmetric and Hermitian standard eigenproblems call respectively:

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

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 Running

To use PRIMME, follow this 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 to solve a real symmetric standard eigenproblems call:

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

   To solve Hermitian standard eigenproblems call:

```
ret = zprimme (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. Before exiting, free the work arrays in PRIMME:

```
primme_Free (&primme);
```

## 2.2 Parameters Guide

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

*Basic*
`int` *n*, matrix dimension.
`void (*` *matrixMatvec* `)(...)`, matrix-vector product.
`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*
`int` *procID*
`int` *nLocal*
`void (*` *globalSumDouble* `)(...)`

*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*

*Advanced options*
`int` *numOrthoConst*, orthogonal constrains to the eigenvectors.
`int` *dynamicMethodSwitch*
`int` *locking*
`int` *maxMatvecs*

```
int maxOuterIterations
int intWorkSize
long int realWorkSize
int iseed [4]
int * intWork
void * realWork
double aNorm
int printLevel
FILE * outputFile
double * ShiftsForPreconditioner
struct restarting_params restartingParams
struct correction_params correctionParams
struct primme_stats stats
struct stackTraceNode *stackTrace
```

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 *globalSumDouble* are also required.

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

## 2.3 Interface Description

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

### 2.3.1 dprimme

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

Solve a real symmetric standard eigenproblem.

**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.
- **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.
- **evecs** – array at least of size *nLocal* times *numEvals* to store columnwise the (local part of the) computed eigenvectors.
- **primme** – parameters structure.

**Returns** error indicator; see *Error Codes*.

### 2.3.2 zprimme

int **zprimme**(double **evals*, Complex_Z **evecs*, double **resNorms*, *primme_params* **primme*)
Solve a Hermitian standard eigenproblem; see function *dprimme()*.

---

**Note:** PRIMME uses a structure called `Complex_Z` to define complex numbers. `Complex_Z` is defined in `PRIMMESRC/COMMONSRC/Complexz.h`. In future versions of PRIMME, `Complex_Z` will be replaced by `complex double` from the C99 standard. Because the two types are binary compatible, we strongly recommend that calling programs use the C99 type to maintain future compatibility. See examples in `TEST` such as `ex_zseq.c` and `ex_zseqf77.c`.

---

### 2.3.3 primme_initialize

void **primme_initialize**(*primme_params* **primme*)
Set PRIMME parameters structure to the default values.

**Parameters**

- **primme** – parameters structure.

### 2.3.4 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

  *DYNAMIC*
  *DEFAULT_MIN_TIME*
  *DEFAULT_MIN_MATVECS*
  *Arnoldi*
  *GD*
  *GD_plusK*
  *GD_Olsen_plusK*
  *JD_Olsen_plusK*
  *RQI*
  *JDQR*
  *JDQMR*
  *JDQMR_ETol*
  *SUBSPACE_ITERATION*
  *LOBPCG_OrthoBasis*
  *LOBPCG_OrthoBasis_Window*

- **primme** – parameters structure.

See also *Preset Methods*.

### 2.3.5 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.

### 2.3.6 primme_Free

void **primme_Free** (*primme_params *primme*)
Free memory allocated by PRIMME.

**Parameters**

- **primme** – parameters structure.

CHAPTER

THREE

# FORTRAN LIBRARY INTERFACE

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

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

## 3.1 primme_initialize_f77

**primme_initialize_f77** (primme)
: Set PRIMME parameters structure to the default values.

  **Parameters**

  - **primme** (ptr) – (output) parameters structure.

## 3.2 primme_set_method_f77

**primme_set_method_f77** (method, primme, ierr)
: Set PRIMME parameters to one of the preset configurations.

  **Parameters**

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

```
PRIMMEF77_DYNAMIC
PRIMMEF77_DEFAULT_MIN_TIME
PRIMMEF77_DEFAULT_MIN_MATVECS
PRIMMEF77_Arnoldi
PRIMMEF77_GD
PRIMMEF77_GD_plusK
PRIMMEF77_GD_Olsen_plusK
PRIMMEF77_JD_Olsen_plusK
PRIMMEF77_RQI
PRIMMEF77_JDQR
PRIMMEF77_JDQMR
PRIMMEF77_JDQMR_ETol
PRIMMEF77_SUBSPACE_ITERATION
```

```
PRIMMEF77_LOBPCG_OrthoBasis
PRIMMEF77_LOBPCG_OrthoBasis_Window
```

See *primme_preset_method*.

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

## 3.3 primme_Free_f77

**primme_Free_f77** (primme)
Free memory allocated by PRIMME.

**Parameters**

- **primme** (ptr) – parameters structure.

## 3.4 dprimme_f77

**dprimme_f77** (evals, evecs, resNorms, primme, ierr)
Solve a real symmetric standard eigenproblem.

**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.
- **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.
- **evecs(*)** (*double precision*) – (input/output) array at least of size *nLocal* times *numEvals* to store columnwise the (local part of the) computed eigenvectors.
- **primme** (ptr) – parameters structure.
- **ierr** (*integer*) – (output) error indicator; see *Error Codes*.

## 3.5 zprimme_f77

**zprimme_f77** (evals, evecs, resNorms, primme, ierr)
Solve a Hermitian standard eigenproblem. The arguments have the same meaning as in function *dprimme_f77()*.

**Parameters**

- **evals(*)** (*double precision*) – (output)
- **resNorms(*)** (*double precision*) – (output)
- **evecs(*)** (*complex double precision*) – (input/output)
- **primme** (ptr) – (input) parameters structure.
- **ierr** (*integer*) – (output) error indicator; see *Error Codes*.

## 3.6 primmetop_set_member_f77

**primmetop_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:

*PRIMMEF77_n*
*PRIMMEF77_matrixMatvec*
*PRIMMEF77_applyPreconditioner*
*PRIMMEF77_numProcs*
*PRIMMEF77_procID*
*PRIMMEF77_commInfo*
*PRIMMEF77_nLocal*
*PRIMMEF77_globalSumDouble*
*PRIMMEF77_numEvals*
*PRIMMEF77_target*
*PRIMMEF77_numTargetShifts*
*PRIMMEF77_targetShifts*
*PRIMMEF77_locking*
*PRIMMEF77_initSize*
*PRIMMEF77_numOrthoConst*
*PRIMMEF77_maxBasisSize*
*PRIMMEF77_minRestartSize*
*PRIMMEF77_maxBlockSize*
*PRIMMEF77_maxMatvecs*
*PRIMMEF77_maxOuterIterations*
*PRIMMEF77_intWorkSize*
*PRIMMEF77_realWorkSize*
*PRIMMEF77_iseed*
*PRIMMEF77_intWork*
*PRIMMEF77_realWork*
*PRIMMEF77_aNorm*
*PRIMMEF77_eps*
*PRIMMEF77_printLevel*
*PRIMMEF77_outputFile*
*PRIMMEF77_matrix*
*PRIMMEF77_preconditioner*
*PRIMMEF77_restartingParams_scheme*.
*PRIMMEF77_restartingParams_maxPrevRetain*
*PRIMMEF77_correctionParams_precondition*
*PRIMMEF77_correctionParams_robustShifts*
*PRIMMEF77_correctionParams_maxInnerIterations*
*PRIMMEF77_correctionParams_projectors_LeftQ*

*PRIMMEF77_correctionParams_projectors_LeftX*
*PRIMMEF77_correctionParams_projectors_RightQ*
*PRIMMEF77_correctionParams_projectors_RightX*
*PRIMMEF77_correctionParams_projectors_SkewQ*
*PRIMMEF77_correctionParams_projectors_SkewX*
*PRIMMEF77_correctionParams_convTest*
*PRIMMEF77_correctionParams_relTolBase*
*PRIMMEF77_stats_numOuterIterations*
*PRIMMEF77_stats_numRestarts*
*PRIMMEF77_stats_numMatvecs*
*PRIMMEF77_stats_numPreconds*
*PRIMMEF77_stats_elapsedTime*
*PRIMMEF77_dynamicMethodSwitch*
*PRIMMEF77_massMatrixMatvec*

- **value** – (input) value to set.

---

**Note:** **Don't use** this function inside PRIMME's callback functions, e.g., *matrixMatvec* or *applyPreconditioner*, or in functions called by these functions. In those cases use *primme_set_member_f77()*.

---

## 3.7 primmetop_get_member_f77

**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 function *primmetop_set_member_f77()*.
- **value** – (output) value of the field.

---

**Note:** **Don't use** this function 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 `PRIMMEF77_matrixMatvec`, `PRIMMEF77_applyPreconditioner`, `PRIMMEF77_commInfo`, `PRIMMEF77_intWork`, `PRIMMEF77_realWork`, `PRIMMEF77_matrix` and `PRIMMEF77_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, PRIMMEF77_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, PRIMMEF77_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.

---

## 3.8 primmetop_get_prec_shift_f77

**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.

## 3.9 primme_set_member_f77

**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 the vales defined in *primmetop_set_member_f77()*.
- **value** – (input) value to set.

---

**Note:** Use this function 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 function *primmetop_set_member_f77()*.

---

## 3.10 primme_get_member_f77

**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 function *primmetop_set_member_f77()*.

- **value** – (output) value of the field.

**Note:** Use this function 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 function *primmetop_get_member_f77()*.

**Note:** When `label` is one of `PRIMMEF77_matrixMatvec`, `PRIMMEF77_applyPreconditioner`, `PRIMMEF77_commInfo`, `PRIMMEF77_intWork`, `PRIMMEF77_realWork`, `PRIMMEF77_matrix` and `PRIMMEF77_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, PRIMMEF77_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, PRIMMEF77_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.

## 3.11 primme_get_prec_shift_f77

**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 function exclusively inside the function *matrixMatvec*, *massMatrixMatvec*, or *applyPreconditioner*. Otherwise use the function *primmetop_get_prec_shift_f77()*.

CHAPTER

# FOUR

# APPENDIX

## 4.1 primme_params

**primme_params**
Structure to set the problem matrices and eigensolver options.

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*, void **y*, int **blockSize*, *primme_params* **primme*)
Block matrix-multivector multiplication, $y = Ax$ in solving $Ax = \lambda x$ or $Ax = \lambda Bx$.

**Parameters**

- **x** – one dimensional array containing the `blockSize` vectors packed one after the other (i.e., the leading dimension is the vector size), each of size *nLocal*. The real type is `double*` and `Complex_Z*` when called from *dprimme()* and *zprimme()* respectively.
- **y** – one dimensional array containing the `blockSize` vectors packed one after the other (i.e., the leading dimension is the vector size), each of size *nLocal*. The real type is `double*` and `Complex_Z*` when called from *dprimme()* and *zprimme()* respectively.
- **blockSize** – number of vectors in x and y.
- **primme** – parameters structure.

Input/output:

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

---

**Note:** Argument `blockSize` is passed by reference to make easier the interface to other languages (like Fortran).

---

void **(*applyPreconditioner)** (void **x*, void **y*, int **blockSize*, struct *primme_params* **primme*)
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 $\sigma$. The function follows the convention of *matrixMatvec*.

Input/output:

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

void **(*massMatrixMatvec)**(void *x, void *y, int *blockSize, struct *primme_params* *primme)
Block matrix-multivector multiplication, $y = Bx$ in solving $Ax = \lambda Bx$. The function follows the convention of *matrixMatvec*.

Input/output:

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

**Warning:** Generalized eigenproblems not implemented in current version. This member is included for future compatibility.

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 0;
*dprimme()* sets this field to 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* and *globalSumDouble*.

Input/output:

*primme_initialize()* sets this field to NULL;

void **(*globalSumDouble)**(double *sendBuf, double *recvBuf, int *count, *primme_params* *primme)
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.

Input/output:

*primme_initialize()* sets this field to an internal function;
*dprimme()* sets this field to an internal function if *numProcs* is 1 and *globalSumDouble* is NULL;
this field is read by *dprimme()*.

When MPI is used this can be a simply wrapper to MPI_Allreduce().

```
void par_GlobalSumDouble(void *sendBuf, void *recvBuf, int *count,
                         primme_params *primme) {
   MPI_Comm communicator = *(MPI_Comm *) primme->commInfo;
   MPI_Allreduce(sendBuf, recvBuf, *count, MPI_DOUBLE, MPI_SUM,
                 communicator);
}
```

---

**Note:** Argument count is passed by reference to make easier the interface to other languages (like Fortran).

---

---

**Note:** The arguments sendBuf and recvBuf are always double arrays and count is always the number of double elements in both arrays, even for *zprimme()*.

---

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 than 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* or *primme_closest_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* or *primme_closest_abs*.

Input/output:

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

The i-th shift (or the last one, if it is not given) is taken into account in finding the i-th eigenvalue.

---

**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.

---

int **printLevel**

The level of message reporting from the code. One of:

- 0: silent.
- 1: print some error messages when these occur.
- 2: as 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: 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: as 3, and info about targeted eigenpairs every inner iteration:

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

- 5: 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 outpufile | awk '{print $8" "$14}' > out
grep INN outpufile | awk '{print $3" "$11}' > inn
```

Then in Matlab:

```
plot(out(:,1),out(:,2),'bo');hold; plot(inn(:,1),inn(:,2),'r');
```

Or in gnuplot:

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

---

double **aNorm**
An estimate of the norm of $A$, which is used in the 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_initialize()* sets this field to 0.0;
this field is read and written by *dprimme()*.

double **eps**
An eigenpairs is marked as converged when the 2-norm of the residual is less than *eps* * *aNorm*. The residual vector is $Ax - \lambda x$ or $Ax - \lambda Bx$.

Input/output:

*primme_initialize()* sets this field to $10^{-12}$;
this field is read 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()*.

int **dynamicMethodSwitch**
If this value is 1, it alternates dynamically between *DEFAULT_MIN_TIME* and *DEFAULT_MIN_MATVECS*, trying to identify the fastest method.

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

-1: use *DEFAULT_MIN_MATVECS* next time.

-2: use *DEFAULT_MIN_TIME* next time.

-3: close call, use *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). Otherwise 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 0;

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()*.

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.

---

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()*.

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()*.

int **intWorkSize**
If *dprimme()* or *zprimme()* is called with all arguments as NULL except for *primme_params* then PRIMME returns immediately with *intWorkSize* containing the size *in bytes* of the integer workspace that will be required by the parameters set in PRIMME.

Otherwise if *intWorkSize* is not 0, it should be the size of the integer work array *in bytes* that the user provides in *intWork*. If *intWorkSize* is 0, the code will allocate the required space, which can be freed later by calling *primme_Free()*.

Input/output:

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

long int **realWorkSize**
If *dprimme()* or *zprimme()* is called with all arguments as NULL except for *primme_params* then PRIMME returns immediately with *realWorkSize* containing the size *in bytes* of the real workspace that will be required by the parameters set in PRIMME.

Otherwise if *realWorkSize* is not 0, it should be the size of the real work array *in bytes* that the user provides in *realWork*. If *realWorkSize* is 0, the code will allocate the required space, which can be freed later by calling *primme_Free()*.

Input/output:

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

int ***intWork**
Integer work array.

If NULL, the code will allocate its own workspace. If the provided space is not enough, the code will free it and allocate a new space.

On exit, the first element shows if a locking problem has occurred. Using locking for large *numEvals* may, in some rare cases, cause some pairs to be practically converged, in the sense that their components are in the basis of `evecs`. If this is the case, a Rayleigh Ritz on returned `evecs` would provide the accurate eigenvectors (see *[r4]*).

Input/output:

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

void ***realWork**
Real work array.

If NULL, the code will allocate its own workspace. If the provided space is not enough, the code will free it and allocate a new space.

Input/output:

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

int **iseed**
The `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 ***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_restartscheme **restartingParams.scheme**
Select a restarting strategy:

- `primme_thick`, Thick restarting. This is the most efficient and robust in the general case.
- `primme_dtr`, Dynamic thick restarting. Helpful without preconditioning but it is expensive to implement.

Input/output:

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

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 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 $\Lambda$ 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 $\Delta$ 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.

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()*.

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()*.

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()*.

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()*.

int **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()*.

## 4.2 Error Codes

The functions *dprimme()* and *zprimme()* return one of the next values:

- 0: success.
- 1: reported only amount of required memory.
- -1: failed in allocating int or real workspace.
- -2: malloc failed in allocating a permutation integer array.
- -3: main_iter() encountered problem; the calling stack of the functions where the error occurred was printed in `stderr`.
- -4: if argument `primme` is NULL.
- -5: if *n* <= 0 or *nLocal* <= 0.
- -6: if *numProcs* < 1.
- -7: if *matrixMatvec* is NULL.
- -8: if *applyPreconditioner* is NULL and *precondition* is not NULL.
- -9: if *globalSumDouble* is NULL.

- -10: if *numEvals* > *n*.
- -11: if *numEvals* < 0.
- -12: if *eps* > 0 and *eps* < machine precision.
- -13: if *target* is not properly defined.
- -14: if *target* is one of *primme_closest_geq*, *primme_closest_leq* or *primme_closest_abs* but *numTargetShifts* <= 0 (no shifts).
- -15: if *target* is one of *primme_closest_geq*, *primme_closest_leq* or *primme_closest_abs* but *targetShifts* is NULL (no shifts array).
- -16: if *numOrthoConst* < 0 or *numOrthoConst* >= *n*. (no free dimensions left).
- -17: if *maxBasisSize* < 2.
- -18: if *minRestartSize* <= 0.
- -19: if *maxBlockSize* <= 0.
- -20: if *maxPrevRetain* < 0.
- -21: if *scheme* is not one of *primme_thick* or *primme_dtr*.
- -22: if *initSize* < 0.
- -23: if not *locking* and *initSize* > *maxBasisSize*.
- -24: if *locking* and *initSize* > *numEvals*.
- -25: if *maxPrevRetain* + *minRestartSize* >= *maxBasisSize*.
- -26: if *minRestartSize* >= *n*.
- -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, but not `evecs` and `resNorms`.
- -31: if `evecs` is NULL, but not `evals` and `resNorms`.
- -32: if `resNorms` is NULL, but not `evecs` and `evals`.

## 4.3 Preset Methods

**primme_preset_method**

**DEFAULT_MIN_TIME**
Set as *JDQMR_ETol* when *target* is either `primme_smallest` or `primme_largest`, and as *JDQMR* otherwise. This method is usually the fastest if the cost of the matrix vector product is inexpensive.

**DEFAULT_MIN_MATVECS**
Currently set as *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.

**DYNAMIC**
Switches to the best method dynamically; currently, between methods *DEFAULT_MIN_TIME* and *DEFAULT_MIN_MATVECS*.

With *DYNAMIC* *primme_set_method()* sets *dynamicMethodSwitch* = 1 and makes the same changes as for method *JDQMR_ETol* when *target* is either `primme_smallest` or `primme_largest`, or as for method *JDQMR* otherwise.

**Arnoldi**
Arnoldi implemented à la Generalized Davidson.

With *Arnoldi* *primme_set_method()* sets:

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

**GD**
Generalized Davidson.

With *GD* *primme_set_method()* sets:

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

**GD_plusK**
GD with locally optimal restarting.

With *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.

**GD_Olsen_plusK**
GD+k and the cheap Olsen's Method.

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

**JD_Olsen_plusK**
GD+k and Olsen's Method.

With *JD_Olsen_plusK* *primme_set_method()* makes the same changes as for method *GD_plusK* and also sets *robustShifts* = 1, *RightX* to 1, and *SkewX* to 1.

**RQI**
(Accelerated) Rayleigh Quotient Iteration.

With *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.

---

**JDQR**
Jacobi-Davidson with fixed number of inner steps.

With *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*.

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

With *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*.

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

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

**SUBSPACE_ITERATION**
Subspace iteration.

With *SUBSPACE_ITERATION* *primme_set_method()* sets:

- *locking* = 1;
- *maxBasisSize* = *numEvals* * 2;
- *minRestartSize* = *numEvals*;

- *maxBlockSize* = *numEvals*;
- *scheme* = *primme_thick*;
- *maxPrevRetain* = 0;
- *robustShifts* = 0;
- *maxInnerIterations* = 0;
- *RightX* = 1;
- *SkewX* = 0.

**LOBPCG_OrthoBasis**
LOBPCG with orthogonal basis.

With *LOBPCG_OrthoBasis* *primme_set_method()* sets:

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

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

With *LOBPCG_OrthoBasis_Window* *primme_set_method()* sets:

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

CHAPTER

FIVE

# INDICES AND TABLES

- genindex
- search

[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.

[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.

## D



## I



## P

## S



## Z