# User Documentation for KINSOL v2.5.0

Aaron M. Collier, Alan C. Hindmarsh, Radu Serban, and Carol S. Woodward
*Center for Applied Scientific Computing*
*Lawrence Livermore National Laboratory*

November 6, 2006

UCRL-SM-208116

This research was supported under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

KINSOL is part of a software family called SUNDIALS: SUite of Nonlinear and DIfferential/ALgebraic equation Solvers. This suite consists of CVODE, KINSOL, and IDA, and variants of these. KINSOL is a general-purpose nonlinear system solver based on Newton-Krylov solver technology.

## 1.1 Historical Background

The first nonlinear solver packages based on Newton-Krylov methods were written in FORTRAN. In particular, the NKSOL package, written at LLNL, was the first Newton-Krylov solver package written for solution of systems arising in solution of partial differential equations [2]. This FORTRAN code made use of Newton's method to solve the discrete nonlinear systems and applied a preconditioned Krylov linear solver for solution of the Jacobian system at each nonlinear iteration. The key to the Newton-Krylov method was that the matrix-vector multiplies required by the Krylov method could effectively be approximated by a finite difference of the nonlinear system-defining function, preventing a requirement for the formation of the actual Jacobian matrix. Significantly less memory was required for the solver as a result.

In the late 1990's, there was a push at LLNL to rewrite the nonlinear solver into C and port it to distributed memory parallel machines. Both Newton and Krylov methods are easily implemented in parallel, and this effort gave rise to the KINSOL package. KINSOL is similar to NKSOL in functionality, except that it provides for more options in the choice of linear system tolerances and has a more modular design to provide flexibility for future enhancements.

At present, KINSOL contains three Krylov methods: the GMRES (Generalized Minimal RESidual) [9], Bi-CGStab (Bi-Conjugate Gradient Stabilized) [10], and TFQMR (Transpose-Free Quasi-Minimal Residual) [7] linear iterative methods. As Krylov methods, these require almost no matrix storage as compared to direct methods. However, the algorithms allow for a user-supplied preconditioner matrix, and for most problems preconditioning is essential for an efficient solution. For very large nonlinear algebraic systems, the Krylov methods are preferable over direct linear solver methods, and are often the only feasible choice. Among the three Krylov methods in KINSOL, we recommend GMRES as the best overall choice. However, users are encouraged to compare all three, especially if encountering convergence failures with GMRES. Bi-CGStab and TFQMR have an advantage in storage requirements, in that the number of workspace vectors they require is fixed, while that number for GMRES depends on the desired Krylov subspace size.

In the process of translating NKSOL into C, the overall KINSOL organization has been changed considerably. One key feature of the KINSOL organization is that a separate module devoted to vector operations has been created. This module facilitated extension to multiprosessor environments with minimal impact on the rest of the solver. The new vector module design is shared across the SUNDIALS suite. This NVECTOR module is written in terms of abstract vector operations with the actual routines attached by a particular implementation (such as serial or parallel) of NVECTOR. This allows writing the SUNDIALS solvers in a manner independent of the actual NVECTOR implementation (which can be user-supplied), as well as allowing more than one NVECTOR module linked into an executable file.

There are several motivations for choosing the C language for KINSOL. First, a general movement away from FORTRAN and toward C in scientific computing is apparent. Second, the pointer, structure, and dynamic memory allocation features in C are extremely useful in software of this complexity, with the great variety of method options offered. Finally, we prefer C over C++ for KINSOL because of the wider availability of C compilers, the potentially greater efficiency of C, and the greater ease of interfacing the solver to applications written in FORTRAN.

## 1.2 Changes from previous versions

### Changes in v2.5.0

The main changes in this release involve a rearrangement of the entire SUNDIALS source tree (see §4.1). At the user interface level, the main impact is in the mechanism of including SUNDIALS header files which must now include the relative path (e.g. `#include <cvode/cvode.h>`). Additional changes were made to the build system: all exported header files are now installed in separate subdirectories of the instaltion *include* directory.

The functions in the generic dense linear solver (`sundials_dense` and `sundials_smalldense`) were modified to work for rectangular $m \times n$ matrices ($m \leq n$), while the factorization and solution functions were renamed to `DenseGETRF`/`denGETRF` and `DenseGETRS`/`denGETRS`, respectively. The factorization and solution functions in the generic band linear solver were renamed `BandGBTRF` and `BandGBTRS`, respectively.

### Changes in v2.4.0

KINSPBCG, KINSPTFQMR, KINDENSE, and KINBAND modules have been added to interface with the Scaled Preconditioned Bi-CGStab (SPBCG), Scaled Preconditioned Transpose-Free Quasi-Minimal Residual (SPTFQMR), DENSE, and BAND linear solver modules, respectively (for details see Chapter 5). Corresponding additions were made to the FORTRAN interface module FKINSOL. At the same time, function type names for Scaled Preconditioned Iterative Linear Solvers were added for the user-supplied Jacobian-times-vector and preconditioner setup and solve functions.

Regarding the FORTRAN interface module FKINSOL, optional inputs are now set using `FKINSETIIN` (integer inputs), `FKINSETRIN` (real inputs), and `FKINSETVIN` (vector inputs). Optional outputs are still obtained from the `IOUT` and `ROUT` arrays which are owned by the user and passed as arguments to `FKINMALLOC`.

The KINDENSE and KINBAND linear solver modules include support for nonlinear residual monitoring which can be used to control Jacobian updating.

To reduce the possibility of conflicts, the names of all header files have been changed by adding unique prefixes (`kinsol_` and `sundials_`). When using the default installation procedure, the header files are exported under various subdirectories of the target `include` directory. For more details see §2.

### Changes in v2.3.0

The user interface has been further refined. Several functions used for setting optional inputs were combined into a single one. Additionally, to resolve potential variable scope issues, all SUNDIALS solvers release user data right after its use. The build systems has been further improved to make it more robust.

### Changes in v2.2.1

The changes in this minor SUNDIALS release affect only the build system.

### Changes in v2.2.0

The major changes from the previous version involve a redesign of the user interface across the entire SUNDIALS suite. We have eliminated the mechanism of providing optional inputs and extracting optional statistics from the solver through the `iopt` and `ropt` arrays. Instead, KINSOL now provides a set of routines (with prefix `KINSet`) to change the default values for various quantities controlling the solver and a set of extraction routines (with prefix `KINGet`) to extract statistics after return from the main solver routine. Similarly, each linear solver module provides its own set of `set`- and `get`-type routines. For more details see Chapter 5.

Additionally, the interfaces to several user-supplied routines (such as those providing Jacobian-vector products and preconditioner information) were simplified by reducing the number of arguments. The same information that was previously accessible through such arguments can now be obtained through `set`-type functions.

## 1.3 Reading this User Guide

This user guide is a combination of general usage instructions and specific examples. We expect that some readers will want to concentrate on the general instructions, while others will refer mostly to the examples, and the organization is intended to accommodate both styles.

There are different possible levels of usage of KINSOL. The most casual user, with a small nonlinear system, can get by with reading all of Chapter 3, then Chapter 5 through §5.5.3 only, and looking at examples in [3]. In a different direction, a more expert user with a nonlinear system may want to (a) use a package preconditioner (§5.7), (b) supply his/her own Jacobian or preconditioner routines (§5.6), (c) supply a new NVECTOR module (Chapter 7), or even (d) supply a different linear solver module (§4.2 and Chapter 9).

The structure of this document is as follows:

- In Chapter 2 we begin with instructions for the installation of KINSOL, within the structure of SUNDIALS.
- In Chapter 3, we provide short descriptions of the numerical methods implemented by KINSOL for the solution of nonlinear systems.
- The following chapter describes the structure of the SUNDIALS suite of solvers (§4.1) and the software organization of the KINSOL solver (§4.2).
- Chapter 5 is the main usage document for KINSOL for C applications. It includes a complete description of the user interface for the solution of nonlinear algebraic systems.
- In Chapter 6, we describe FKINSOL, an interface module for the use of KINSOL with FORTRAN applications.
- Chapter 7 gives a brief overview of the generic NVECTOR module shared among the various components of SUNDIALS, and details on the two NVECTOR implementations provided with SUNDIALS: a serial implementation (§7.1) and a parallel implementation based on MPI (§7.2).
- Chapter 8 describes the interfaces to the linear solver modules, so that a user can provide his/her own such module.
- Chapter 9 describes in detail the generic linear solvers shared by all SUNDIALS solvers.
- Finally, Chapter 10 lists the constants used for input to and output from KINSOL.

Finally, the reader should be aware of the following notational conventions in this user guide: program listings and identifiers (such as `KINMalloc`) within textual explanations appear in typewriter type style; fields in C structures (such as *content*) appear in italics; and packages or modules are written in all capitals. In the index, page numbers that appear in bold indicate the main reference for that entry.

**Acknowledgments.** We wish to acknowledge the contributions to previous versions of the KINSOL code and user guide of Allan G. Taylor.

# Chapter 2

# KINSOL Installation Procedure

The installation of KINSOL is accomplished by installing the SUNDIALS suite as a whole, according to the instructions that follow. The same procedure applies whether or not the downloaded file contains solvers other than KINSOL. [1]

The SUNDIALS suite (or individual solvers) are distributed as compressed archives (`.tgz`). The name of the distribution archive is of the form *solver*-`x.y.z.tgz`, where *solver* is one of: `sundials`, `cvode`, `cvodes`, `ida`, or `kinsol`, and `x.y.z` represents the version number (of the SUNDIALS suite or of the individual solver).

To begin the installation, first uncompress and expand the sources, by issuing

```
% tar xzf solver-x.y.z.tgz
```

This will extract source files under a directory *solver*-`x.y.z`.

The installation procedure outlined below will work on commodity LINUX/UNIX systems without modification. However, users are still encouraged to carefully read the entire chapter before attempting to install the SUNDIALS suite, in case non-default choices are desired for compilers, compilation options, or the like. In lieu of reading the option list below, the user may invoke the configuration script with the help flag to view a complete listing of available options, which may be done by issuing

```
% ./configure --help
```

from within the directory created above.

In the remainder of this chapter, we make the following distinctions:

- *srcdir*

  is the directory *solver*-`x.y.z` created above; i.e., the directory containing the SUNDIALS sources.

- *builddir*

  is the directory under which SUNDIALS is built; i.e., the directory from within which the `configure` command is issued. Usually, this is the same as *srcdir*.

- *instdir*

  is the directory under which the SUNDIALS exported header files and libraries will be installed. Typically, header files are exported under a directory *instdir*`/include` while libraries are installed under *instdir*`/lib`, with *instdir* specified with the `--prefix` flag to `configure`. See §2.1 for more details on the installation directories, including the special cases of the SUNDIALS examples and the SUNDIALSTB toolbox.

*Note*: The installation directory *instdir* should *not* be the same as the source directory *srcdir*. !

[1] Files for both the serial and parallel versions of KINSOL are included in the distribution. For users in a serial computing environment, the files specific to parallel environments (which may be deleted) are as follows: all files in `src/nvec_par`; `kinbbdpre.c, kinbbdpre_impl.h` (in `src/kinsol/`); `kinbbdpre.h` (in `include/kinsol/`); `fkinbbd.c, fkinbbd.h` (in `src/kinsol/fcmix/`); all files in `examples/kinsol/parallel/`; all files in `examples/kinsol/fcmix_parallel/`. (By "serial version" of KINSOL we mean the KINSOL solver with the serial NVECTOR module attached, and similarly for "parallel version".)

The installation steps for SUNDIALS can be as simple as

```
% tar xzf solver-x.y.z.tgz
% cd solver-x.y.z
% ./configure
% make
% make install
```

in which case the SUNDIALS header files and libraries are installed under `/usr/local/include` and `/usr/local/lib`, respectively. Note that, by default, neither the example programs nor the SUNDIALSTB toolbox are built and installed.

If disk space is a priority, then to delete all temporary files created by building SUNDIALS, issue

```
% make clean
```

To prepare the SUNDIALS distribution for a new install (using, for example, different options and/or installation destinations), issue

```
% make distclean
```

## 2.1 Configuration options

The installation procedure given above will generally work without modification; however, if the system includes multiple MPI implementations, then certain configure script-related options may be used to indicate which MPI implementation should be used. Also, if the user wants to use non-default language compilers, then, again, the necessary shell environment variables must be appropriately redefined. The remainder of this section provides explanations of available configure script options.

### General options

`--prefix=PREFIX`

Location for architecture-independent files.

Default: `PREFIX`=/usr/local

`--exec-prefix=EPREFIX`

Location for architecture-dependent files.

Default: `EPREFIX`=/usr/local

`--includedir=DIR`

Alternate location for installation of header files.

Default: `DIR=PREFIX/include`

`--libdir=DIR`

Alternate location for installation of libraries.

Default: `DIR=EPREFIX/lib`

`--disable-`*solver*

Although each existing solver module is built by default, support for a given solver can be explicitly disabled using this option. The valid values for *solver* are: `cvode`, `cvodes`, `ida`, and `kinsol`.

`--enable-examples`

Available example programs are *not* built by default. Use this option to enable compilation of all pertinent example programs. Upon completion of the `make` command, the example executables will be created under solver-specific subdirectories of *builddir*/`examples`:

*builddir*/examples/*solver*/serial : serial C examples

*builddir*/examples/*solver*/parallel : parallel C examples

*builddir*/examples/*solver*/fcmix_serial : serial Fortran examples

*builddir*/examples/*solver*/fcmix_parallel : parallel Fortran examples

*Note*: Some of these subdirectories may not exist depending upon the solver and/or the configuration options given.

`--with-examples-instdir=EXINSTDIR`

Alternate location for example executables and sample output files (valid only if examples are enabled). Note that installtion of example files can be completely disabled by issuing `EXINSTDIR=no` (in case building the examples is desired only as a test of the sundials libraries).

Default: `DIR=EPREFIX/examples`

`--with-cppflags=ARG`

Specify additional C preprocessor flags (e.g., `ARG=-I<include_dir>` if necessary header files are located in nonstandard locations).

`--with-cflags=ARG`

Specify additional C compilation flags.

`--with-ldflags=ARG`

Specify additional linker flags (e.g., `ARG=-L<lib_dir>` if required libraries are located in nonstandard locations).

`--with-libs=ARG`

Specify additional libraries to be used (e.g., `ARG=-l<foo>` to link with the library named `libfoo.a` or `libfoo.so`).

`--with-precision=ARG`

By default, sundials will define a real number (internally referred to as `realtype`) to be a double-precision floating-point numeric data type (`double` C-type); however, this option may be used to build sundials with `realtype` alternatively defined as a single-precision floating-point numeric data type (`float` C-type) if `ARG=single`, or as a `long double` C-type if `ARG=extended`.

Default: `ARG=double`

Users should *not* build sundials with support for single-precision floating-point arithmetic on 32- or 64-bit systems. This will almost certainly result in unreliable numerical solutions. The configuration option `--with-precision=single` is intended for systems on which single-precision arithmetic involves at least 14 decimal digits.

## Options for Fortran support

`--disable-fcmix`

Using this option will disable all Fortran support. The fcvode, fkinsol, fida, and fnvector modules will not be built, regardless of availability.

`--with-fflags=ARG`

Specify additional Fortran compilation flags.

The configuration script will attempt to automatically determine the function name mangling scheme required by the specified Fortran compiler, but the following two options may be used to override the default behavior.

`--with-f77underscore=ARG`

This option pertains to the FCVODE, FKINSOL, FIDA, and FNVECTOR FORTRAN-C interface modules and is used to specify the number of underscores to append to function names so FORTRAN routines can properly link with the associated SUNDIALS libraries. Valid values for `ARG` are: `none`, `one` and `two`.

Default: `ARG=one`

`--with-f77case=ARG`

Use this option to specify whether the external names of the FCVODE, FKINSOL, FIDA, and FNVECTOR FORTRAN-C interface functions should be lowercase or uppercase so FORTRAN routines can properly link with the associated SUNDIALS libraries. Valid values for `ARG` are: `lower` and `upper`.

Default: `ARG=lower`

## Options for MPI support

The following configuration options are only applicable to the parallel SUNDIALS packages:

`--disable-mpi`

Using this option will completely disable MPI support.

`--with-mpicc=ARG`

`--with-mpif77=ARG`

By default, the configuration utility script will use the MPI compiler scripts named `mpicc` and `mpif77` to compile the parallelized SUNDIALS subroutines; however, for reasons of compatibility, different executable names may be specified via the above options. Also, `ARG=no` can be used to disable the use of MPI compiler scripts, thus causing the serial C and FORTRAN compilers to be used to compile the parallelized SUNDIALS functions and examples.

`--with-mpi-root=MPIDIR`

This option may be used to specify which MPI implementation should be used. The SUNDIALS configuration script will automatically check under the subdirectories `MPIDIR/include` and `MPIDIR/lib` for the necessary header files and libraries. The subdirectory `MPIDIR/bin` will also be searched for the C and FORTRAN MPI compiler scripts, unless the user uses `--with-mpicc=no` or `--with-mpif77=no`.

`--with-mpi-incdir=INCDIR`

`--with-mpi-libdir=LIBDIR`

`--with-mpi-libs=LIBS`

These options may be used if the user would prefer not to use a preexisting MPI compiler script, but instead would rather use a serial complier and provide the flags necessary to compile the MPI-aware subroutines in SUNDIALS.

Often an MPI implementation will have unique library names and so it may be necessary to specify the appropriate libraries to use (e.g., `LIBS=-lmpich`).

Default: `INCDIR=MPIDIR/include` and `LIBDIR=MPIDIR/lib`

`--with-mpi-flags=ARG`

Specify additional MPI-specific flags.

## Options for library support

By default, only static libraries are built, but the following option may be used to build shared libraries on supported platforms.

`--enable-shared`

Using this particular option will result in both static and shared versions of the available SUNDIALS libraries being built if the system supports shared libraries. To build only shared libraries also specify `--disable-static`.

*Note*: The FCVODE, FKINSOL, and FIDA libraries can only be built as static libraries because they contain references to externally defined symbols, namely user-supplied FORTRAN subroutines. Although the FORTRAN interfaces to the serial and parallel implementations of the supplied NVECTOR module do not contain any unresolvable external symbols, the libraries are still built as static libraries for the purpose of consistency.

## Options for Matlab support

The following options are relevant only for configuring and building the SUNDIALSTB Matlab toolbox:

`--enable-sundialsTB`

The SUNDIALSTB Matlab toolbox is *not* built by default. Use this option to enable configuration and compilation of the `mex` files. Upon completion of the `make` command, the following `mex` files will be created:

*builddir*`/sundialsTB/cvodes/cvm/cvm.`*mexext*

*builddir*`/sundialsTB/idas/idm/idm.`*mexext*

*builddir*`/sundialsTB/kinsol/kim/kim.`*mexext*

where *mexext* is the platform-specific extension of `mex` files.

`--with-sundialsTB-instdir=STBINSTDIR`

Alternate location for the installed SUNDIALSTB toolbox (valid only if SUNDIALSTB is enabled). As for the example programs, installation of SUNDIALSTB can be completely disabled by issuing `STBINSTDIR=no` (in case building the toolbox is desired but its installtion will be done manually afterwards). Otherwise, all required SUNDIALSTB files will be installed under the directory `STBINSTDIR/sundialsTB`.

Default: `DIR=MATLAB/toolbox` (see below for the definition of `MATLAB`).

`--with-matlab=MATLAB`

This option can be used to specify the location of the Matlab executable. The default is to search the path.

`--with-mexopts=ARG`

Specify the `mex` options file to be used.

Default: Standard Matlab `mex` options file.

`--with-mexflags=ARG`

Specify the `mex` compiler flags to be used.

Default: `ARG=-O`

`--with-mexldadd=ARG`

Specify additional `mex` linker flags.

Default: none

### Environment variables

The following environment variables can be locally (re)defined for use during the configuration of SUNDIALS. See the next section for illustrations of these.

`CC`

`F77`

Since the configuration script uses the first C and FORTRAN compilers found in the current executable search path, then each relevant shell variable (`CC` and `F77`) must be locally (re)defined in order to use a different compiler. For example, to use `xcc` (executable name of chosen compiler) as the C language compiler, use `CC=xcc` in the configure step.

`CFLAGS`

`FFLAGS`

Use these environment variables to override the default C and FORTRAN compilation flags.

## 2.2 Configuration examples

The following examples are meant to help demonstrate proper usage of the configure options.

To build SUNDIALS using the default C and Fortran compilers, and default `mpicc` and `mpif77` parallel compilers, enable compilation of examples, build the Matlab `mex` files for SUNDIALSTB, and install it under `/home/myname/matlab/sundialsTB`, use

```
% configure --prefix=/home/myname/sundials --enable-examples \
            --enable-sundialsTB --with-sundialsTB-instdir=/home/myname/matlab
```

To disable installation of the examples, use:

```
% configure --prefix=/home/myname/sundials \
            --enable-examples --with-examples-instdir=no \
            --enable-sundialsTB --with-sundialsTB-instdir=/home/myname/matlab
```

The following example builds SUNDIALS using `gcc` as the serial C compiler, `g77` as the serial FORTRAN compiler, `mpicc` as the parallel C compiler, `mpif77` as the parallel FORTRAN compiler, and appends the `-g3` compilaton flag to the list of default flags:

```
% configure CC=gcc F77=g77 --with-cflags=-g3 --with-fflags=-g3 \
            --with-mpicc=/usr/apps/mpich/1.2.4/bin/mpicc \
            --with-mpif77=/usr/apps/mpich/1.2.4/bin/mpif77
```

The next example again builds SUNDIALS using `gcc` as the serial C compiler, but the `--with-mpicc=no` option explicitly disables the use of the corresponding MPI compiler script. In addition, since the `--with-mpi-root` option is given, the compilation flags `-I/usr/apps/mpich/1.2.4/include` and `-L/usr/apps/mpich/1.2.4/lib` are passed to `gcc` when compiling the MPI-enabled functions. The `--disable-examples` option explicitly disables the examples (which means a FORTRAN compiler is not required). The `--with-mpi-libs` option is required so that the configure script can check if `gcc` can link with the appropriate MPI library.

```
% configure CC=gcc --disable-examples --with-mpicc=no \
            --with-mpi-root=/usr/apps/mpich/1.2.4 \
            --with-mpi-libs=-lmpich
```

## 2.3 Installed libraries and exported header files

Using the standard SUNDIALS build system, the command

```
% make install
```

will install the libraries under *libdir* and the public header files under *includedir*. The default values for these directories are *instdir*/`lib` and *instdir*/`include`, respectively, but can be changed using the configure script options `--prefix`, `--exec-prefix`, `--includedir` and `--libdir` (see §2.1). For example, a global installation of SUNDIALS on a *NIX system could be accomplished using

```
% configure --prefix=/opt/sundials-2.1.1
```

Although all installed libraries reside under *libdir*, the public header files are further organized into subdirectories under *includedir*.

The installed libraries and exported header files are listed for reference in Table 2.1. The file extension *.lib* is typically `.so` for shared libraries and `.a` for static libraries (see *Options for library support* for additional details).

A typical user program need not explicitly include any of the shared SUNDIALS header files from under the *includedir*/`sundials` directory since they are explicitly included by the appropriate solver header files (*e.g.*, `cvode_dense.h` includes `sundials_dense.h`). However, it is both legal and safe to do so (*e.g.*, the functions declared in `sundials_smalldense.h` could be used in building a preconditioner).

## 2.4 Building SUNDIALS without the configure script

If the `configure` script cannot be used (*e.g.*, when building SUNDIALS under Microsoft Windows without using Cygwin), or if the user prefers to own the build process (*e.g.*, when SUNDIALS is incorporated into a larger project with its own build system), then the header and source files for a given module can be copied from the *srcdir* to some other location and compiled separately.

The following files are required to compile a SUNDIALS solver module:

- public header files located under *srcdir*/`include/`*solver*
- implementation header files and source files located under *srcdir*/`src/`*solver*
- (optional) FORTRAN/C interface files located under *srcdir*/`src/`*solver*/`fcmix`
- shared public header files located under *srcdir*/`include/sundials`
- shared source files located under *srcdir*/`src/sundials`
- (optional) NVECTOR_SERIAL header and source files located under *srcdir*/`include/nvector` and *srcdir*/`src/nvec_ser`
- (optional) NVECTOR_PARALLEL header and source files located under *srcdir*/`include/nvector` and *srcdir*/`src/nvec_par`
- configuration header file `sundials_config.h` (see below)

A sample header file that, appropriately modified, can be used as `sundials_config.h` (otherwise created automatically by the `configure` script) is provided below. The various preprocessor macros defined within `sundials_config.h` have the following uses:

- Precision of the SUNDIALS `realtype` type

  Only one of the macros `SUNDIALS_SINGLE_PRECISION`, `SUNDIALS_DOUBLE_PRECISION` and `SUNDIALS_EXTENDED_PRECISION` should be defined to indicate if the SUNDIALS `realtype` type is an alias for `float`, `double`, or `long double`, respectively.

Table 2.1: SUNDIALS libraries and header files (names are relative to *libdir* for libraries and to *includedir* for header files)

| | | | |
|---|---|---|---|
| SHARED | Libraries | n/a | |
| | Header files | sundials/sundials_types.h | sundials/sundials_math.h |
| | | sundials/sundials_config.h | sundias/sundials_nvector.h |
| | | sundials/sunials_smalldense.h | sundials/sundials_dense.h |
| | | sundials/sundials_iterative.h | sundials/sundials_band.h |
| | | sundials/sundials_spbcgs.h | sundials/sundials_sptfqmr.h |
| | | sundials/sundials_spgmr.h | |
| NVECTOR_SERIAL | Libraries | libsundials_nvecserial.*lib* | libsundials_fnvecserial.a |
| | Header files | nvector/nvector_serial.h | |
| NVECTOR_PARALLEL | Libraries | libsundials_nvecparallel.*lib* | libsundials_fnvecparallel.a |
| | Header files | nvector/nvector_parallel.h | |
| CVODE | Libraries | libsundials_cvode.*lib* | libsundials_fcvode.a |
| | Header files | cvode/cvode.h | |
| | | cvode/cvode_dense.h | cvode/cvode_band.h |
| | | cvode/cvode_diag.h | cvode/cvode_spils.h |
| | | cvode/cvode_bandpre.h | cvode/cvode_bbdpre.h |
| | | cvode/cvode_spgmr.h | cvode/cvode_spbcgs.h |
| | | cvode/cvode_sptfqmr.h | cvode/cvode_impl.h |
| CVODES | Libraries | libsundials_cvodes.*lib* | |
| | Header files | cvodes/cvodes.h | |
| | | cvodes/cvodes_dense.h | cvodes/cvodes_band.h |
| | | cvodes/cvodes_diag.h | cvodes/cvodes_spils.h |
| | | cvodes/cvodes_bandpre.h | cvodes/cvodes_bbdpre.h |
| | | cvodes/cvodes_spgmr.h | cvodes/cvodes_spbcgs.h |
| | | cvodes/cvodes_sptfqmr.h | cvodes/cvodes_impl.h |
| | | cvodes/cvodea_impl.h | |
| IDA | Libraries | libsundials_ida.*lib* | libsundials_fida.a |
| | Header files | ida/ida.h | |
| | | ida/ida_dense.h | ida/ida_band.h |
| | | ida/ida_spils.h | ida/ida_spgmr.h |
| | | ida/ida_spbcgs.h | ida/ida_sptfqmr.h |
| | | ida/ida_bbdpre.h | ida/ida_impl.h |
| KINSOL | Libraries | libsundials_kinsol.*lib* | libsundials_fkinsol.a |
| | Header files | kinsol/kinsol.h | |
| | | kinsol/kinsol_dense.h | kinsol/kinsol_band.h |
| | | kinsol/kinsol_spils.h | kinsol/kinsol_spgmr.h |
| | | kinsol/kinsol_spbcgs.h | kinsol/kinsol_sptfqmr.h |
| | | kinsol/kinsol_bbdpre.h | kinsol/kinsol_impl.h |

- Use of generic math functions

  If SUNDIALS_USE_GENERIC_MATH is defined, then the functions in sundials_math.(h,c) will use the pow, sqrt, fabs, and exp functions from the standard math library (see math.h), regardless of the definition of realtype. Otherwise, if realtype is defined to be an alias for the float C-type, then SUNDIALS will use powf, sqrtf, fabsf, and expf. If realtype is instead defined to be a synonym for the long double C-type, then powl, sqrtl, fabsl, and expl will be used.

  *Note*: Although the powf/powl, sqrtf/sqrtl, fabsf/fabsl, and expf/expl routines are not specified in the ANSI C standard, they are ISO C99 requirements. Consequently, these routines will only be used if available.

- FORTRAN name-mangling scheme

  The macros given below are used to transform the C-language function names defined in the FORTRAN-C inteface modules in a manner consistent with the preferred FORTRAN compiler, thus allowing native C functions to be called from within a FORTRAN subroutine. The name-mangling scheme can be specified either by appropriately defining the parameterized macros (using the stringization operator, ##, if necessary)

  – F77_FUNC(name,NAME)
  – F77_FUNC_(name,NAME)

  or by defining *one* macro from each of the following lists:

  – SUNDIALS_CASE_LOWER or SUNDIALS_CASE_UPPER
  – SUNDIALS_UNDERSCORE_NONE, SUNDIALS_UNDERSCORE_ONE, or SUNDIALS_UNDERSCORE_TWO

  For example, to specify that mangled C-language function names should be lowercase with one underscore appended include either

  ```
  #define F77_FUNC(name,NAME) name ## _
  #define F77_FUNC_(name,NAME) name ## _
  ```

  or

  ```
  #define SUNDIALS_CASE_LOWER 1
  #define SUNDIALS_UNDERSCORE_ONE 1
  ```

  in the sundials_config.h header file.

- Use of an MPI communicator other than MPI_COMM_WORLD in FORTRAN

  If the macro SUNDIALS_MPI_COMM_F2C is defined, then the MPI implementation used to build SUNDIALS defines the type MPI_Fint and the function MPI_Comm_f2c, and it is possible to use MPI communicators other than MPI_COMM_WORLD with the FORTRAN-C interface modules.

```c
/*
 * -----------------------------------------------------------------
 * Copyright (c) 2005, The Regents of the University of California.
 * Produced at the Lawrence Livermore National Laboratory.
 * All rights reserved.
 * For details, see sundials/shared/LICENSE.
 * -----------------------------------------------------------------
 * SUNDIALS configuration header file
 * -----------------------------------------------------------------
 */


/* Define SUNDIALS version number
 * ------------------------------ */

#define SUNDIALS_PACKAGE_VERSION "2.2.1"

/* Define precision of SUNDIALS data type 'realtype'
 * ------------------------------------------------- */

/* Define SUNDIALS data type 'realtype' as 'double' */
#define SUNDIALS_DOUBLE_PRECISION 1

/* Define SUNDIALS data type 'realtype' as 'float' */
/* #define SUNDIALS_SINGLE_PRECISION 1 */

/* Define SUNDIALS data type 'realtype' as 'long double' */
/* #define SUNDIALS_EXTENDED_PRECISION 1 */

/* Use generic math functions
 * -------------------------- */

#define SUNDIALS_USE_GENERIC_MATH 1

/* FCMIX: Define Fortran name-mangling macro
 * ----------------------------------------- */

#define F77_FUNC(name,NAME) name ## _
#define F77_FUNC_(name,NAME) name ## _

/* FCMIX: Define case of function names
 * ------------------------------------ */

/* FCMIX: Make function names lowercase */
/* #define SUNDIALS_CASE_LOWER 1 */

/* FCMIX: Make function names uppercase */
/* #define SUNDIALS_CASE_UPPER 1 */

/* FCMIX: Define number of underscores to append to function names
 * --------------------------------------------------------------- */

/* FCMIX: Do NOT append any underscores to functions names */
/* #define SUNDIALS_UNDERSCORE_NONE 1 */

/* FCMIX: Append ONE underscore to function names */
/* #define SUNDIALS_UNDERSCORE_ONE 1 */

/* FCMIX: Append TWO underscores to function names */
/* #define SUNDIALS_UNDERSCORE_TWO 1 */

/* FNVECTOR: Allow user to specify different MPI communicator
 * ---------------------------------------------------------- */

#define SUNDIALS_MPI_COMM_F2C 1
```

# Chapter 3

# Mathematical Considerations

KINSOL solves nonlinear algebraic systems in real $N$-space,

$$F(u) = 0\,, \quad F : \mathbf{R}^N \rightarrow \mathbf{R}^N\,, \tag{3.1}$$

given an initial guess $u_0$.

### Basic Newton iteration

Depending on the linear solver used, KINSOL can employ either an Inexact Newton method [1, 2, 4, 5, 8], or a Modified Newton method. At the highest level, KINSOL implements the following iteration scheme:

1. Set $u_0$ = an initial guess
2. For $n = 0, 1, 2, ...$ until convergence do:
   (a) Solve $J(u_n)\delta_n = -F(u_n)$
   (b) Set $u_{n+1} = u_n + \lambda\delta_n$, $0 < \lambda \leq 1$
   (c) Test for convergence

Here, $u_n$ is the $n$th iterate to $u$, and $J(u) = F'(u)$ is the system Jacobian. At each stage in the iteration process, a scalar multiple of the step $\delta_n$, is added to $u_n$ to produce a new iterate, $u_{n+1}$. A test for convergence is made before the iteration continues.

### Newton method variants

For solving the linear system given in step 2(a), KINSOL provides a choice of five methods:

- dense direct solver,
- band direct solver,
- scaled preconditioned GMRES (Generalized Minimal Residual method),
- scaled preconditioned Bi-CGStab (Bi-Conjugate Gradient Stable method), or
- scaled preconditioned TFQMR (Transpose-Free Quasi-Minimal Residual method).

When using one of the direct linear solvers, the linear system in 2(a) is solved exactly thus resulting in a Modified Newton method (the Jacobian matrix is normally outdated; see below[1]). Note that the direct linear solvers (dense and band) can only be used with serial vector representations.

[1] KINSOL allows the user to enforce a Jacobian evaluation at each iteration thus allowing for an Exact Newton iteration.

On the other hand, using any of the iterative linear solvers (GMRES, Bi-CGStab, or TFQMR), the linear system in 2(a) is solved only approximately thus resulting in an Inexact Newton method. In the latter case, right preconditioning is available if the preconditioning setup and solve routines are supplied by the user, in which case the iterative method is applied to the linear systems $(JP^{-1})(P\delta) = -F$, where $P$ denotes the right preconditioning matrix.

### Jacobian information update strategy

In general, unless specified otherwise by the user, KINSOL strives to update Jacobian information (the actual system Jacobian $J$ in the case of direct linear solvers, or the preconditioner matrix $P$ in the case of iterative linear solvers) as infrequently as possible to balance the high costs of matrix operations against other costs. Specifically, these updates occur when:

- the problem is initialized,
- $\|\lambda\delta_{n-1}\|_{D_u,\infty} > 1.5$ (Inexact Newton only),
- `mbset`$= 10$ nonlinear iterations have passed since the last update,
- the linear solver failed recoverably with outdated Jacobian information,
- the global strategy failed with outdated Jacobian information, or
- $\|\lambda\delta_n\|_{D_u,\infty} <$ STEPTOL with outdated Jacobian information.

KINSOL allows, through optional solver inputs, changes to the above strategy. Indeed, the user can disable the initial Jacobian information evaluation or change the default value of `mbset`, the number of nonlinear iterations after which a Jacobian information update is enforced.

### Scaling

To address the case of ill-conditioned nonlinear systems, KINSOL allows prescribing scaling factors both for the solution vector and for the residual vector. For scaling to be used, the user should supply values $D_u$, which are diagonal elements of the scaling matrix such that $D_u u_n$ has all components roughly the same magnitude when $u_n$ is close to a solution, and $D_F$, which are diagonal scaling matrix elements such that $D_F F$ has all components roughly the same magnitude when $u_n$ is not too close to a solution. In the text below, we use the following scaled norms:

$$\|z\|_{D_u} = \|D_u z\|_2, \quad \|z\|_{D_F} = \|D_F z\|_2, \quad \|z\|_{D_u,\infty} = \|D_u z\|_\infty, \quad \text{and} \quad \|z\|_{D_F,\infty} = \|D_F z\|_\infty \tag{3.2}$$

where $\|\cdot\|_\infty$ is the max norm. When scaling values are provided for the solution vector, these values are automatically incorporated into the calculation of the perturbations used for the default difference quotient approximations for Jacobian information; see (3.6) and (3.8) below.

### Globalization strategy

Two methods of applying a computed step $\delta_n$ to the previously computed solution vector are implemented. The first and simplest is the standard Newton strategy which applies step 2(b) as above with $\lambda$ always set to 1. The other method is a global strategy, which attempts to use the direction implied by $\delta_n$ in the most efficient way for furthering convergence of the nonlinear problem. This technique is implemented in the second strategy, called Linesearch. This option employs both the $\alpha$ and $\beta$ conditions of the Goldstein-Armijo linesearch given in [5] for step 2(b), where $\lambda$ is chosen to guarantee a sufficient decrease in $F$ relative to the step length as well as a minimum step length relative to the initial rate of decrease of $F$. One property of the algorithm is that the full Newton step tends to be taken close to the solution.

KINSOL implements a backtracking algorithm to first find the value $\lambda$ such that $u_n + \lambda\delta_n$ satisfies the sufficient decrease condition (or $\alpha$-condition)

$$F(u_n + \lambda\delta_n) \le F(u_n) + \alpha\nabla F(u_n)\lambda\delta_n \,,$$

where $\alpha = 10^{-4}$. Although backtracking in itself guarantees that the step is not too small, KINSOL secondly relaxes $\lambda$ to satisfy the so-called $\beta$-condition (equivalent to Wolfe's curvature condition):

$$F(u_n + \lambda\delta_n) \geq F(u_n) + \beta\nabla F(u_n)\lambda\delta_n$$

where $beta = 0.9$. During this second phase, $\lambda$ is allowed to vary in the interval $[\lambda_{min}, \lambda_{max}]$ where

$$\lambda_{min} = \frac{\text{STEPTOL}}{\|\bar{\delta}_n\|_\infty}, \quad \bar{\delta}_n^j = \frac{\delta_n^j}{1/D_u^j + |u^j|}$$

and $\lambda_{max}$ corresponds to the maximum feasible step size at the current iteration (typically $\lambda_{max} = \text{STEPMAX}/\|\delta_n\|_{D_u}$). In the above expressions, $v^j$ indicates the $j$th component of a vector $v$.

For more details, the reader is referred to [5].

### Nonlinear iteration stopping criteria

Stopping criteria for the Newton method are applied to both of the nonlinear residual and the step length. For the former, the Newton iteration must pass a stopping test

$$\|F(u_n)\|_{D_F,\infty} < \text{FTOL}\,,$$

where FTOL is an input scalar tolerance with a default value of $U^{1/3}$. For the latter, the Newton method will terminate when the maximum scaled step is below a given tolerance

$$\|\lambda\delta_n\|_{D_u,\infty} < \text{STEPTOL}\,,$$

where STEPTOL is an input scalar tolerance with a default value of $U^{2/3}$. Only the first condition (small residual) is considered a successful completion of KINSOL. The second condition (small step) may indicate that the iteration is stalled near a point for which the residual is still unacceptable.

### Additional constraints

As a user option, KINSOL permits the application of inequality constraints, $u^i > 0$ and $u^i < 0$, as well as $u^i \geq 0$ and $u^i \leq 0$, where $u^i$ is the $i$th component of $u$. Any such constraint, or no constraint, may be imposed on each component. KINSOL will reduce step lengths in order to ensure that no constraint is violated. Specifically, if a new Newton iterate will violate a constraint, the maximum (over all $i$) step length along the Newton direction that will satisfy all constraints is found and $\delta_n$ in Step 2(b) is scaled to take a step of that length.

### Residual monitoring for Modified Newton method

When using a Modified Newton method (i.e.; when a direct linear solver is used), in addition to the strategy described below for the update of the Jacobian matrix, KINSOL also provides an optional nonlinear residual monitoring scheme to control when the system Jacobian is updated. Specifically, a Jacobian update will also occur when `mbsetsub`= 5 nonlinear iterations have passed since the last update and

$$\|D_uF(u_n)\|_2 > \omega\|D_uF(u_m)\|_2\,,$$

where $u_n$ is the current iterate and $u_m$ is the iterate at the last Jacobian update. The scalar $\omega$ is given by

$$\omega = \min\left(\omega_{min}\, e^{\max(0,\beta-1)}, \omega_{max}\right)\,, \tag{3.3}$$

with $\beta$ defined as

$$\beta = \frac{\|D_uF(u_n)\|_2}{\text{FTOL}}\,, \tag{3.4}$$

where FTOL is the input scalar tolerance discussed before. Optionally, a constant value $\omega_{const}$ can be used for the parameter $\omega$.

The constants controlling the nonlinear residual monitoring algorithm can be changed from their default values through optional inputs to KINSOL. These include the parameters $\omega_{min}$ and $\omega_{max}$, the constant value $\omega_{const}$, and the threshold `mbsetsub`.

### Stopping criteria for iterative linear solvers

When using an Inexact Newton method (i.e.; when an iterative linear solver is used), the convergence of the overall nonlinear solver is intimately coupled with the accuracy with which the linear solver in 2(a) above is solved. KINSOL provides three options for stopping criteria for the linear system solver, including the two algorithms of Eisenstat and Walker [6]. More precisely, the Krylov iteration must pass a stopping test

$$\|J\delta_n + F\|_{D_F} < (\eta_n + U)\|F\|_{D_F} \,,$$

where $\eta_n$ is one of:

**Eisenstat and Walker Choice 1**

$$\eta_n = \frac{|\; \|F(u_n)\|_{D_F} - \|F(u_{n-1}) + J(u_{n-1})\delta_n\|_{D_F} \;|}{\|F(u_{n-1})\|_{D_F}} \,,$$

**Eisenstat and Walker Choice 2**

$$\eta_n = \gamma \left( \frac{\|F(u_n)\|_{D_F}}{\|F(u_{n-1})\|_{D_F}} \right)^{\alpha} \,,$$

where default values of $\gamma$ and $\alpha$ are 0.9 and 2, respectively.

**Constant $\eta$**

$$\eta_n = constant,$$

with 0.1 as the default.

The default strategy is "Eisenstat and Walker Choice 1". For both options 1 and 2, appropriate safeguards are incorporated to ensure that $\eta$ does not decrease too quickly [6].

### Difference quotient Jacobian approximations

With the direct dense and band methods, the Jacobian may be supplied by a user routine, or approximated by difference quotients, at the user's option. In the latter case, we use the usual approximation

$$J^{ij} = [F^i(u + \sigma_j e^j) - F^i(u)]/\sigma_j \,. \tag{3.5}$$

The increments $\sigma_j$ are given by

$$\sigma_j = \sqrt{U}\;\max\left\{|u^j|, 1/D_u^j\right\} \,, \tag{3.6}$$

where $U$ is the unit roundoff. In the dense case, this scheme requires $N$ evaluations of $F$, one for each column of $J$. In the band case, the columns of $J$ are computed in groups, by the Curtis-Powell-Reid algorithm, with the number of $F$ evaluations equal to the bandwidth.

In the case of a Krylov method, Jacobian information is needed only as matrix-vector products $Jv$. If a routine for $Jv$ is not supplied, these products are approximated by directional difference quotients as

$$J(u)v \approx [F(u + \sigma v) - F(u)]/\sigma \tag{3.7}$$

where $u$ is the current approximation to a root of (3.1), and $\sigma$ is a scalar. The choice of $\sigma$ is taken from [2] and is given by

$$\sigma = \frac{\max\{|u^T v|, u_{typ}^T |v|\}}{\|v\|_2} \mathrm{sign}(u^T v)\sqrt{U} \,, \tag{3.8}$$

where $u_{typ}$ is a vector of typical values for the absolute values of the solution (and can be taken to be inverses of the scale factors given for $u$ as described below), and $U$ is unit roundoff. Convergence of the Newton method is maintained as long as the value of $\sigma$ remains appropriately small as shown in [1].

# Chapter 4

# Code Organization

## 4.1 SUNDIALS organization

The family of solvers referred to as SUNDIALS consists of the solvers CVODE (for ODE systems), KINSOL (for nonlinear algebraic systems), and IDA (for differential-algebraic systems). In addition, variants of these which also do sensitivity analysis calculations are available or in development. CVODES, an extension of CVODE that provides both forward and adjoint sensitivity capabilities is available, while IDAS is currently in development.

The various solvers of this family share many subordinate modules. For this reason, it is organized as a family, with a directory structure that exploits that sharing (see Fig. 4.1). The following is a list of the solver packages presently available:

- CVODE, a solver for stiff and nonstiff ODEs $dy/dt = f(t, y)$;
- CVODES, a solver for stiff and nonstiff ODEs $dy/dt = f(t, y, p)$ with sensitivity analysis capabilities;
- KINSOL, a solver for nonlinear algebraic systems $F(u) = 0$;
- IDA, a solver for differential-algebraic systems $F(t, y, y') = 0$.

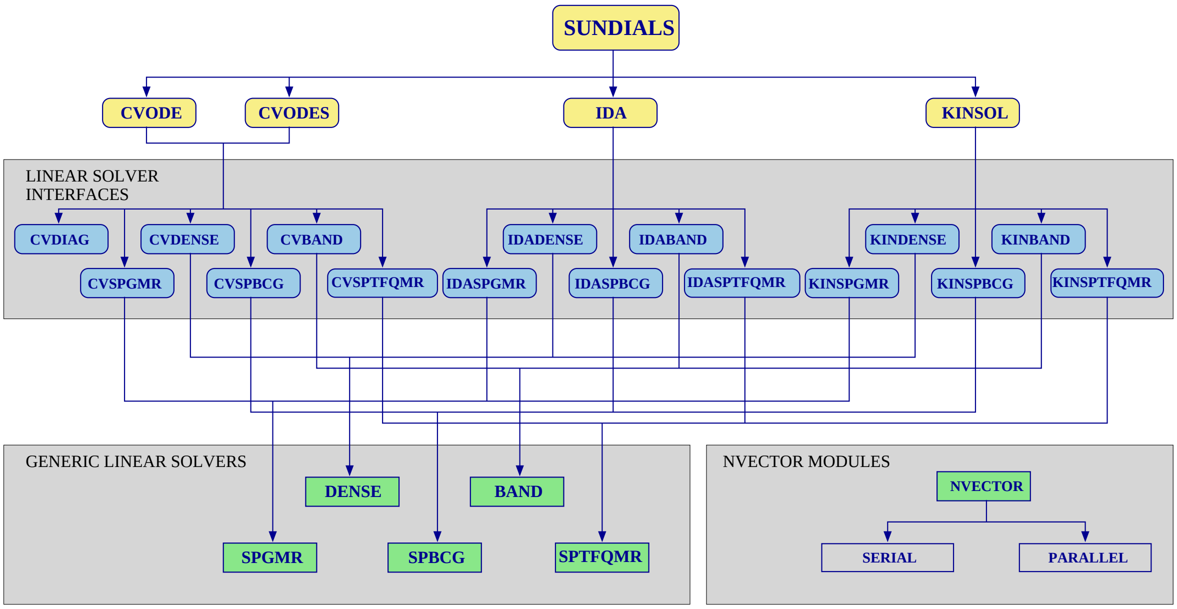
## 4.2 KINSOL organization

The KINSOL package is written in the ANSI C language. This section summarizes the basic structure of the package, although knowledge of this structure is not necessary for its use.

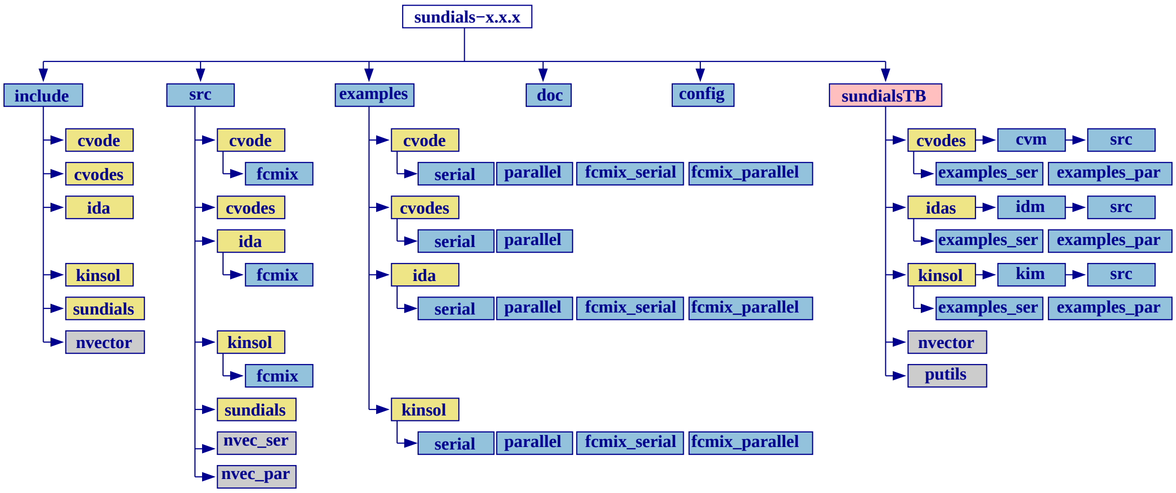
The overall organization of the KINSOL package is shown in Figure 4.2. The central solver module, implemented in the files `kinsol.h`, `kinsol_impl.h` and `kinsol.c`, deals with the solution of a nonlinear algebraic system using either an Inexact Newton method or a line search method for the global strategy. Although this module contains logic for the Newton iteration, it has no knowledge of the method used to solve the linear systems that arise. For any given user problem, one of the linear system modules is specified, and is then invoked as needed.

At present, the package includes the following five KINSOL linear system modules:

- KINDENSE: LU factorization and backsolving with dense matrices;
- KINBAND: LU factorization and backsolving with banded matrices;
- KINSPGMR: scaled preconditioned GMRES method;
- KINSPBCG: scaled preconditioned Bi-CGStab method;
- KINSPTFQMR: scaled preconditioned TFQMR method.




(a) High-level diagram




(b) Directory structure of the source tree

Figure 4.1: Organization of the SUNDIALS suite

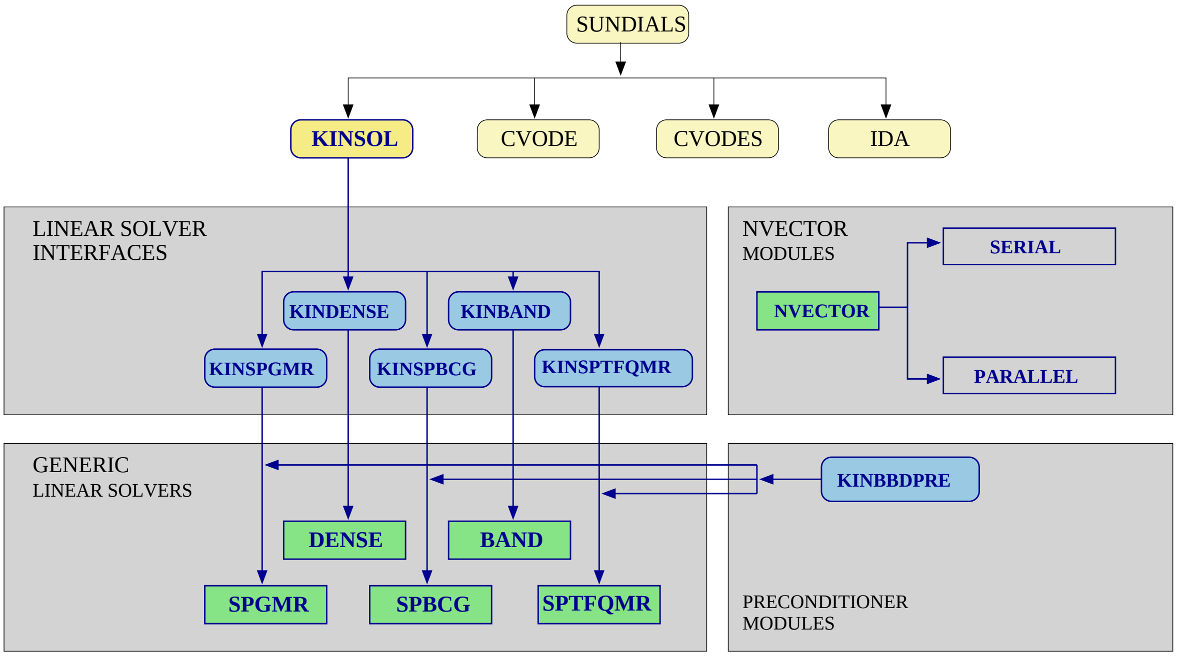



Figure 4.2: Overall structure diagram of the KINSOL package. Modules specific to KINSOL are distinguished by rounded boxes, while generic solver and auxiliary modules are in rectangular boxes. Grayed boxes refer to the encompassing SUNDIALS structure.

This set of linear solver modules is intended to be expanded in the future as new algorithms are developed.

In the case of the direct methods KINDENSE and KINBAND, the package includes an algorithm for the approximation of the Jacobian by difference quotients, but the user also has the option of supplying the Jacobian (or an approximation to it) directly. In the case of the Krylov methods KINSPGMR, KINSPBCG and KINSPTFQMR, the package includes an algorithm for the approximation by difference quotients of the product between the Jacobian matrix and a vector of appropriate length. Again, the user has the option of providing a routine for this operation. For the Krylov methods, the preconditioning must be supplied by the user, in twho phases: setup (preprocessing of Jacobian data) and solve.

Each KINSOL linear solver module consists of four routines, devoted to (1) memory allocation and initialization, (2) setup of the matrix data involved, (3) solution of the system, and (4) freeing of memory. The setup and solution phases are separate because the evaluation of Jacobians and preconditioners is done only periodically during the integration, as required to achieve convergence. The call list within the central KINSOL module to each of the associated functions is fixed, thus allowing the central module to be completely independent of the linear system method.

These modules are also decomposed in another way. Each of the modules KINDENSE, KINBAND, KINSPGMR, KINSPBCG, and KINSPTFQMR is a set of interface routines built on top of a generic solver module, name DENSE, BAND, SPGMR, SPBCG, and SPTFQMR, respectively. The interface deals with the use of these methods in the KINSOL context, whereas the generic solver is independent of the context. While the generic solvers here were generated with SUNDIALS in mind, our intention is that they be usable in other applications as general-purpose solvers. This separation also allows for any generic solver to be replaced by an improved version, with no necessity to revise the KINSOL package elsewhere.

KINSOL also provides a preconditioner module called KINBBDPRE for use with any of the Krylov iterative liear solvers. It works in conjunction with NVECTOR_PARALLEL and generates a preconditioner that is a block-diagonal matrix with each block being a band matrix, as further described in §5.7.

All state information used by KINSOL to solve a given problem is saved in a structure, and a pointer to that structure is returned to the user. There is no global data in the KINSOL package, and so in this respect it is reentrant. State information specific to the linear solver is saved in a separate structure, a pointer to which resides in the KINSOL memory structure. The reentrancy of KINSOL was motivated by the anticipated multicomputer extension, but is also essential in a uniprocessor setting where two or more different problems are solved by intermixed calls to the package from one user program.

# Chapter 5

# Using KINSOL for C Applications

This chapter is concerned with the use of KINSOL for the solution of nonlinear systems. The following subsections treat the header files, the layout of the user's main program, description of the KINSOL user-callable routines, and user-supplied functions. The final section describes the FORTRAN/C interface module, which supports users with applications written in FORTRAN77. The listings of the sample programs in the companion document [3] may also be helpful. Those codes may be used as templates (with the removal of some lines involved in testing), and are included in the KINSOL package.

The user should be aware that not all linear solver modules are compatible with all NVECTOR implementations. For example, NVECTOR_PARALLEL is not compatible with the direct dense or direct band linear solvers since these linear solver modules need to form the complete system Jacobian. The following KINSOL modules can only be used with NVECTOR_SERIAL: KINDENSE and KINBAND. The preconditioner module KINBBDPRE can only be used with NVECTOR_PARALLEL.

KINSOL uses various constants for both input and output. These are defined as needed in this chapter, but for convenience are also listed separately in Chapter 10.

## 5.1 Access to library and header files

At this point, it is assumed that the installation of KINSOL, following the procedure described in Chapter 2, has been completed successfully.

Regardless of where the user's application program resides, its associated compilation and load commands must make reference to the appropriate locations for the library and header files required by KINSOL. The relevant library files are

- *libdir*/`libsundials_kinsol.`*lib*,
- *libdir*/`libsundials_nvec*.`*lib* (one or two files),

where the file extension *.lib* is typically `.so` for shared libraries and `.a` for static libraries. The relevant header files are located in the subdirectories

- *incdir*/`include`
- *incdir*/`include/kinsol`
- *incdir*/`include/sundials`

The directories *libdir* and *incdir* are the install libray and include directories. For a default installation, these are *builddir*/`lib` and *builddir*/`include`, respectively, where *builddir* was defined in Chapter 2.

## 5.2 Data types

The sundials_types.h file contains the definition of the type realtype, which is used by the SUNDIALS solvers for all floating-point data. The type realtype can be float, double, or long double, with the default being double. The user can change the precision of the SUNDIALS solvers arithmetic at the configuration stage (see §2.1).

Additionally, based on the current precision, sundials_types.h defines BIG_REAL to be the largest value representable as a realtype, SMALL_REAL to be the smallest value representable as a realtype, and UNIT_ROUNDOFF to be the difference between 1.0 and the minimum realtype greater than 1.0.

Within SUNDIALS, real constants are set by way of a macro called RCONST. It is this macro that needs the ability to branch on the definition realtype. In ANSI C, a floating-point constant with no suffix is stored as a double. Placing the suffix "F" at the end of a floating point constant makes it a float, whereas using the suffix "L" makes it a long double. For example,

```
#define A 1.0
#define B 1.0F
#define C 1.0L
```

defines A to be a double constant equal to 1.0, B to be a float constant equal to 1.0, and C to be a long double constant equal to 1.0. The macro call RCONST(1.0) automatically expands to 1.0 if realtype is double, to 1.0F if realtype is float, or to 1.0L if realtype is long double. SUNDIALS uses the RCONST macro internally to declare all of its floating-point constants.

A user program which uses the type realtype and the RCONST macro to handle floating-point constants is precision-independent except for any calls to precision-specific standard math library functions. (Our example programs use both realtype and RCONST.) Users can, however, use the type double, float, or long double in their code (assuming the typedef for realtype matches this choice). Thus, a previously existing piece of ANSI C code can use SUNDIALS without modifying the code to use realtype, so long as the SUNDIALS libraries use the correct precision (for details see §2.1).

## 5.3 Header files

The calling program must include several header files so that various macros and data types can be used. The header file that is always required is:

- kinsol.h, the header file for KINSOL, which defines several types and various constants, and includes function prototypes.

kinsol.h also includes sundials_types.h, which defines the types realtype and booleantype and constants FALSE and TRUE.

The calling program must also include an NVECTOR implementation header file (see Chapter 7 for details). For the two NVECTOR implementations that are included in the KINSOL package, the corresponding header files are:

- nvector_serial.h, which defines the serial implementation, NVECTOR_SERIAL;
- nvector_parallel.h, which defines the parallel MPI implementation, NVECTOR_PARALLEL.

Note that both of these files include in turn the header file sundials_nvector.h, which defines the abstract N_Vector type.

Finally, a linear solver module header file is required. The header files corresponding to the various linear solver options in KINSOL are:

- kinsol_dense.h, which is used with the dense direct linear solver in the context of KINSOL. This in turn includes a header file (sundials_dense.h) which defines the DenseMat type and corresponding accessor macros;

- `kinsol_band.h`, which is used with the band direct linear solver in the context of KINSOL. This in turn includes a header file (`sundials_band.h`) which defines the `BandMat` type and corresponding accessor macros;
- `kinsol_spgmr.h`, which is used with the Krylov solver SPGMR in the context of KINSOL;
- `kinsol_spbcgs.h`, which is used with the Krylov solver SPBCG in the context of KINSOL;
- `kinsol_sptfqmr.h`, which is used with the Krylov solver SPTFQMR in the context of KINSOL;

The header files for the Krylov iterative solvers include `kinsol_spils.h` which defined common fuunctions and which in turn includes a header file (`sundials_iterative.h`) which enumerates the kind of preconditioning and for the choices for the Gram-Schmidt process for SPGMR.

Other headers may be needed, according to the choice of preconditioner, etc. For example, in the `kinkryx` example [3], preconditioning is done with a block-diagonal matrix. For this, the header `sundials_smalldense.h` is included.

## 5.4 A skeleton of the user's main program

The following is a skeleton of the user's main program (or calling program) for the solution of a nonlinear problem. Some steps are independent of the NVECTOR implementation used; where this is not the case, usage specifications are given for the two implementations provided with KINSOL: Steps marked with [**P**] correspond to NVECTOR_PARALLEL, while steps marked with [**S**] correspond to NVECTOR_SERIAL.

1. **[P] Initialize MPI**

   Call `MPI_Init(&argc, &argv);` to initialize MPI if used by the user's program, aside from the internal use in NVECTOR_PARALLEL. Here `argc` and `argv` are the command line argument counter and array received by `main`.

2. **Set problem dimensions**

   [**S**] Set `N`, the problem size $N$.

   [**P**] Set `Nlocal`, the local vector length (the sub-vector length for this process); `N`, the global vector length (the problem size $N$, and the sum of all the values of `Nlocal`); and the active set of processes.

3. **Set vector with initial guess**

   To set the vector `u` of initial values, use functions defined by a particular NVECTOR implementation. If a `realtype` array `udata` already exists, containing the initial guess of $u_0$, make the call:

   [**S**] `u = N_VMake_Serial(N, udata);`

   [**P**] `u = N_VMake_Parallel(comm, Nlocal, N, udata);`

   Otherwise, make the call:

   [**S**] `u = N_VNew_Serial(N);`

   [**P**] `u = N_VNew_Parallel(comm, Nlocal, N);`

   and load initial values into the structure defined by:

   [**S**] `NV_DATA_S(u)`

   [**P**] `NV_DATA_P(u)`

   Here `comm` is the MPI communicator, set in one of two ways: If a proper subset of active processes is to be used, `comm` must be set by suitable MPI calls. Otherwise, to specify that all processes are to be used, `comm` must be `MPI_COMM_WORLD`.

4. **Create KINSOL object**

   Call `kin_mem = KINCreate();` to create the KINSOL memory block. `KINCreate` returns a pointer to the KINSOL memory structure. See §5.5.1 for details.

5. **Set optional inputs**

   Call `KINSet*` routines to change from their default values any optional inputs that control the behavior of KINSOL. See §5.5.4 for details.

6. **Allocate internal memory**

   Call `KINMalloc(...);` to specify the problem defining function $F$, allocate internal memory for KINSOL, and initialize KINSOL. `KINMalloc` returns a flag to indicate success or an illegal argument value. See §5.5.1 for details.

7. **Attach linear solver module**

   Initialize the linear solver module with one of the following calls (for details see §5.5.2).

   [**S**] `ier = KINDense(...);`

   [**S**] `ier = KINBand(...);`

   `ier = KINSpgmr(...);`

   `ier = KINSpbcg(...);`

   `ier = KINSptfqmr(...);`

8. **Set linear solver optional inputs**

   Call `KIN*Set*` functions from the selected linear solver module to change optional inputs specific to that linear solver. See §5.5.4 for details.

9. **Solve problem**

   Call `ier = KINSol(...);` to solve the nonlinear problem for a given initial guess. See §5.5.3 for details.

10. **Get optional outputs**

    Call `KINGet*` and `KIN*Get*` functions to obtain optional output. See §5.5.5 for details.

11. **Deallocate memory for solution vector**

    Upon completion of the solution, deallocate memory for the vector `u` by calling the destructor function defined by the NVECTOR implementation:

    [**S**] `N_VDestroy_Serial(u);`

    [**P**] `N_VDestroy_Parallel(u);`

12. **Free solver memory**

    Call `KINFree(&kin_mem);` to free the memory allocated for KINSOL.

13. **[P] Finalize MPI**

    Call `MPI_Finalize();` to terminate MPI.

## 5.5 User-callable functions

This section describes the KINSOL functions that are called by the user to set up and solve a nonlinear problem. Some of these are required. However, starting with §5.5.4, the functions listed involve optional inputs/outputs or restarting, and those paragraphs can be skipped for a casual use of KINSOL. In any case, refer to §5.4 for the correct order of these calls.

### 5.5.1 KINSOL initialization and deallocation functions

The following three functions must be called in the order listed. The last one is to be called only after the problem solution is complete, as it frees the KINSOL memory block created and allocated by the first two calls.

`KINCreate`

**Call** `kin_mem = KINCreate();`

**Description** The function `KINCreate` instantiates a KINSOL solver object.

**Arguments** This function has no arguments.

**Return value** If successful, `KINCreate` returns a pointer to the newly created KINSOL memory block (of type `void *`). If an error occurred, `KINCreate` prints an error message to `stderr` and returns `NULL`.

`KINMalloc`

**Call** `flag = KINMalloc(kin_mem, func, tmpl);`

**Description** The function `KINMalloc` specifies the problem-defining function, allocates internal memory, and initializes KINSOL.

**Arguments**
- `kin_mem` (`void *`) pointer to the KINSOL memory block returned by `KINCreate`.
- `func` (`KINSysFn`) is the C function which computes $F$ in the nonlinear problem. This function has the form `func(u, fval, f_data)` (for full details see §5.6.1).
- `tmpl` (`N_Vector`) is an `N_Vector` which is used as a template to create (by cloning) necessary vectors in `kin_mem`.

**Return value** The return flag `flag` (of type `int`) will be one of the following:
- `KIN_SUCCESS` The call to `KINMalloc` was successful.
- `KIN_MEM_NULL` The kinsol memory block was not initialized through a previous call to `KINCreate`.
- `KIN_MEM_FAIL` A memory allocation request has failed.
- `KIN_ILL_INPUT` An input argument to `KINMalloc` has an illegal value.

**Notes** If an error occurred, `KINMalloc` sends an error message to the error handler function.

`KINFree`

**Call** `KINFree(&kin_mem);`

**Description** The function `KINFree` frees the memory allocated by a previous call to `KINMalloc`.

**Arguments** The argument is the address of the pointer to the KINSOL memory block returned by `KINCreate` (of type `void *`).

**Return value** The function `KINFree` has no return value.

### 5.5.2 Linear solver specification functions

As previously explained, Newton iteration requires the solution of linear systems of the form (2). There are five KINSOL linear solvers currently available for this task: KINDENSE, KINBAND, KINSPGMR, KINSPBCG, and KINSPTFQMR. The first two are direct solvers and their names indicate the type of approximation used for the Jacobian $J = \partial F/\partial u$; KINDENSE and KINBAND work with dense and banded approximations to $J$, respectively. The last three KINSOL linear solvers — KINSPGMR, KINSPBCG, and

KINSPTFQMR — are Krylov iterative solvers, which use scaled preconditioned GMRES, scaled preconditioned Bi-CGStab, and scaled preconditioned TFQMR, respectively. Together, they are referred to as KINSPILS (from scaled preconditioned iterative linear solvers).

To specify a KINSOL linear solver, after the call to `KINCreate` but before any calls to `KINSol`, the user's program must call one of the functions `KINDense`, `KINBand`, `KINSpgmr`, `KINSpbcg`, or `KINSptfqmr`, as documented below. The first argument passed to these functions is the KINSOL memory pointer returned by `KINCreate`. A call to one of these functions links the main KINSOL nonlinear solver to a linear solver and allows the user to specify parameters which are specific to a particular solver, such as the half-bandwidths in the KINBAND case. The use of each of the linear solvers involves certain constants and possibly some macros, that are likely to be needed in the user code. These are available in the corresponding header file associated with the linear solver, as specified below.

In each case, the linear solver module used by KINSOL is actually built on top of a generic linear system solver, which may be of interest in itself. These generic solvers, denoted DENSE, BAND, SPGMR, SPBCG, and SPTFQMR, are described separately in Chapter 9.

`KINDense`

Call `flag = KINDense(kin_mem, N);`

Description The function `KINDense` selects the KINDENSE linear solver.

The user's main function must include the `kinsol_dense.h` header file.

Arguments
- `kin_mem` (`void *`) pointer to the KINSOL memory block.
- `N` (`long int`) problem dimension.

Return value The return value `flag` (of type `int`) is one of

- `KINDENSE_SUCCESS` The KINDENSE initialization was successful.
- `KINDENSE_MEM_NULL` The `kin_mem` pointer is `NULL`.
- `KINDENSE_ILL_INPUT` The KINDENSE solver is not compatible with the current NVECTOR module.
- `KINDENSE_MEM_FAIL` A memory allocation request failed.

Notes The KINDENSE linear solver may not be compatible with a particular implementation of the NVECTOR module. Of the two NVECTOR modules provided by SUNDIALS, only NVECTOR_SERIAL is compatible, while NVECTOR_PARALLEL is not.

`KINBand`

Call `flag = KINBand(kin_mem, N, mupper, mlower);`

Description The function `KINBand` selects the KINBAND linear solver.

The user's main function must include the `kinsol_band.h` header file.

Arguments
- `kin_mem` (`void *`) pointer to the KINSOL memory block.
- `N` (`long int`) problem dimension.
- `mupper` (`long int`) upper half-bandwidth of the problem Jacobian (or of the approximation of it).
- `mlower` (`long int`) lower half-bandwidth of the problem Jacobian (or of the approximation of it).

Return value The return value `flag` (of type `int`) is one of

- `KINBAND_SUCCESS` The KINBAND initialization was successful.
- `KINBAND_MEM_NULL` The `kin_mem` pointer is `NULL`.
- `KINBAND_ILL_INPUT` The KINBAND solver is not compatible with the current NVECTOR module, or one of the Jacobian half-bandwidths is outside its valid range ($0 \ldots$ N$-1$).
- `KINBAND_MEM_FAIL` A memory allocation request failed.

Notes The KINBAND linear solver may not be compatible with a particular implementation of the NVECTOR module. Of the two NVECTOR modules provided by SUNDIALS, only NVECTOR_SERIAL is compatible, while NVECTOR_PARALLEL is not. The half-bandwidths are to be set so that the nonzero locations $(i, j)$ in the banded (approximate) Jacobian satisfy $-$`mlower` $\leq j - i \leq$ `mupper`.

**`KINSpgmr`**

Call `flag = KINSpgmr(kin_mem, maxl);`

Description The function `KINSpgmr` selects the KINSPGMR linear solver.

The user's main function must include the `kinsol_spgmr.h` header file.

Arguments
- `kin_mem` (`void *`) pointer to the KINSOL memory block.
- `maxl` (`int`) maximum dimension of the Krylov subspace to be used. Pass 0 to use the default value `KINSPILS_MAXL`= 5.

Return value The return value `flag` (of type `int`) is one of:

- `KINSPILS_SUCCESS` The KINSPGMR initialization was successful.
- `KINSPILS_MEM_NULL` The `kin_mem` pointer is `NULL`.
- `KINSPILS_ILL_INPUT` The NVECTOR module used does not implement a required operation.
- `KINSPILS_MEM_FAIL` A memory allocation request failed.

Notes The KINSPGMR solver uses a scaled preconditioned GMRES iterative method to solve the linear system (2).

Within KINSOL, only right preconditioning is available. For specification of the preconditioner, see §5.5.4 and §5.6.

If preconditioning is done, user-supplied functions define the right preconditioner matrices $P$, which approximate the Newton matrix from (2).

**`KINSpbcg`**

Call `flag = KINSpbcg(kin_mem, maxl);`

Description The function `KINSpbcg` selects the KINSPBCG linear solver.

The user's main function must include the `kinsol_spbcgs.h` header file.

Arguments
- `kin_mem` (`void *`) pointer to the KINSOL memory block.
- `maxl` (`int`) maximum dimension of the Krylov subspace to be used. Pass 0 to use the default value `KINSPILS_MAXL`= 5.

Return value The return value `flag` (of type `int`) is one of:

- `KINSPILS_SUCCESS` The KINSPBCG initialization was successful.
- `KINSPILS_MEM_NULL` The `kin_mem` pointer is `NULL`.
- `KINSPILS_ILL_INPUT` The NVECTOR module used does not implement a required operation.
- `KINSPILS_MEM_FAIL` A memory allocation request failed.

Notes The KINSPBCG solver uses a scaled preconditioned Bi-CGStab iterative method to solve the linear system (2).

Within KINSOL, only right preconditioning is available. For specification of the preconditioner, see §5.5.4 and §5.6.

If preconditioning is done, user-supplied functions define the right preconditioner matrices $P$, which approximate the Newton matrix from (2).

`KINSptfqmr`

Call `flag = KINSptfqmr(kin_mem, maxl);`

Description The function `KINSptfqmr` selects the KINSPTFQMR linear solver.

The user's main function must include the `kinsol_sptfqmr.h` header file.

Arguments
- `kin_mem` (`void *`) pointer to the KINSOL memory block.
- `maxl` (`int`) maximum dimension of the Krylov subspace to be used. Pass 0 to use the default value `KINSPILS_MAXL`= 5.

Return value The return value `flag` (of type `int`) is one of:

- `KINSPILS_SUCCESS` The KINSPTFQMR initialization was successful.
- `KINSPILS_MEM_NULL` The `kin_mem` pointer is `NULL`.
- `KINSPILS_ILL_INPUT` The NVECTOR module used does not implement a required operation.
- `KINSPILS_MEM_FAIL` A memory allocation request failed.

Notes The KINSPTFQMR solver uses a scaled preconditioned TFQMR iterative method to solve the linear system (2).

Within KINSOL, only right preconditioning is available. For specification of the preconditioner, see §5.5.4 and §5.6.

If preconditioning is done, user-supplied functions define the right preconditioner matrices $P$, which approximate the Newton matrix from (2).

### 5.5.3 KINSOL solver function

This is the central step in the solution process — the call to solve the nonlinear algebraic system.

`KINSol`

Call `flag = KINSol(kin_mem, u, strategy, u_scale, f_scale);`

Description The function `KINSol` computes an approximate solution to the nonlinear system.

Arguments
- `kin_mem` (`void *`) pointer to the KINSOL memory block.
- `u` (`N_Vector`) vector set to initial guess by user before calling `KINSol`, but which upon return contains an approximate solution of the nonlinear system $F(u) = 0$ the computed solution vector.
- `strategy` (`int`) globalization strategy applied to the Newton method. It must be one of `KIN_NONE` or `KIN_LINESEARCH`.
- `u_scale` (`N_Vector`) vector containing diagonal elements of scaling matrix $D_u$ for vector `u` chosen so that the components of $D_u$·`u` (as a matrix multiplication) all have about the same magnitude when `u` is close to a root of $F(u)$.
- `f_scale` (`N_Vector`) vector containing diagonal elements of scaling matrix $D_F$ for $F(u)$ chosen so that the components of $D_F \cdot F($`u`$)$ (as a matrix multiplication) all have roughly the same magnitude when `u` is not too near a root of $F(u)$.

Return value On return, `KINSol` returns the approximate solution in the vector `u`. The return value `flag` (of type `int`) will be one of the following:

`KIN_SUCCESS`

`KINSol` succeeded; the scaled norm of $F(u)$ is less than `fnormtol`.

`KIN_INITIAL_GUESS_OK`

The guess `u` $= u_0$ satisfied the system $F(u) = 0$ within the tolerances specified.

`KIN_STEP_LT_STPTOL`

KINSOL stopped based on scaled step length. This means that the current iterate may be an approximate solution of the given nonlinear system, but it is also quite possible that the algorithm is "stalled" (making insufficient progress) near an invalid solution, or that the scalar `scsteptol` is too large (see `KINSetScaledStepTol` in §5.5.4 to change `scsteptol` from its default value).

`KIN_MEM_NULL`

The KINSOL memory block pointer was `NULL`.

`KIN_ILL_INPUT`

An input parameter was invalid.

`KIN_NO_MALLOC`

The KINSOL memory was not allocated by a call to `KINMalloc`.

`KIN_LINESEARCH_NONCONV`

The line search algorithm was unable to find an iterate sufficiently distinct from the current iterate, or could not find an iterate satisfying the sufficient decrease condition.

Failure to satisfy the sufficient decrease condition could mean the current iterate is "close" to an approximate solution of the given nonlinear system, the finite difference approximation of the matrix-vector product $J(u)v$ is inaccurate, or the real scalar `scsteptol` is too large.

`KIN_MAXITER_REACHED`

The maximum number of nonlinear iterations has been reached.

`KIN_MXNEWT_5X_EXCEEDED`

Five consecutive steps have been taken that satisfy the inequality $\|D_u p\|_{L2} > 0.99$ `mxnewtstep`, where $p$ denotes the current step and `mxnewtstep` is a scalar upper bound on the scaled step length.

Such a failure may mean that $\|D_F F(u)\|_{L2}$ asymptotes from above to a finite value, or the real scalar `mxnewtstep` is too small.

`KIN_LINESEARCH_BCFAIL`

The line search algorithm was unable to satisfy the "beta-condition" for `MXNBCF` $+1$ nonlinear iterations (not necessarily consecutive), which may indicate the algorithm is making poor progress.

`KIN_LINSOLV_NO_RECOVERY`

The user-supplied routine `psolve` encountered a recoverable error, but the preconditioner is already current.

`KIN_LINIT_FAIL`

The linear solver initialization routine (`linit`) encountered an error.

`KIN_LSETUP_FAIL`

The user-supplied routine `pset` (used to set up the preconditioner data) encountered an unrecoverable error.

`KIN_LSOLVE_FAIL`

Either the user-supplied routine `psolve` (used to to solve the preconditioned linear system) encountered an unrecoverable error, or the linear solver routine (`lsolve`) encountered an error condition.

`KIN_SYSFUNC_FAIL`

The system function failed in an unrecoverable manner.

`KIN_FIRST_SYSFUNC_ERR`

The system function failed recoverably at the first call.

`KIN_REPTD_SYSFUNC_ERR`

The system function had repeated recoverable errors. No recovery is possible.

Notes The components of vectors `u_scale` and `f_scale` should be strictly positive.

`KIN_SUCCESS` = 0, `KIN_INITIAL_GUESS_OK` = 1, and `KIN_STEP_LT_STPTOL` = 2. All remaining return values are negative and therefore a test `flag` $< 0$ will trap all `KINSol` failures.

### 5.5.4 Optional input functions

KINSOL provides an extensive list of functions that can be used to change from their default values various optional input parameters that control the behavior of the KINSOL solver. Table 5.1 lists all optional input functions in KINSOL which are then described in detail in the remainder of this section, beginning with those for the main KINSOL solver and continuing with those for the linear solver modules. For the most casual use of KINSOL, the reader can skip to §5.6.

We note that, on error return, all of these functions also send an error message to the error handler function. We also note that all error return values are negative, so a test `flag` $< 0$ will catch any error.

#### 5.5.4.1 Main solver optional input functions

The calls listed here can be executed in any order.
However, if `KINSetErrHandlerFn` or `KINSetErrFile` are to be called, that call should be first, in order to take effect for any later error message.

`KINSetErrHandlerFn`

Call `flag = KINSetErrHandlerFn(kin_mem, ehfun, eh_data);`

Description The function `KINSetErrHandlerFn` specifies the optional user-defined function to be used in handling error messages.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.

`ehfun` (`KINErrHandlerFn`) is the C error handler function (see §5.6.2).

`eh_data` (`void *`) pointer to user data passed to `ehfun` every time it is called.

Return value The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS` The function `ehfun` and data pointer `eh_data` have been successfully set.

`KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.

Notes The default internal error handler function directs error messages to the file specified by the file pointer `errfp` (see `KINSetErrFile` below).

Error messages indicating that the KINSOL solver memory is `NULL` will always be directed to `stderr`.

`KINSetErrFile`

Call `flag = KINSetErrFile(kin_mem, errfp);`

Description The function `KINSetErrFile` specifies the pointer to the file where all KINSOL messages should be directed in case the default KINSOL error handler function is used.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.

`errfp` (`FILE *`) pointer to output file.

Return value The return value `flag` (of type `int`) is one of

`KIN_SUCCESS` The optional value has been successfully set.

`KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.

Table 5.1: Optional inputs for KINSOL, KINDENSE, and KINSPILS

| Optional input | Function name | Default |
|---|---|---|
| **KINSOL main solver** | | |
| Error handler function | `KINSetErrHandlerFn` | internal fn. |
| Pointer to an error file | `KINSetErrFile` | `stderr` |
| Info handler function | `KINSetInfoHandlerFn` | internal fn. |
| Pointer to an info file | `KINSetInfoFile` | `stdout` |
| Data for problem-defining function | `KINSetFdata` | `NULL` |
| Verbosity level of output | `KINSetPrintLevel` | 0 |
| Max. number of nonlinear iterations | `KINSetNumMaxIters` | 200 |
| No initial preconditioner setup | `KINSetNoInitSetup` | `FALSE` |
| No residual monitoring* | `KINSetNoResMon` | `FALSE` |
| Max. iterations without prec. setup | `KINSetMaxSetupCalls` | 10 |
| Max. iterations without residual check* | `KINSetMaxSubSetupCalls` | 5 |
| Form of $\eta$ coefficient | `KINSetEtaForm` | `KIN_ETACHOICE1` |
| Constant value of $\eta$ | `KINSetEtaConstValue` | 0.1 |
| Values of $\gamma$ and $\alpha$ | `KINSetEtaParams` | 0.9 and 2.0 |
| Values of $\omega_{min}$ and $\omega_{max}$* | `KINSetResMonParams` | 0.00001 and 0.9 |
| Constant value of $\omega$* | `KINSetResMonConstValue` | 0.9 |
| Lower bound on $\epsilon$ | `KINSetNoMinEps` | `FALSE` |
| Max. scaled length of Newton step | `KINSetMaxNewtonStep` | $1000\|D_u u_0\|_2$ |
| Max. number of $\beta$-condition failures | `KINSetMaxBetaFails` | 10 |
| Rel. error for F.D. $Jv$ | `KINSetRelErrFunc` | $\sqrt{\text{uround}}$ |
| Function-norm stopping tolerance | `KINSetFuncNormTol` | $\sqrt[3]{\text{uround}}$ |
| Scaled-step stopping tolerance | `KINSetScaledSteptol` | $\text{uround}^{2/3}$ |
| Inequality constraints on solution | `KINSetConstraints` | `NULL` |
| Nonlinear system function | `KINSetSysFunc` | none |
| **KINDENSE linear solver** | | |
| Dense Jacobian function and data | `KINDenseSetJacFn` | internal DQ, `NULL` |
| **KINBAND linear solver** | | |
| Band Jacobian function and data | `KINBandSetJacFn` | internal DQ, `NULL` |
| **KINSPILS linear solvers** | | |
| Max. number of restarts** | `KINSpilsSetMaxRestarts` | 0 |
| Preconditioner functions and data | `KINSpilsSetPreconditioner` | `NULL`, `NULL`, `NULL` |
| Jacobian-times-vector function and data | `KINSpilsSetJacTimesVecFn` | internal DQ, `NULL` |

* Only for KINDENSE and KINBAND

** Only for KINSPGMR

Notes The default value for `errfp` is `stderr`.

Passing a value of `NULL` disables all future error message output (except for the case in which the KINSOL memory pointer is `NULL`).

If `KINSetErrFile` is to be called, it should be called before any other optional input functions, in order to take effect for any later error message.

`KINSetInfoHandlerFn`

Call `flag = KINSetInfoHandlerFn(kin_mem, ihfun, ih_data);`

Description The function `KINSetInfoHandlerFn` specifies the optional user-defined function to be used in handling informative (non-error) messages.

Arguments
`kin_mem` (`void *`) pointer to the KINSOL memory block.
`ihfun` (`KINInfoHandlerFn`) is the C info handler function (see §5.6.3).
`ih_data` (`void *`) pointer to user data passed to `ihfun` every time it is called.

Return value The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS` The function `ihfun` and data pointer `ih_data` have been successfully set.
`KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.

Notes The default internal info handler function directs informative (non-error) messages to the file specified by the file pointer `infofp` (see `KINSetInfoFile` below).

`KINSetInfoFile`

Call `flag = KINSetInfoFile(kin_mem, infofp);`

Description The function `KINSetInfoFile` specifies the pointer to the file where all informative (non-error) messages should be directed.

Arguments
`kin_mem` (`void *`) pointer to the KINSOL memory block.
`infofp` (`FILE *`) pointer to output file.

Return value The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS` The optional value has been successfully set.
`KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.

Notes The default value for `infofp` is `stdout`.

`KINSetPrintLevel`

Call `flag = KINSetPrintLevel(kin_mem, printfl);`

Description The function `KINSetPrintLevel` specifies the level of verbosity of the output.

Arguments
`kin_mem` (`void *`) pointer to the KINSOL memory block.
`printfl` (`int`) flag indicating the level of verbosity. Must be one of:

0 no information displayed.

1 for each nonlinear iteration display the following information: the scaled Euclidean $\ell_2$ norm of the system function evaluated at the current iterate, the scaled norm of the Newton step (only if using `KIN_NONE`), and the number of function evaluations performed so far.

2 display level 1 output and the following values for each iteration:
$\|F(u)\|_{D_F}$ (only for `KIN_NONE`).
$\|F(u)\|_{D_F,\infty}$ (for `KIN_NONE` and `KIN_LINESEARCH`).

3 display level 2 output plus additional values used by the global strategy (only if using `KIN_LINESEARCH`), and statistical information for the linear solver.

Return value The return value `flag` (of type `int`) is one of:

| | |
|---|---|
| `KIN_SUCCESS` | The optional value has been successfully set. |
| `KIN_MEM_NULL` | The `kin_mem` pointer is `NULL`. |
| `KIN_ILL_INPUT` | The argument `printfl` had an illegal value. |

Notes The default value for `printfl` is 0.

`KINSetFdata`

Call `flag = KINSetFdata(kin_mem, f_data);`

Description The function `KINSetFdata` specifies the pointer to user-defined memory that is to be passed to the user-supplied function implementing the nonlinear system residual.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`f_data` (`void *`) pointer to the user-defined memory.

Return value The return value `flag` (of type `int`) is one of:

| | |
|---|---|
| `KIN_SUCCESS` | The optional value has been successfully set. |
| `KIN_MEM_NULL` | The `kin_mem` pointer is `NULL`. |

Notes The default value for `f_data` is `NULL`.

`KINSetNumMaxIters`

Call `flag = KINSetNumMaxIters(kin_mem, mxiter);`

Description The function `KINSetNumMaxIters` specifies the maximum number of nonlinear iterations allowed.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`mxiter` (`long int`) maximum number of nonlinear iterations.

Return value The return value `flag` (of type `int`) is one of:

| | |
|---|---|
| `KIN_SUCCESS` | The optional value has been successfully set. |
| `KIN_MEM_NULL` | The `kin_mem` pointer is `NULL`. |
| `KIN_ILL_INPUT` | The maximum number of iterations was non-positive. |

Notes The default value for `mxiter` is `MXITER_DEFAULT` = 200.

`KINSetNoInitSetup`

Call `flag = KINSetNoInitSetup(kin_mem, noInitSetup);`

Description The function `KINSetNoInitSetup` specifies whether an initial call to the preconditioner setup function should be made or not.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`noInitSetup` (`booleantype`) flag controlling whether or not an initial call to the preconditioner setup function is made.

Return value The return value `flag` (of type `int`) is one of:

| | |
|---|---|
| `KIN_SUCCESS` | The optional value has been successfully set. |
| `KIN_MEM_NULL` | The `kin_mem` pointer is `NULL`. |

Notes The default value for `noInitSetup` is `FALSE`, meaning that an initial call to the preconditioner setup function will be made.

`KINSetNoResMon`

Call `flag = KINSetNoResMon(kin_mem, noNNIResMon);`

Description The function `KINSetNoResMon` specifies whether or not the nonlinear residual monitoring scheme is used to control Jacobian updating.

Arguments
- `kin_mem` (`void *`) pointer to the KINSOL memory block.
- `noNNIResMon` (`booleantype`) flag controlling whether or not residual monitoring is used.

Return value The return value `flag` (of type `int`) is one of:
- `KIN_SUCCESS` The optional value has been successfully set.
- `KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.

Notes When using a direct solver, the default value for `noNNIResMon` is `FALSE`, meaning that the nonlinear residual will be monitored. Otherwise, `noNNIResMon` is set to `TRUE`.

! Residual monitoring is only available for use with the direct linear solver modules (meaning KINDENSE and KINBAND).

`KINSetMaxSetupCalls`

Call `flag = KINSetMaxSetupCalls(kin_mem, msbset);`

Description The function `KINSetMaxSetupCalls` specifies the maximum number of nonlinear iterations that can be performed between calls to the preconditioner setup function.

Arguments
- `kin_mem` (`void *`) pointer to the KINSOL memory block.
- `msbset` (`long int`) maximum number of nonlinear iterations without a call to the preconditioner setup function.

Return value The return value `flag` (of type `int`) is one of:
- `KIN_SUCCESS` The optional value has been successfully set.
- `KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.
- `KIN_ILL_INPUT` The argument `msbset` was negative.

Notes The default value for `msbset` is `MSBSET_DEFAULT` $= 10$.

`KINSetMaxSubSetupCalls`

Call `flag = KINSetMaxSubSetupCalls(kin_mem, msbsetsub);`

Description The function `KINSetMaxSubSetupCalls` specifies the maximum number of nonlinear iterations between checks by the residual monitoring algorithm.

Arguments
- `kin_mem` (`void *`) pointer to the KINSOL memory block.
- `msbsetsub` (`long int`) maximum number of nonlinear iterations without checking the nonlinear residual.

Return value The return value `flag` (of type `int`) is one of:
- `KIN_SUCCESS` The optional value has been successfully set.
- `KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.
- `KIN_ILL_INPUT` The argument `msbsetsub` was negative.

Notes The default value for `msbsetsub` is `MSBSET_SUB_DEFAULT` $= 5$.

! Residual monitoring is only available for use with the direct linear solver modules (meaning KINDENSE and KINBAND).

`KINSetEtaForm`

Call `flag = KINSetEtaForm(kin_mem, etachoice);`

Description The function `KINSetEtaForm` specifies the method for computing the value of the $\eta$ coefficient used in the calculation of the linear solver convergence tolerance.

Arguments
- `kin_mem` (`void *`) pointer to the KINSOL memory block.
- `etachoice` (`int`) flag indicating the method for computing $\eta$. `etachoice` must be one of `KIN_ETACHOICE1`, `KIN_ETACHOICE2`, or `KIN_ETACONSTANT` (see Chapter 3 for details).

Return value The return value `flag` (of type `int`) is one of:
- `KIN_SUCCESS` The optional value has been successfully set.
- `KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.
- `KIN_ILL_INPUT` The argument `etachoice` had an illegal value.

Notes The default value for `etachoice` is `KIN_ETACHOICE1`.

`KINSetEtaConstValue`

Call `flag = KINSetEtaConstValue(kin_mem, eta);`

Description The function `KINSetEtaConstValue` specifies the constant value for $\eta$ in the case `etachoice = KIN_ETACONSTANT`.

Arguments
- `kin_mem` (`void *`) pointer to the KINSOL memory block.
- `eta` (`realtype`) constant value for $\eta$.

Return value The return value `flag` (of type `int`) is one of:
- `KIN_SUCCESS` The optional value has been successfully set.
- `KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.
- `KIN_ILL_INPUT` The argument `eta` had an illegal value

Notes The default value for `eta` is 0.1. The valid values are $0.0 <$ `eta` $\leq 1.0$.

`KINSetEtaParams`

Call `flag = KINSetEtaParams(kin_mem, egamma, ealpha);`

Description The function `KINSetEtaParams` specifies the parameters $\gamma$ and $\alpha$ in the formula for $\eta$, in the case `etachoice = KIN_ETACHOICE2`.

Arguments
- `kin_mem` (`void *`) pointer to the KINSOL memory block.
- `egamma` (`realtype`) value of the $\gamma$ parameter.
- `ealpha` (`realtype`) value of the $\alpha$ parameter.

Return value The return value `flag` (of type `int`) is one of:
- `KIN_SUCCESS` The optional values have been successfully set.
- `KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.
- `KIN_ILL_INPUT` One of the arguments `egamma` or `ealpha` had an illegal value.

Notes The default values for `egamma` and `ealpha` are 0.9 and 2.0, respectively.

The valid values for `ealpha` are $1.0 <$ `ealpha` $\leq 2.0$. If `ealpha` $= 0.0$, then its value is set to 2.0.

The valid values for `egamma` are $0.0 <$ `egamma` $\leq 1.0$. If `egamma` $= 0.0$, then its value is set to 0.9.

`KINSetResMonConstValue`

Call `flag = KINSetResMonConstValue(kin_mem, omegaconst);`

Description The function `KINSetResMonConstValue` specifies the constant value for $\omega$ when using residual monitoring.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`omegaconst` (`realtype`) constant value for $\omega$.

Return value The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS` The optional value has been successfully set.
`KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.
`KIN_ILL_INPUT` The argument `omegaconst` had an illegal value

Notes The default value for `omega` is 0.9. The valid values are 0.0 < `omega` < 1.0.

`KINSetResMonParams`

Call `flag = KINSetResMonParams(kin_mem, omegamin, omegamax);`

Description The function `KINSetResMonParams` specifies the parameters $\omega_{min}$ and $\omega_{max}$ in the formula for $\omega$.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`omegamin` (`realtype`) value of the $\omega_{min}$ parameter.
`omegamax` (`realtype`) value of the $\omega_{max}$ parameter.

Return value The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS` The optional values have been successfully set.
`KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.
`KIN_ILL_INPUT` One of the arguments `omegamin` or `omegamax` had an illegal value.

Notes The default values for `omegamin` and `omegamax` are 0.00001 and 0.9, respectively.

The valid values for `omegamin` and `omegamax` are 0.0 < `omegamin` < `omegamax` < 1.0. If `omegamin` = 0.0, then its value is set to 0.00001. If `omegamax` = 0.0, then its value is set to 0.9.

`KINSetNoMinEps`

Call `flag = KINSetNoMinEps(kin_mem, noMinEps);`

Description The function `KINSetNoMinEps` specifies a flag that controls whether or not the value of $\epsilon$, the scaled linear residual tolerance, is bounded from below.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`noMinEps` (`booleantype`) flag controlling the bound on $\epsilon$.

Return value The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS` The optional value has been successfully set.
`KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.

Notes The default value for `noMinEps` is `FALSE`.

`KINSetMaxNewtonStep`

Call `flag = KINSetMaxNewtonStep(kin_mem, mxnewtstep);`

Description The function `KINSetMaxNewtonStep` specifies the maximum allowable scaled length of the Newton step.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.

mxnewtstep (realtype) maximum scaled step length.

Return value The return value flag (of type int) is one of:

KIN_SUCCESS The optional value has been successfully set.

KIN_MEM_NULL The kin_mem pointer is NULL.

KIN_ILL_INPUT The maximum step was non-positive.

Notes The default value of mxnewtstep is $1000 \|u_0\|_{D_u}$, where $u_0$ is the initial guess.

KINSetMaxBetaFails

Call `flag = KINSetMaxBetaFails(kin_mem, mxnbcf);`

Description The function KINSetMaxBetaFails specifies the maximum number of $\beta$-condition failures in the linesearch algorithm.

Arguments kin_mem (void *) pointer to the KINSOL memory block.

mxnbcf (realtype) maximum number of $\beta$-condition failures.

Return value The return value flag (of type int) is one of:

KIN_SUCCESS The optional value has been successfully set.

KIN_MEM_NULL The kin_mem pointer is NULL.

KIN_ILL_INPUT mxnbcf was non-positive.

Notes The default value of mxnbcf is MXNBCF_DEFAULT = 10.

KINSetRelErrFunc

Call `flag = KINSetRelErrFunc(kin_mem, relfunc);`

Description The function KINSetRelErrFunc specifies the relative error in computing $F(u)$, which is used in the difference quotient approximation of the Jacobian-vector product.

Arguments kin_mem (void *) pointer to the KINSOL memory block.

relfunc (realtype) relative error in $F(u)$.

Return value The return value flag (of type int) is one of:

KIN_SUCCESS The optional value has been successfully set.

KIN_MEM_NULL The kin_mem pointer is NULL.

KIN_ILL_INPUT The relative error was non-positive.

Notes The default value for relfunc is $\sqrt{\text{unit roundoff}}$.

KINSetFuncNormTol

Call `flag = KINSetFuncNormTol(kin_mem, fnormtol);`

Description The function KINSetFuncNormTol specifies the scalar used as a stopping tolerance on the scaled maximum norm of the system function $F(u)$.

Arguments kin_mem (void *) pointer to the KINSOL memory block.

fnormtol (realtype) tolerance for stopping based on scaled function norm.

Return value The return value flag (of type int) is one of:

KIN_SUCCESS The optional value has been successfully set.

KIN_MEM_NULL The kin_mem pointer is NULL.

KIN_ILL_INPUT The tolerance was non-positive.

Notes The default value for fnormtol is $\sqrt[3]{\text{unit roundoff}}$.

`KINSetScaledStepTol`

Call `flag = KINSetScaledStepTol(kin_mem, scsteptol);`

Description The function `KINSetScaledStepTol` specifies the scalar used as a stopping tolerance on the minimum scaled step length.

Arguments
`kin_mem` (`void *`) pointer to the KINSOL memory block.
`scsteptol` (`realtype`) tolerance for stopping based on scaled step length..

Return value The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS` The optional value has been successfully set.
`KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.
`KIN_ILL_INPUT` The tolerance was non-positive.

Notes The default value for `scsteptol` is (unit roundoff)$^{2/3}$.

`KINSetConstraints`

Call `flag = KINSetConstraints(kin_mem, constraints);`

Description The function `KINSetConstraints` specifies a vector that defines inequality constraints for each component of the solution vector $u$.

Arguments
`kin_mem` (`void *`) pointer to the KINSOL memory block.
`constraints` (`N_Vector`) vector of constraint flags. If `constraints[i]` is

0.0 then no constraint is imposed on $u_i$.
1.0 then $u_i$ will be constrained to be $u_i \geq 0.0$.
$-1.0$ then $u_i$ will be constrained to be $u_i \leq 0.0$.
2.0 then $u_i$ will be constrained to be $u_i > 0.0$.
$-2.0$ then $u_i$ will be constrained to be $u_i < 0.0$.

Return value The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS` The optional value has been successfully set.
`KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.
`KIN_ILL_INPUT` The constraint vector contains illegal values.

Notes The presence of a non-`NULL` constraints vector that is not 0.0 in all components will cause constraint checking to be performed.

The function creates a private copy of the constraints vector. Consequently, the user-supplied vector can be freed after the function call, and the constraints can only be changed by calling this function.

`KINSetSysFunc`

Call `flag = KINSetSysFunc(kin_mem, func);`

Description The function `KINSetSysFunc` specifies the user-provided function that evaluates the nonlinear system function $F(u)$.

Arguments
`kin_mem` (`void *`) pointer to the KINSOL memory block.
`func` (`KINSysFn`) user-supplied function that evaluates $F(u)$.

Return value The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS` The optional value has been successfully set.
`KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.
`KIN_ILL_INPUT` The argument `func` was `NULL`.

Notes The nonlinear system function is initially specified through `KINMalloc`. The option of changing the system function is provided for a user who wishes to solve several problems of the same size but with different functions.

#### 5.5.4.2 Dense linear solver

The KINDENSE solver needs a function to compute a dense approximation to the Jacobian matrix $J(u)$. This function must be of type `KINDenseJacFn`. The user can supply his/her own dense Jacobian function, or use the default difference quotient function `KINDenseDQJac` that comes with the KINDENSE solver. To specify a user-supplied Jacobian function `djac` and associated user data `jac_data`, KINDENSE provides the function `KINDenseSetJacFn`. The KINDENSE solver passes the pointer `jac_data` to its dense Jacobian function. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied Jacobian function, without using global data in the program. The pointer `jac_data` may be identical to `f_data`, if the latter was specified through `KINSetFdata`.

`KINDenseSetJacFn`

**Call** `flag = KINDenseSetJacFn(kin_mem, djac, jac_data);`

**Description** The function `KINDenseSetJacFn` specifies the dense Jacobian approximation function to be used and the pointer to user data.

**Arguments**
- `kin_mem` (`void *`) pointer to the KINSOL memory block.
- `djac` (`KINDenseJacFn`) user-defined dense Jacobian approximation function.
- `jac_data` (`void *`) pointer to the user-defined data structure.

**Return value** The return value `flag` (of type `int`) is one of
- `KINDENSE_SUCCESS` The optional value has been successfully set.
- `KINDENSE_MEM_NULL` The `kin_mem` pointer is `NULL`.
- `KINDENSE_LMEM_NULL` The KINDENSE linear solver has not been initialized.

**Notes** By default, KINDENSE uses the difference quotient function `KINDenseDQJac`. If `NULL` is passed to `djac`, this default function is used.

The function type `KINDenseJacFn` is described in §5.6.4.

#### 5.5.4.3 Band linear solver

The KINDENSE solver needs a function to compute a banded approximation to the Jacobian matrix $J(u)$. This function must be of type `KINBandJacFn`. The user can supply his/her own banded Jacobian approximation function, or use the default difference quotient function `KINBandDQJac` that comes with the KINBAND solver. To specify a user-supplied Jacobian function `bjac` and associated user data `jac_data`, KINBAND provides the function `KINBandSetJacFn`. The KINBAND solver passes the pointer `jac_data` to its banded Jacobian approximation function. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied Jacobian function, without using global data in the program. The pointer `jac_data` may be identical to `f_data`, if the latter was specified through `KINSetFdata`.

`KINBandSetJacFn`

**Call** `flag = KINBandSetJacFn(cvode_mem, bjac, jac_data);`

**Description** The function `KINBandSetJacFn` specifies the banded Jacobian approximation function to be used and the pointer to user data.

**Arguments**
- `kin_mem` (`void *`) pointer to the KINSOL memory block.
- `bjac` (`KINBandJacFn`) user-defined banded Jacobian approximation function.
- `jac_data` (`void *`) pointer to the user-defined data structure.

**Return value** The return value `flag` (of type `int`) is one of
- `KINBAND_SUCCESS` The optional value has been successfully set.
- `KINBAND_MEM_NULL` The `kin_mem` pointer is `NULL`.

KINBAND_LMEM_NULL The KINBAND linear solver has not been initialized.

Notes: By default, KINBAND uses the difference quotient function `KINBandDQJac`. If `NULL` is passed to `bjac`, this default function is used.

The function type `KINBandJacFn` is described in §5.6.5.

#### 5.5.4.4 SPILS linear solvers

If any preconditioning is to be done with one of the KINSPILS linear solvers, then the user must supply a preconditioner solve function `psolve` and specify its name in a call to `KINSpilsSetPreconditioner`.

The evaluation and preprocessing of any Jacobian-related data needed by the user's preconditioner solve function is done in the optional user-supplied function `psetup`. Both of these functions are fully specified in §5.6. If used, the `psetup` function should also be specified in the call to `KINSpilsSetPreconditioner`. Optionally, a KINSPILS solver passes the pointer `p_data` received through `KINSpilsSetPreconditioner` to the preconditioner `psetup` and `psolve` functions. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied preconditioner functions without using global data in the program. The pointer `p_data` may be identical to `f_data`, if the latter was specified through `KINSetFdata`.

Ther KINSPILS solvers require a function to compute an approximation to the product between the Jacobian matrix $J(u)$ and a vector $v$. The user can supply his/her own Jacobian-times-vector approximation function, or use the difference quotient function `KINSpilsDQJtimes` that comes with the KINSPILS solvers. A user-defined Jacobian-vector function must be of type `KINSpilsJacTimesVecFn` and can be specified through a call to `KINSpilsSetJacTimesVecFn` (see §5.6.6 for specification details). As with the preconditioner user data structure `p_data`, the user can also specify, in the call to `KINSpilsSetJacTimesVecFn`, a pointer to a user-defined data structure, `jac_data`, which the KINSPILS solver passes to the Jacobian-times-vector function `jtimes` each time it is called. The pointer `jac_data` may be identical to `p_data` and/or `f_data`.

**`KINSpilsSetPreconditioner`**

Call: `flag = KINSpilsSetPreconditioner(kin_mem, psetup, psolve, p_data);`

Description: The function `KINSpilsSetPreconditioner` specifies the preconditioner setup and solve functions and the pointer to user data.

Arguments:
- `kin_mem` (`void *`) pointer to the KINSOL memory block.
- `psetup` (`KINSpilsPrecSetupFn`) user-defined preconditioner setup function.
- `psolve` (`KINSpilsPrecSolveFn`) user-defined preconditioner solve function.
- `p_data` (`void *`) pointer to the user-defined data structure.

Return value: The return value `flag` (of type `int`) is one of
- `KINSPILS_SUCCESS` The optional value has been successfully set.
- `KINSPILS_MEM_NULL` The `kin_mem` pointer is `NULL`.
- `KINSPILS_LMEM_NULL` The KINSPILS linear solver has not been initialized.

Notes: The function type `KINSpilsPrecSolveFn` is described in §5.6.7. The function type `KINSpilsPrecSetupFn` is described in §5.6.8.

**`KINSpilsSetJacTimesVecFn`**

Call: `flag = KINSpilsSetJacTimesVecFn(kin_mem, jtimes, jac_data);`

Description: The function `KINSpilsSetJacTimesFn` specifies the Jacobian-vector function to be used and the pointer to user data.

Arguments:
- `kin_mem` (`void *`) pointer to the KINSOL memory block.
- `jtimes` (`KINSpilsJacTimesVecFn`) user-defined Jacobian-vector product function.
- `jac_data` (`void *`) pointer to the user-defined data structure.

Return value The return value `flag` (of type `int`) is one of

- `KINSPILS_SUCCESS` The optional value has been successfully set.
- `KINSPILS_MEM_NULL` The `kin_mem` pointer is `NULL`.
- `KINSPILS_LMEM_NULL` The KINSPILS linear solver has not been initialized.

Notes By default, the KINSPILS linear solvers use an internal difference quotient function `KINSpilsDQJtimes`. If `NULL` is passed to `jtimes`, this default function is used.

The function type `KINSpilsJacTimesVecFn` is described in §5.6.6.

`KINSpilsSetMaxRestarts`

Call `flag = KINSpilsSetMaxRestarts(kin_mem, maxrs);`

Description The function `KINSpilsSetMaxRestarts` specifies the maximum number of times the iterative linear solver can be restarted.

Arguments
- `kin_mem` (`void *`) pointer to the KINSOL memory block.
- `maxrs` (`int`) maximum number of restarts.

Return value The return value `flag` (of type `int`) is one of:

- `KINSPILS_SUCCESS` The optional value has been successfully set.
- `KINSPILS_ILL_INPUT` The maximum number of restarts specified is negative.
- `KINSPILS_MEM_NULL` The `kin_mem` pointer is `NULL`.
- `KINSPILS_LMEM_NULL` The linear solver has not been initialized.

Notes The default value is 0 (meaning no restarts).

This option is available only for the KINSPGMR linear solver.

### 5.5.5 Optional output functions

KINSOL provides an extensive list of functions that can be used to obtain solver performance information. Table 5.2 lists all optional output functions in KINSOL, which are then described in detail in the remainder of this section, beginning with those for the main KINSOL solver and continuing with those for the linear solver modules. Where the name of an output from a linear solver module would otherwise conflict with the name of an optional output from the main solver, a suffix `LS` (for Linear Solver) has been added here (*e.g.*, `lenrwLS`).

Table 5.2: Optional outputs from KINSOL, KINDENSE, KINBAND, and KINSPILS

| **Optional output** | **Function name** |
|---|---|
| **KINSOL main solver** | |
| Size of KINSOL real and integer workspaces | `KINGetWorkSpace` |
| Number of function evaluations | `KINGetNumFuncEvals` |
| Number of nonlinear iterations | `KINGetNumNolinSolvIters` |
| Number of $\beta$-condition failures | `KINGetNumBetaCondFails` |
| Number of backtrack operations | `KINGetNumBacktrackOps` |
| Scaled norm of $F$ | `KINGetFuncNorm` |
| Scaled norm of the step | `KINGetStepLength` |
| **KINDENSE linear solver** | |
| Size of KINDENSE real and integer workspaces | `KINDenseGetWorkSpace` |
| No. of Jacobian evaluations | `KINDenseGetNumJacEvals` |
| No. of r.h.s. calls for finite diff. Jacobian evals. | `KINDenseGetNumFuncEvals` |
| Last return from a KINDENSE function | `KINDenseGetLastFlag` |

*continued on next page*

| *continued from last page* | |
|---|---|
| **Optional output** | **Function name** |
| **KINBAND linear solver** | |
| Size of KINBAND real and integer workspaces | KINBandGetWorkSpace |
| No. of Jacobian evaluations | KINBandGetNumJacEvals |
| No. of r.h.s. calls for finite diff. Jacobian evals. | KINBandGetNumFuncEvals |
| Last return from a KINBAND function | KINBandGetLastFlag |
| **KINSPILS linear solvers** | |
| Size of real and integer workspaces | KINSpilsGetWorkSpace |
| No. of linear iterations | KINSpilsGetNumLinIters |
| No. of linear convergence failures | KINSpilsGetNumConvFails |
| No. of preconditioner evaluations | KINSpilsGetNumPrecEvals |
| No. of preconditioner solves | KINSpilsGetNumPrecSolves |
| No. of Jacobian-vector product evaluations | KINSpilsGetNumJtimesEvals |
| No. of fct. calls for finite diff. Jacobian-vector evals. | KINSpilsGetNumFuncEvals |
| Last return from a linear solver function | KINSpilsGetLastFlag |

#### 5.5.5.1 Main solver optional output functions

KINSOL provides several user-callable functions that can be used to obtain different quantities that may be of interest to the user, such as solver workspace requirements and solver performance statistics. These optional output functions are described next.

`KINGetWorkSpace`

Call `flag = KINGetWorkSpace(kin_mem, &lenrw, &leniw);`

Description The function `KINGetWorkSpace` returns the KINSOL integer and real workspace sizes.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`lenrw` (`long int`) the number of `realtype` values in the KINSOL workspace.
`leniw` (`long int`) the number of integer values in the KINSOL workspace.

Return value The return value `flag` (of type `int`) is one of:
`KIN_SUCCESS` The optional output values have been successfully set.
`KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.

Notes In terms of the problem size $N$, the actual size of the real workspace is $17+5N$ `realtype` words. The real workspace is increased by an additional $N$ words if constraint checking is enabled (see `KINSetConstraints`).

The actual size of the integer workspace (without distinction between `int` and `long int`) is $22+5N$ (increased by $N$ if constraint checking is enabled).

`KINGetNumFuncEvals`

Call `flag = KINGetNumFuncEvals(kin_mem, &nfevals);`

Description The function `KINGetNumFuncEvals` returns the number of evaluations of the system function.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`nfevals` (`long int`) number of calls to the user-supplied function that evaluates $F(u)$.

Return value The return value `flag` (of type `int`) is one of:
`KIN_SUCCESS` The optional output value has been successfully set.
`KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.

KINGetNumNonlinSolvIters

Call `flag = KINGetNumNonlinSolvIters(kin_mem, &nniters);`

Description The function `KINGetNumNonlinSolvIters` returns the number of nonlinear iterations.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`nniters` (`long int`) number of nonlinear iterations.

Return value The return value `flag` (of type `int`) is one of:
`KIN_SUCCESS` The optional output value has been successfully set.
`KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.

KINGetNumBetaCondFails

Call `flag = KINGetNumBetaCondFails(kin_mem, &nbcfails);`

Description The function `KINGetNumBetaCondFails` returns the number of $\beta$-condition failures.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`nbcfails` (`long int`) number of $\beta$-condition failures.

Return value The return value `flag` (of type `int`) is one of:
`KIN_SUCCESS` The optional output value has been successfully set.
`KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.

KINGetNumBacktrackOps

Call `flag = KINGetNumBacktrackOps(kin_mem, &nbacktr);`

Description The function `KINGetNumBacktrackOps` returns the number of backtrack operations (step length adjustments) performed by the line search algorithm.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`nbacktr` (`long int`) number of backtrack operations.

Return value The return value `flag` (of type `int`) is one of:
`KIN_SUCCESS` The optional output value has been successfully set.
`KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.

KINGetFuncNorm

Call `flag = KINGetFuncNorm(kin_mem, &fnorm);`

Description The function `KINGetFuncNorm` returns the scaled Euclidean $\ell_2$ norm of the nonlinear system function $F(u)$ evaluated at the current iterate.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`fnorm` (`realtype`) current scaled norm of $F(u)$.

Return value The return value `flag` (of type `int`) is one of:
`KIN_SUCCESS` The optional output value has been successfully set.
`KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.

KINGetStepLength

Call `flag = KINGetStepLength(kin_mem, &steplength);`

Description The function `KINGetStepLength` returns the scaled Euclidean $\ell_2$ norm of the step used during the previous iteration.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.

steplength (realtype) scaled norm of the Newton step.

Return value The return value flag (of type int) is one of:

KIN_SUCCESS The optional output value has been successfully set.

KIN_MEM_NULL The kin_mem pointer is NULL.

#### 5.5.5.2 Dense linear solver

The following optional outputs are available from the KINDENSE module: workspace requirements, number of calls to the Jacobian routine, number of calls to the system function routine for finite-difference Jacobian approximation, and last return value from a KINDENSE function.

**KINDenseGetWorkSpace**

Call `flag = KINDenseGetWorkSpace(kin_mem, &lenrwLS, &leniwLS);`

Description The function KINDenseGetWorkSpace returns the KINDENSE real and integer workspace sizes.

Arguments kin_mem (void *) pointer to the KINSOL memory block.

lenrwLS (long int) the number of realtype values in the KINDENSE workspace.

leniwLS (long int) the number of integer values in the KINDENSE workspace.

Return value The return value flag (of type int) is one of

KINDENSE_SUCCESS The optional output value has been successfully set.

KINDENSE_MEM_NULL The kin_mem pointer is NULL.

KINDENSE_LMEM_NULL The KINDENSE linear solver has not been initialized.

Notes In terms of the problem size $N$, the actual size of the real workspace is $2N^2$ realtype words, and the actual size of the integer workspace is $N$ integer words.

**KINDenseGetNumJacEvals**

Call `flag = KINDenseGetNumJacEvals(kin_mem, &njevals);`

Description The function KINDenseGetNumJacEvals returns the number of calls to the dense Jacobian approximation function.

Arguments kin_mem (void *) pointer to the KINSOL memory block.

njevals (long int) the number of calls to the Jacobian function.

Return value The return value flag (of type int) is one of

KINDENSE_SUCCESS The optional output value has been successfully set.

KINDENSE_MEM_NULL The kin_mem pointer is NULL.

KINDENSE_LMEM_NULL The KINDENSE linear solver has not been initialized.

**KINDenseGetNumFuncEvals**

Call `flag = KINDenseGetNumFuncEvals(kin_mem, &nfevalsLS);`

Description The function KINDenseGetNumFuncEvals returns the number of calls to the user system function due to the finite difference dense Jacobian approximation.

Arguments kin_mem (void *) pointer to the KINSOL memory block.

nfevalsLS (long int) the number of calls to the user system function.

Return value The return value flag (of type int) is one of

KINDENSE_SUCCESS The optional output value has been successfully set.

KINDENSE_MEM_NULL The kin_mem pointer is NULL.

KINDENSE_LMEM_NULL The KINDENSE linear solver has not been initialized.

Notes The value nfevalsLS is incremented only if the default KINDenseDQJac difference quotient function is used.

KINDenseGetLastFlag

Call flag = KINDenseGetLastFlag(kin_mem, &lsflag);

Description The function KINDenseGetLastFlag returns the last return value from a KINDENSE routine.

Arguments kin_mem (void *) pointer to the KINSOL memory block.
lsflag (int) the value of the last return flag from a KINDENSE function.

Return value The return value flag (of type int) is one of

KINDENSE_SUCCESS The optional output value has been successfully set.
KINDENSE_MEM_NULL The kin_mem pointer is NULL.
KINDENSE_LMEM_NULL The KINDENSE linear solver has not been initialized.

Notes If the KINDENSE setup function failed (KINSol returned KIN_LSETUP_FAIL), then lsflag is equal to the column index (numbered from one) at which a zero diagonal element was encountered during the LU factorization of the dense Jacobian matrix.

#### 5.5.5.3 Band linear solver

The following optional outputs are available from the KINBAND module: workspace requirements, number of calls to the Jacobian routine, number of calls to the right-hand side routine for finite-difference Jacobian approximation, and last return value from a KINBAND function.

KINBandGetWorkSpace

Call flag = KINBandGetWorkSpace(kin_mem, &lenrwLS, &leniwLS);

Description The function KINBandGetWorkSpace returns the KINBAND real and integer workspace sizes.

Arguments kin_mem (void *) pointer to the CVODE memory block.
lenrwLS (long int) the number of realtype values in the KINBAND workspace.
leniwLS (long int) the number of integer values in the KINBAND workspace.

Return value The return value flag (of type int) is one of

KINBAND_SUCCESS The optional output value has been successfully set.
KINBAND_MEM_NULL The kin_mem pointer is NULL.
KINBAND_LMEM_NULL The KINBAND linear solver has not been initialized.

Notes In terms of the problem size $N$ and Jacobian half-bandwidths, the actual size of the real workspace is (2 mupper+3 mlower +2) $N$ realtype words, and the actual size of the integer workspace is $N$ integer words.

KINBandGetNumJacEvals

Call flag = KINBandGetNumJacEvals(kin_mem, &njevals);

Description The function KINBandGetNumJacEvals returns the number of calls to the banded Jacobian approximation function.

Arguments kin_mem (void *) pointer to the CVODE memory block.
njevals (long int) the number of calls to the Jacobian function.

Return value The return value flag (of type int) is one of

| | |
|---|---|
| KINBAND_SUCCESS | The optional output value has been successfully set. |
| KINBAND_MEM_NULL | The kin_mem pointer is NULL. |
| KINBAND_LMEM_NULL | The KINBAND linear solver has not been initialized. |

**KINBandGetNumRhsEvals**

Call `flag = KINBandGetNumFuncEvals(kin_mem, &nfevalsLS);`

Description The function `KINBandGetNumFuncEvals` returns the number of calls to the user system function due to the finite difference banded Jacobian approximation.

Arguments `kin_mem` (`void *`) pointer to the CVODE memory block.

`nfevalsLS` (`long int`) the number of calls to the user system function.

Return value The return value `flag` (of type `int`) is one of

| | |
|---|---|
| KINBAND_SUCCESS | The optional output value has been successfully set. |
| KINBAND_MEM_NULL | The kin_mem pointer is NULL. |
| KINBAND_LMEM_NULL | The KINBAND linear solver has not been initialized. |

Notes The value `nfevalsLS` is incremented only if the default `KINBandDQJac` difference quotient function is used.

**KINBandGetLastFlag**

Call `flag = KINBandGetLastFlag(kin_mem, &lsflag);`

Description The function `KINBandGetLastFlag` returns the last return value from a KINBAND routine.

Arguments `kin_mem` (`void *`) pointer to the CVODE memory block.

`lsflag` (`int`) the value of the last return flag from a KINBAND function.

Return value The return value `flag` (of type `int`) is one of

| | |
|---|---|
| KINBAND_SUCCESS | The optional output value has been successfully set. |
| KINBAND_MEM_NULL | The kin_mem pointer is NULL. |
| KINBAND_LMEM_NULL | The KINBAND linear solver has not been initialized. |

Notes If the KINBAND setup sunction failed (`KINSol` returned `KIN_LSETUP_FAIL`), the value `lsflag` is equal to the column index (numbered from one) at which a zero diagonal element was encountered during the LU factorization of the banded Jacobian matrix.

#### 5.5.5.4 SPILS linear solvers

The following optional outputs are available from the KINSPILS modules: workspace requirements, number of linear iterations, number of linear convergence failures, number of calls to the preconditioner setup and solve routines, number of calls to the Jacobian-vector product routine, number of calls to the system function routine for finite-difference Jacobian-vector product approximation, and last return value from a linear solver function.

**KINSpilsGetWorkSpace**

Call `flag = KINSpilsGetWorkSpace(kin_mem, &lenrwLS, &leniwLS);`

Description The function `KINSpilsGetWorkSpace` returns the global sizes of the linear solver real and integer workspaces.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.

`lenrwSG` (`long int`) the number of `realtype` values in the linear solver workspace.

`leniwSG` (`long int`) the number of integer values in the linear solver workspace.

Return value The return value flag (of type int) is one of:

| | |
|---|---|
| KINSPILS_SUCCESS | The optional output values have been successfully set. |
| KINSPILS_MEM_NULL | The kin_mem pointer is NULL. |
| KINSPILS_LMEM_NULL | The KINSPGMR linear solver has not been initialized. |

Notes In terms of the problem size $N$ and maximum subspace size maxl, the actual size of the real workspace is roughly:
$(\texttt{maxl}+3) * N+$ maxl $*(\texttt{maxl}+4) + 1$ realtype words for KINSPGMR,
$7 * N$ realtype words for KINSPBCG,
and $11 * N$ realtype words for KINSPTFQMR.

In a parallel setting, this value is global — summed over all processes.

### KINSpilsGetNumLinIters

Call `flag = KINSpilsGetNumLinIters(kin_mem, &nliters);`

Description The function KINSpilsGetNumLinIters returns the cumulative number of linear iterations.

Arguments kin_mem (void *) pointer to the KINSOL memory block.
nliters (long int) the current number of linear iterations.

Return value The return value flag (of type int) is one of:

| | |
|---|---|
| KINSPILS_SUCCESS | The optional output value has been successfully set. |
| KINSPILS_MEM_NULL | The kin_mem pointer is NULL. |
| KINSPLIS_LMEM_NULL | The linear solver module has not been initialized. |

### KINSpilsGetNumConvFails

Call `flag = KINSpilsGetNumConvFails(kin_mem, &nlcfails);`

Description The function KINSpilsGetNumConvFails returns the cumulative number of linear convergence failures.

Arguments kin_mem (void *) pointer to the KINSOL memory block.
nlcfails (long int) the current number of linear convergence failures.

Return value The return value flag (of type int) is one of:

| | |
|---|---|
| KINSPILS_SUCCESS | The optional output value has been successfully set. |
| KINSPILS_MEM_NULL | The kin_mem pointer is NULL. |
| KINSPILS_LMEM_NULL | The linear solver module has not been initialized. |

### KINSpilsGetNumPrecEvals

Call `flag = KINSpilsGetNumPrecEvals(kin_mem, &npevals);`

Description The function KINSpilsGetNumPrecEvals returns the number of preconditioner evaluations, i.e., the number of calls made to psetup.

Arguments kin_mem (void *) pointer to the KINSOL memory block.
npevals (long int) the current number of calls to psetup.

Return value The return value flag (of type int) is one of:

| | |
|---|---|
| KINSPILS_SUCCESS | The optional output value has been successfully set. |
| KINSPILS_MEM_NULL | The kin_mem pointer is NULL. |
| KINSPILS_LMEM_NULL | The linear solver module has not been initialized. |

`KINSpilsGetNumPrecSolves`

Call `flag = KINSpilsGetNumPrecSolves(kin_mem, &npsolves);`

Description The function `KINSpilsGetNumPrecSolves` returns the cumulative number of calls made to the preconditioner solve function, `psolve`.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`npsolves` (`long int`) the current number of calls to `psolve`.

Return value The return value `flag` (of type `int`) is one of:

| | |
|---|---|
| `KINSPILS_SUCCESS` | The optional output value has been successfully set. |
| `KINSPILS_MEM_NULL` | The `kin_mem` pointer is `NULL`. |
| `KINSPILS_LMEM_NULL` | The linear solver module has not been initialized. |

`KINSpilsGetNumJtimesEvals`

Call `flag = KINSpilsGetNumJtimesEvals(kin_mem, &njvevals);`

Description The function `KINSpilsGetNumJtimesEvals` returns the cumulative number made to the Jacobian-vector product function, `jtimes`.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`njvevals` (`long int`) the current number of calls to `jtimes`.

Return value The return value `flag` (of type `int`) is one of:

| | |
|---|---|
| `KINSPILS_SUCCESS` | The optional output value has been successfully set. |
| `KINSPILS_MEM_NULL` | The `kin_mem` pointer is `NULL`. |
| `KINSPILS_LMEM_NULL` | The linear solver module has not been initialized. |

`KINSpilsGetNumFuncEvals`

Call `flag = KINSpilsGetNumFuncEvals(kin_mem, &nfevalsLS);`

Description The function `KINSpilsGetNumFuncEvals` returns the number of calls to the user system function for finite difference Jacobian-vector product approximations.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`nfevalsLS` (`long int`) the number of calls to the user system function.

Return value The return value `flag` (of type `int`) is one of:

| | |
|---|---|
| `KINSPILS_SUCCESS` | The optional output value has been successfully set. |
| `KINSPILS_MEM_NULL` | The `kin_mem` pointer is `NULL`. |
| `KINSPILS_LMEM_NULL` | The linear solver module has not been initialized. |

Notes The value `nfevalsLS` is incremented only if the default `KINSpilsDQJtimes` difference quotient function is used.

`KINSpilsGetLastFlag`

Call `flag = KINSpilsGetLastFlag(kin_mem, &lsflag);`

Description The function `KINSpilsGetLastFlag` returns the last return value from a KINSPILS routine.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`lsflag` (`int`) the value of the last return flag from a KINSPILS function.

Return value The return value `flag` (of type `int`) is one of:

| | |
|---|---|
| `KINSPILS_SUCCESS` | The optional output value has been successfully set. |
| `KINSPILS_MEM_NULL` | The `kin_mem` pointer is `NULL`. |

KINSPILS_LMEM_NULL The linear solver module has not been initialized.

Notes If the KINSPILS setup function failed (KINSOL returned KIN_LSETUP_FAIL), lsflag will be SPGMR_PSET_FAIL_UNREC, SPBCG_PSET_FAIL_UNREC, or SPTFQMR_PSET_FAIL_UNREC.

If the KINSPGMR solve function failed (KINSol returned KIN_LSOLVE_FAIL), lsflag contains the error return flag from SpgmrSolve and will be one of: SPGMR_MEM_NULL, indicating that the SPGMR memory is NULL; SPGMR_ATIMES_FAIL_UNREC, indicating an unrecoverable failure in the Jacobian-times-vector function; SPGMR_PSOLVE_FAIL_UNREC, indicating that the preconditioner solve function psolve failed unrecoverably; SPGMR_GS_FAIL, indicating a failure in the Gram-Schmidt procedure; or SPGMR_QRSOL_FAIL, indicating that the matrix $R$ was found to be singular during the QR solve phase.

If the KINSPBCG solve function failed (KINSol returned KIN_LSOLVE_FAIL), lsflag contains the error return flag from SpbcgSolve and will be one of: SPBCG_MEM_NULL, indicating that the SPBCG memory is NULL; SPBCG_ATIMES_FAIL_UNREC, indicating an unrecoverable failure in the Jacobian-times-vector function; or SPBCG_PSOLVE_FAIL_UNREC, indicating that the preconditioner solve function psolve failed unrecoverably.

If the KINSPTFQMR solve function failed (KINSol returned KIN_LSOLVE_FAIL), lsflag contains the error return flag from SptfqmrSolve and will be one of: SPTFQMR_MEM_NULL, indicating that the SPTFQMR memory is NULL; SPTFQMR_ATIMES_FAIL_UNREC, indicating an unrecoverable failure in the Jacobian-times-vector function; or SPTFQMR_PSOLVE_FAIL_UNREC, indicating that the preconditioner solve function psolve failed unrecoverably.

## 5.6 User-supplied functions

The user-supplied functions consist of one function defining the nonlinear system, (optionally) a function that handles error and warning messages, (optionally) a function that handles informational messages, (optionally) a function that provides Jacobian-related information for the linear solver, and (optionally) one or two functions that define the preconditioner for use in any of the Krylov iterative algorithms.

### 5.6.1 Problem-defining function

The user must provide a function of type KINSysFn defined as follows:

KINSysFn

Definition `typedef void (*KINSysFn)(N_Vector u, N_Vector fval, void *f_data);`

Purpose This function computes $F(u)$ for a given value of the vector $u$.

Arguments
- u is the current value of the variable vector, $u$.
- fval is the output vector $F(u)$.
- f_data is a pointer to user data, same as the pointer f_data passed to KINSetFdata.

Return value A KINSysFn function should return 0 if successful, a positive value if a recoverable error occurred (in which case KINSOL will attempt to correct), or a negative value if it failed unrecoverably (in which case the solution process is halted and KIN_SYSFUNC_FAIL is returned).

Notes Allocation of memory for fval is handled within KINSOL.

### 5.6.2 Error message handler function

As an alternative to the default behavior of directing error and warning messages to the file pointed to by errfp (see KINSetErrFile), the user may provide a function of type KINErrHandlerFn to process any such messages. The function type KINErrHandlerFn is defined as follows:

`KINErrHandlerFn`

**Definition**
```
typedef void (*KINErrHandlerFn)(int error_code,
                                const char *module, const char *function,
                                char *msg, void *eh_data);
```

**Purpose** This function processes error and warning messages from KINSOL and its sub-modules.

**Arguments**
- `error_code` is the error code.
- `module` is the name of the KINSOL module reporting the error.
- `function` is the name of the function in which the error occurred.
- `msg` is the error message.
- `eh_data` is a pointer to user data, the same as the `eh_data` parameter passed to `KINSetErrHandlerFn`.

**Return value** A `KINErrHandlerFn` function has no return value.

**Notes** `error_code` is negative for errors and positive (`KIN_WARNING`) for warnings. If a function returning a pointer to memory (*e.g.*, `KINBBDPrecAlloc`) encounters an error, it sets `error_code` to 0 before returning `NULL`.

### 5.6.3 Informational message handler function

As an alternative to the default behavior of directing informational (meaning non-error) messages to the file pointed to by `infofp` (see `KINSetInfoFile`), the user may provide a function of type `KINInfoHandlerFn` to process any such messages. The function type `KINInfoHandlerFn` is defined as follows:

`KINInfoHandlerFn`

**Definition**
```
typedef void (*KINInfoHandlerFn)(const char *module, const char *function,
                                 char *msg, void *ih_data);
```

**Purpose** This function processes informational messages from KINSOL and its sub-modules.

**Arguments**
- `module` is the name of the KINSOL module reporting the information.
- `function` is the name of the function reporting the information.
- `msg` is the message.
- `ih_data` is a pointer to user data, the same as the `ih_data` parameter passed to `KINSetInfoHandlerFn`.

**Return value** A `KINInfoHandlerFn` function has no return value.

### 5.6.4 Jacobian information (direct method with dense Jacobian)

If the direct linear solver with dense treatment of the Jacobian is used (i.e. `KINDense` is called in Step 7 of §5.4), the user may provide a function of type `KINDenseJacFn` defined by

`KINDenseJacFn`

**Definition**
```
typedef int (*KINDenseJacFn)(long int N, DenseMat J,
                             N_Vector u, N_Vector fu, void *jac_data,
                             N_Vector tmp1, N_Vector tmp2);
```

**Purpose** This function computes the dense Jacobian $J(u)$ (or an approximation to it).

**Arguments**
- `N` is the problem size.
- `J` is the output Jacobian matrix.
- `u` is the current (unscaled) iterate.
- `fu` is the current value of the vector $F(u)$.

jac_data is a pointer to user data — the same as the jac_data parameter passed to KINDenseSetJacData.

tmp1

tmp2 are pointers to memory allocated for variables of type N_Vector which can be used by KINDenseJacFn as temporary storage or work space.

Return value A function of type KINDenseJacFn should return 0 if successful or a non-zero value otherwise.

Notes A user-supplied dense Jacobian function must load the N by N dense matrix J with an approximation to the Jacobian matrix $J$ at u. Only nonzero elements need to be loaded into J because J is set to the zero matrix before the call to the Jacobian function. The type of J is DenseMat.

The accessor macros DENSE_ELEM and DENSE_COL allow the user to read and write dense matrix elements without making explicit references to the underlying representation of the DenseMat type. DENSE_ELEM(J, i, j) references the (i, j)-th element of the dense matrix J (i, j= $0 \dots N-1$). This macro is for use in small problems in which efficiency of access is not a major concern. Thus, in terms of indices $m$ and $n$ running from 1 to $N$, the Jacobian element $J_{m,n}$ can be loaded with the statement DENSE_ELEM(J, m-1, n-1) = $J_{m,n}$. Alternatively, DENSE_COL(J, j) returns a pointer to the storage for the jth column of J (j= $0 \dots N-1$), and the elements of the jth column are then accessed via ordinary array indexing. Thus $J_{m,n}$ can be loaded with the statements col_n = DENSE_COL(J, n-1); col_n[m-1] = $J_{m,n}$. For large problems, it is more efficient to use DENSE_COL than to use DENSE_ELEM. Note that both of these macros number rows and columns starting from 0, not 1.

The DenseMat type and the accessor macros DENSE_ELEM and DENSE_COL are documented in §9.1.

If the user's KINDenseJacFn function uses difference quotient approximations, it may need to access quantities not in the call list. To obtain these, use the KINGet* functions described in §5.5.5.1. The unit roundoff can be accessed as UNIT_ROUNDOFF defined in sundials_types.h.

### 5.6.5 Jacobian information (direct method with banded Jacobian)

If the direct linear solver with banded treatment of the Jacobian is used (*i.e.*, KINBand is called in Step 7 of §5.4), the user may provide a function of type KINBandJacFn defined as follows:

KINBandJacFn

Definition
```
typedef int (*KINBandJacFn)(long int N, long int mupper,
                            long int mlower, BandMat J,
                            N_Vector u, N_Vector fu, void *jac_data,
                            N_Vector tmp1, N_Vector tmp2);
```

Purpose This function computes the banded Jacobian $J(u)$ (or a banded approximation to it).

Arguments N is the problem size.

mlower

mupper are the lower and upper half-bandwidths of the Jacobian.

J is the output Jacobian matrix.

u is the current (unscaled) iterate.

fu is the current value of the vector $F(u)$.

jac_data is a pointer to user data — the same as the jac_data parameter passed to KINBandSetJacData.

tmp1

`tmp2` are pointers to memory allocated for variables of type `N_Vector` which can be used by `KINBandJacFn` as temporary storage or work space.

Return value A function of type `KINBandJacFn` should return 0 if successful or a non-zero value otherwise.

Notes A user-supplied band Jacobian function must load the band matrix `J` of type `BandMat` with the elements of the Jacobian $J$ at `u`. Only nonzero elements need to be loaded into `J` because `J` is preset to zero before the call to the Jacobian function.

The accessor macros `BAND_ELEM`, `BAND_COL`, and `BAND_COL_ELEM` allow the user to read and write band matrix elements without making specific references to the underlying representation of the `BandMat` type. `BAND_ELEM(J, i, j)` references the (`i`, `j`)th element of the band matrix `J`, counting from 0. This macro is for use in small problems in which efficiency of access is not a major concern. Thus, in terms of indices $m$ and $n$ running from 1 to $N$ with $(m, n)$ within the band defined by `mupper` and `mlower`, the Jacobian element $J_{m,n}$ can be loaded with the statement `BAND_ELEM(J, m-1, n-1)` = $J_{m,n}$. The elements within the band are those with `-mupper` $\leq$ `m-n` $\leq$ `mlower`. Alternatively, `BAND_COL(J, j)` returns a pointer to the diagonal element of the `j`th column of `J`, and if we assign this address to `realtype *col_j`, then the `i`th element of the `j`th column is given by `BAND_COL_ELEM(col_j, i, j)`, counting from 0. Thus for $(m, n)$ within the band, $J_{m,n}$ can be loaded by setting `col_n = BAND_COL(J, n-1); BAND_COL_ELEM(col_n, m-1, n-1)` = $J_{m,n}$. The elements of the `j`th column can also be accessed via ordinary array indexing, but this approach requires knowledge of the underlying storage for a band matrix of type `BandMat`. The array `col_n` can be indexed from $-$`mupper` to `mlower`. For large problems, it is more efficient to use the combination of `BAND_COL` and `BAND_COL_ELEM` than to use the `BAND_ELEM`. As in the dense case, these macros all number rows and columns starting from 0, not 1.

The `BandMat` type and the accessor macros `BAND_ELEM`, `BAND_COL`, and `BAND_COL_ELEM` are documented in §9.2.

If the user's `KINBandJacFn` function uses difference quotient approximations, it may need to access quantities not in the call list. To obtain these, use the `KINGet*` functions described in §5.5.5.1. The unit roundoff can be accessed as `UNIT_ROUNDOFF` defined in `sundials_types.h`.

### 5.6.6 Jacobian information (matrix-vector product)

If one of the Krylov iterative linear solvers SPGMR, SPBCG, or SPTFQMR is selected (`KINSp*` is called in step 7 of §5.4), the user may provide a function of type `KINSpilsJacTimesVecFn` in the following form:

`KINSpilsJacTimesVecFn`

Definition
```
typedef int (*KINSpilsJacTimesVecFn)(N_Vector v, N_Vector Jv,
                                     N_Vector u, booleantype new_u,
                                     void *jac_data);
```

Purpose This function computes the product $Jv$ (or an approximation to it).

Arguments
- `v` is the vector by which the Jacobian must be multiplied to the right.
- `Jv` is the output vector computed.
- `u` is the current value of the dependent variable vector.
- `new_u` is a flag (reset by user) indicating if the iterate has been updated in the interim — Jacobian needs to be updated/reevaluated, if appropriate, unless `new_u = FALSE`.
- `jac_data` is a pointer to user data — the same as the `jac_data` parameter passed to `KINSp*SetJacTimesVecFn`.

Return value The value to be returned by the Jacobian-times-vector function should be 0 if successful. If a recoverable failure occured, the return value should be positive. In this case, KINSOL will attempt to correct by calling the preconditioner setup function if the preconditioner information. If this information is current, KINSOL halts. If the Jacobian-times-vector function encounters an unrecoverable error, it should return a negative value, prompting KINSOL to halt.

Notes If a user-defined routine is not given, then an internal KINSPGMR function, using difference quotient approximations, is used.

If the user-provided `KINSpilsJacTimesVecFn` function needs the unit roundoff, this can be accessed as `UNIT_ROUNDOFF` defined in `sundials_types.h`.

The user is responsible for resetting the value of `new_u` to `FALSE`.

!

### 5.6.7 Preconditioning (linear system solution)

If preconditioning is used, then the user must provide a C function to solve the linear system $Pz = r$ where $P$ is the preconditioner matrix. This function must be of type `KINSpilsPrecSolveFn`, defined as follows:

`KINSpilsPrecSolveFn`

Definition
```
typedef int (*KINSpilsPrecSolveFn)(N_Vector u, N_Vector uscale,
                                   N_Vector fval, N_Vector fscale,
                                   N_Vector v, void *prec_data,
                                   N_Vector tmp);
```

Purpose This function solves the preconditioning system $Pz = r$.

Arguments
- `u` is the current (unscaled) value of the iterate.
- `uscale` is a vector containing diagonal elements of the scaling matrix for `u`.
- `fval` is the vector $F(u)$ evaluated at `u`.
- `fscale` is a vector containing diagonal elements of the scaling matrix for `fval`.
- `v` on input, `v` is set to the right-hand side vector of the linear system, `r`. On output, `v` must contain the solution `z` of the linear system $Pz = r$.
- `prec_data` is a pointer to user data - the same as the `prec_data` parameter passed to the function `KINSpilsSetPreconditioner`.
- `tmp` is a pointer to memory allocated for a variable of type `N_Vector` which can be used for work space.

Return value The value to be returned by the preconditioner solve function is a flag indicating whether it was successful. This value should be 0 if successful, positive for a recoverable error, and negative for an unrecoverable error.

Notes If the preconditioner solve function fails recoverably and if the preconditioner information (set by the preconditioner setup function) is out of date, KINSOL attempts to correct by calling the setup function. If the preconditioner data is current, KINSOL halts.

### 5.6.8 Preconditioning (Jacobian data)

If the user's preconditioner requires that any Jacobian-related data be evaluated or preprocessed, then this needs to be done in a user-supplied C function of type `KINSpilsPrecSetupFn`, defined as follows:

`KINSpilsPrecSetupFn`

Definition
```
typedef int (*KINSpilsPrecSetupFn)(N_Vector u, N_Vector uscale,
                                   N_Vector fval, N_Vector fscale,
                                   void *prec_data, N_Vector tmp1,
                                   N_Vector tmp2);
```

Purpose: This function evaluates and/or preprocesses Jacobian-related data needed by the preconditioner.

Arguments: The arguments of a `KINSpilsPrecSetupFn` are as follows:

- `u` is the current (unscaled) value of the iterate.
- `uscale` is a vector containing diagonal elements of the scaling matrix for `u`.
- `fval` is the vector $F(u)$ evaluated at `u`.
- `fscale` is a vector containing diagonal elements of the scaling matrix for `fval`.
- `prec_data` is a pointer to user data - the same as the `prec_data` parameter passed to the function `KINSpilsSetPreconditioner`.
- `tmp1`
- `tmp2` are pointers to memory allocated for variables of type `N_Vector` which can be used by `KINSpilsPrecSetupFn` as temporary storage or work space.

Return value: The value to be returned by the preconditioner setup function is a flag indicating whether it was successful. This value should be 0 if successful, any other value resulting in halting the KINSOL solver.

Notes: The user-supplied preconditioner setup subroutine should compute the right preconditioner matrix $P$ (stored in the memory block referenced by the `prec_data` pointer) used to form the scaled preconditioned linear system

$$(D_F J(u) P^{-1} D_u^{-1}) \cdot (D_u P x) = -D_F F(u)\,,$$

where $D_u$ and $D_F$ denote the diagonal scaling matrices whose diagonal elements are stored in the vectors `uscale` and `fscale`, respectively.

The preconditioner setup routine will not be called prior to every call made to the preconditioner solve function, but will instead be called only as often as necessary to achieve convergence of the Newton iteration.

If the preconditioner solve routine requires no preparation, then a preconditioner setup function need not be given.

## 5.7 A parallel band-block-diagonal preconditioner module

The efficiency of Krylov iterative methods for the solution of linear systems can be greatly enhanced through preconditioning. For problems in which the user cannot define a more effective, problem-specific preconditioner, KINSOL provides a band-block-diagonal preconditioner module KINBBDPRE, to be used with the parallel `N_Vector` module described in §7.2.

This module provides a preconditioner matrix for KINSOL that is block-diagonal with banded blocks. The blocking corresponds to the distribution of the dependent variable vector $u$ amongst the processes. Each preconditioner block is generated from the Jacobian of the local part (associated with the current process) of a given function $G(u)$ approximating $F(u)$ ($G = F$ is allowed). The blocks are generated by each process via a difference quotient scheme, utilizing a specified band structure. This structure is given by upper and lower half-bandwidths, `mudq` and `mldq`, defined as the number of non-zero diagonals above and below the main diagonal, respectively. However, from the resulting approximate Jacobain blocks, only a matrix of bandwidth `mu` + `ml` +1 is retained.

Neither pair of parameters need be the true half-bandwidths of the Jacobian of the local block of $G$, if smaller values provide a more efficient preconditioner. Such an efficiency gain may occur if the couplings in the system outside a certain bandwidth are considerably weaker than those within the band. Reducing mu and ml while keeping mudq and mldq at their true values, discards the elements outside the narrower band. Reducing both pairs has the additional effect of lumping the outer Jacobian elements into the computed elements within the band, and requires more caution and experimentation to see whether the lower cost of narrower band matrices offsets the loss of accuracy in the blocks.

The KINBBDPRE module calls two user-provided functions to construct $P$: a required function Gloc (of type KINLocalFn) which approximates the nonlinear system function function $G(u) \approx F(u)$ and which is computed locally, and an optional function Gcomm (of type KINCommFn) which performs all interprocess communication necessary to evaluate the approximate function $G$. These are in addition to the user-supplied nonlinear system function that evaluates $F(u)$. Both functions take as input the same pointer f_data as that passed by the user to KINSetFdata and passed to the user's function func, and neither function has a return value. The user is responsible for providing space (presumably within f_data) for components of u that are communicated by Gcomm from the other processes, and that are then used by Gloc, which is not expected to do any communication.

**KINLocalFn**

Definition
```
typedef void (*KINLocalFn)(long int Nlocal, N_Vector u,
                           N_Vector gval, void *f_data);
```

Purpose: This function computes $G(\mathtt{u})$, and outputs the resulting vector as gval.

Arguments:
- Nlocal is the local vector length.
- u is the current value of the iterate.
- gval is the output vector.
- f_data is a pointer to user data - the same as the f_data parameter passed to KINSetFdata.

Return value: A KINLocalFn function type does not have a return value.

Notes: This function assumes that all interprocess communication of data needed to calculate gval has already been done, and this data is accessible within f_data.

Memory for u and gval is handled within the preconditioner module.

The case where $G$ is mathematically identical to $F$ is allowed.

**KINCommFn**

Definition
```
typedef void (*KINCommFn)(long int Nlocal, N_Vector u, void *f_data);
```

Purpose: This function performs all interprocess communications necessary for the execution of the gloc function above, using the input vector u.

Arguments:
- Nlocal is the local vector length.
- u is the current value of the iterate.
- f_data is a pointer to user data - the same as the f_data parameter passed to KINSetFdata.

Return value: A KINCommFn function type does not have a return value.

Notes: The Gcomm function is expected to save communicated data in space defined within the structure f_data.

Each call to the Gcomm function is preceded by a call to the system function func with the same u argument. Thus Gcomm can omit any communications done by func if relevant to the evaluation of Gloc. If all necessary communication was done in func, then Gcomm = NULL can be passed in the call to KINBBDPrecAlloc (see below).

Besides the header files required for the solution of a nonlinear problem (see §5.3), to use the KINBBDPRE module, the main program must include the header file kinbbdpre.h which declares the needed function prototypes.

The following is a summary of the usage of this module and describes the sequence of calls in the user main program. Steps that are unchanged from the user main program presented in §5.4 are grayed out.

1. **Initialize MPI**
2. **Set problem dimensions**
3. **Set vector with initial guess**
4. **Create KINSOL object**
5. **Set optional inputs**
6. **Allocate internal memory**
7. **Initialize the KINBBDPRE preconditioner module**

   Specify the upper and lower half-bandwidth pairs `mu`,`ml` and `mukeep`,`mlkeep`, and call

   ```
   bbd_data = KINBBDPrecAlloc(kin_mem, Nlocal, mudq, mldq,
                              mukeep, mlkeep, dq_rel_u, Gloc, Gcomm);
   ```

   to allocate memory for and initialize a data structure `bbd_data` to be passed to the appropriate KINSPILS linear solver. The last two arguments of `KINBBDPrecAlloc` are the two user-supplied functions described above.

8. **Attach the Kyrlov linear solver, one of:**

   ```
   flag = KINBBDSpgmr(kin_mem, maxl, bbd_data);
   flag = KINBBDSpbcg(kin_mem, maxl, bbd_data);
   flag = KINBBDSptfqmr(kin_mem, maxl, bbd_data);
   ```

   Each function `KINBBDSp*` is a wrapper around the corresponding specification function `KINSp*` and performs the following actions:

   - Attaches the KINSPILS linear solver to the main KINSOL solver memory;
   - Sets the preconditioner data structure for KINBBDPRE;
   - Sets the preconditioner setup function for KINBBDPRE;
   - Sets the preconditioner solve function for KINBBDPRE;

   The argument `maxl` is described below. The last argument of `KINBBDSp*` is the pointer to the KINBBDPRE data returned by `KINBBDPrecAlloc`.

9. **Set linear solver optional inputs**

   Note that the user should not overwrite the preconditioner data, setup function, or solve function through calls to KINSPILS optional input functions.

10. **Solve problem**
11. **Get optional output**
12. **Deallocate memory for solution vector**
13. **Free the KINBBDPRE data structure**

    ```
    KINBBDPrecFree(&bbd_data);
    ```

14. **Free solver memory**
15. **Finalize MPI**

The user-callable functions that initialize, attach, and deallocate the KINBBDPRE preconditioner module (steps 7, 8, and 13 above) are described in more detail below.

KINBBDPrecAlloc

Call `bbd_data = KINBBDPrecAlloc(kin_mem, Nlocal, mudq, mldq, mukeep, mlkeep, dq_rel_u, Gloc, Gcomm);`

Description The function `KINBBDPrecAlloc` initializes and allocates memory for the KINBBDPRE preconditioner.

Arguments
- `kin_mem` (`void *`) pointer to the KINSOL memory block.
- `Nlocal` (`long int`) local vector length.
- `mudq` (`long int`) upper half-bandwidth to be used in the difference-quotient Jacobian approximation.
- `mldq` (`long int`) lower half-bandwidth to be used in the difference-quotient Jacobian approximation.
- `mukeep` (`long int`) upper half-bandwidth of the retained banded approximate Jacobian block.
- `mlkeep` (`long int`) lower half-bandwidth of the retained banded approximate Jacobian block.
- `dq_rel_u` (`realtype`) the relative increment in components of `u` used in the difference quotient approximations. The default is `dq_rel_u`= $\sqrt{\text{unit roundoff}}$, which can be specified by passing `dq_rel_u`= 0.0.
- `Gloc` (`KINLocalFn`) the C function which computes the approximation $G(u) \approx F(u)$.
- `Gcomm` (`KINCommFn`) the optional C function which performs all interprocess communication required for the computation of $G(u)$.

Return value If successful, `KINBBDPrecAlloc` returns a pointer to the newly created KINBBDPRE memory block (of type `void *`). If an error occurred, `KINBBDPrecAlloc` returns `NULL`.

Notes If one of the half-bandwidths `mudq` or `mldq` to be used in the difference-quotient calculation of the approximate Jacobian is negative or exceeds the value `Nlocal`$-1$, it is replaced with 0 or `Nlocal`$-1$ accordingly.

The half-bandwidths `mudq` and `mldq` need not be the true half-bandwidths of the Jacobian of the local block of $G$, when smaller values may provide greater efficiency.

Also, the half-bandwidths `mukeep` and `mlkeep` of the retained banded approximate Jacobian block may be even smaller, to reduce storage and computation costs further.

For all four half-bandwidths, the values need not be the same for every process.

KINBBDSpgmr

Call `flag = KINBBDSpgmr(kin_mem, maxl, bbd_data);`

Description The function `KINBBDSpgmr` links the KINBBDPRE data to the KINSPGMR linear solver and attaches the latter to the KINSOL memory block.

Arguments
- `kin_mem` (`void *`) pointer to the KINSOL memory block.
- `maxl` (`int`) maximum dimension of the Krylov subspace to be used. Pass 0 to use the default value `KINSPILS_MAXL`= 5.
- `bbd_data` (`void *`) pointer to the KINBBDPRE data structure.

Return value The return value `flag` (of type `int`) is one of:

| | |
|---|---|
| `KINSPILS_SUCCESS` | The KINSPGMR initialization was successful. |
| `KINSPILS_MEM_NULL` | The `kin_mem` pointer is `NULL`. |
| `KINSPILS_ILL_INPUT` | The NVECTOR module used does not implement a required operation. |
| `KINSPILS_MEM_FAIL` | A memory allocation request failed. |
| `KINBBDPRE_PDATA_NULL` | The KINBBDPRE preconditioner has not been initialized. |

`KINBBDSpbcg`

Call `flag = KINBBDSpbcg(kin_mem, maxl, bbd_data);`

Description The function `KINBBDSpbcg` links the KINBBDPRE data to the KINSPBCG linear solver and attaches the latter to the KINSOL memory block.

Arguments
`kin_mem` (`void *`) pointer to the KINSOL memory block.
`maxl` (`int`) maximum dimension of the Krylov subspace to be used. Pass 0 to use the default value `KINSPILS_MAXL`= 5.
`bbd_data` (`void *`) pointer to the KINBBDPRE data structure.

Return value The return value `flag` (of type `int`) is one of:

| | |
|---|---|
| `KINSPILS_SUCCESS` | The KINSPGMR initialization was successful. |
| `KINSPILS_MEM_NULL` | The `kin_mem` pointer is `NULL`. |
| `KINSPILS_ILL_INPUT` | The NVECTOR module used does not implement a required operation. |
| `KINSPILS_MEM_FAIL` | A memory allocation request failed. |
| `KINBBDPRE_PDATA_NULL` | The KINBBDPRE preconditioner has not been initialized. |

`KINBBDSptfqmr`

Call `flag = KINBBDSptfqmr(kin_mem, maxl, bbd_data);`

Description The function `KINBBDSptfqmr` links the KINBBDPRE data to the KINSPTFQMR linear solver and attaches the latter to the KINSOL memory block.

Arguments
`kin_mem` (`void *`) pointer to the KINSOL memory block.
`maxl` (`int`) maximum dimension of the Krylov subspace to be used. Pass 0 to use the default value `KINSPILS_MAXL`= 5.
`bbd_data` (`void *`) pointer to the KINBBDPRE data structure.

Return value The return value `flag` (of type `int`) is one of:

| | |
|---|---|
| `KINSPILS_SUCCESS` | The KINSPGMR initialization was successful. |
| `KINSPILS_MEM_NULL` | The `kin_mem` pointer is `NULL`. |
| `KINSPILS_ILL_INPUT` | The NVECTOR module used does not implement a required operation. |
| `KINSPILS_MEM_FAIL` | A memory allocation request failed. |
| `KINBBDPRE_PDATA_NULL` | The KINBBDPRE preconditioner has not been initialized. |

`KINBBDPrecFree`

Call `KINBBDPrecFree(&bbd_data);`

Description The function `KINBBDPrecFree` frees the pointer allocated by `KINBBDPrecAlloc`.

Arguments The only argument of `KINBBDPrecFree` is the pointer to the KINBBDPRE data structure (of type `void *`).

Return value The function `KINBBDPrecFree` has no return value.

The following two optional output functions are available for use with the KINBBDPRE module:

`KINBBDPrecGetWorkSpace`

Call `flag = KINBBDPrecGetWorkSpace(bbd_data, &lenrwBBDP, &leniwBBDP);`

Description The function `KINBBDPrecGetWorkSpace` returns the local KINBBDPRE real and integer workspace sizes.

Arguments `bbd_data` (`void *`) pointer to the KINBBDPRE data structure.

lenrwBBDP (long int) local number of realtype values in the KINBBDPRE workspace.

leniwBBDP (long int) local number of integer values in the KINBBDPRE workspace.

Return value The return value flag (of type int) is one of:

KINBBDPRE_SUCCESS The optional output values have been successfully set.

KINBBDPRE_PDATA_NULL The KINBBDPRE preconditioner has not been initialized.

Notes In terms of the local vector dimension Nlocal and smu $=\min(N_l - 1,$ mukeep $+$ mlkeep$)$, the actual size of the real workspace is $(2$ mlkeep $+$ mukeep $+$ smu $+2)$ Nlocal realtype words, and the actual size of the integer workspace is Nlocal integer words. These values are local to the current processor.

The workspaces referred to here exist in addition to those given by the corresponding KINSp*GetWorkSpace function.

KINBBDPrecGetNumGfnEvals

Call `flag = KINBBDPrecGetNumGfnEvals(bbd_data, &ngevalsBBDP);`

Description The function KINBBDPrecGetNumGfnEvals returns the number of calls to the user Gloc function due to the finite difference approximation of the Jacobian blocks used within KINBBDPRE's preconditioner setup function.

Arguments bbd_data (void *) pointer to the KINBBDPRE data structure.

ngevalsBBDP (long int) the number of calls to the user Gloc function.

Return value The return value flag (of type int) is one of:

KINBBDPRE_SUCCESS The optional output value has been successfully set.

KINBBDPRE_PDATA_NULL The KINBBDPRE preconditioner has not been initialized.

KINBBDPrecGetReturnFlagName

Call `name = KINBBDPrecGetReturnFlagName(flag);`

Description The function KINBBDPrecGetReturnFlagName returns the name of the KINBBDPRE constant corresponding to flag.

Arguments The only argument, of type int is a return flag from a KINBBDPRE function.

Return value The return value is a string containing the name of the corresponding constant.

In addition to the ngevalsBBDP Gloc evaluations, the costs associated with KINBBDPRE also include nlinsetups LU factorizations, nlinsetups calls to Gcomm, npsolves banded backsolve calls, and nfevalsLS right-hand side function evaluations, where nlinsetups is an optional KINSOL output and npsolves and nfevalsLS are linear solver optional outputs (see §5.5.5).

# Chapter 6

# FKINSOL, an Interface Module for FORTRAN Applications

The FKINSOL interface module is a package of C functions which support the use of the KINSOL solver, for the solution nonlinear systems $F(u) = 0$, in a mixed FORTRAN/C setting. While KINSOL is written in C, it is assumed here that the user's calling program and user-supplied problem-defining routines are written in FORTRAN. This package provides the necessary interface to KINSOL for both the serial and the parallel NVECTOR implementations.

## 6.1 Important note on portability

In this package, the names of the interface functions, and the names of the FORTRAN user routines called by them, appear as dummy names which are mapped to actual values by a series of definitions in the header files `fkinsol.h`and `fkinbbd.h`. By default, those mapping definitions depend in turn on the C macro `F77_FUNC` defined in the header file `sundials_config.h` by `configure`. However, the set of flags `SUNDIALS_CASE_UPPER`, `SUNDIALS_CASE_LOWER`, `SUNDIALS_UNDERSCORE_NONE`, `SUNDIALS_UNDERSCORE_ONE`, and `SUNDIALS_UNDERSCORE_TWO` can be explicitly defined in the header file `sundials_config.h` when configuring SUNDIALS via the options `--with-f77underscore` and `--with-f77case` to override the default behavior if necessary (see Chapter 2). Either way, the names into which the dummy names are mapped are in upper or lower case and have up to two underscores appended.

The user must also ensure that variables in the user FORTRAN code are declared in a manner consistent with their counterparts in KINSOL. All real variables must be declared as `REAL`, `DOUBLE PRECISION`, or perhaps as `REAL*`$n$, where $n$ denotes the number of bytes, depending on whether KINSOL was built in single, double or extended precision (see Chapter 2). Moreover, some of the FORTRAN integer variables must be declared as `INTEGER*4` or `INTEGER*8` according to the C type `long int`. These integer variables include: the array of integer optional inputs and outputs (`IOUT`), problem dimensions (`NEQ`, `NLOCAL`, `NGLOBAL`), and Jacobian half-bandwidths (`MU` and `ML`). This is particularly important when using KINSOL and the FKINSOL package on 64-bit architectures.

## 6.2 FKINSOL routines

The user-callable functions, with the corresponding KINSOL functions, are as follows:

- Interface to the NVECTOR modules
  - `FNVINITS` (defined by NVECTOR_SERIAL) interfaces to `N_VNewEmpty_Serial`.
  - `FNVINITP` (defined by NVECTOR_PARALLEL) interfaces to `N_VNewEmpty_Parallel`.
- Interface to the main KINSOL module

- FKINMALLOC interfaces to KINCreate, KINSetFdata, and KINMalloc.
- FKINSETIIN and FKINSETRIN interface to KINSet* functions.
- FKINSETVIN interfaces to KINSetConstraints.
- FKINSOL interfaces to KINSol, KINGet* functions, and to the optional output functions for the selected linear solver module.
- FKINFREE interfaces to KINFree.

• Interface to the linear solver modules

- FKINDENSE interfaces to KINDense.
- FKINDENSESETJAC interfaces to KINDenseSetJacFn.
- FKINBAND interfaces to KINBand.
- FKINBANDSETJAC interfaces to KINBandSetJacFn.
- FKINSPGMR interfaces to KINSpgmr and SPGMR optional input functions.
- FKINSPBCG interfaces to KINSpbcg and SPBCG optional input functions.
- FKINSPTFQMR interfaces to KINSptfqmr and SPTFQMR optional input functions.
- FKINSPILSSETJAC interfaces to KINSpilsSetJacTimesVecFn.
- FKINSPILSSETPREC interfaces to KINSpilsSetPreconditioner.

The user-supplied functions, each listed with the corresponding interface function which calls it (and its type within KINSOL), are as follows:

| FKINSOL routine (FORTRAN) | KINSOL function (C) | KINSOL function type |
|---|---|---|
| FKFUN | FKINfunc | KINSysFn |
| FKDJAC | FKINDenseJac | KINDenseJacFn |
| FKBJAC | FKINBandJac | KINBandJacFn |
| FKPSET | FKINPSet | KINSpilsPrecSetupFn |
| FKPSOL | FKINPSol | KINSpilsPrecSolveFn |
| FKJTIMES | FKINJtimes | KINSpilsJacTimesVecFn |

In contrast to the case of direct use of KINSOL, the names of all user-supplied routines here are fixed, in order to maximize portability for the resulting mixed-language program.

## 6.3 Usage of the FKINSOL interface module

The usage of FKINSOL requires calls to a few different interface functions, depending on the method options selected, and one or more user-supplied routines which define the problem to be solved. These function calls and user routines are summarized separately below. Some details are omitted, and the user is referred to the description of the corresponding KINSOL functions for information on the arguments of any given user-callable interface routine, or of a given user-supplied function called by an interface function.

Steps marked with [**S**] in the instructions below apply to the serial NVECTOR implementation (NVECTOR_SERIAL) only, while those marked with [**P**] apply to NVECTOR_PARALLEL.

1. **Nonlinear system function specification**

   The user must in all cases supply the following FORTRAN routine

```
SUBROUTINE FKFUN (U, FVAL, IER)
DIMENSION U(*), FVAL(*)
```

It must set the FVAL array to $F(u)$, the system function, as a function of U$= u$. IER is an error return flag which should be set to 0 if successful, a positive value if a recoverable error occurred (in which case KINSOL will attempt to corret), or a negative value if it failed unrecoverably (in which case the the solution process is halted).

2. NVECTOR **module initialization**

   [**S**] To initialize the serial NVECTOR module, the user must make the following call:

   ```
   CALL FNVINITS (KEY, NEQ, IER)
   ```

   where KEY is the solver id (KEY $= 3$ for KINSOL), NEQ is the size of vectors, and IER is a return completion flag which is 0 on success and $-1$ if a failure occurred.

   [**P**] To initialize the parallel vector module, the user must make the following call:

   ```
   CALL FNVINITP (COMM, KEY, NLOCAL, NGLOBAL, IER)
   ```

   in which the arguments are: COMM = MPI communicator, KEY = 3, NLOCAL = the local size of vectors on this processor, and NGLOBAL = the system size (and the global size of all vectors, equal to the sum of all values of NLOCAL). The return flag IER is set to 0 on a successful return and to $-1$ otherwise.

   If the header file sundials_config.h defines SUNDIALS_MPI_COMM_F2C to be 1 (meaning the MPI implementation used to build SUNDIALS includes the MPI_Comm_f2c function), then COMM can be any valid MPI communicator. Otherwise, MPI_COMM_WORLD will be used, so just pass an integer value as a placeholder. !

3. **Problem specification**

   To set various problem and solution parameters and allocate internal memory, make the following call:

   $\boxed{\texttt{FKINMALLOC}}$

   | | |
   |---|---|
   | Call | CALL FKINMALLOC (IOUT, ROUT, IER) |
   | Description | This function specifies the optional output arrays, allocates internal memory, and initializes KINSOL. |
   | Arguments | IOUT is an integer array for integer optional outputs.<br>ROUT is a real array for real optional outputs. |
   | Return value | IER is the return completion flag. Values are 0 for successful return and $-1$ otherwise. See printed message for details in case of failure. |
   | Notes | The user integer data array IOUT must be declared as INTEGER*4 or INTEGER*8 according to the C type long int.<br>The optional outputs associated with the main KINSOL integrator are listed in Table 6.2. |

4. **Linear solver specification**

   The solution method in KINSOL involves the solution of linear systems related to the Jacobian of the nonlinear system. KINSOL presently includes five choices for the treatment of these systems, and the user of FKINSOL must call a routine with a specific name to make the desired choice.

   [**S**] **Dense treatment of the linear system**

   The user must make the call:

   ```
   CALL FKINDENSE (NEQ, IER)
   ```

where NEQ is the size of the nonlinear system. The argument IER is an error return flag which is 0 for success , $-1$ if a memory allocation failure occurred, or $-2$ for illegal input. As an option when using the DENSE linear solver, the user may supply a routine that computes a dense approximation of the system Jacobian $J = \partial F/\partial u$. If supplied, it must have the following form:

```
 SUBROUTINE FKDJAC (NEQ, U, FVAL, DJAC,
&                   WK1, WK2, IER)
 DIMENSION U(*), FVAL(*), DJAC(NEQ,*),
&          WK1(*), WK2(*)
```

Typically this routine will use only NEQ, U, and DJAC. It must compute the Jacobian and store it columnwise in DJAC. The input arguments U and FVAL contain the current values of $u$, and $F(u)$, respectively. The vectors WK1 and WK2 of length NEQ are provided as work space for use in FKDJAC. IER is an error return flag which should be set to 0 if successful, a positive value if a recoverable error occurred (in which case KINSOL will attempt to correct), or a negative value if FKDJAC failed unrecoverably (in which case the solution process is halted).

If the FKDJAC routine is provided, then, following the call to FKINDENSE, the user must make the call:

```
CALL FKINDENSESETJAC (FLAG, IER)
```

with FLAG $\neq 0$ to specify use of the user-supplied Jacobian approximation. The argument IER is an error return flag which is 0 for success or non-zero if an error occurred.

Optional outputs specific to the DENSE case are listed in Table 6.2.

**[S] Band treatment of the linear system**

The user must make the call:

```
CALL FKINBAND (NEQ, MU, ML, IER)
```

The arguments are: MU, the upper half-bandwidth; ML, the lower half-bandwidth; and IER an error return flag which is 0 for success , $-1$ if a memory allocation failure occurred, or $-2$ in case an input has an illegal value.

As an option when using the BAND linear solver, the user may supply a routine that computes a band approximation of the system Jacobian $J = \partial F/\partial u$. If supplied, it must have the following form:

```
 SUBROUTINE FKBJAC (NEQ, MU, ML, MDIM, U, FVAL, BJAC,
&                   WK1, WK2, IER)
 DIMENSION U(*), FVAL(*), BJAC(MDIM,*),
&          WK1(*), WK2(*)
```

Typically this routine will use only NEQ, MU, ML, U, and BJAC. It must load the MDIM by N array BJAC with the Jacobian matrix at the current $u$ in band form. Store in BJAC($k, j$) the Jacobian element $J_{i,j}$ with $k = i-j+$ MU $+1$ ($k = 1 \cdots$ ML + MU + 1) and $j = 1 \cdots N$. The input arguments U, and FVAL contain the current values of $u$, and $F(u)$, respectively. The vectors WK1 and WK2 of length NEQ are provided as work space for use in FKBJAC. IER is an error return flag which should be set to 0 if successful, a positive value if a recoverable error occurred (in which case KINSOL will attempt to correct), or a negative value if FKBJAC failed unrecoverably (in which case the solution process is halted).

If the FKBJAC routine is provided, then, following the call to FKINBAND, the user must make the call:

```
CALL FKINBANDSETJAC (FLAG, IER)
```

with FLAG $\neq 0$ to specify use of the user-supplied Jacobian approximation. The argument IER is an error return flag which is 0 for success or non-zero if an error occurred.

Optional outputs specific to the BAND case are listed in Table 6.2.

**[S][P] SPGMR treatment of the linear systems**

For the Scaled Preconditioned GMRES solution of the linear systems, the user must make the call

```
CALL FKINSPGMR (MAXL, MAXLRST, IER)
```

The arguments are as follows. MAXL is the maximum Krylov subspace dimension. MAXLRST is the maximum number of restarts. IER is an error return flag which is 0 to indicate success, $-1$ if a memory allocation failure occurred, or $-2$ to indicate an illegal input.

Optional outputs specific to the SPGMR case are listed in Table 6.2.

For descriptions of the relevant optional user-supplied routines, see **User-supplied routines for SPGMR/SPBCG/SPTFQMR** below.

**[S][P] SPBCG treatment of the linear systems**

For the Scaled Preconditioned Bi-CGStab solution of the linear systems, the user must make the call

```
CALL FKINSPBCG (MAXL, IER)
```

Its arguments are the same as those with the same names for FKINSPGMR.

Optional outputs specific to the SPBCG case are listed in Table 6.2.

For descriptions of the relevant optional user-supplied routines, see **User-supplied routines for SPGMR/SPBCG/SPTFQMR** below.

**[S][P] SPTFQMR treatment of the linear systems**

For the Scaled Preconditioned Transpose-Reee Quasi-Minimal Residual solution of the linear systems, the user must make the call

```
CALL FKINSPTFQMR (MAXL, IER)
```

Its arguments are the same as those with the same names for FKINSPGMR.

Optional outputs specific to the SPTFQMR case are listed in Table 6.2.

For descriptions of the relevant optional user-supplied routines, see below.

**[S][P] Functions used by SPGMR/SPBCG/SPTFQMR**

An optional user-supplied routine, FKINJTIMES (see below), can be provided for Jacobian-vector products. If it is, then, following the call to FKINSPGMR, FKINSPBCG, or FKINSPTFQMR, the user must make the call:

```
CALL FKINSPILSSETJAC (FLAG, IER)
```

with FLAG $\neq 0$ to specify use of the user-supplied Jacobian-times-vector approximation. The argument IER is an error return flag which is 0 for success or non-zero if an error occurred.

If preconditioning is to be done, then the user must call

```
CALL FKINSPILSSETPREC (FLAG, IER)
```

with FLAG $\neq 0$. The return flag IER is 0 if successful, or negative if a memory error occurred. In addition, the user program must include preconditioner routines FKPSOL and FKPSET (see below).

**[S][P] User-supplied routines for SPGMR/SPBCG/SPTFQMR**

With treatment of the linear systems by any of the Krylov iterative solvers, there are three optional user-supplied routines — FKINJTIMES, FKPSOL, and FKPSET. The specifications for these routines are given below.

As an option when using the SPGMR, SPBCG, or SPTFQMR linear solvers, the user may supply a routine that computes the product of the system Jacobian $J = \partial F/\partial u$ and a given vector $v$. If supplied, it must have the following form:

```
SUBROUTINE FKINJTIMES (V, FJV, NEWU, U, IER)
DIMENSION V(*), FJV(*), U(*)
```

Typically this routine will use only NEQ, U, V, and FJV. It must compute the product vector $Jv$, where the vector $v$ is stored in V, and store the product in FJV. The input argument U contains the current value of $u$. On return, set IER = 0 if FKINJTIMES was successful, and nonzero otherwise.

If preconditioning is to be included, the following routine must be supplied, for solution of the preconditioner linear system:

```
SUBROUTINE FKPSOL (U, USCALE, FVAL, FSCALE, VTEM, FTEM, IER)
DIMENSION U(*), USCALE(*), FVAL(*), FSCALE(*), VTEM(*), FTEM(*)
```

Typically this routine will use only U, FVAL, VTEM and FTEM. It must solve the preconditioned linear system $Pz = r$, where $r =$ VTEM is input, and store the solution $z$ in VTEM as well. Here $P$ is the right preconditioner. If scaling is being used, the routine supplied must also account for scaling on either coordinate or function value, as given in the arrays USCALE and FSCALE, respectively.

If the user's preconditioner requires that any Jacobian-related data be evaluated or preprocessed, then the following routine can be used for the evaluation and preprocessing of the preconditioner:

```
SUBROUTINE FKPSET (U, USCALE, FVAL, FSCALE, VTEMP1, VTEMP2, IER)
DIMENSION U(*), USCALE(*), FVAL(*), FSCALE(*), VTEMP1(*), VTEMP2(*)
```

It must perform any evaluation of Jacobian-related data and preprocessing needed for the solution of the preconditioned linear systems by FKPSOL. The variables U through FSCALE are for use in the preconditioning setup process. Typically, the system function FKFUN is called before any calls to FKPSET, so that FVAL will have been updated. U is the current solution iterate. The arrays VTEMP1 and VTEMP2 are available for work space. If scaling is being used, USCALE and FSCALE are available for those operations requiring scaling. NEQ is the problem size.

On return, set IER $= 0$ if FKPSET was successful or set IER $= 1$ if an error occurred.

! If the user calls FKINSPILSSETPREC, the routine FKPSET must be provided, even if it is not needed and must return IER=0.

5. **Problem solution**

Solving the nonlinear system is accomplished by making the following call:

```
CALL FKINSOL (U, GLOBALSTRAT, USCALE, FSCALE, IER)
```

The arguments are as follows. U is an array containing the initial guess on input, and the solution on return. GLOBALSTRAT is an integer (type INTEGER) defining the global strategy choice (1 specifies Inexact Newton, while 2 indicates line search). USCALE is an array of scaling factors for the U vector.

FSCALE is an array of scaling factors for the FVAL vector. IER is an integer completion flag and will have one of the following values: 0 to indicate success, 1 to indicate that the initial guess satisfies $F(u) = 0$ within tolerances, 2 to indicate apparent stalling (small step), or a negative value to indicate an error or failure. These values correspond to the KINSol returns (see §5.5.3 and §10.2). The values of the optional outputs are available in IOPT and ROPT (see Table 6.2).

6. **Memory deallocation**

   To free the internal memory created by the call to FKINMALLOC, make the call

   ```
   CALL FKINFREE
   ```

## 6.4 FKINSOL optional input and output

In order to keep the number of user-callable FKINSOL interface routines to a minimum, optional inputs to the KINSOL solver are passed through only two routines: FKINSETIIN for integer optional inputs and FKINSETRIN for real optional inputs. These functions should be called as follows:

```
CALL FKINSETIIN (KEY, IVAL, IER)
CALL FKINSETRIN (KEY, RVAL, IER)
```

where KEY is a quoted string indicating which optional input is set (see Table 6.1), IVAL is the integer input value to the used, RVAL is the real input value to be used, and IER is an integer return flag which is set to 0 on success and a negative value if a filure occurred.

The optional outputs from the KINSOL solver are accessed not through individual functions, but rather through a pair of arrays, IOUT (integer type) of dimension at least 15, and ROUT (real type) of dimension at least 2. These arrays are owned (and allocated) by the user and are passed as arguments to FKINMALLOC. Table 6.2 lists the entries in these two arrays and specifies the optional variable as well as the KINSOL function which is actually called to extract the optional output.

For more details on the optional inputs and outputs, see §5.5.4 and §5.5.5.

## 6.5 Usage of the FKINBBD interface to KINBBDPRE

The FKINBBD interface sub-module is a package of C functions which, as part of the FKINSOL interface module, support the use of the KINSOL solver with the parallel NVECTOR_PARALLEL module and the KINBBDPRE preconditioner module (see §5.7), for the solution of nonlinear problems in a mixed Fortran/C setting.

The user-callable functions in this package, with the corresponding KINSOL and KINBBDPRE functions, are as follows:

- FKINBBDINIT interfaces to KINBBDPrecAlloc.
- FKINBBDSPGMR interfaces to KINBBDSpgmr and SPGMR optional input functions.
- FKINBBDSPBCG interfaces to KINBBDSpbcg and SPBCG optional input functions.
- FKINBBDSPTFQMR interfaces to KINBBDSptfmqr and SPTFQMR optional input functions.
- FKINBBDOPT interfaces to KINBBDPRE optional output functions.
- FKINBBDFREE interfaces to KINBBDPrecFree.

In addition to the Fortran right-hand side function FKFUN, the user-supplied functions used by this package, are listed below, each with the corresponding interface function which calls it (and its type within KINBBDPRE or KINSOL):

Table 6.1: Keys for setting FKINSOL optional inputs

Integer optional inputs FKINSETIIN

| Key | Optional input | Default value |
|---|---|---|
| PRNT_LEVEL | Verbosity level of output | 0 |
| MAX_NITER | Maximum no. of nonlinear iterations | 200 |
| ETA_FORM | Form of $\eta$ coefficient | 1 (KIN_ETACHOICE1) |
| MAX_SETUPS | Maximum no. of iterations without prec. setup | 10 |
| MAX_SP_SETUPS | Maximum no. of iterations without residual check | 5 |
| NO_INIT_SETUP | No initial preconditioner setup | FALSE |
| NO_MIN_EPS | Lower bound on $\epsilon$ | FALSE |
| NO_RES_MON | No residual monitoring | FALSE |

Real optional inputs (FKINSETRIN)

| Key | Optional input | Default value |
|---|---|---|
| FNORM_TOL | Function-norm stopping tolerance | $\sqrt[3]{\text{uround}}$ |
| SSTEP_TOL | Scaled-step stopping tolerance | $uround^{2/3}$ |
| MAX_STEP | Max. scaled length of Newton step | $1000\|D_u u_0\|_2$ |
| RERR_FUNC | Relative error for F.D. $Jv$ | $\sqrt{\text{uround}}$ |
| ETA_CONST | Constant value of $\eta$ | 0.1 |
| ETA_PARAMS | Values of $\gamma$ and $\alpha$ | 0.9 and 2.0 |
| RMON_CONST | Constant value of $\omega$ | 0.9 |
| RMON_PARAMS | Values of $\omega_{min}$ and $\omega_{max}$ | 0.00001 and 0.9 |

| FKINBBD routine (FORTRAN) | KINSOL function (C) | KINSOL function type |
|---|---|---|
| FKLOCFN | FKINgloc | KINLocalFn |
| FKCOMMF | FKINgcomm | KINCommFn |
| FKJTIMES | FKINJtimes | KINSpilsJacTimesVecFn |

As with the rest of the FKINSOL routines, the names of all user-supplied routines here are fixed, in order to maximize portability for the resulting mixed-language program. Additionally, based on flags discussed above in §6.2, the names of the user-supplied routines are mapped to actual values through a series of definitions in the header file `fkinbbd.h`.

The following is a summary of the usage of this module. Steps that are unchanged from the main program described in §6.3 are grayed-out.

1. **Nonlinear system function specification**

2. NVECTOR **module initialization**

3. **Problem specification**

4. **Linear solver specification**

   To initialize the KINBBDPRE preconditioner, make the following call:

   ```
   CALL FKINBBDINIT (NLOCAL, MUDQ, MLDQ, MU, ML, IER)
   ```

   The arguments are as follows. NLOCAL is the local size of vectors for this process. MUDQ and MLDQ are the upper and lower half-bandwidths to be used in the computation of the local Jacobian blocks by difference quotients; these may be smaller than the true half-bandwidths of the Jacobian of the local block of $G$, when smaller values may provide greater efficiency. MU and ML are the upper and lower half-bandwidths of the band matrix that is retained as an approximation of the local Jacobian block; these may be smaller than MUDQ and MLDQ. IER is a return completion flag. A

Table 6.2: Description of the FKINSOL optional output arrays IOUT and ROUT

Integer output array IOUT

| Index | Optional output | KINSOL function |
|---|---|---|
| | KINSOL main solver | |
| 1 | LENRW | KINGetWorkSpace |
| 2 | LENIW | KINGetWorkSpace |
| 3 | NNI | KINGetNumNonlinSolvIters |
| 4 | NFE | KINGetNumFuncEvals |
| 5 | NBCF | KINGetNumBetaCondFails |
| 6 | NBKTRK | KINGetNumBacktrackOps |
| | KINDENSE linear solver | |
| 7 | LENRWLS | KINDenseGetWorkSpace |
| 8 | LENIWLS | KINDenseGetWorkSpace |
| 9 | LS_FLAG | KINDenseGetLastFlag |
| 10 | NFELS | KINDenseGetNumFuncEvals |
| 11 | NJE | KINDenseGetNumJacEvals |
| | KINBAND linear solver | |
| 7 | LENRWLS | KINBandGetWorkSpace |
| 8 | LENIWLS | KINBandGetWorkSpace |
| 9 | LS_FLAG | KINBandGetLastFlag |
| 10 | NFELS | KINBandGetNumFuncEvals |
| 11 | NJE | KINBandGetNumJacEvals |
| | KINSPGMR, KINSPBCG, KINSPTFQMR linear solvers | |
| 7 | LENRWLS | KINSpilsGetWorkSpace |
| 8 | LENIWLS | KINSpilsGetWorkSpace |
| 9 | LS_FLAG | KINSpilsGetLastFlag |
| 10 | NFELS | KINSpilsGetNumFuncEvals |
| 11 | NJTV | KINSpilsGetNumJacEvals |
| 12 | NPE | KINSpilsGetNumPrecEvals |
| 13 | NPS | KINSpilsGetNumPrecSolves |
| 14 | NLI | KINSpilsGetNumLinIters |
| 15 | NCFL | KINSpilsGetNumConvFails |

Real output array ROUT

| Index | Optional output | KINSOL function |
|---|---|---|
| 1 | FNORM | KINGetFuncNorm |
| 2 | SSTEP | KINGetStepLength |

value of 0 indicates success, while a value of $-1$ indicates that a memory failure occurred or that an input had an illegal value.

To specify the SPGMR linear system solver and use the KINBBDPRE preconditioner, make the following call:

```
CALL FKINBBDSPGMR (MAXL, MAXLRST, IER)
```

Its arguments are the same as those of FKINSPGMR (see step 4 in §6.3).

To specify the SPBCG linear system solver and use the KINBBDPRE preconditioner, make the following call:

```
CALL FKINBBDSPBCG (MAXL, IER)
```

Its arguments are the same as those of FKINSPGMR (see step 4 in §6.3).

To specify the SPTFQMR linear system solver and use the KINBBDPRE preconditioner, make the following call:

```
CALL FKINBBDSPTFQMR (MAXL, IER)
```

Its arguments are the same as those of FKINSPGMR (see step 4 in §6.3).

Optionally, to specify that SPGMR, SPBCG, or SPTFQMR should use the supplied FKJTIMES, make the call

```
CALL FKINSPILSSETJAC (FLAG, IER)
```

with FLAG $\neq 0$. (see step 4 in §6.3).

5. **Problem solution**

6. KINBBDPRE **Optional outputs**

   Optional outputs specific to the SPGMR, SPBCG, or SPTFQMR solver are listed in Table 6.2. To obtain the optional outputs associated with the KINBBDPRE module, make the following call:

   ```
   CALL FKINBBDOPT (LENRBBD, LENIBBD, NGEBBD)
   ```

   The arguments returned are as follows. LENRBBD is the length of real preconditioner work space, in realtype words. LENIBBD is the length of integer preconditioner work space, in integer words. These sizes are local to the current process. NGEBBD is the cumulative number of $G(u)$ evaluations (calls to FKLOCFN) so far.

7. **Memory deallocation**

   To free the internal memory created by the call to FKINBBDINIT, before calling FKINFREE and FNVFREEP, the user must call

   ```
   CALL FKINBBDFREE
   ```

8. **User-supplied routines**

   The following two routines must be supplied for use with the KINBBDPRE module:

   ```
   SUBROUTINE FKLOCFN (NLOC, ULOC, GLOC, IER)
   DIMENSION ULOC(*), GLOC(*)
   ```

This routine is to evaluate the function $G(u)$ approximating $F$ (possibly identical to $F$), in terms of the array ULOC (of length NLOC), which is the sub-vector of $u$ local to this processor. The resulting (local) sub-vector is to be stored in the array GLOC. IER is an error return flag which should be set to 0 if successful, a positive value if a recoverable error occurred (in which case KINSOL will attempt to correct), or a negative value if FKLOCFN failed unrecoverably (in which case the solution process is halted).

```
SUBROUTINE FKCOMMFN (NLOC, ULOC, IER)
DIMENSION ULOC(*)
```

This routine is to perform the inter-processor communication necessary for the FKLOCFN routine. Each call to FKCOMMFN is preceded by a call to the system function routine FKFUN with the same argument ULOC. IER is an error return flag which should be set to 0 if successful, a positive value if a recoverable error occurred (in which case KINSOL will attempt to correct), or a negative value if FKCOMMFN failed recoverably (in which case the solution process is halted).

The subroutine FKCOMMFN must be supplied even if it is not needed and must return IER=0.

Optionally, the user can supply a routine FKINJTIMES for the evaluation of Jacobian-vector products, as described above in step 4 in §6.3.

# Chapter 7

# Description of the NVECTOR module

The SUNDIALS solvers are written in a data-independent manner. They all operate on generic vectors (of type `N_Vector`) through a set of operations defined by the particular NVECTOR implementation. Users can provide their own specific implementation of the NVECTOR module or use one of two provided within SUNDIALS, a serial and an MPI parallel implementations.

The generic `N_Vector` type is a pointer to a structure that has an implementation-dependent *content* field containing the description and actual data of the vector, and an *ops* field pointing to a structure with generic vector operations. The type `N_Vector` is defined as

```
typedef struct _generic_N_Vector *N_Vector;

struct _generic_N_Vector {
    void *content;
    struct _generic_N_Vector_Ops *ops;
};
```

The `_generic_N_Vector_Ops` structure is essentially a list of pointers to the various actual vector operations, and is defined as

```
struct _generic_N_Vector_Ops {
  N_Vector    (*nvclone)(N_Vector);
  N_Vector    (*nvcloneempty)(N_Vector);
  void        (*nvdestroy)(N_Vector);
  void        (*nvspace)(N_Vector, long int *, long int *);
  realtype*   (*nvgetarraypointer)(N_Vector);
  void        (*nvsetarraypointer)(realtype *, N_Vector);
  void        (*nvlinearsum)(realtype, N_Vector, realtype, N_Vector, N_Vector);
  void        (*nvconst)(realtype, N_Vector);
  void        (*nvprod)(N_Vector, N_Vector, N_Vector);
  void        (*nvdiv)(N_Vector, N_Vector, N_Vector);
  void        (*nvscale)(realtype, N_Vector, N_Vector);
  void        (*nvabs)(N_Vector, N_Vector);
  void        (*nvinv)(N_Vector, N_Vector);
  void        (*nvaddconst)(N_Vector, realtype, N_Vector);
  realtype    (*nvdotprod)(N_Vector, N_Vector);
  realtype    (*nvmaxnorm)(N_Vector);
  realtype    (*nvwrmsnorm)(N_Vector, N_Vector);
  realtype    (*nvwrmsnormmask)(N_Vector, N_Vector, N_Vector);
  realtype    (*nvmin)(N_Vector);
```

```
  realtype    (*nvwl2norm)(N_Vector, N_Vector);
  realtype    (*nvl1norm)(N_Vector);
  void        (*nvcompare)(realtype, N_Vector, N_Vector);
  booleantype (*nvinvtest)(N_Vector, N_Vector);
  booleantype (*nvconstrmask)(N_Vector, N_Vector, N_Vector);
  realtype    (*nvminquotient)(N_Vector, N_Vector);
};
```

The generic NVECTOR module defines and implements the vector operations acting on `N_Vector`. These routines are nothing but wrappers for the vector operations defined by a particular NVECTOR implementation, which are accessed through the *ops* field of the `N_Vector` structure. To illustrate this point we show below the implementation of a typical vector operation from the generic NVECTOR module, namely `N_VScale`, which performs the scaling of a vector `x` by a scalar `c`:

```
void N_VScale(realtype c, N_Vector x, N_Vector z)
{
   z->ops->nvscale(c, x, z);
}
```

Table 7.1 contains a complete list of all vector operations defined by the generic NVECTOR module.

Finally, note that the generic NVECTOR module defines the functions `N_VCloneVectorArray` and `N_VCloneEmptyVectorArray`. Both functions create (by cloning) an array of `count` variables of type `N_Vector`, each of the same type as an existing `N_Vector`. Their prototypes are

```
N_Vector *N_VCloneVectorArray(int count, N_Vector w);
N_Vector *N_VCloneEmptyVectorArray(int count, N_Vector w);
```

and their definitions are based on the implementation-specific `N_VClone` and `N_VCloneEmpty` operations, respectively.

An array of variables of type `N_Vector` can be destroyed by calling `N_VDestroyVectorArray`, whose prototype is

```
void N_VDestroyVectorArray(N_Vector *vs, int count);
```

and whose definition is based on the implementation-specific `N_VDestroy` operation.

A particular implementation of the NVECTOR module must:

- Specify the *content* field of `N_Vector`.
- Define and implement the vector operations. Note that the names of these routines should be unique to that implementation in order to permit using more than one NVECTOR module (each with different `N_Vector` internal data representations) in the same code.
- Define and implement user-callable constructor and destructor routines to create and free an `N_Vector` with the new *content* field and with *ops* pointing to the new vector operations.
- Optionally, define and implement additional user-callable routines acting on the newly defined `N_Vector` (e.g., a routine to print the content for debugging purposes).
- Optionally, provide accessor macros as needed for that particular implementation to be used to access different parts in the *content* field of the newly defined `N_Vector`.

Table 7.1: Description of the NVECTOR operations

| Name | Usage and Description |
|---|---|
| N_VClone | `v = N_VClone(w);`<br>Creates a new `N_Vector` of the same type as an existing vector `w` and sets the *ops* field. It does not copy the vector, but rather allocates storage for the new vector. |
| N_VCloneEmpty | `v = N_VCloneEmpty(w);`<br>Creates a new `N_Vector` of the same type as an existing vector `w` and sets the *ops* field. It does not allocate storage for the data array. |
| N_VDestroy | `N_VDestroy(v);`<br>Destroys the `N_Vector v` and frees memory allocated for its internal data. |
| N_VSpace | `N_VSpace(nvSpec, &lrw, &liw);`<br>Returns storage requirements for one `N_Vector`. `lrw` contains the number of realtype words and `liw` contains the number of integer words. |
| N_VGetArrayPointer | `vdata = N_VGetArrayPointer(v);`<br>Returns a pointer to a `realtype` array from the `N_Vector v`. Note that this assumes that the internal data in `N_Vector` is a contiguous array of `realtype`. This routine is only used in the solver-specific interfaces to the dense and banded linear solvers, as well as the interfaces to the banded preconditioners provided with SUNDIALS. |
| N_VSetArrayPointer | `N_VSetArrayPointer(vdata, v);`<br>Overwrites the data in an `N_Vector` with a given array of `realtype`. Note that this assumes that the internal data in `N_Vector` is a contiguous array of `realtype`. This routine is only used in the interfaces to the dense linear solver. |
| N_VLinearSum | `N_VLinearSum(a, x, b, y, z);`<br>Performs the operation $z = ax + by$, where $a$ and $b$ are scalars and $x$ and $y$ are of type `N_Vector`: $z_i = ax_i + by_i$, $i = 0, \ldots, n-1$. |
| N_VConst | `N_VConst(c, z);`<br>Sets all components of the `N_Vector z` to `c`: $z_i = c$, $i = 0, \ldots, n-1$. |
| N_VProd | `N_VProd(x, y, z);`<br>Sets the `N_Vector z` to be the component-wise product of the `N_Vector` inputs `x` and `y`: $z_i = x_i y_i$, $i = 0, \ldots, n-1$. |
| N_VDiv | `N_VDiv(x, y, z);`<br>Sets the `N_Vector z` to be the component-wise ratio of the `N_Vector` inputs `x` and `y`: $z_i = x_i / y_i$, $i = 0, \ldots, n-1$. The $y_i$ may not be tested for 0 values. It should only be called with an `x` that is guaranteed to have all nonzero components. |

*continued on next page*

*continued from last page*

| Name | Usage and Description |
|---|---|
| N_VScale | `N_VScale(c, x, z);`<br>Scales the `N_Vector` `x` by the scalar `c` and returns the result in `z`: $z_i = cx_i,\ i = 0, \ldots, n-1$. |
| N_VAbs | `N_VAbs(x, z);`<br>Sets the components of the `N_Vector` `z` to be the absolute values of the components of the `N_Vector` `x`: $y_i = \lvert x_i \rvert,\ i = 0, \ldots, n-1$. |
| N_VInv | `N_VInv(x, z);`<br>Sets the components of the `N_Vector` `z` to be the inverses of the components of the `N_Vector` `x`: $z_i = 1.0/x_i,\ i = 0, \ldots, n-1$. This routine may not check for division by 0. It should be called only with an `x` which is guaranteed to have all nonzero components. |
| N_VAddConst | `N_VAddConst(x, b, z);`<br>Adds the scalar `b` to all components of `x` and returns the result in the `N_Vector` `z`: $z_i = x_i + b,\ i = 0, \ldots, n-1$. |
| N_VDotProd | `d = N_VDotProd(x, y);`<br>Returns the value of the ordinary dot product of `x` and `y`: $d = \sum_{i=0}^{n-1} x_i y_i$. |
| N_VMaxNorm | `m = N_VMaxNorm(x);`<br>Returns the maximum norm of the `N_Vector` `x`: $m = \max_i \lvert x_i \rvert$. |
| N_VWrmsNorm | `m = N_VWrmsNorm(x, w)`<br>Returns the weighted root-mean-square norm of the `N_Vector` `x` with weight vector `w`: $m = \sqrt{\left( \sum_{i=0}^{n-1} (x_i w_i)^2 \right) / n}$. |
| N_VWrmsNormMask | `m = N_VWrmsNormMask(x, w, id);`<br>Returns the weighted root mean square norm of the `N_Vector` `x` with weight vector `w` built using only the elements of `x` corresponding to nonzero elements of the `N_Vector` `id`:<br>$m = \sqrt{\left( \sum_{i=0}^{n-1} (x_i w_i \operatorname{sign}(id_i))^2 \right) / n}$. |
| N_VMin | `m = N_VMin(x);`<br>Returns the smallest element of the `N_Vector` `x`: $m = \min_i x_i$. |
| N_VWL2Norm | `m = N_VWL2Norm(x, w);`<br>Returns the weighted Euclidean $\ell_2$ norm of the `N_Vector` `x` with weight vector `w`: $m = \sqrt{\sum_{i=0}^{n-1} (x_i w_i)^2}$. |
| N_VL1Norm | `m = N_VL1Norm(x);`<br>Returns the $\ell_1$ norm of the `N_Vector` `x`: $m = \sum_{i=0}^{n-1} \lvert x_i \rvert$. |
| N_VCompare | `N_VCompare(c, x, z);`<br>Compares the components of the `N_Vector` `x` to the scalar `c` and returns an `N_Vector` `z` such that: $z_i = 1.0$ if $\lvert x_i \rvert \geq c$ and $z_i = 0.0$ otherwise. |

*continued on next page*

| *continued from last page* | |
|---|---|
| **Name** | **Usage and Description** |
| N_VInvTest | `t = N_VInvTest(x, z);`<br>Sets the components of the N_Vector `z` to be the inverses of the components of the N_Vector `x`, with prior testing for zero values: $z_i = 1.0/x_i,\ i = 0, \dots, n-1$. This routine returns `TRUE` if all components of `x` are nonzero (successful inversion) and returns `FALSE` otherwise. |
| N_VConstrMask | `t = N_VConstrMask(c, x, m);`<br>Performs the following constraint tests: $x_i > 0$ if $c_i = 2$, $x_i \geq 0$ if $c_i = 1$, $x_i \leq 0$ if $c_i = -1$, $x_i < 0$ if $c_i = -2$. There is no constraint on $x_i$ if $c_i = 0$. This routine returns `FALSE` if any element failed the constraint test, `TRUE` if all passed. It also sets a mask vector `m`, with elements equal to 1.0 where the constraint test failed, and 0.0 where the test passed. This routine is used only for constraint checking. |
| N_VMinQuotient | `minq = N_VMinQuotient(num, denom);`<br>This routine returns the minimum of the quotients obtained by termwise dividing $\texttt{num}_i$ by $\texttt{denom}_i$. A zero element in `denom` will be skipped. If no such quotients are found, then the large value `BIG_REAL` (defined in the header file `sundials_types.h`) is returned. |

## 7.1 The NVECTOR_SERIAL implementation

The serial implementation of the NVECTOR module provided with SUNDIALS, NVECTOR_SERIAL, defines the *content* field of `N_Vector` to be a structure containing the length of the vector, a pointer to the beginning of a contiguous data array, and a boolean flag *own_data* which specifies the ownership of *data*.

```
struct _N_VectorContent_Serial {
  long int length;
  booleantype own_data;
  realtype *data;
};
```

The following five macros are provided to access the content of an NVECTOR_SERIAL vector. The suffix _S in the names denotes serial version.

- `NV_CONTENT_S`

  This routine gives access to the contents of the serial vector `N_Vector`.

  The assignment `v_cont` = `NV_CONTENT_S(v)` sets `v_cont` to be a pointer to the serial `N_Vector` content structure.

  Implementation:

  ```
  #define NV_CONTENT_S(v) ( (N_VectorContent_Serial)(v->content) )
  ```

- `NV_OWN_DATA_S`, `NV_DATA_S`, `NV_LENGTH_S`

  These macros give individual access to the parts of the content of a serial `N_Vector`.

  The assignment `v_data = NV_DATA_S(v)` sets `v_data` to be a pointer to the first component of the data for the `N_Vector` `v`. The assignment `NV_DATA_S(v) = v_data` sets the component array of `v` to be `v_data` by storing the pointer `v_data`.

  The assignment `v_len = NV_LENGTH_S(v)` sets `v_len` to be the length of `v`. On the other hand, the call `NV_LENGTH_S(v) = len_v` sets the length of `v` to be `len_v`.

Implementation:

```
#define NV_OWN_DATA_S(v) ( NV_CONTENT_S(v)->own_data )
#define NV_DATA_S(v) ( NV_CONTENT_S(v)->data )
#define NV_LENGTH_S(v) ( NV_CONTENT_S(v)->length )
```

- `NV_Ith_S`

  This macro gives access to the individual components of the data array of an `N_Vector`.

  The assignment `r = NV_Ith_S(v,i)` sets `r` to be the value of the `i`-th component of `v`. The assignment `NV_Ith_S(v,i) = r` sets the value of the `i`-th component of `v` to be `r`.

  Here $i$ ranges from 0 to $n-1$ for a vector of length $n$.

  Implementation:

  ```
  #define NV_Ith_S(v,i) ( NV_DATA_S(v)[i] )
  ```

The NVECTOR_SERIAL module defines serial implementations of all vector operations listed in Table 7.1. Their names are obtained from those in Table 7.1 by appending the suffix `_Serial`. The module NVECTOR_SERIAL provides the following additional user-callable routines:

- `N_VNew_Serial`

  This function creates and allocates memory for a serial `N_Vector`. Its only argument is the vector length.

  ```
  N_Vector N_VNew_Serial(long int vec_length);
  ```

- `N_VNewEmpty_Serial`

  This function creates a new serial `N_Vector` with an empty (`NULL`) data array.

  ```
  N_Vector N_VNewEmpty_Serial(long int vec_length);
  ```

- `N_VMake_Serial`

  This function creates and allocates memory for a serial vector with user-provided data array.

  ```
  N_Vector N_VMake_Serial(long int vec_length, realtype *v_data);
  ```

- `N_VCloneVectorArray_Serial`

  This function creates (by cloning) an array of `count` serial vectors.

  ```
  N_Vector *N_VCloneVectorArray_Serial(int count, N_Vector w);
  ```

- `N_VCloneVectorArrayEmpty_Serial`

  This function creates (by cloning) an array of `count` serial vectors, each with an empty (`NULL`) data array.

  ```
  N_Vector *N_VCloneVectorArrayEmpty_Serial(int count, N_Vector w);
  ```

- `N_VDestroyVectorArray_Serial`

  This function frees memory allocated for the array of `count` variables of type `N_Vector` created with `N_VCloneVectorArray_Serial` or with `N_VCloneVectorArrayEmpty_Serial`.

  ```
  void N_VDestroyVectorArray_Serial(N_Vector *vs, int count);
  ```

- `N_VPrint_Serial`

  This function prints the content of a serial vector to `stdout`.

  ```
  void N_VPrint_Serial(N_Vector v);
  ```

**Notes**

- When looping over the components of an `N_Vector v`, it is more efficient to first obtain the component array via `v_data = NV_DATA_S(v)` and then access `v_data[i]` within the loop than it is to use `NV_Ith_S(v,i)` within the loop.
- `N_VNewEmpty_Serial`, `N_VMake_Serial`, and `N_VCloneVectorArrayEmpty_Serial` set the field *own_data* = `FALSE`. `N_VDestroy_Serial` and `N_VDestroyVectorArray_Serial` will not attempt to free the pointer *data* for any `N_Vector` with *own_data* set to `FALSE`. In such a case, it is the user's responsibility to deallocate the *data* pointer.
- To maximize efficiency, vector operations in the NVECTOR_SERIAL implementation that have more than one `N_Vector` argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the same internal representations.

## 7.2 The NVECTOR_PARALLEL implementation

The parallel implementation of the NVECTOR module provided with SUNDIALS, NVECTOR_PARALLEL, defines the *content* field of `N_Vector` to be a structure containing the global and local lengths of the vector, a pointer to the beginning of a contiguous local data array, an MPI communicator, an a boolean flag *own_data* indicating ownership of the data array *data*.

```
struct _N_VectorContent_Parallel {
  long int local_length;
  long int global_length;
  booleantype own_data;
  realtype *data;
  MPI_Comm comm;
};
```

The following seven macros are provided to access the content of a NVECTOR_PARALLEL vector. The suffix _P in the names denotes parallel version.

- `NV_CONTENT_P`

  This macro gives access to the contents of the parallel vector `N_Vector`.

  The assignment `v_cont = NV_CONTENT_P(v)` sets `v_cont` to be a pointer to the `N_Vector` content structure of type `struct _N_VectorParallelContent`.

  Implementation:

  ```
  #define NV_CONTENT_P(v) ( (N_VectorContent_Parallel)(v->content) )
  ```

- `NV_OWN_DATA_P`, `NV_DATA_P`, `NV_LOCLENGTH_P`, `NV_GLOBLENGTH_P`

  These macros give individual access to the parts of the content of a parallel `N_Vector`.

  The assignment `v_data = NV_DATA_P(v)` sets `v_data` to be a pointer to the first component of the local data for the `N_Vector v`. The assignment `NV_DATA_P(v) = v_data` sets the component array of `v` to be `v_data` by storing the pointer `v_data`.

  The assignment `v_llen = NV_LOCLENGTH_P(v)` sets `v_llen` to be the length of the local part of `v`. The call `NV_LENGTH_P(v) = llen_v` sets the local length of `v` to be `llen_v`.

  The assignment `v_glen = NV_GLOBLENGTH_P(v)` sets `v_glen` to be the global length of the vector `v`. The call `NV_GLOBLENGTH_P(v) = glen_v` sets the global length of `v` to be `glen_v`.

  Implementation:

  ```
  #define NV_OWN_DATA_P(v)   ( NV_CONTENT_P(v)->own_data )

  #define NV_DATA_P(v)       ( NV_CONTENT_P(v)->data )
  ```

```
#define NV_LOCLENGTH_P(v)  ( NV_CONTENT_P(v)->local_length )
#define NV_GLOBLENGTH_P(v) ( NV_CONTENT_P(v)->global_length )
```

- `NV_COMM_P`

  This macro provides access to the MPI communicator used by the NVECTOR_PARALLEL vectors.

  Implementation:

  ```
  #define NV_COMM_P(v) ( NV_CONTENT_P(v)->comm )
  ```

- `NV_Ith_P`

  This macro gives access to the individual components of the local data array of an `N_Vector`.

  The assignment `r = NV_Ith_P(v,i)` sets `r` to be the value of the `i`-th component of the local part of `v`. The assignment `NV_Ith_P(v,i) = r` sets the value of the `i`-th component of the local part of `v` to be `r`.

  Here $i$ ranges from 0 to $n-1$, where $n$ is the local length.

  Implementation:

  ```
  #define NV_Ith_P(v,i) ( NV_DATA_P(v)[i] )
  ```

The NVECTOR_PARALLEL module defines parallel implementations of all vector operations listed in Table 7.1 Their names are obtained from those in Table 7.1 by appending the suffix `_Parallel`. The module NVECTOR_PARALLEL provides the following additional user-callable routines:

- `N_VNew_Parallel`

  This function creates and allocates memory for a parallel vector.

  ```
  N_Vector N_VNew_Parallel(MPI_Comm comm,
                           long int local_length,
                           long int global_length);
  ```

- `N_VNewEmpty_Parallel`

  This function creates a new parallel `N_Vector` with an empty (`NULL`) data array.

  ```
  N_Vector N_VNewEmpty_Parallel(MPI_Comm comm,
                                long int local_length,
                                long int global_length);
  ```

- `N_VMake_Parallel`

  This function creates and allocates memory for a parallel vector with user-provided data array.

  ```
  N_Vector N_VMake_Parallel(MPI_Comm comm,
                            long int local_length,
                            long int global_length,
                            realtype *v_data);
  ```

- `N_VCloneVectorArray_Parallel`

  This function creates (by cloning) an array of `count` parallel vectors.

  ```
  N_Vector *N_VCloneVectorArray_Parallel(int count, N_Vector w);
  ```

- `N_VCloneVectorArrayEmpty_Parallel`

  This function creates (by cloning) an array of `count` parallel vectors, each with an empty (`NULL`) data array.

```
N_Vector *N_VCloneVectorArrayEmpty_Parallel(int count, N_Vector w);
```

- `N_VDestroyVectorArray_Parallel`

  This function frees memory allocated for the array of `count` variables of type `N_Vector` created with `N_VCloneVectorArray_Parallel` or with `N_VCloneVectorArrayEmpty_Parallel`.

  ```
  void N_VDestroyVectorArray_Parallel(N_Vector *vs, int count);
  ```

- `N_VPrint_Parallel`

  This function prints the content of a parallel vector to stdout.

  ```
  void N_VPrint_Parallel(N_Vector v);
  ```

**Notes**

- When looping over the components of an `N_Vector v`, it is more efficient to first obtain the local component array via `v_data = NV_DATA_P(v)` and then access `v_data[i]` within the loop than it is to use `NV_Ith_P(v,i)` within the loop.

- `N_VNewEmpty_Parallel`, `N_VMake_Parallel`, and `N_VCloneVectorArrayEmpty_Parallel` set the field *own_data* = `FALSE`. `N_VDestroy_Parallel` and `N_VDestroyVectorArray_Parallel` will not attempt to free the pointer *data* for any `N_Vector` with *own_data* set to `FALSE`. In such a case, it is the user's responsibility to deallocate the *data* pointer.

- To maximize efficiency, vector operations in the NVECTOR_PARALLEL implementation that have more than one `N_Vector` argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the same internal representations.

## 7.3 NVECTOR functions used by KINSOL

In Table 7.2 below, we list the vector functions in the NVECTOR module within the KINSOL package. The table also shows, for each function, which of the code modules uses the function. The KINSOL column shows function usage within the main solver module, while the remaining five columns show function usage within each of the four KINSOL linear solvers (KINSPILS stands for any of KINSPGMR, KINSPBCG, or KINSPTFQMR), the KINBBDPRE preconditioner module, and the FKINSOL module.

There is one subtlety in the KINSPILS column hidden by the table explained here for the case of the KINSPGMR module. The dot product function `N_VDotProd` is called both within the implementation file `kinsol_spgmr.c` for the KINSPGMR solver and within the implementation files `sundials_spgmr.c` and `sundials_iterative.c` for the generic SPGMR solver upon which the KINSPGMR solver is implemented. Also, although `N_VDiv` is not called within the implementation file `kinsol_spgmr.c`, it is called within the implementation file `sundials_spgmr.c` and so is required by the KINSPGMR solver module. This issue does not arise for the other two KINSOL linear solvers because the generic DENSE and BAND solvers (used in the implementation of KINDENSE and KINBAND) do not make calls to any vector functions.

At this point, we should emphasize that the KINSOL user does not need to know anything about the usage of vector functions by the KINSOL code modules in order to use KINSOL. The information is presented as an implementation detail for the interested reader.

The vector functions listed in Table 7.1 that are *not* used by KINSOL are: `N_VAddConst`, `N_VWrmsNorm`, `N_VWrmsNormMask`, `N_VCompare`, and `N_VInvTest`. Therefore a user-supplied NVECTOR module for KINSOL could omit these five functions.

Table 7.2: List of vector functions usage by KINSOL code modules

| | KINSOL | KINDENSE | KINBAND | KINSPILS | KINBBDPRE | FKINSOL |
|---|---|---|---|---|---|---|
| N_VClone | ✓ | | | ✓ | ✓ | |
| N_VCloneEmpty | | | | | | ✓ |
| N_VDestroy | ✓ | | | ✓ | ✓ | ✓ |
| N_VSpace | ✓ | | | | | |
| N_VGetArrayPointer | | ✓ | ✓ | | ✓ | ✓ |
| N_VSetArrayPointer | | ✓ | | | | ✓ |
| N_VLinearSum | ✓ | ✓ | | ✓ | | |
| N_VConst | | | | ✓ | | |
| N_VProd | ✓ | ✓ | ✓ | ✓ | | |
| N_VDiv | ✓ | | | ✓ | | |
| N_VScale | ✓ | ✓ | ✓ | ✓ | ✓ | |
| N_VAbs | ✓ | | | | | |
| N_VInv | ✓ | | | | | |
| N_VDotProd | | ✓ | ✓ | ✓ | | |
| N_VMaxNorm | ✓ | | | | | |
| N_VMin | ✓ | | | | | |
| N_VWL2Norm | ✓ | ✓ | ✓ | ✓ | | |
| N_VL1Norm | | | | ✓ | | |
| N_VConstrMask | ✓ | | | | | |
| N_VMinQuotient | ✓ | | | | | |

# Chapter 8

# Providing Alternate Linear Solver Modules

The central KINSOL module interfaces with the linear solver module to be used by way of calls to four routines. These are denoted here by `linit`, `lsetup`, `lsolve`, and `lfree`. Briefly, their purposes are as follows:

- `linit`: initialize and allocate memory specific to the linear solver;
- `lsetup`: evaluate and preprocess the Jacobian or preconditioner;
- `lsolve`: solve the linear system;
- `lfree`: free the linear solver memory.

A linear solver module must also provide a user-callable specification routine (like that described in §5.5.2) which will attach the above four routines to the main KINSOL memory block. The KINSOL memory block is a structure defined in the header file `kinsol_impl.h`. A pointer to such a structure is defined as the type `KINMem`. The four fields in a `KINMem` structure that must point to the linear solver's functions are `kin_linit`, `kin_lsetup`, `kin_lsolve`, and `kin_lfree`, respectively. Note that of the four interface routines, only the `lsolve` routine is required. The `lfree` routine must be provided only if the solver specification routine makes any memory allocation. For consistency with the existing KINSOL linear solver modules, we recommend that the return value of the specification function be 0 for a successful return or a negative value if an error occurs (the pointer to the main KINSOL memory block is `NULL`, an input is illegal, the NVECTOR implementation is not compatible, a memory allocation fails, etc.)

To facilitate data exchange between the four interface functions, the field `kin_lmem` in the KINSOL memory block can be used to attach a linear solver-specific memory block.

These four routines that interface between KINSOL and the linear solver module necessarily have fixed call sequences. Thus, a user wishing to implement another linear solver within the KINSOL package must adhere to this set of interfaces. The following is a complete description of the call list for each of these routines. Note that the call list of each routine includes a pointer to the main KINSOL memory block, by which the routine can access various data related to the KINSOL solution. The contents of this memory block are given in the file `kinsol_impl.h` (but not reproduced here, for the sake of space).

## 8.1 Initialization function

The type definition of `linit` is

`linit`

Definition `int (*linit)(KINMem kin_mem);`

Purpose The purpose of `linit` is to complete initializations for a specific linear solver, such as counters and statistics.

Arguments `kin_mem` is the KINSOL memory pointer of type `KINMem`.

Return value An `linit` function should return 0 if it has successfully initialized the KINSOL linear solver and −1 otherwise.

## 8.2 Setup function

The type definition of `lsetup` is

`lsetup`

Definition `int (*lsetup)(KINMem kin_mem);`

Purpose The job of `lsetup` is to prepare the linear solver for subsequent calls to `lsolve`. It may recompute Jacobian-related data if it deems necessary.

Arguments `kin_mem` is the KINSOL memory pointer of type `KINMem`.

Return value The `lsetup` routine should return 0 if successful and a non-zero value otherwise.

## 8.3 Solve function

The type definition of `lsolve` is

`lsolve`

Definition
```
int (*lsolve)(KINMem kin_mem, N_Vector x,
              N_Vector b, realtype *res_norm);
```

Purpose The routine `lsolve` must solve the linear equation $Jx = b$, where $J = \partial F/\partial u$ is evaluated at the current iterate and the right-hand side vector $b$ is input.

Arguments
- `kin_mem` is the KINSOL memory pointer of type `KINMem`.
- `x` is a vector set to an initial guess prior to calling `lsolve`. On return it should contain the solution to $Jx = b$.
- `b` is the right-hand side vector $b$, set to $-F(u)$, evaluated at the current iterate.
- `res_norm` holds the value of the $L_2$ norm of the residual vector upon return.

Return value `lsolve` should return 0 if successful. If an error occurs and recovery could be possible by calling again the `lsetup` function, then it should return a positive value. Otherwise, `lsolve` should return a negative value.

## 8.4 Memory deallocation function

The type definition of `lfree` is

`lfree`

Definition `void (*lfree)(KINMem kin_mem);`

Purpose The routine `lfree` should free any linear solver memory allocated by the `linit` routine.

Arguments `kin_mem` is the KINSOL memory pointer of type `KINMem`.

Return value This routine has no return value.

Notes — This routine is called once a problem has been completed and the linear solver is no longer needed.

# Chapter 9

# Generic Linear Solvers in SUNDIALS

In this chapter, we describe five generic linear solver code modules that are included in SUNDIALS, but which are of potential use as generic packages in themselves, either in conjunction with the use of KINSOL or separately. These modules are:

- The DENSE matrix package, which includes functions for small dense matrices treated as simple array types.
- The BAND matrix package, which includes the matrix type `BandMat`, macros and functions for `BandMat` matrices.
- The SPGMR package, which includes a solver for the scaled preconditioned GMRES method.
- The SPBCG package, which includes a solver for the scaled preconditioned Bi-CGStab method.
- The SPTFQMR package, which includes a solver for the scaled preconditioned TFQMR method.

For reasons related to installation, the names of the files involved in these generic solvers begin with the prefix `sundials_`. But despite this, each of the solvers is in fact generic, in that it is usable completely independently of SUNDIALS.

For the sake of space, the functions for `DenseMat` and `BandMat` matrices and the functions in SPGMR, SPBCG, and SPTFQMR are only summarized briefly, since they are less likely to be of direct use in connection with KINSOL. The functions for small dense matrices are fully described, because we expect that they will be useful in the implementation of preconditioners used with the combination of KINSOL and the KINSPGMR, KINSPBCG, or KINSPTFQMR solver.

## 9.1 The DENSE module

Relative to the SUNDIALS *srcdir*, the files comprising the DENSE generic linear solver are as follows:

- header files (located in *srcdir*`/include/sundials`)
  `sundials_dense.h` `sundials_smalldense.h`
  `sundials_types.h` `sundials_math.h` `sundials_config.h`
- source files (located in *srcdir*`/src/sundials`)
  `sundials_dense.c` `sundials_smalldense.c` `sundials_math.c`

Only two of the preprocessing directives in the header file `sundials_config.h` are relevant to the DENSE package by itself (see §2.4 for details):

- (required) definition of the precision of the SUNDIALS type `realtype`. One of the following lines must be present:
  `#define SUNDIALS_DOUBLE_PRECISION 1`
  `#define SUNDIALS_SINGLE_PRECISION 1`
  `#define SUNDIALS_EXTENDED_PRECISION 1`
- (optional) use of generic math functions: `#define SUNDIALS_USE_GENERIC_MATH 1`

The `sundials_types.h` header file defines the SUNDIALS `realtype` and `booleantype` types and the macro `RCONST`, while the `sundials_math.h` header file is needed for the `ABS` macro and `RAbs` function.

The eight files listed above can be extracted from the SUNDIALS *srcdir* and compiled by themselves into a DENSE library or into a larger user code.

### 9.1.1 Type DenseMat

The type `DenseMat` is defined to be a pointer to a structure with the number of rows, number of columns, and a data field:

```
typedef struct {
  long int M;
  long int N;
  realtype  **data;
} *DenseMat;
```

The *M* and *N* fields indicates the number of columns and rows, respectively, of a dense matrix, while the *data* field is a two dimensional array used for component storage. The elements of a dense matrix are stored columnwise (i.e columns are stored one on top of the other in memory). If `A` is of type `DenseMat`, then the (`i`,`j`)-th element of `A` (with $0 \leq$ `i` $<$ `M` and $0 \leq$ `j` $<$ `N`) is given by the expression `(A->data)[j][i]` or by the expression `(A->data)[0][j*M+i]`. The macros below allow a user to efficiently access individual matrix elements without writing out explicit data structure references and without knowing too much about the underlying element storage. The only storage assumption needed is that elements are stored columnwise and that a pointer to the `j`-th column of elements can be obtained via the `DENSE_COL` macro. Users should use these macros whenever possible.

### 9.1.2 Accessor Macros

The following two macros are defined by the DENSE module to provide access to data in the `DenseMat` type:

- `DENSE_ELEM`

  Usage : `DENSE_ELEM(A,i,j) = a_ij;` or `a_ij = DENSE_ELEM(A,i,j);`

  `DENSE_ELEM` references the (`i`,`j`)-th element of the $M \times N$ `DenseMat A`, $0 \leq$ `i` $< M$, $0 \leq$ `j` $< N$.

- `DENSE_COL`

  Usage : `col_j = DENSE_COL(A,j);`

  `DENSE_COL` references the `j`-th column of the $M \times N$ `DenseMat A`, $0 \leq$ `j` $< N$. The type of the expression `DENSE_COL(A,j)` is `realtype *` . After the assignment in the usage above, `col_j` may be treated as an array indexed from 0 to $M - 1$. The (`i, j`)-th element of `A` is referenced by `col_j[i]`.

### 9.1.3 Functions

The following functions for `DenseMat` matrices are available in the DENSE package. For full details, see the header file `sundials_dense.h`.

- `DenseAllocMat`: allocation of a `DenseMat` matrix;
- `DenseAllocPiv`: allocation of a pivot array for use with `DenseGETRF`/`DenseGETRS`;
- `DenseGETRF`: LU factorization with partial pivoting;
- `DenseGETRS`: solution of $Ax = b$ using LU factorization (for square matrices $A$);
- `DenseZero`: load a matrix with zeros;
- `DenseCopy`: copy one matrix to another;
- `DenseScale`: scale a matrix by a scalar;
- `DenseAddI`: increment a square matrix by the identity matrix;
- `DenseFreeMat`: free memory for a `DenseMat` matrix;
- `DenseFreePiv`: free memory for a pivot array;
- `DensePrint`: print a `DenseMat` matrix to standard output.

### 9.1.4 Small Dense Matrix Functions

The following functions for small dense matrices are available in the DENSE package:

- `denalloc`

  `denalloc(m,n)` allocates storage for an `m` by `n` dense matrix. It returns a pointer to the newly allocated storage if successful. If the memory request cannot be satisfied, then `denalloc` returns `NULL`. The underlying type of the dense matrix returned is `realtype**`. If we allocate a dense matrix `realtype** a` by `a = denalloc(m,n)`, then `a[j][i]` references the (`i`,`j`)-th element of the matrix `a`, $0 \leq$ `i` $<$ `m`, $0 \leq$ `j` $< n$, and `a[j]` is a pointer to the first element in the `j`-th column of `a`. The location `a[0]` contains a pointer to `m` $\times$ `n` contiguous locations which contain the elements of `a`.

- `denallocpiv`

  `denallocpiv(n)` allocates an array of `n` integers. It returns a pointer to the first element in the array if successful. It returns `NULL` if the memory request could not be satisfied.

- `denGETRF`

  `denGETRF(a,m,n,p)` factors the `m` by `n` dense matrix `a`, using Gaussian elimination with row pivoting. It overwrites the elements of `a` with its LU factors and keeps track of the pivot rows chosen in the pivot array `p`.

  A successful LU factorization leaves the matrix `a` and the pivot array `p` with the following information:

  1. `p[k]` contains the row number of the pivot element chosen at the beginning of elimination step `k`, `k` $= 0, 1, ...,$`n`$-1$.
  2. If the unique LU factorization of `a` is given by $Pa = LU$, where $P$ is a permutation matrix, $L$ is an `m` by `n` lower trapezoidal matrix with all diagonal elements equal to 1, and $U$ is an `n` by `n` upper triangular matrix, then the upper triangular part of `a` (including its diagonal) contains $U$ and the strictly lower trapezoidal part of `a` contains the multipliers, $I - L$. If `a` is square, $L$ is a unit lower triangular matrix.

     `denGETRF` returns 0 if successful. Otherwise it encountered a zero diagonal element during the factorization, indicating that the matrix `a` does not have full column rank. In this case it returns the column index (numbered from one) at which it encountered the zero.

- `denGETRS`

  `denGETRS(a,n,p,b)` solves the `n` by `n` linear system $ax = b$. It assumes that `a` (of size `n` × `n`) has been LU-factored and the pivot array `p` has been set by a successful call to `denGETRF(a,n,n,p)`. The solution $x$ is written into the `b` array.

- `denzero`

  `denzero(a,m,n)` sets all the elements of the `m` by `n` dense matrix `a` to be 0.0;

- `dencopy`

  `dencopy(a,b,m,n)` copies the `m` by `n` dense matrix `a` into the `m` by `n` dense matrix `b`;

- `denscale`

  `denscale(c,a,m,n)` scales every element in the `m` by `n` dense matrix `a` by `c`;

- `denaddI`

  `denaddI(a,n)` increments the `n` by `n` dense matrix `a` by the identity matrix;

- `denfreepiv`

  `denfreepiv(p)` frees the pivot array `p` allocated by `denallocpiv`;

- `denfree`

  `denfree(a)` frees the dense matrix `a` allocated by `denalloc`;

- `denprint`

  `denprint(a,m,n)` prints the `m` by `n` dense matrix `a` to standard output as it would normally appear on paper. It is intended as a debugging tool with small values of `n`. The elements are printed using the `%g` option. A blank line is printed before and after the matrix.

## 9.2 The BAND module

Relative to the SUNDIALS *srcdir*, the files comprising the BAND generic linear solver are as follows:

- header files (located in *srcdir*/`include/sundials`)
  `sundials_band.h`
  `sundials_types.h` `sundials_math.h` `sundials_config.h`

- source files (located in *srcdir*/`src/sundials`)
  `sundials_band.c` `sundials_math.c`

Only two of the preprocessing directives in the header file `sundials_config.h` are required to use the BAND package by itself (see §2.4 for details):

- (required) definition of the precision of the SUNDIALS type `realtype`. One of the following lines must be present:
  `#define SUNDIALS_DOUBLE_PRECISION 1`
  `#define SUNDIALS_SINGLE_PRECISION 1`
  `#define SUNDIALS_EXTENDED_PRECISION 1`

- (optional) use of generic math functions:
  `#define SUNDIALS_USE_GENERIC_MATH 1`

The `sundials_types.h` header file defines of the SUNDIALS `realtype` and `booleantype` types and the macro `RCONST`, while the `sundials_math.h` header file is needed for the `MIN`, `MAX`, and `ABS` macros and `RAbs` function.

The six files listed above can be extracted from the SUNDIALS *srcdir* and compiled by themselves into a BAND library or into a larger user code.

### 9.2.1 Type BandMat

The type `BandMat` is the type of a large band matrix A (possibly distributed). It is defined to be a pointer to a structure defined by:

```
typedef struct {
  long int size;
  long int mu, ml, smu;
  realtype **data;
} *BandMat;
```

The fields in the above structure are:

- *size* is the number of columns (which is the same as the number of rows);
- *mu* is the upper half-bandwidth, $0 \leq mu \leq size-1$;
- *ml* is the lower half-bandwidth, $0 \leq ml \leq size-1$;
- *smu* is the storage upper half-bandwidth, $mu \leq smu \leq size-1$. The `BandGBTRF` routine writes the LU factors into the storage for A. The upper triangular factor U, however, may have an upper half-bandwidth as big as min($size-1$,$mu+ml$) because of partial pivoting. The *smu* field holds the upper half-bandwidth allocated for A.
- *data* is a two dimensional array used for component storage. The elements of a band matrix of type `BandMat` are stored columnwise (i.e. columns are stored one on top of the other in memory). Only elements within the specified half-bandwidths are stored.

  If we number rows and columns in the band matrix starting from 0, then

  - *data[0]* is a pointer to $(smu+ml+1)*size$ contiguous locations which hold the elements within the band of A
  - *data[j]* is a pointer to the uppermost element within the band in the j-th column. This pointer may be treated as an array indexed from $smu-mu$ (to access the uppermost element within the band in the j-th column) to $smu+ml$ (to access the lowest element within the band in the j-th column). Indices from 0 to $smu-mu-1$ give access to extra storage elements required by `BandGBTRF`.
  - *data[j][i-j+smu]* is the $(i,j)$-th element, $j-mu \leq i \leq j+ml$.

The macros below allow a user to access individual matrix elements without writing out explicit data structure references and without knowing too much about the underlying element storage. The only storage assumption needed is that elements are stored columnwise and that a pointer into the `j`-th column of elements can be obtained via the `BAND_COL` macro. Users should use these macros whenever possible.

See Figure 9.1 for a diagram of the `BandMat` type.

### 9.2.2 Accessor Macros

The following three macros are defined by the BAND module to provide access to data in the `BandMat` type:

- `BAND_ELEM`

  Usage : `BAND_ELEM(A,i,j) = a_ij;` or `a_ij = BAND_ELEM(A,i,j);`

  `BAND_ELEM` references the `(i,j)`-th element of the $N \times N$ band matrix `A`, where $0 \leq$ `i`, `j` $\leq N-1$. The location `(i,j)` should further satisfy `j`$-$`(A->mu)` $\leq$ `i` $\leq$ `j`$+$`(A->ml)`.

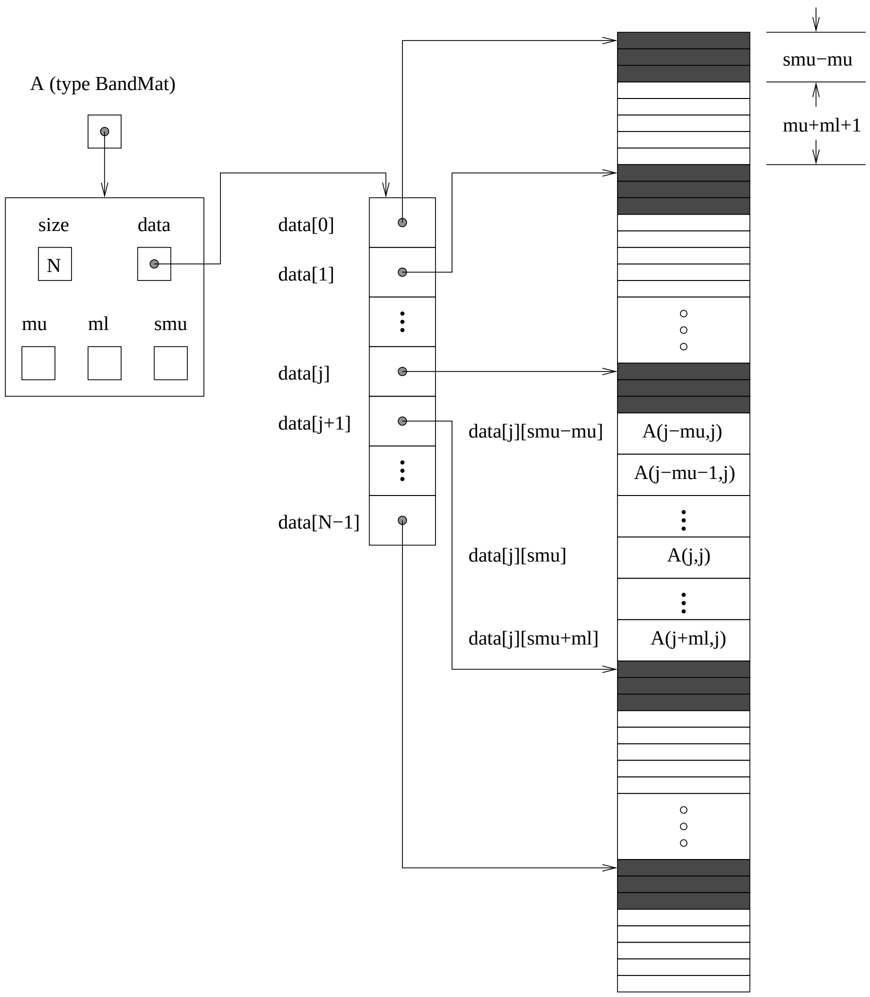



Figure 9.1: Diagram of the storage for a band matrix of type `BandMat`. Here `A` is an $N \times N$ band matrix of type `BandMat` with upper and lower half-bandwidths `mu` and `ml`, respectively. The rows and columns of `A` are numbered from 0 to $N-1$ and the $(i,j)$-th element of `A` is denoted `A(i,j)`. The greyed out areas of the underlying component storage are used by the `BandGBTRF` and `BandGBTRS` routines.

- BAND_COL

  Usage : `col_j = BAND_COL(A,j);`

  BAND_COL references the diagonal element of the j-th column of the $N \times N$ band matrix A, $0 \leq$ j $\leq N-1$. The type of the expression BAND_COL(A,j) is realtype *. The pointer returned by the call BAND_COL(A,j) can be treated as an array which is indexed from −(A->mu) to (A->ml).

- BAND_COL_ELEM

  Usage : `BAND_COL_ELEM(col_j,i,j) = a_ij;` or `a_ij = BAND_COL_ELEM(col_j,i,j);`

  This macro references the (i,j)-th entry of the band matrix A when used in conjunction with BAND_COL to reference the j-th column through col_j. The index (i,j) should satisfy j−(A->mu) ≤ i ≤ j+(A->ml).

### 9.2.3 Functions

The following functions for BandMat matrices are available in the BAND package. For full details, see the header file sundials_band.h.

- BandAllocMat: allocation of a BandMat matrix;
- BandAllocPiv: allocation of a pivot array for use with BandGBTRF/BandGBTRS;
- BandGBTRF: LU factorization with partial pivoting;
- BandGBTRS: solution of $Ax = b$ using LU factorization;
- BandZero: load a matrix with zeros;
- BandCopy: copy one matrix to another;
- BandScale: scale a matrix by a scalar;
- BandAddI: increment a matrix by the identity matrix;
- BandFreeMat: free memory for a BandMat matrix;
- BandFreePiv: free memory for a pivot array;
- BandPrint: print a BandMat matrix to standard output.

## 9.3 The SPGMR module

The SPGMR package, in the files sundials_spgmr.h and sundials_spgmr.c, includes an implementation of the scaled preconditioned GMRES method. A separate code module, implemented in sundials_iterative.(h,c), contains auxiliary functions that support SPGMR, as well as the other Krylov solvers in SUNDIALS (SPBCG and SPTFQMR). For full details, including usage instructions, see the header files sundials_spgmr.h and sundials_iterative.h.

Relative to the SUNDIALS *srcdir*, the files comprising the SPGMR generic linear solver are as follows:

- header files (located in *srcdir*/include/sundials)
  sundials_spgmr.h sundials_iterative.h sundials_nvector.h
  sundials_types.h sundials_math.h sundials_config.h
- source files (located in *srcdir*/src/sundials)
  sundials_spgmr.c sundials_iterative.c sundials_nvector.c

Only two of the preprocessing directives in the header file sundials_config.h are required to use the SPGMR package by itself (see §2.4 for details):

- (required) definition of the precision of the SUNDIALS type `realtype`. One of the following lines must be present:
  `#define SUNDIALS_DOUBLE_PRECISION 1`
  `#define SUNDIALS_SINGLE_PRECISION 1`
  `#define SUNDIALS_EXTENDED_PRECISION 1`
- (optional) use of generic math functions:
  `#define SUNDIALS_USE_GENERIC_MATH 1`

The `sundials_types.h` header file defines the SUNDIALS `realtype` and `booleantype` types and the macro `RCONST`, while the `sundials_math.h` header file is needed for the `MAX` and `ABS` macros and `RAbs` and `RSqrt` functions.

The generic NVECTOR files, `sundials_nvector.(h,c)` are needed for the definition of the generic `N_Vector` type and functions. The NVECTOR functions used by the SPGMR module are: `N_VDotProd`, `N_VLinearSum`, `N_VScale`, `N_VProd`, `N_VDiv`, `N_VConst`, `N_VClone`, `N_VCloneVectorArray`, `N_VDestroy`, and `N_VDestroyVectorArray`.

The SPGMR package can only be used in conjunction with an actual NVECTOR implementation library, such as the NVECTOR_SERIAL or NVECTOR_PARALLEL provided with SUNDIALS.

The nine files listed above can be extracted from the SUNDIALS *srcdir* and compiled by themselves into an SPGMR library or into a larger user code.

### 9.3.1 Functions

The following functions are available in the SPGMR package:

- `SpgmrMalloc`: allocation of memory for `SpgmrSolve`;
- `SpgmrSolve`: solution of $Ax = b$ by the SPGMR method;
- `SpgmrFree`: free memory allocated by `SpgmrMalloc`.

The following functions are available in the support package `sundials_iterative.(h,c)`:

- `ModifiedGS`: performs modified Gram-Schmidt procedure;
- `ClassicalGS`: performs classical Gram-Schmidt procedure;
- `QRfact`: performs QR factorization of Hessenberg matrix;
- `QRsol`: solves a least squares problem with a Hessenberg matrix factored by `QRfact`.

## 9.4 The SPBCG module

The SPBCG package, in the files `sundials_spbcgs.h` and `sundials_spbcgs.c`, includes an implementation of the scaled preconditioned Bi-CGStab method. For full details, including usage instructions, see the file `sundials_spbcgs.h`.

The SPBCG package can only be used in conjunction with an actual NVECTOR implementation library, such as the NVECTOR_SERIAL or NVECTOR_PARALLEL provided with SUNDIALS.

The files needed to use the SPBCG module by itself are the same as for the SPGMR module, with `sundials_spbcgs.(h,c)` replacing `sundials_spgmr.(h,c)`.

### 9.4.1 Functions

The following functions are available in the SPBCG package:

- `SpbcgMalloc`: allocation of memory for `SpbcgSolve`;
- `SpbcgSolve`: solution of $Ax = b$ by the SPBCG method;
- `SpbcgFree`: free memory allocated by `SpbcgMalloc`.

## 9.5 The SPTFQMR module

The SPTFQMR package, in the files sundials_sptfqmr.h and sundials_sptfqmr.c, includes an implementation of the scaled preconditioned TFQMR method. For full details, including usage instructions, see the file sundials_sptfqmr.h.

The SPTFQMR package can only be used in conjunction with an actual NVECTOR implementation library, such as the NVECTOR_SERIAL or NVECTOR_PARALLEL provided with SUNDIALS.

The files needed to use the SPTFQMR module by itself are the same as for the SPGMR module, with sundials_sptfqmr.(h,c) replacing sundials_spgmr.(h,c).

### 9.5.1 Functions

The following functions are available in the SPTFQMR package:

- SptfqmrMalloc: allocation of memory for SptfqmrSolve;
- SptfqmrSolve: solution of $Ax = b$ by the SPTFQMR method;
- SptfqmrFree: free memory allocated by SptfqmrMalloc.

Chapter 10

# KINSOL Constants

Below we list all input and output constants used by the main solver and linear solver modules, together with their numerical values and a short description of their meaning.

## 10.1 KINSOL input constants

| KINSOL **main solver module** | | |
|---|---|---|
| KIN_ETACHOICE1 | 1 | Use Eisenstat and Walker Choice 1 for $\eta$. |
| KIN_ETACHOICE2 | 2 | Use Eisenstat and Walker Choice 2 for $\eta$. |
| KIN_ETACONSTANT | 3 | Use constant value for $\eta$. |
| KIN_NONE | 0 | Use inexact Newton globalization. |
| KIN_LINESEARCH | 1 | Use line search globalization. |

| **Iterative linear solver module** | | |
|---|---|---|
| PREC_NONE | 0 | No preconditioning |
| PREC_RIGHT | 2 | Preconditioning on the right. |
| MODIFIED_GS | 1 | Use modified Gram-Schmidt procedure. |
| CLASSICAL_GS | 2 | Use classical Gram-Schmidt procedure. |

## 10.2 KINSOL output constants

| KINSOL **main solver module** | | |
|---|---|---|
| KIN_SUCCESS | 0 | Successful function return. |
| KIN_INITIAL_GUESS_OK | 1 | The initial user-supplied guess already satisfies the stopping criterion. |
| KIN_STEP_LT_STPTOL | 2 | The stopping tolerance on scaled step length was satisfied. |
| KIN_MEM_NULL | -1 | The `kin_mem` argument was `NULL`. |
| KIN_ILL_INPUT | -2 | One of the function inputs is illegal. |
| KIN_NO_MALLOC | -3 | The KINSOL memory was not allocated by a call to `KINMalloc`. |
| KIN_MEM_FAIL | -4 | A memory allocation failed. |
| KIN_LINESEARCH_NONCONV | -5 | The line search algorithm was unable to find an iterate sufficiently distinct from the current iterate. |

| | | |
|---|---|---|
| KIN_MAXITER_REACHED | -6 | The maximum number of nonlinear iterations has been reached. |
| KIN_MXNEWT_5X_EXCEEDED | -7 | Five consecutive steps have been taken that satisfy a scaled step length test. |
| KIN_LINESEARCH_BCFAIL | -8 | The line search algorithm was unable to satisfy the $\beta$-condition for `nbcfails` iterations. |
| KIN_LINSOLV_NO_RECOVERY | -9 | The user-supplied routine preconditioner slve function failed recoverably, but the preconditioner is already current. |
| KIN_LINIT_FAIL | -10 | The linear solver's initialization function failed. |
| KIN_LSETUP_FAIL | -11 | The linear solver's setup function failed in an unrecoverable manner. |
| KIN_LSOLVE_FAIL | -12 | The linear solver's solve function failed in an unrecoverable manner. |
| KIN_SYSFUNC_FAIL | -13 | The system function failed in an unrecoverable manner. |
| KIN_FIRST_SYSFUNC_ERR | -14 | The system function failed recoverably at the first call. |
| KIN_REPTD_SYSFUNC_ERR | -15 | The system function had repeated recoverable errors. |

### KINDENSE linear solver module

| | | |
|---|---|---|
| KINDENSE_SUCCESS | 0 | Successful function return. |
| KINDENSE_MEM_NULL | -1 | The `kin_mem` argument was `NULL`. |
| KINDENSE_LMEM_NULL | -2 | The KINDENSE linear solver has not been initialized. |
| KINDENSE_ILL_INPUT | -3 | The KINDENSE solver is not compatible with the current NVECTOR module. |
| KINDENSE_MEM_FAIL | -4 | A memory allocation request failed. |

### KINBAND linear solver module

| | | |
|---|---|---|
| KINBAND_SUCCESS | 0 | Successful function return. |
| KINBAND_MEM_NULL | -1 | The `kin_mem` argument was `NULL`. |
| KINBAND_LMEM_NULL | -2 | The KINBAND linear solver has not been initialized. |
| KINBAND_ILL_INPUT | -3 | The KINBAND solver is not compatible with the current NVECTOR module, or an input value was illegal. |
| KINBAND_MEM_FAIL | -4 | A memory allocation request failed. |

### KINSPILS linear solver modules

| | | |
|---|---|---|
| KINSPILS_SUCCESS | 0 | Successful function return. |
| KINSPILS_MEM_NULL | -1 | The `kin_mem` argument was `NULL`. |
| KINSPILS_LMEM_NULL | -2 | The linear solver has not been initialized. |
| KINSPILS_ILL_INPUT | -3 | The solver is not compatible with the current NVECTOR module, or an input value was illegal. |
| KINSPILS_MEM_FAIL | -4 | A memory allocation request failed. |

### SPGMR generic linear solver module

| | | |
|---|---|---|
| SPGMR_SUCCESS | 0 | Converged. |
| SPGMR_RES_REDUCED | 1 | No convergence, but the residual norm was reduced. |

| | | |
|---|---|---|
| SPGMR_CONV_FAIL | 2 | Failure to converge. |
| SPGMR_QRFACT_FAIL | 3 | A singular matrix was found during the QR factorization. |
| SPGMR_PSOLVE_FAIL_REC | 4 | The preconditioner solve function failed recoverably. |
| SPGMR_MEM_NULL | -1 | The SPGMR memory is NULL |
| SPGMR_ATIMES_FAIL | -2 | The Jacobian tims vector function failed. |
| SPGMR_PSOLVE_FAIL_UNREC | -3 | The preconditioner solve function failed unrecoverably. |
| SPGMR_GS_FAIL | -4 | Failure in the Gram-Schmidt procedure. |
| SPGMR_QRSOL_FAIL | -5 | The matrix $R$ was found to be singular during the QR solve phase. |

**SPBCG generic linear solver module**

| | | |
|---|---|---|
| SPBCG_SUCCESS | 0 | Converged. |
| SPBCG_RES_REDUCED | 1 | No convergence, but the residual norm was reduced. |
| SPBCG_CONV_FAIL | 2 | Failure to converge. |
| SPBCG_PSOLVE_FAIL_REC | 3 | The preconditioner solve function failed recoverably. |
| SPBCG_MEM_NULL | -1 | The SPBCG memory is NULL |
| SPBCG_ATIMES_FAIL | -2 | The Jacobian tims vector function failed. |
| SPBCG_PSOLVE_FAIL_UNREC | -3 | The preconditioner solve function failed unrecoverably. |

**SPTFQMR generic linear solver module**

| | | |
|---|---|---|
| SPTFQMR_SUCCESS | 0 | Converged. |
| SPTFQMR_RES_REDUCED | 1 | No convergence, but the residual norm was reduced. |
| SPTFQMR_CONV_FAIL | 2 | Failure to converge. |
| SPTFQMR_PSOLVE_FAIL_REC | 3 | The preconditioner solve function failed recoverably. |
| SPTFQMR_MEM_NULL | -1 | The SPTFQMR memory is NULL |
| SPTFQMR_ATIMES_FAIL | -2 | The Jacobian tims vector function failed. |
| SPTFQMR_PSOLVE_FAIL_UNREC | -3 | The preconditioner solve function failed unrecoverably. |

**KINBBDPRE preconditioner module**

| | | |
|---|---|---|
| KINBBDPRE_SUCCESS | 0 | Successful function return. |
| KINBBDPRE_PDATA_NULL | -11 | The preconditioner module has not been initialized. |

# Bibliography


[1] P. N. Brown. A local convergence theory for combined inexact-Newton/finite difference projection methods. *SIAM J. Numer. Anal.*, 24(2):407–434, 1987.

[2] P. N. Brown and Y. Saad. Hybrid Krylov Methods for Nonlinear Systems of Equations. *SIAM J. Sci. Stat. Comput.*, 11:450–481, 1990.

[3] A. M. Collier and R. Serban. Example Programs for KINSOL v2.4.0. Technical Report UCRL-SM-208114, LLNL, 2006.

[4] R. S. Dembo, S. C. Eisenstat, and T. Steihaug. Inexact Newton Methods. *SIAM J. Numer. Anal.*, 19:400–408, 1982.

[5] J. E. Dennis and R. B. Schnabel. *"Numerical Methods for Unconstrained Optimization and Nonlinear Equations"*. SIAM, Philadelphia, 1996.

[6] S. C. Eisenstat and H. F. Walker. Choosing the Forcing Terms in an Inexact Newton Method. *SIAM J. Sci. Comput.*, 17:16–32, 1996.

[7] R. W. Freund. A Transpose-Free Quasi-Minimal Residual Algorithm for Non-Hermitian Linear Systems. *SIAM J. Sci. Comp.*, 14:470–482, 1993.

[8] C. T. Kelley. *Iterative Methods for Solving Linear and Nonlinear Equations*. SIAM, Philadelphia, 1995.

[9] Y. Saad and M. H. Schultz. GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. *SIAM J. Sci. Stat. Comp.*, 7:856–869, 1986.

[10] H. A. Van Der Vorst. Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems. *SIAM J. Sci. Stat. Comp.*, 13:631–644, 1992.

# Index