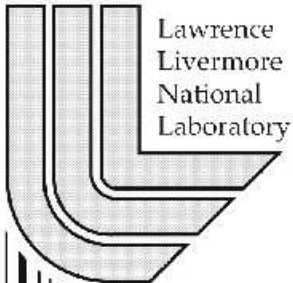


User Documentation for KINSOL v2.2.0

*Aaron M. Collier, Alan C. Hindmarsh,
Radu Serban, and Carol S. Woodward*

U.S. Department of Energy



Lawrence
Livermore
National
Laboratory

November 2004

DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

This is a preprint of a paper intended for publication in a journal or proceedings. Since changes may be made before publication, this preprint is made available with the understanding that it will not be cited or reproduced without the permission of the author.

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.

User Documentation for KINSOL v2.2.0

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

Contents

List of Tables	v
List of Figures	vii
1 Introduction	1
1.1 Historical Background	1
1.2 Changes in Version 2.2.0	2
1.3 Reading this User Guide	2
2 KINSOL Installation Procedure	3
2.1 Installation steps	3
2.2 Configuration options	4
2.3 Configuration examples	8
3 Mathematical Considerations	9
4 Code Organization	11
4.1 SUNDIALS organization	11
4.2 KINSOL organization	11
5 Using KINSOL	15
5.1 Data types	15
5.2 Header Files	16
5.3 A Skeleton of the User's Main Program	16
5.4 User-callable functions	19
5.4.1 KINSOL initialization and deallocation functions	19
5.4.2 Linear solver specification function	20
5.4.3 KINSOL solver function	20
5.4.4 Optional input functions	22
5.4.5 Optional output functions	30
5.5 User-supplied functions	35
5.5.1 Problem-defining function	35
5.5.2 Jacobian information (SPGMR matrix-vector product)	35
5.5.3 Preconditioning (SPGMR linear system solution)	36
5.5.4 Preconditioning (SPGMR Jacobian data)	36
5.6 A parallel band-block-diagonal preconditioner module	37
5.7 FKINSOL, a FORTRAN-C interface module	41
5.7.1 FKINSOL routines	41
5.7.2 FKINSOL optional input and output	42
5.7.3 Usage of the FKINSOL interface module	42
5.7.4 Usage of the FKINBBD interface to KINBBDPRE	46

6	Description of the NVECTOR module	49
6.1	The NVECTOR_SERIAL implementation	53
6.2	The NVECTOR_PARALLEL implementation	55
6.3	NVECTOR functions used by KINSOL	58
7	Providing Alternate Linear Solver Modules	59
8	Generic Linear Solvers in SUNDIALS	61
8.1	The SPGMR Module	61
	Bibliography	63
	Index	65

List of Tables

2.1	SUNDIALS libraries and header files	5
5.1	Optional inputs for KINSOL and KINSPGMR	22
5.2	Optional outputs from KINSOL and KINSPGMR	30
5.3	Description of the FKINSOL optional input-output arrays IOPT and ROPT	43
6.1	Description of the NVECTOR operations	51
6.2	List of vector functions usage by KINSOL code modules	58

List of Figures

4.1	Organization of the SUNDIALS suite	12
4.2	Overall structure diagram of the KINSOL package	13
5.1	Diagram of the user program and KINSOL package for the solution of nonlinear systems	17

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 [3]. 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.

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.

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 multiprocessor 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.

1.2 Changes in Version 2.2.0

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 §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

The structure of this document is as follows:

- The next section discusses how to install the KINSOL package.
- In §3, we provide short descriptions of the numerical methods implemented by KINSOL for the solution of nonlinear systems.
- The following section describes the structure of the SUNDIALS suite of solvers (§4.1) and the software organization of the KINSOL solver (§4.2).
- In §5, we give an overview of the usage of KINSOL, as well as a complete description of the user interface and of the user-defined routines for solution of nonlinear systems.
- Section 6 gives a brief overview of the generic NVECTOR module shared among the various components of SUNDIALS, as well as details of the two NVECTOR implementations provided with SUNDIALS: a serial implementation (§6.1) and a parallel implementation, based on MPI (§6.2).
- Section 8 describes the generic linear solvers shared by all SUNDIALS solvers.

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.

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 procedure for KINSOL is the same whether or not the downloaded file contains SUNDIALS solvers other than KINSOL.

Generally speaking, the installation procedure outlined in §2.1 below will work on commodity LINUX/UNIX systems without modification. Users are still encouraged, however, 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 `sundials` directory.

In the descriptions below, *build_tree* refers to the directory under which the user wants to build and/or install the SUNDIALS package. By default, the SUNDIALS libraries and header files are installed under the subdirectories *build_tree/lib* and *build_tree/include*, respectively. Also, *source_tree* refers to the directory where the SUNDIALS source code is located. The chosen *build_tree* may be different from the *source_tree*, thus allowing for multiple installations of the SUNDIALS suite with different configuration options.

Concerning the installation procedure outlined below, after invoking the `tar` command with the appropriate options, the contents of the SUNDIALS archive (or the *source_tree*) will be extracted to a directory named `sundials`. Since the name of the extracted directory is not version-specific it is recommended that the user refrain from extracting the archive to a directory containing a previous version/release of the SUNDIALS suite. If the user is only upgrading and the previous installation of SUNDIALS is not needed, then the user may remove the previous installation by issuing

```
% rm -rf sundials
```

from a shell command prompt.

Even though the installation procedure given below presupposes that the user will use the default vector modules supplied with the distribution, using the SUNDIALS suite with a user-supplied vector module normally will not require any changes to the build procedure.

2.1 Installation steps

To install the SUNDIALS suite, given a downloaded file named *sundials_file.tar.gz*, issue the following commands from a shell command prompt, while within the directory where *source_tree* is to be located. The names of installed libraries and header files are listed in Table 2.1 for reference. (For brevity, the corresponding `.c` files are not listed.) Regarding the file extension *.lib* appearing in Table 2.1, shared libraries generally have an extension of *.so* and static libraries have an extension of *.a*. (See *Options for library support* for additional details.)

1. `gunzip sundials_file.tar.gz`
2. `tar -xf sundials_file.tar` [creates `sundials` directory]
3. `cd build_tree`
4. `path_to_source_tree/configure options` [options can be absent]
5. `make`
6. `make install`
7. `make examples`
8. If system storage space conservation is a priority, then issue
`% make clean`
 and/or
`% make examples_clean`
 from a shell command prompt to remove unneeded object files.

2.2 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=build_tree`

`--includedir=DIR`

Alternate location for installation of header files.

Default: `DIR=PREFIX/include`

`--libdir=DIR`

Alternate location for installation of libraries.

Default: `DIR=PREFIX/lib`

`--disable-examples`

All available example programs are automatically built unless this option is given. The example executables are stored under the following subdirectories of the associated solver:

`build_tree/solver/examples_ser` : serial C examples

`build_tree/solver/examples_par` : parallel C examples (MPI-enabled)

`build_tree/solver/fcmix/examples_ser` : serial FORTRAN examples

`build_tree/solver/fcmix/examples_par` : parallel FORTRAN examples (MPI-enabled)

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

Table 2.1: SUNDIALS libraries and header files

Module	Libraries	Header files
SHARED	<code>libsundials_shared.lib</code>	<code>sundialstypes.h</code> <code>sundialsmath.h</code> <code>sundials_config.h</code> <code>dense.h</code> <code>smalldense.h</code> <code>band.h</code> <code>spgmr.h</code> <code>iterative.h</code> <code>nvector.h</code>
NVECTOR_SERIAL	<code>libsundials_nvecserial.lib</code> <code>libsundials_fnvecserial.a</code>	<code>nvector_serial.h</code>
NVECTOR_PARALLEL	<code>libsundials_nvecparallel.lib</code> <code>libsundials_fnvecparallel.a</code>	<code>nvector_parallel.h</code>
CVODE	<code>libsundials_cvode.lib</code> <code>libsundials_fcvcde.a</code>	<code>cvode.h</code> <code>cvdense.h</code> <code>cvband.h</code> <code>cvdiag.h</code> <code>cvspgmr.h</code> <code>cvbandpre.h</code> <code>cvbbdpre.h</code>
CVODES	<code>libsundials_cvodes.lib</code>	<code>cvodes.h</code> <code>cvodea.h</code> <code>cvdense.h</code> <code>cvband.h</code> <code>cvdiag.h</code> <code>cvspgmr.h</code> <code>cvbandpre.h</code> <code>cvbbdpre.h</code>
IDA	<code>libsundials_ida.lib</code>	<code>ida.h</code> <code>idadense.h</code> <code>idaband.h</code> <code>idaspgmr.h</code> <code>idabbdppe.h</code>
KINSOL	<code>libsundials_kinsol.lib</code> <code>libsundials_fkingsol.a</code>	<code>kinsol.h</code> <code>kinspgmr.h</code> <code>kinbbdppe.h</code>

--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`.

--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 non-standard 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`

Options for Fortran support

--disable-f77

Using this option will disable all FORTRAN support. The `FCVODE`, `FKINSOL` 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 `FKINSOL`, `FCVODE` 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 `FKINSOL`, `FCVODE` 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 compiler 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`, `LIBDIR=MPIDIR/lib` and `LIBS=-lmpi`

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 and FKINSOL 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 cross-compilation

If the SUNDIALS suite will be cross-compiled (meaning the build procedure will not be completed on the actual destination system, but rather on an alternate system with a different architecture) then the following two options should be used:

`--build=BUILD`

This particular option is used to specify the canonical system/platform name for the build system.

`--host=HOST`

If cross-compiling, then the user must use this option to specify the canonical system/platform name for the destination system.

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.

2.3 Configuration examples

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

```
% 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 above 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** compilation flag to the list of default flags.

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

This 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 disables the examples (which means a FORTRAN compiler is not required). The **--with-mpi-libs** option is still needed so that the configure script can check if **gcc** can link with the appropriate MPI library as **-lmpi** is the internal default.

Chapter 3

Mathematical Considerations

KINSOL solves nonlinear algebraic systems in real space, which we write as

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

given an initial guess u_0 .

KINSOL employs the Inexact Newton method developed in [1, 3, 5] and further described in [6, 8], resulting in the following iteration:

Inexact Newton iteration

1. Set u_0 = an initial guess
2. For $n = 0, 1, 2, \dots$ until convergence do:
 - (a) Approximately 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. As this code module is anticipated for use on large systems, only iterative methods are provided to solve the system in step 2(a). These solutions are only approximate. At each stage in the iteration process, a scalar multiple of the approximate solution, δ_n , is added to u_n to produce a new iterate, u_{n+1} . A test for convergence is made before the iteration continues.

The linear iterative method currently implemented is one of the class of Krylov methods, GMRES [2, 9], provided through the SPGMR module common to all SUNDIALS codes. Use of SPGMR provides a linear solver which, by default, is applied in a matrix-free manner, with matrix-vector products Jv obtained by either finite difference quotients or a user-supplied routine. In the case where finite differences are used, the matrix-vector product $J(u)v$ is approximated by a quotient of the form given by

$$J(u)v \approx [F(u + \sigma v) - F(u)]/\sigma \quad (3.2)$$

where u is the current approximation to a root of (3.1), and σ is a scalar. The choice of σ is taken from [3] and is given by

$$\sigma = \frac{\max\{|u^T v|, \text{typ}u^T |v|\}}{\|v\|_2} \text{sign}(u^T v) \sqrt{U}, \quad (3.3)$$

where $\text{typ}u$ 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 σ remains appropriately small as shown in [1].

To the above methods are added scaling and preconditioning. Scaling is allowed for both the solution vector and the system function 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:

$$\|z\|_{D_u} = \|D_u z\|_2, \quad \|z\|_{D_F} = \|D_F z\|_2, \quad \|z\|_{D_u, \infty} = \|D_u z\|_\infty, \quad \text{and} \quad \|z\|_{D_F, \infty} = \|D_F z\|_\infty \quad (3.4)$$

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 σ in (3.3). Additionally, right preconditioning is provided if the preconditioning setup and solve routines are supplied by the user. In this case, GMRES is applied to the linear systems $(JP^{-1})(P\delta) = -F$, where P denotes the right preconditioning matrix.

Two methods of applying a computed step δ_n to the previously computed solution vector are implemented. The first and simplest is the Inexact Newton strategy which applies step 2(b) as above with λ always set to 1. The other method is a global strategy, which attempts to use the direction implied by δ_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 α and β conditions of the Goldstein-Armijo linesearch given in [6] for step 2(b), where λ 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. For more details, the reader is referred to [6].

Stopping criteria for the Newton method can be required for either or 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

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

where STEPTOL is an input scalar tolerance with a default value of $U^{2/3}$.

Three options for stopping criteria for the linear system solve are implemented, including the two algorithms of Eisenstat and Walker [7]. The Krylov iteration must pass a stopping test

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

where η_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 γ and α are 0.9 and 2, respectively.

- $\eta_n = \text{constant}$ with 0.1 as the default.

The default is Eisenstat and Walker Choice 1. For both options 1 and 2, appropriate safeguards are incorporated to ensure that η does not decrease too quickly [7].

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 δ_n in Step 2(b) is scaled to take a step of that length.

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.

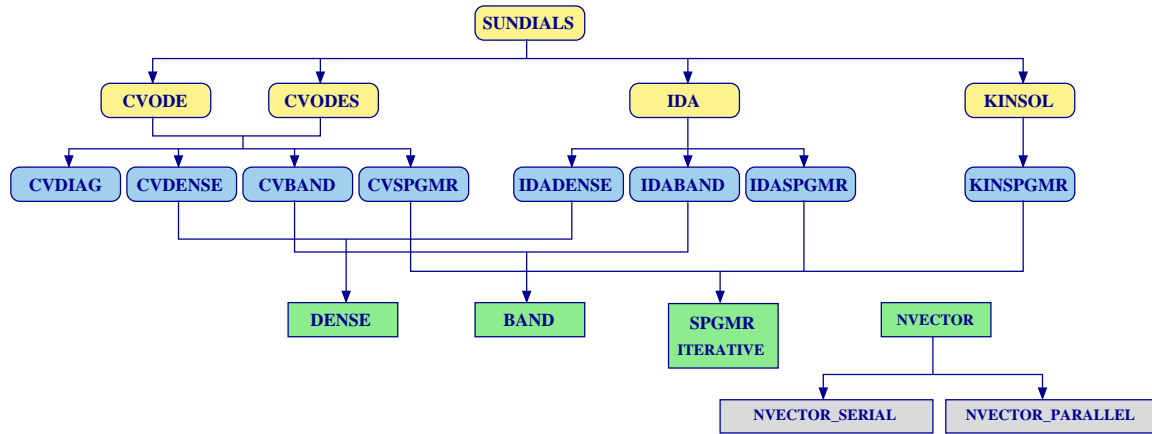
The central solver module, implemented in the files `kinsol.h` and `kinsol.c`, deals with the solution of a nonlinear algebraic system using either an Inexact Newton method or a line search method for the global strategy. Although this module contains logic for the Newton iteration, it has no knowledge of the method used to solve the linear systems that arise. For any given user problem, the user must specify which linear solver module to use.

At present, the package includes the following KINSOL linear system module:

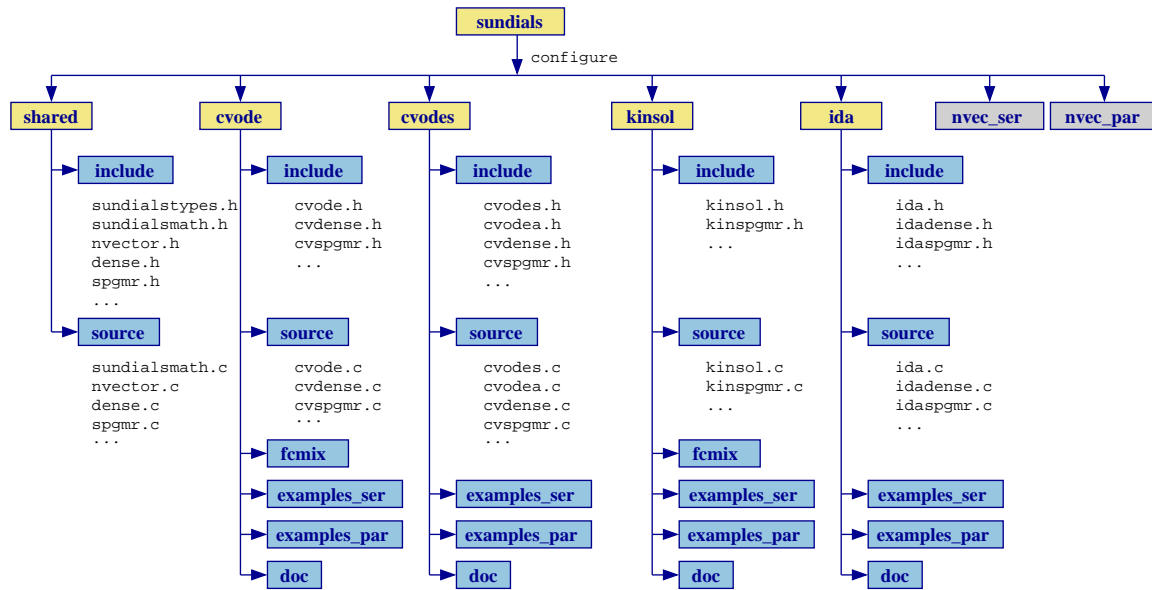
- KINSPGMR: scaled preconditioned GMRES method.

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

The KINSPGMR package includes an algorithm for the approximation by difference quotients of the product between the Jacobian matrix and a vector of appropriate length. The user has the option of providing a routine for this operation. With KINSPGMR, the preconditioning must be supplied by the user, in two phases: setup (preprocessing of preconditioner data) and solve.



(a) High-level diagram



(b) Directory structure

Figure 4.1: Organization of the SUNDIALS suite

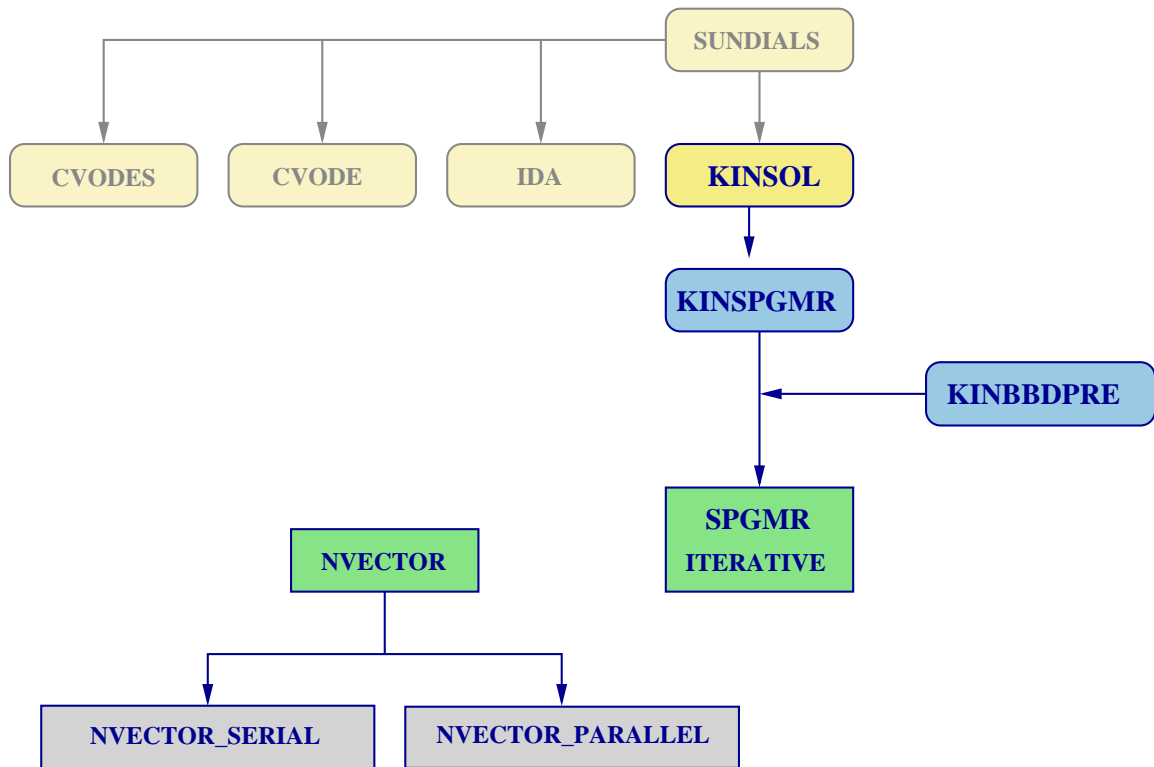


Figure 4.2: Overall structure diagram of the KINSOL package. Modules specific to KINSOL are distinguished by rounded boxes, while generic solver and auxiliary modules are in rectangular boxes. Grayed boxes refer to the encompassing SUNDIALS structure.

A 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.

Linear solver modules are also decomposed in another way. The module KINSPGMR is a set of interface routines built on top of a generic solver module SPGMR. 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 which works in conjunction with NVECTOR_PARALLEL to generate a preconditioner that is a block-diagonal matrix with each block being a band matrix, as further described in §5.6.

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

This section 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 listings of the sample programs in the companion document [4] may also be helpful. Those codes are intended to serve as templates and are included in the KINSOL package.

5.1 Data types

The `sundialtypes.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 §2.2).

Additionally, based on the current precision, `sundialtypes.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 §2.2).

5.2 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 `sundialstypes.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 §6 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 `nvector.h`, which defines the abstract `N_Vector` type.

Finally, a linear solver module header file is required. At the present time, KINSOL offers only a Krylov linear solver, `KINSPGMR`, whose corresponding header file is `kinspgmr.h`. This in turn includes a header file (`iterative.h`) which enumerates the kind of preconditioning and the choices for the Gram-Schmidt process.

Other headers may be needed, according to the choice of preconditioner, etc. For example, in the `kinwebs` example [4], preconditioning is done with a block-diagonal matrix. For this, the header `smalldense.h` is included.

5.3 A Skeleton of the User's Main Program

A high-level view of the combined user program and KINSOL package is shown in Figure 5.1. The following is a skeleton of the user's main program (or calling program) for the solution of a nonlinear problem. Most 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. Initialize MPI

[P] `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 = NV_Make_Serial(N, udata);`

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

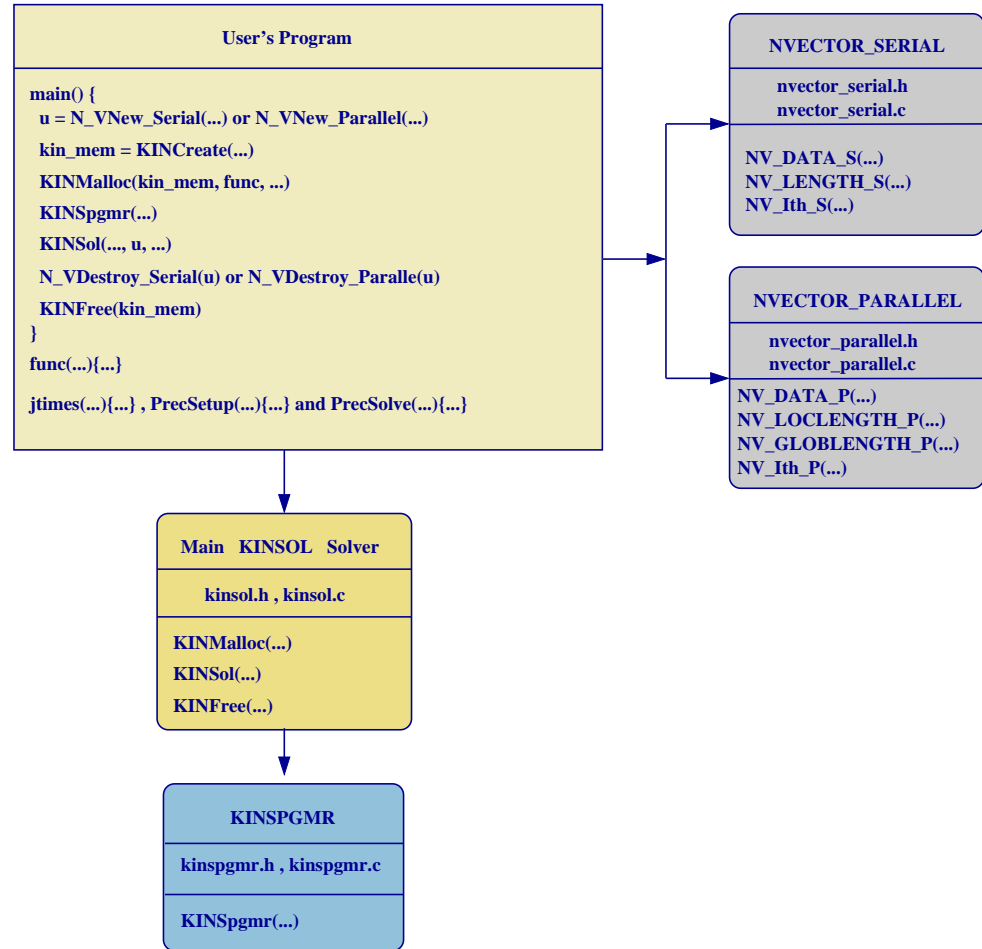


Figure 5.1: Diagram of the user program and KINSOL package for the solution of nonlinear systems

Otherwise, make the call:

[S] `u = NV_New_Serial(N);`

[P] `u = NV_New_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.

5. Set optional inputs

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

6. Allocate internal memory

Call `KINMalloc(...)`; to specify the problem defining function F , allocate internal memory for KINSOL, and initialize KINSOL. `KINMalloc` returns an error flag to indicate success or an illegal argument value (for details see §5.4.1).

7. Attach linear solver module

Initialize the linear solver module by calling `KINSpngr(...)`; to specify the maximum dimension of the Krylov subspace.

8. Set linear solver optional inputs

Call `KINSpngrSet*` routines to change optional inputs for the KINSPGMR linear solver.

9. Solve problem

Call `KINsol(...)`; to solve the nonlinear problem for a given initial guess (see §5.4.3 for details).

10. Get optional outputs

Call `KINGet*` functions to obtain optional output from KINSOL, and call `KINSpngrGet*` functions for optional outputs from KINSPGMR. See §5.4.5.

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] `NV_Destroy_Serial(u);`

[P] `NV_Destroy_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.

5.4 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 §5.4.4, 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 §5.3 for the correct order of these calls.

5.4.1 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 §5.5.1).
`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` also prints an error message to the file specified by the optional input `errfp`.

KINFree

Call `KINFree(kin_mem);`

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

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

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

5.4.2 Linear solver specification function

As previously explained, Newton iteration requires the solution of linear systems of the form (2). At the present time there is only one solver available for this task, KINSPGMR. This is an iterative solver that uses a scaled preconditioned GMRES method.

To attach the KINSPGMR linear solver, after the call to `KINCreate` but before any call to `KINSol`, the user's program must call `KINSpgrmr`, as documented below. The first argument passed to this function is the KINSOL memory pointer returned by `KINCreate`. The call to this function links the linear solver to the main KINSOL memory block and allows the user to specify parameters for KINSPGMR.

The KINSPGMR linear solver is actually built on top of a generic linear system solver, which may be of interest in itself. This generic solver, SPGMR, is described separately in §8.

KINSpgrmr	
Call	<code>flag = KINSpgrmr(kin_mem, maxl);</code>
Description	The function <code>KINSpgrmr</code> selects the KINSPGMR linear solver.
Arguments	<code>kin_mem</code> (<code>void *</code>) pointer to the KINSOL memory block.
	<code>maxl</code> (<code>int</code>) maximum dimension of the Krylov subspace to be used. Pass 0 to use the default value <code>KINSPGMR_MAXL= 5</code> .
Return value	<p>The return value <code>flag</code> (of type <code>int</code>) is one of:</p> <p><code>KINSPGMR_SUCCESS</code> The KINSPGMR initialization was successful.</p> <p><code>KINSPGMR_MEM_NULL</code> The <code>kin_mem</code> pointer is NULL.</p> <p><code>KINSPGMR_ILL_INPUT</code> The NVECTOR module used does not implement a required operation.</p> <p><code>KINSPGMR_MEM_FAIL</code> A memory allocation request failed.</p>
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 §5.4.4 and §5.5.
	If preconditioning is done, user-supplied functions define the right preconditioner matrices P , which approximate the Newton matrix from (2).

5.4.3 KINSOL solver function

KINSol	
Call	<code>flag = KINSol(kin_mem, u, strategy, u_scale, f_scale);</code>
Description	The function <code>KINSol</code> computes an approximate solution of the nonlinear system.
Arguments	<code>kin_mem</code> (<code>void *</code>) pointer to the KINSOL memory block.
	<code>u</code> (<code>N_Vector</code>) vector set to initial guess by user before calling <code>KINSol</code> , but which upon return contains an approximate solution of the nonlinear system $F(u) = 0$ the computed solution vector.
	<code>strategy</code> globalization strategy applied to the Newton method. It must be one of <code>KIN_INEXACT_NEWTON</code> or <code>KIN_LINESEARCH</code> .
	<code>u_scale</code> vector containing diagonal elements of scaling matrix D_u for vector <code>u</code> chosen so that the components of $D_u \cdot u$ (as a matrix multiplication) all have about the same magnitude when <code>u</code> is close to a root of $F(u)$.
	<code>f_scale</code> 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 <code>u</code> 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.

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 §5.4.4 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 $\|D_u p\|_{L_2} > 0.99 \text{mxnewtstep}$, where p denotes the current step and `mxnewtstep` is a scalar upper bound on the scaled step length.

Such a failure may mean that $\|D_F F(u)\|_{L_2}$ 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 solve the preconditioned linear system) encountered an unrecoverable error, or the linear solver routine (`lsolve`) encountered an error condition.

Table 5.1: Optional inputs for KINSOL and KINSPGMR

Optional input	Function name	Default
KINSOL main solver		
Pointer to an error file	KINSetErrFile	stderr
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	KINSetNoPrecInit	FALSE
Max. iterations without prec. setup	KINSetMaxPrecCalls	10
Form of η coefficient	KINSetEtaForm	KIN_ETACHOICE1
Constant value of η	KINSetEtaConstValue	0.1
Values of γ and α	KINSetEtaParams	0.9 and 2.0
Lower bound on ϵ	KINSetNoMinEps	FALSE
Max. scaled length of Newton step	KINSetMaxNewtonStep	$1000\ D_u u_0\ _2$
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
KINSPGMR linear solver		
Max. number of restarts	KINSpgrmrSetMaxRestarts	0
Preconditioner solve function	KINSpgrmrSetPrecSolveFn	NULL
Preconditioner setup function	KINSpgrmrSetPrecSetupFn	NULL
Data for preconditioner functions	KINSpgrmrSetPrecData	NULL
Jacobian-vector product function	KINSpgrmrSetJacTimesVecFn	internal DQ
Data for Jacobian-vector product function	KINSpgrmrSetJacData	NULL

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.

5.4.4 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. For the most casual use of KINSOL, the reader can skip to §5.5.

We note that, on error return, all of these functions also print an error message to `stderr` (or to the file pointed to by `errfp` if already specified). We also note that all error return values are negative, so a test `flag < 0` will catch any error.

Main solver optional input functions

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

KINSetErrFile

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

Description	The function <code>KINSetErrFile</code> specifies the pointer to the file where all KINSOL error messages should be directed.
Arguments	<code>kin_mem</code> (<code>void *</code>) pointer to the KINSOL memory block. <code>errfp</code> (<code>FILE *</code>) pointer to output file.
Return value	The return value <code>flag</code> (of type <code>int</code>) is one of: <code>KIN_SUCCESS</code> The optional value has been successfully set. <code>KIN_MEM_NULL</code> The <code>kin_mem</code> pointer is <code>NULL</code> .
Notes	The default value for <code>errfp</code> is <code>stderr</code> . Passing a value <code>NULL</code> disables all future error message output (except for the case in which the KINSOL memory pointer is <code>NULL</code>).

`KINSetInfoFile`

Call	<code>flag = KINSetInfoFile(kin_mem, infofp);</code>
Description	The function <code>KINSetInfoFile</code> specifies the pointer to the file where all informative messages should be directed.
Arguments	<code>kin_mem</code> (<code>void *</code>) pointer to the KINSOL memory block. <code>infofp</code> (<code>FILE *</code>) pointer to output file.
Return value	The return value <code>flag</code> (of type <code>int</code>) is one of: <code>KIN_SUCCESS</code> The optional value has been successfully set. <code>KIN_MEM_NULL</code> The <code>kin_mem</code> pointer is <code>NULL</code> .
Notes	The default value for <code>infofp</code> is <code>stderr</code> .

`KINSetPrintLevel`

Call	<code>flag = KINSetPrintLevel(kin_mem, printf1);</code>
Description	The function <code>KINSetPrintLevel</code> specifies the level of verbosity of the output.
Arguments	<code>kin_mem</code> (<code>void *</code>) pointer to the KINSOL memory block. <code>printf1</code> (<code>int</code>) flag indicating the level of verbosity. Must be one of: <ul style="list-style-type: none"> 0 no information displayed. 1 for each nonlinear iteration display the following information: the scaled Euclidean ℓ_2 norm of the system function evaluated at the current iterate, the scaled norm of the Newton step (only if using <code>KIN_INEXACT_NEWTON</code>), 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 <code>KIN_INEXACT_NEWTON</code>). $\ F(u)\ _{D_F,\infty}$ (for <code>KIN_INEXACT_NEWTON</code> and <code>KIN_LINESEARCH</code>). 3 display level 2 output plus additional values used by the global strategy (only if using <code>KIN_LINESEARCH</code>), and statistical information for the linear solver.
Return value	The return value <code>flag</code> (of type <code>int</code>) is one of: <code>KIN_SUCCESS</code> The optional value has been successfully set. <code>KIN_MEM_NULL</code> The <code>kin_mem</code> pointer is <code>NULL</code> . <code>KIN_ILL_INPUT</code> The argument <code>printf1</code> had an illegal value.
Notes	The default value for <code>printf1</code> 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`.

KINSetNoPrecInit

Call `flag = KINSetNoPrecInit(kin_mem, noPrecInit);`

Description The function `KINSetNoPrecInit` 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.
`noPrecInit` (`boolean` type) 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 `noPrecInit` is `FALSE`, meaning that an initial call to the preconditioner setup function will be made.

KINSetMaxPrecCalls

Call `flag = KINSetMaxPrecCalls(kin_mem, msbpre);`

Description The function `KINSetMaxPrecCalls` 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.
`msbpre` (`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 `msbpre` was negative.

Notes The default value for `msbpre` is `MSBPRES = 10`.

KINSetEtaForm

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

Description The function `KINSetEtaForm` specifies the method for computing the value of the η 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 η . `etachoice` must be one of `KIN_ETACHOICE1`, `KIN_ETACHOICE2`, or `KIN_ETACONSTANT` (see §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 η in the case `etachoice = KIN_ETACONSTANT`.

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

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 $0.0 < \mathbf{eta} \leq 1.0$.

KINSetEtaParams

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

Description The function `KINSetEtaParams` specifies the parameters γ and α in the formula for η , in the case `etachoice = KIN_ETACHOICE2`.

Arguments `kin_mem` (void *) pointer to the KINSOL memory block.
`egamma` (realtype) value of the γ parameter.
`ealpha` (realtype) value of the α 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 $1.0 < \text{ealpha} \leq 2.0$. If `ealpha` = 0.0, then its value is set to 2.0.
- The valid values for `egamma` are $0.0 < \text{egamma} \leq 1.0$. If `egamma` = 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 ϵ , 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 ϵ .
- 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.

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	<code>flag = KINSetFuncNormTol(kin_mem, fnormtol);</code>
Description	The function <code>KINSetFuncNormTol</code> specifies the scalar used as a stopping tolerance on the scaled maximum norm of the system function $F(u)$.
Arguments	<code>kin_mem</code> (void *) pointer to the KINSOL memory block. <code>fnormtol</code> (realtype) tolerance for stopping based on scaled function norm.
Return value	The return value <code>flag</code> (of type <code>int</code>) is one of: <code>KIN_SUCCESS</code> The optional value has been successfully set. <code>KIN_MEM_NULL</code> The <code>kin_mem</code> pointer is NULL. <code>KIN_ILL_INPUT</code> The tolerance was non-positive.
Notes	The default value for <code>fnormtol</code> is $\sqrt[3]{\text{unit roundoff}}$.

KINSetScaledStepTol

Call	<code>flag = KINSetScaledStepTol(kin_mem, scsteptol);</code>
Description	The function <code>KINSetScaledStepTol</code> specifies the scalar used as a stopping tolerance on the minimum scaled step length.
Arguments	<code>kin_mem</code> (void *) pointer to the KINSOL memory block. <code>scsteptol</code> (realtype) tolerance for stopping based on scaled step length..
Return value	The return value <code>flag</code> (of type <code>int</code>) is one of: <code>KIN_SUCCESS</code> The optional value has been successfully set. <code>KIN_MEM_NULL</code> The <code>kin_mem</code> pointer is NULL. <code>KIN_ILL_INPUT</code> The tolerance was non-positive.
Notes	The default value for <code>scsteptol</code> is $(\text{unit roundoff})^{2/3}$.

KINSetConstraints

Call	<code>flag = KINSetConstraints(kin_mem, constraints);</code>
Description	The function <code>KINSetConstraints</code> specifies a vector that defines inequality constraints for each component of the solution vector u .
Arguments	<code>kin_mem</code> (void *) pointer to the KINSOL memory block. <code>constraints</code> (N_Vector) vector of constraint flags. If <code>constraints[i]</code> is 0.0 then no constraint is imposed on u_i . 1.0 then u_i will be constrained to be $u_i > 0.0$. -1.0 then u_i will be constrained to be $u_i < 0.0$. 2.0 then u_i will be constrained to be $u_i \geq 0.0$. -2.0 then u_i will be constrained to be $u_i \leq 0.0$.
Return value	The return value <code>flag</code> (of type <code>int</code>) is one of: <code>KIN_SUCCESS</code> The optional value has been successfully set. <code>KIN_MEM_NULL</code> The <code>kin_mem</code> pointer is NULL.
Notes	The presence of a non-NULL constraints vector that is not 0.0 in all components will cause constraint checking to be performed.

KINSetSysFunc

Call	<code>flag = KINSetSysFunc(kin_mem, func);</code>
Description	The function <code>KINSetSysFunc</code> specifies the user-provided function that evaluates the nonlinear system function $F(u)$.
Arguments	<code>kin_mem</code> (void *) pointer to the KINSOL memory block. <code>func</code> (KINSysFn) user-supplied function that evaluates $F(u)$.
Return value	The return value <code>flag</code> (of type <code>int</code>) is one of: <code>KIN_SUCCESS</code> The optional value has been successfully set. <code>KIN_MEM_NULL</code> The <code>kin_mem</code> pointer is NULL. <code>KIN_ILL_INPUT</code> The argument <code>func</code> was NULL.
Notes	The nonlinear system function is initially specified through <code>KINMalloc</code> . 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.

Linear solver optional input functions

The KINSPGMR linear solver module allows for various optional inputs, which are described here. The call to `KINSpgrmr` is used to communicate the maximum dimension of the Krylov subspace to be used (`maxl`).

If preconditioning is to be done within the SPGMR method, then the user must supply a preconditioner solve function `psolve` and specify it through a call to `KINSpgrmrSetPrecSolveFn`. 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 §5.5. If used, the `psetup` function should be specified through a call to `KINSpgrmrSetPrecSetupFn`. Optionally, the KINSPGMR solver passes the pointer it receives through `KINSpgrmrSetPrecData` to the preconditioner setup and solve 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 `prec_data` may be identical to `f_data`, if the latter was specified through `KINSetFdata`.

The KINSPGMR solver requires 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-vector product approximation function, or use the difference quotient function `KINSpgrmrDQJtimes` that comes with the KINSPGMR solver. A user-defined Jacobian-vector function must be of type `KINSpgrmrJtimesFn` and can be specified through a call to `KINSpgrmrSetJacTimesVecFn` (see §5.5 for specification details). As with the preconditioner user data structure `prec_data`, the user can specify, through a call to `KINSpgrmrSetJacData`, a pointer to a user-defined data structure, `jac_data`, which the KINSPGMR solver passes to the Jacobian-vector product function `jtimes` each time it is called. The pointer `jac_data` may be identical to `prec_data` and/or `f_data`.

KINSpgrmrSetMaxRestarts

Call	<code>flag = KINSpgrmrSetMaxRestarts(kin_mem, maxrs);</code>
Description	The function <code>KINSpgrmrSetMaxRestarts</code> specifies the maximum number of times the SPGMR linear solver can be restarted.
Arguments	<code>kin_mem</code> (void *) pointer to the KINSOL memory block. <code>maxrs</code> (int) maximum number of restarts.
Return value	The return value <code>flag</code> (of type <code>int</code>) is one of: <code>KINSPGMR_SUCCESS</code> The optional value has been successfully set. <code>KINSPGMR_ILL_INPUT</code> The maximum number of restarts specified is negative. <code>KINSPGMR_MEM_NULL</code> The <code>kin_mem</code> pointer is NULL. <code>KINSPGMR_LMEM_NULL</code> The KINSPGMR linear solver has not been initialized.

KINSpGmrSetPrecSolveFn

Call `flag = KINSpGmrSetPrecSolveFn(kin_mem, psolve);`

Description The function `KINSpGmrSetPrecSolveFn` specifies the preconditioner solve function.

Arguments `kin_mem` (void *) pointer to the KINSOL memory block.
`psolve` (`KINSpGmrPrecSolveFn`) user-defined preconditioner solve function.

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

- `KINSPGMR_SUCCESS` The optional value has been successfully set.
- `KINSPGMR_MEM_NULL` The `kin_mem` pointer is NULL.
- `KINSPGMR_LMEM_NULL` The `KINSPGMR` linear solver has not been initialized.

Notes The function type `KINSpGmrPrecSolveFn` is described in §5.5.3.

KINSpGmrSetPrecSetupFn

Call `flag = KINSpGmrSetPrecSetupFn(kin_mem, psetup);`

Description The function `KINSpGmrSetPrecSetupFn` specifies the preconditioner preprocessing function.

Arguments `kin_mem` (void *) pointer to the KINSOL memory block.
`psetup` (`KINSpGmrPrecSetupFn`) user-defined preconditioner setup function.

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

- `KINSPGMR_SUCCESS` The optional value has been successfully set.
- `KINSPGMR_MEM_NULL` The `kin_mem` pointer is NULL.
- `KINSPGMR_LMEM_NULL` The `KINSPGMR` linear solver has not been initialized.

Notes The function type `KINSpGmrPrecSetupFn` is described in §5.5.4.

KINSpGmrSetPrecData

Call `flag = KINSpGmrSetPrecData(kin_mem, prec_data);`

Description The function `KINSpGmrSetPrecData` specifies the data structure to be passed to the user supplied preconditioner setup and solve functions each time they are called.

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

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

- `KINSPGMR_SUCCESS` The optional value has been successfully set.
- `KINSPGMR_MEM_NULL` The `kin_mem` pointer is NULL.
- `KINSPGMR_LMEM_NULL` The `KINSPGMR` linear solver has not been initialized.

KINSpGmrSetJacTimesVecFn

Call `flag = KINSpGmrSetJacTimesVecFn(kin_mem, jtimes);`

Description The function `KINSpGmrSetJacTimesVecFn` specifies the Jacobian-vector product function to be used.

Arguments `kin_mem` (void *) pointer to the KINSOL memory block.
`jtimes` (`KINSpGmrJacTimesVecFn`) user-defined Jacobian-vector product function.

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

- `KINSPGMR_SUCCESS` The optional value has been successfully set.
- `KINSPGMR_MEM_NULL` The `kin_mem` pointer is NULL.

Table 5.2: Optional outputs from KINSOL and KINSPGMR

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 β -condition failures	KINGetNumBetaCondFails
Number of backtrack operations	KINGetNumBacktrackOps
Scaled norm of F	KINGetFuncNorm
Scaled norm of the step	KINGetStepLength
KINSPGMR linear solver	
Size of KINSPGMR real and integer workspaces	KINSpgrmrGetWorkSpace
No. of linear iterations	KINSpgrmrGetNumLinIters
No. of linear convergence failures	KINSpgrmrGetNumConvFails
No. of preconditioner evaluations	KINSpgrmrGetNumPrecEvals
No. of preconditioner solves	KINSpgrmrGetNumPrecSolves
No. of Jacobian-vector product evaluations	KINSpgrmrGetNumJtimesEvals
No. of fct. calls for finite diff. Jacobian-vector evals.	KINSpgrmrGetNumFuncEvals
Last return from a KINSPGMR function	KINSpgrmrGetLastFlag

KINSPGMR_LMEM_NULL The KINSPGMR linear solver has not been initialized.

Notes By default, KINSPGMR uses the difference quotient function KINSpgrmrDQJtimes. If NULL is passed to jtimes, this default function is used.

The function type KINSpgrmrJacTimesVecFn is described in §5.5.2.

KINSpgrmrSetJacData

Call `flag = KINSpgrmrSetJacData(kin_mem, jac_data);`

Description The function KINSpgrmrSetJacData specifies the data structure to be passed to the user supplied Jacobian-vector function each time it is called.

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

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

KINSPGMR_SUCCESS The optional value has been successfully set.

KINSPGMR_MEM_NULL The `kin_mem` pointer is NULL.

KINSPGMR_LMEM_NULL The KINSPGMR linear solver has not been initialized.

5.4.5 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.

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.

KINGGetWorkSpace

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

Description The function `KINGGetWorkSpace` 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 $5N$ **realtype** words.

KINGGetNumFuncEvals

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

Description The function `KINGGetNumFuncEvals` 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.

KINGGetNumNonlinSolvIters

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

Description The function `KINGGetNumNonlinSolvIters` 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.

KINGGetNumBetaCondFails

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

Description The function `KINGGetNumBetaCondFails` returns the number of β -condition failures.

Arguments `kin_mem` (void *) pointer to the KINSOL memory block.
 `nbcfails` (long int) number of β -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 ℓ_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 ℓ_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.

Linear solver optional output functions

The functions available to access various optional outputs that describe the performance of the KINSPGMR module are described below.

KINSpgrmrGetWorkSpace

Call `flag = KINSpgrmrGetWorkSpace(kin_mem, &lenrwSG, &leniwSG);`

Description The function `KINSpgrmrGetWorkSpace` returns the real and integer workspace sizes used by KINSPGMR.

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

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

Notes KINSPGMR_LMEM_NULL The KINSPGMR linear solver has not been initialized.
 In terms of the problem size N and maximum subspace size `maxl`, the actual size of the real workspace is $(\text{maxl}+5) * N + \text{maxl} * (\text{maxl}+4) + 1$ `realtype` words. (In a parallel setting, this value is global - summed over all processes.)

KINSpgrmrGetNumLinIters

Call `flag = KINSpgrmrGetNumLinIters(kin_mem, &nliters);`
Description The function `KINSpgrmrGetNumLinIters` 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:
 KINSPGMR_SUCCESS The optional output value has been successfully set.
 KINSPGMR_MEM_NULL The `kin_mem` pointer is NULL.
 KINSPGMR_LMEM_NULL The KINSPGMR linear solver has not been initialized.

KINSpgrmrGetNumConvFails

Call `flag = KINSpgrmrGetNumConvFails(kin_mem, &nlcfails);`
Description The function `KINSpgrmrGetNumConvFails` 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:
 KINSPGMR_SUCCESS The optional output value has been successfully set.
 KINSPGMR_MEM_NULL The `kin_mem` pointer is NULL.
 KINSPGMR_LMEM_NULL The KINSPGMR linear solver has not been initialized.

KINSpgrmrGetNumPrecEvals

Call `flag = KINSpgrmrGetNumPrecEvals(kin_mem, &npevals);`
Description The function `KINSpgrmrGetNumPrecEvals` 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:
 KINSPGMR_SUCCESS The optional output value has been successfully set.
 KINSPGMR_MEM_NULL The `kin_mem` pointer is NULL.
 KINSPGMR_LMEM_NULL The KINSPGMR linear solver has not been initialized.

KINSpgrmrGetNumPrecSolves

Call `flag = KINSpgrmrGetNumPrecSolves(kin_mem, &npsolves);`
Description The function `KINSpgrmrGetNumPrecSolves` 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:

KINSPGMR_SUCCESS The optional output value has been successfully set.
 KINSPGMR_MEM_NULL The `kin_mem` pointer is NULL.
 KINSPGMR_LMEM_NULL The KINSPGMR linear solver has not been initialized.

KINSpgmrGetNumJtimesEvals

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

Description The function `KINSpgmrGetNumJtimesEvals` 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:

KINSPGMR_SUCCESS The optional output value has been successfully set.
 KINSPGMR_MEM_NULL The `kin_mem` pointer is NULL.
 KINSPGMR_LMEM_NULL The KINSPGMR linear solver has not been initialized.

KINSpgmrGetNumRhsEvals

Call `flag = KINSpgmrGetNumRhsEvals(kin_mem, &nfevalsSG);`

Description The function `KINSpgmrGetNumRhsEvals` returns the number of calls to the user right-hand side function for finite difference Jacobian-vector product approximations.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`nfevalsSG` (`long int`) the number of calls to the user right-hand side function.

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

KINSPGMR_SUCCESS The optional output value has been successfully set.
 KINSPGMR_MEM_NULL The `kin_mem` pointer is NULL.
 KINSPGMR_LMEM_NULL The KINSPGMR linear solver has not been initialized.

Notes The value `nfevalsSG` is incremented only if the default `KINSpgmrDQJtimes` difference quotient function is used.

KINSpgmrGetLastFlag

Call `flag = KINSpgmrGetLastFlag(kin_mem, &flag);`

Description The function `KINSpgmrGetLastFlag` returns the last return value from a KINSPGMR routine.

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

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

KINSPGMR_SUCCESS The optional output value has been successfully set.
 KINSPGMR_MEM_NULL The `kin_mem` pointer is NULL.
 KINSPGMR_LMEM_NULL The KINSPGMR linear solver has not been initialized.

Notes If the KINSPGMR setup function failed (KINSOL returned `KIN_LSETUP_FAIL`), `flag` contains the return value of the preconditioner setup function `psetup`.

If the KINSPGMR solve function failed (KINSOL returned `KIN_LSOLVE_FAIL`), `flag` contains the error return flag from `SpgmrSolve` and will be one of: `SPGMR_CONV_FAIL`, indicating a failure to converge; `SPGMR_QRFACT_FAIL`, indicating a singular matrix found

during the QR factorization; `SPGMR_PSOLVE_FAIL_REC`, indicating that the preconditioner solve function `psolve` failed recoverably; `SPGMR_MEM_NULL`, indicating that the SPGMR memory is NULL; `SPGMR_ATIMES_FAIL`, indicating a failure in the Jacobian-vector product 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.

5.5 User-supplied functions

The user-supplied functions consist of one function defining the nonlinear system, (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 the SPGMR algorithm.

5.5.1 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 type does not have a return value.
Notes Allocation of memory for `fval` is handled within `KINSOL`.

5.5.2 Jacobian information (SPGMR matrix-vector product)

The user may provide a function of type `KINSpgrmrJacTimesVecFn` to evaluate Jacobian-vector products for the `KINSPGMR` linear solver module. This function has the following form:

KINSpgrmrJacTimesVecFn

Definition `typedef int (*KINSpgrmrJacTimesVecFn)(N_Vector v, N_Vector Jv,
N_Vector u, booleantype *new_u,
void *jac_data);`
Purpose This function computes the product $Jv = (\partial F / \partial u)v$ (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 (unscaled) value of the iterate.
`new_u` is a flag (reset by user) indicating if the iterate u has been updated in the interim. The Jacobian-vector product 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 `KINSpgrmrSetJacData`.
Return value The value to be returned by the Jacobian-vector product function should be 0 if successful. Any other return value will result in an unrecoverable error of the SPGMR generic solver, in which case the solution process is halted.

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

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

5.5.3 Preconditioning (SPGMR 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 KINSpgrPrecSolveFn, defined as follows:

KINSpgrPrecSolveFn

Definition `typedef int (*KINSpgrPrecSolveFn)(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 KINSpgrSetPrecData.
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.

5.5.4 Preconditioning (SPGMR 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 KINSpgrPrecSetupFn, defined as follows:

KINSpgrPrecSetupFn

Definition `typedef int (*KINSpgrPrecSetupFn)(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 KINSpgrPrecSetupFn 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**.

	<code>prec_data</code> is a pointer to user data - the same as the <code>prec_data</code> parameter passed to the function <code>KINSpgrmrSetPrecData</code> .
	<code>tmp1</code>
	<code>tmp2</code> are pointers to memory allocated for variables of type <code>N_Vector</code> which can be used by <code>KINSpgrmrPrecSetupFn</code> 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, positive for a recoverable error, and negative for an unrecoverable error.
Notes	The user-supplied preconditioner setup subroutine should compute the right preconditioner matrix P (stored in the memory block referenced by the <code>prec_data</code> 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.

5.6 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 `KINBDPRE`, to be used with the parallel `N_Vector` module described in §6.2.

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 $F(u)$. The blocks are generated by each process via a difference quotient scheme, utilizing the assumed banded structure with given half-bandwidths.

The `KINBDPRE` module calls two user-provided functions to construct P : a required function `Gloc` (of type `KINLocalFn`) which approximates the nonlinear system 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	<pre>typedef void (*KINLocalFn)(long int Nlocal, N_Vector u, N_Vector gval, void *f_data);</pre>
Purpose	This function computes $G(u)$, and outputs the resulting vector as <code>gval</code> .
Arguments	<code>Nlocal</code> is the local vector length. <code>u</code> 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 §5.2), 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 §5.3 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-bandwidths `mu`, `m1` and call

```
bbd_data = KINBBDPrecAlloc(kin_mem, Nlocal, mu, m1,
                           dq_rel_u, Gloc, Gcomm);
```

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

8. Attach the KINSPGMR linear solver

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

The function `KINBBDSpgmr` is a wrapper around the KINSPGMR specification function `KINSpgrmr` and performs the following actions:

- Attaches the KINSPGMR linear solver to the main CVODE 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 `KINBBDSpgmr` 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 KINSPGMR 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 three user-callable functions that initialize, attach, and deallocate the KINBBDPRE preconditioner module (steps 7, 8, and 13 above) are described next.

KINBBDPrecAlloc

Call `bbd_data = KINBBDPrecAlloc(kin_mem, Nlocal, mu, ml, 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.
 `mu` (long int) upper half-bandwidth to be used in the difference quotient Jacobian approximation.
 `ml` (long int) lower half-bandwidth to be used in the difference quotient Jacobian approximation.
 `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 The half-bandwidths `mu` and `m1` need not be the true half-bandwidths of the Jacobian of the local block of G , when smaller values may provide a greater efficiency.

Moreover, the half-bandwidth 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 `KINSPGMR_MAXL=5`.
`bbd_data` (`void *`) pointer to the KINBBDPRE data structure.

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

`KINSPGMR_SUCCESS` The KINSPGMR initialization was successful.
`KINSPGMR_MEM_NULL` The `kin_mem` pointer is `NULL`.
`KINSPGMR_ILL_INPUT` The NVECTOR module used does not implement a required operation.
`KINSPGMR_MEM_FAIL` A memory allocation request failed.
`KIN_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:

`KIN_SUCCESS` The optional output values have been successfully set.
`KIN_PDATA_NULL` The KINBBDPRE preconditioner has not been initialized.

KINBBDPrecGetNumGfnEvals

Call	<code>flag = KINBBDPrecGetNumGfnEvals(bbd_data, &ngevalsBBDP);</code>
Description	The function <code>KINBBDPrecGetNumGfnEvals</code> returns the number of calls to the user <code>Gloc</code> function due to the finite difference approximation of the Jacobian blocks used within <code>KINBBDPRE</code> 's preconditioner setup function.
Arguments	<code>bbd_data</code> (void *) pointer to the <code>KINBBDPRE</code> data structure. <code>ngevalsBBDP</code> (long int) the number of calls to the user <code>Gloc</code> function.
Return value	The return value <code>flag</code> (of type <code>int</code>) is one of: <code>KIN_SUCCESS</code> The optional output value has been successfully set. <code>KIN_PDATA_NULL</code> The <code>KINBBDPRE</code> preconditioner has not been initialized.

5.7 FKINSOL, a FORTRAN-C interface module

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, and may also be used with user-supplied mixed FORTRAN/C `NVECTOR` implementations.

5.7.1 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 `NV_New_Serial`.
 - `FNVINITP` (defined by `NVECTOR_PARALLEL`) interfaces to `NV_New_Parallel`.
 - `FNVFREES` (defined by `NVECTOR_SERIAL`) interface to `NV_Destroy_Serial`.
 - `FNVFREEP` (defined by `NVECTOR_PARALLEL`) interfaces to `NV_Destroy_Parallel`.
- Interface to the main `KINSOL` module
 - `FKINMALLOC` interfaces to `KINCreate`, `KINSet*` functions, and `KINMalloc`.
 - `FKINSOL` interfaces to `KINSol`, `KINGet*` functions, and to the optional output functions for the `KINSPGMR` linear solver module.
 - `FKINFREE` interfaces to `KINFree`.
- Interface to the `KINSPGMR` solver module
 - `FKINSPGMR` interfaces to `KINSpgmr` and `SPGMR` optional input functions.
 - `FKINSPGMRSETJAC` interfaces to `KINSpgmrSetJacTimesVecFn`.
 - `FKINSPGMRSETPSOL` interfaces to `KINSpgmrSetPrecSolveFn`.
 - `FKINSPGMRSETPSET` interfaces to `KINSpgmrSetPrecSetupFn`.

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
FKPSOL	FKINPSol	KINSpgrmrPrecSolveFn
FKPSET	FKINPSet	KINSpgrmrPrecSetupFn
FKJTIMES	FKINJtimes	KINSpgrmrJacTimesVecFn

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.

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 `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 `config.h` when configuring SUNDIALS via the `--with-f77underscore` and `--with-f77case` options 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 (`IOPT`), 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.

5.7.2 FKINSOL optional input and output

In order to keep the number of user-callable FKINSOL interface routines to a minimum, optional inputs and outputs to the KINSOL solver and to related modules are not accessed through individual functions, but rather through a pair of arrays, `IOPT` of integer type and `ROPT` of real type. Table 5.3 lists the entries in these two arrays and specifies the FKINSOL user-callable routine which sets/accesses the corresponding optional variable, as well as the KINSOL optional function which is actually called. For more details on the optional inputs and outputs, see §5.4.4 and §5.4.5.

5.7.3 Usage of the FKINSOL interface module

The usage of FKINSOL requires calls to several 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. The usage of FKINSOL with the band-block-diagonal preconditioner module `KINBBDPRE` is described in the next subsection.

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)
  DIMENSION U(*), FVAL(*)
```

Table 5.3: Description of the FKINSOL optional input-output arrays IOPT and ROPT

Integer input-output array IOPT

Index	Optional input	Optional output	KINSOL function
KINSOL main solver			
1	PRINTFL		KINSetPrintLevel
2	MXITER		KINSetNumMaxIters
3	PRECOND_NO_INIT		KINSetNoPrecInit
4			KINGetNumNonlinSolvIters
5			KINGetNumFuncEvals
6			KINGetNumBetaCondFails
7			KINGetNumBacktrackOps
8	ETACHOICE		KINSetEtaForm
9	NO_MIN_EPS		KINSetNoMinEps
KINSPGMR linear solver			
11		NLI	KINSpgmrGetNumLinIters
12		NPE	KINSpgmrGetNumPrecEvals
13		NPS	KINSpgmrGetNumPrecSolves
14		NCFL	KINSpgmrGetNumConvFails
15		LS_FLAG	KINSpgmrGetLastFlag

Real input-output array ROPT

Index	Optional input	Optional output	KINSOL function
1	MXNEWTSTEP	FNORM STEPL	KINSetMaxNewtonStep
2	RELFUNC		KINSetRelErrFunc
3			KINGetFuncNorm
4			KINGetStepLength
5	ETACONST		KINSetEtaConstValue
6	ETAGAMMA		KINSetEtaParams
7	ETAALPHA		KINSetEtaParams

It must set the **FVAL** array to $F(u)$, the system function, as a function of the array **U**. Here **U** and **FVAL** are arrays representing vectors, which are distributed vectors in the parallel case.

2. NVECTOR module initialization

[S] To initialize the serial NVECTOR module, the user must make the following call:

```
CALL FNVINITS(NEQ, IER)
```

where **NEQ** is the size of vectors and **IER** is a return completion flag which is set to 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(NLOCAL, NGLOBAL, IER)
```

in which the arguments are: **NLOCAL** the local size of vectors for this process, **NGLOBAL** the system size (and the global size of vectors, that is the sum of all values of **NLOCAL**). The return completion flag **IER** is set to 0 upon successful return and to -1 otherwise. Note that if MPI was initialized by the user, the communicator must be set to **MPI_COMM_WORLD**. If not, this routine initializes MPI and sets the communicator equal to **MPI_COMM_WORLD**.

3. Problem specification

To set various problem and solution parameters and allocate internal memory, make the following call:

FKINMALLOC

Call	CALL FKINMALLOC(MSBPRE, FNORMTOL, SCSTEPTOL, CONSTRAINTS, & OPTIN, IOPT, ROPT, IER)	
Description	This function provides required problem and solution specifications, specifies optional inputs, allocates internal memory, and initializes KINSOL.	
Arguments	MSBP	is the maximum number of preconditioning solve calls without calling the preconditioning setup routine. A value of 0 indicates the default.
	FNORMTOL	is the tolerance on the scaled maximum norm of $F(u)$ to accept convergence.
	SCSTEPTOL	is the tolerance on minimum scaled step size.
	CONSTRAINTS	is an array of constraint values on the components of the solution u .
	INOPT	is an integer flag indicating whether possible input values in IOPT are to be used for input. A value of 0 means <i>no</i> and a value of 1 indicates <i>yes</i> .
	IOPT	is an array of integer optional inputs and outputs (must be declared as INTEGER*4 or INTEGER*8 according to the C type long int).
	ROPT	is an array of real optional inputs and outputs.
Return value	IER is the return completion flag. Its possible values are 0 indicating success or -1 indicating failure.	
Notes	The optional inputs and outputs associated with the main KINSOL integrator are listed in Table 5.3. If any of the optional inputs are used, the others must be set to zero to indicate default values.	

4. Linear solver specification

The solution method in KINSOL involves the solution of linear systems related to the Jacobian of the nonlinear system.

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 (0 indicates default). **MAXLRST** is the maximum number of linear system restarts (0 indicates default). **IER** is the return completion flag (possible values are 0: success and -1: failure).

As an option when using the SPGMR linear solver, 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 FKJTIMES(V, Z, NEWU, U, IER)
  DIMENSION V(*), Z(*), U(*)
```

This must set the array **Z** to the product Jv , where J is the Jacobian matrix $J = \partial F / \partial u$, and **V** is a given array. Here **U** is an array containing the current value of the unknown vector u . **NEWU** is an input integer indicating whether **U** has changed since **FKJTIMES** was last called (1 = yes, 0 = no). If **FKJTIMES** computes and saves Jacobian data, then no such computation is necessary when **NEWU** = 0. The arguments **V**, **Z**, and **U** are arrays of length **NEQ**, the problem size, or the local length of all distributed vectors in the parallel case. **FKJTIMES** should return **IER** = 0 if successful, or a nonzero **IER** otherwise.

If the user program includes the **FKJTIMES** routine for the evaluation of the Jacobian vector product, the following call must be made:

```
CALL FKINSPGMRSETJAC(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 can be 0 for success or nonzero if an error occurred.

If preconditioning is to be done then, following the call to **FKINSPGMR**, the user must call

```
CALL FKINSPGMRSETPSOL(FLAG, IER)
```

with **FLAG** \neq 0, and the user program must include the following routine 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 $r = \mathbf{VTEM}$ 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, following the call to **FKINSPGMRSETPSOL**, the user must call

```
CALL FKINSPGMRSETPSET(FLAG, IER)
```

with **FLAG** \neq 0. In this case, the user program must also include the following routine 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.

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.

The current values of the optional outputs are available in IOPT and ROPT (see Table 5.3).

6. **Memory deallocation** To free the internal memory created by the call to FKINMALLOC, make the call

```
CALL FKINFREE
```

and then, depending on the NVECTOR version (serial or parallel), either

```
CALL FNVFREES
```

or

```
CALL FNVFREEP
```

respectively.

5.7.4 Usage of the FKINBBB interface to KINBBDPRE

The FKINBBB 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 §5.6), 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.
- 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):

FKINBBD routine (FORTRAN)	KINSOL function (C)	KINSOL function type
FKLOCFN	FKINGloc	KINLocalFn
FKCOMMF	FKINGcomm	KINCommFn
FKJTIMES	FKINJtimes	KINSpgrmrJacTimesVecFn

As with the rest of the FKINSOL routines, the names of the user-supplied routines are mapped to actual values through a series of definitions in the header file `fkinbbd.h` (see §5.7).

The following is a summary of the usage of this module. Steps that are unchanged from the main program described in §5.7.3 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, MU, ML, IER)
```

The arguments are as follows. `NLOCAL` is the local size of vectors for this process. `MU` and `ML` 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. `IER` is a return completion flag. A 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 §5.7.3).

Optionally, to specify that SPGMR should use the supplied FKJTIMES, make the call

```
CALL FKINSPGMRSETJAC(FLAG, IER)
```

with `FLAG` $\neq 0$.

5. Problem solution
6. KINBBDPRE Optional outputs

To obtain the optional outputs associated with the KINBBDPRE module, make the following call:

```
CALL FKINBBDOPT(LENRPW, LENIPW, NGE)
```

The arguments returned are as follows. `LENRPW` is the length of real preconditioner work space, in `realtype` words. This size is local to the current process. `LENIPW` is the length of integer preconditioner work space, in integer words. This size is local to the current process. `NGE` is the cumulative number of $G(u)$ evaluations (calls to FKLOCFN).

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
```

Chapter 6

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);  
    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 (*nvintest)(N_Vector, N_Vector);
booleantype (*nvconstrmask)(N_Vector, N_Vector, N_Vector);
realtype    (*nvminquotient)(N_Vector, N_Vector);
};

```

The generic NVECTOR module also 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 6.1 contains a complete list of all vector operations defined by the generic NVECTOR module.

Finally, note that the generic NVECTOR module defines a function **N_VCloneVectorArray** which creates (by cloning) an array of **count** variables of type **N_Vector**, each of the same type as an existing **N_Vector**. Its prototype is

```
N_Vector *N_VCloneVectorArray(int count, N_Vector w);
```

and its definition is based on the implementation-specific **N_VClone** operation. 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 6.1: Description of the NVECTOR operations

Name	Usage and Description
N_VClone	$v = \text{N_VClone}(w);$ Creates a new N_Vector of the same type as an existing vector w and sets the <i>ops</i> field. It does not copy the vector, but rather allocates storage for the new vector.
N_VDestroy	$\text{N_VDestroy}(v);$ Destroys the N_Vector v and frees memory allocated for its internal data.
N_VSpace	$\text{N_VSpace}(nvSpec, \&lrw, \&liw);$ Returns storage requirements for one N_Vector . lrw contains the number of realtype words and liw contains the number of integer words.
N_VGetArrayPointer	$vdata = \text{N_VGetArrayPointer}(v);$ 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	$\text{N_VSetArrayPointer}(vdata, v);$ 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	$\text{N_VLinearSum}(a, x, b, y, z);$ Performs the operation $z = ax + by$, where <i>a</i> and <i>b</i> are scalars and <i>x</i> and <i>y</i> are of type N_Vector : $z_i = ax_i + by_i, i = 0, \dots, n-1$.
N_VConst	$\text{N_VConst}(c, z);$ Sets all components of the N_Vector z to c : $z_i = c, i = 0, \dots, n-1$.
N_VProd	$\text{N_VProd}(x, y, z);$ Sets the N_Vector z to be the component-wise product of the N_Vector inputs x and y : $z_i = x_i y_i, i = 0, \dots, n-1$.
N_VDiv	$\text{N_VDiv}(x, y, z);$ Sets the N_Vector z to be the component-wise ratio of the N_Vector inputs x and y : $z_i = x_i / y_i, i = 0, \dots, n-1$. 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.
<i>continued on next page</i>	

continued from last page	
Name	Usage and Description
N_VScale	<p><code>N_VScale(c, x, z);</code> Scales the N_Vector <code>x</code> by the scalar <code>c</code> and returns the result in <code>z</code>: $z_i = cx_i, i = 0, \dots, n-1$.</p>
N_VAbs	<p><code>N_VAbs(x, y);</code> Sets the components of the N_Vector <code>y</code> to be the absolute values of the components of the N_Vector <code>x</code>: $y_i = x_i , i = 0, \dots, n-1$.</p>
N_VInv	<p><code>N_VInv(x, z);</code> Sets the components of the N_Vector <code>z</code> to be the inverses of the components of the N_Vector <code>x</code>: $z_i = 1.0/x_i, i = 0, \dots, n-1$. This routine may not check for division by 0. It should be called only with an <code>x</code> which is guaranteed to have all nonzero components.</p>
N_VAddConst	<p><code>N_VAddConst(x, b, z);</code> Adds the scalar <code>b</code> to all components of <code>x</code> and returns the result in the N_Vector <code>z</code>: $z_i = x_i + b, i = 0, \dots, n-1$.</p>
N_VDotProd	<p><code>d = N_VDotProd(x, y);</code> Returns the value of the ordinary dot product of <code>x</code> and <code>y</code>: $d = \sum_{i=0}^{n-1} x_i y_i$.</p>
N_VMaxNorm	<p><code>m = N_VMaxNorm(x);</code> Returns the maximum norm of the N_Vector <code>x</code>: $m = \max_i x_i$.</p>
N_VWrmsNorm	<p><code>m = N_VWrmsNorm(x, w)</code> Returns the weighted root-mean-square norm of the N_Vector <code>x</code> with weight vector <code>w</code>: $m = \sqrt{(\sum_{i=0}^{n-1} (x_i w_i)^2) / n}$.</p>
N_VWrmsNormMask	<p><code>m = N_VWrmsNormMask(x, w, id);</code> Returns the weighted root mean square norm of the N_Vector <code>x</code> with weight vector <code>w</code> built using only the elements of <code>x</code> corresponding to nonzero elements of the N_Vector <code>id</code>: $m = \sqrt{(\sum_{i=0}^{n-1} (x_i w_i \text{sign}(id_i))^2) / n}$.</p>
N_VMin	<p><code>m = N_VMin(x);</code> Returns the smallest element of the N_Vector <code>x</code>: $m = \min_i x_i$.</p>
N_VWL2Norm	<p><code>m = N_VWL2Norm(x, w);</code> Returns the weighted Euclidean ℓ_2 norm of the N_Vector <code>x</code> with weight vector <code>w</code>: $m = \sqrt{\sum_{i=0}^{n-1} (x_i w_i)^2}$.</p>
N_VL1Norm	<p><code>m = N_VL1Norm(x);</code> Returns the ℓ_1 norm of the N_Vector <code>x</code>: $m = \sum_{i=0}^{n-1} x_i$.</p>
continued on next page	

continued from last page	
Name	Usage and Description
N_VCompare	<code>N_VCompare(c, x, z);</code> Compares the components of the <code>N_Vector</code> <code>x</code> to the scalar <code>c</code> and returns an <code>N_Vector</code> <code>z</code> such that: $z_i = 1.0$ if $ x_i \geq c$ and $z_i = 0.0$ otherwise.
N_VInvTest	<code>t = N_VInvTest(x, z);</code> Sets the components of the <code>N_Vector</code> <code>z</code> to be the inverses of the components of the <code>N_Vector</code> <code>x</code> , with prior testing for zero values: $z_i = 1.0/x_i$, $i = 0, \dots, n-1$. This routine returns <code>TRUE</code> if all components of <code>x</code> are nonzero (successful inversion) and returns <code>FALSE</code> otherwise.
N_VConstrMask	<code>t = N_VConstrMask(c, x, m);</code> 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 <code>FALSE</code> if any element failed the constraint test, <code>TRUE</code> if all passed. It also sets a mask vector <code>m</code> , 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	<code>minq = N_VMinQuotient(num, denom);</code> This routine returns the minimum of the quotients obtained by term-wise dividing <code>num_i</code> by <code>denom_i</code> . A zero element in <code>denom</code> will be skipped. If no such quotients are found, then the large value <code>BIG_REAL</code> (defined in the header file <code>sundialstypes.h</code>) is returned.

6.1 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;
    boolean_t 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 6.1 and provides the following 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_VCloneEmpty_Serial`

This function creates a new serial `N_Vector` with an empty (`NULL`) data array by using an existing `N_Vector` as a template.

```
N_Vector N_VCloneEmpty_Serial(N_Vector w);
```

- `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_VNewVectorArray_Serial`

This function creates an array of `count` serial vectors.

```
N_Vector *N_VNewVectorArray_Serial(int count, long int vec_length);
```

- `N_VNewVectorArrayEmpty_Serial`

This function creates an array of `count` serial vectors, each with an empty (`NULL`) data array.

```
N_Vector *N_VNewVectorArrayEmpty_Serial(int count, long int vec_length);
```


- `N_VDestroyVectorArray_Serial`

This function frees memory allocated for the array of count variables of type `N_Vector` created with `N_VNewVectorArray_Serial` or with `N_VNewVectorArrayEmpty_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.
- The `NVECTOR_SERIAL` constructor functions `N_VNewEmpty_Serial`, `N_VCloneEmpty_Serial`, `N_VMake_Serial`, and `N_VNewVectorArrayEmpty_Serial` set the field `own_data = FALSE`. The functions `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.

6.2 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 6.1 and provides the following 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_VCloneEmpty_Parallel`

This function creates a new parallel `N_Vector` with an empty (NULL) data array by using an existing `N_Vector` as a template.

```
N_Vector N_VCloneEmpty_Parallel(N_Vector w);
```

- `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_VNewVectorArray_Parallel`

This function creates an array of `count` parallel vectors.

```
N_Vector *N_VNewVectorArray_Parallel(int count,
                                     MPI_Comm comm,
                                     long int local_length,
                                     long int global_length);
```

- `N_VNewVectorArrayEmpty_Parallel`

This function creates an array of `count` parallel vectors, each with an empty (NULL) data array.

```
N_Vector *N_VNewVectorArrayEmpty_Parallel(int count,
                                           MPI_Comm comm,
                                           long int local_length,
                                           long int global_length);
```

- `N_VDestroyVectorArray_Parallel`

This function frees memory allocated for the array of `count` variables of type `N_Vector` created with `N_VNewVectorArray_Parallel` or with `N_VNewVectorArrayEmpty_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.
- The `NVECTOR_PARALLEL` constructor functions `N_VNewEmpty_Parallel`, `N_VMake_Parallel`, `N_VCloneEmpty_Parallel`, and `N_VNewVectorArrayEmpty_Parallel` set the field `own_data = FALSE`. The functions `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.

6.3 NVECTOR functions used by KINSOL

In Table 6.2 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, the KINSPGMR column shows function usage within the linear solver, the KINBBDPRE column shows function usage within the band-block-diagonal preconditioner module, and the FKINSOL column shows function usage within the FKINSOL interface module.

There is one subtlety in the KINSPGMR column hidden by the table. The dot product function `N_VDotProd` is called both within the implementation file `kinspgmr.c` for the KINSPGMR solver and within the implementation files `spgmr.c` and `iterative.c` for the generic SPGMR solver upon which the KINSPGMR solver is implemented.

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.

Table 6.2: List of vector functions usage by KINSOL code modules

	KINSOL	KINSPGMR	KINBBDPRE	FKINSOL
<code>N_VClone</code>	✓		✓	✓
<code>N_VDestroy</code>	✓		✓	✓
<code>N_VSpace</code>	✓			
<code>N_VGetArrayPointer</code>			✓	✓
<code>N_VSetArrayPointer</code>				✓
<code>N_VLinearSum</code>	✓	✓		
<code>N_VConst</code>		✓		
<code>N_VProd</code>	✓	✓		
<code>N_VDiv</code>	✓			
<code>N_VMinQuotient</code>	✓			
<code>N_VScale</code>	✓	✓	✓	
<code>N_VAbs</code>	✓			
<code>N_VInv</code>	✓			
<code>N_VDotProd</code>		✓		
<code>N_VConstrMask</code>	✓			
<code>N_VMaxNorm</code>	✓			
<code>N_VL1Norm</code>		✓		
<code>N_VWL2Norm</code>	✓	✓		
<code>N_VMin</code>	✓			

The following vector operations listed in Table 6.1 are *not* used by KINSOL: `N_VAddConst`, `N_VWrmsNorm`, `N_VWrmsNormMask`, `N_VCompare`, and `N_VInvTest`. Therefore a user-supplied NVECTOR module for KINSOL could omit these five functions.

Chapter 7

Providing Alternate Linear Solver Modules

The central KINSOL module interfaces with the linear solver module 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 §5.4.2 for KINSPGMR) which will attach the above four routines to the main KINSOL memory block. 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.

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).

Initialization routine. 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.

Notes If an error does occur, an appropriate message should be sent to `kin_mem->kin_errfp`.

Setup routine. The type definition of `lsetup` is

Chapter 8

Generic Linear Solvers in SUNDIALS

In this section, we describe three 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 SPGMR package, which includes a solver for the scaled preconditioned GMRES method.

For the sake of space, the functions for `DenseMat` and `BandMat` matrices and the functions in SPGMR 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 solver.

8.1 The SPGMR Module

The SPGMR package, in the files `spgmr.h` and `spgmr.c`, includes an implementation of the scaled preconditioned GMRES method. A separate code module, `iterative.h` and `iterative.c`, contains auxiliary functions that support SPGMR, and also other Krylov solvers to be added later. For full details, including usage instructions, see the files `spgmr.h` and `iterative.h`.

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 `iterative.h` and `iterative.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`.

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 A. C. Hindmarsh. Reduced Storage Matrix Methods in Stiff ODE Systems. *J. Appl. Math. & Comp.*, 31:49–91, 1989.
- [3] P. N. Brown and Y. Saad. Hybrid Krylov Methods for Nonlinear Systems of Equations. *SIAM J. Sci. Stat. Comput.*, 11:450–481, 1990.
- [4] A. M. Collier and R. Serban. Example Programs for KINSOL v2.2.0. Technical Report UCRL-SM-208114, LLNL, 2004.
- [5] R. S. Dembo, S. C. Eisenstat, and T. Steihaug. Inexact Newton Methods. *SIAM J. Numer. Anal.*, 19:400–408, 1982.
- [6] J. E. Dennis and R. B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. SIAM, Philadelphia, 1996.
- [7] S. C. Eisenstat and H. F. Walker. Choosing the Forcing Terms in an Inexact Newton Method. *SIAM J. Sci. Comput.*, 17:16–32, 1996.
- [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.

Index

- BIG_REAL, 15, 53
- error message, 22
- f_data, 35, 38
- FKFUN, 42
- FKINBBD interface module
 - optional output, 47
 - usage, 46
- FKINBBDFREE, 48
- FKINBBDINIT, 47
- FKINBBDOPT, 47
- FKINBBDSPGMR, 47
- FKINFREE, 46
- FKINMALLOC, 44
- FKINSOL, 46
- FKINSOL interface module
 - interface to the KINBBDPRE module, 48
 - optional input and output, 42
 - usage, 46
 - user-callable functions, 41
 - user-supplied functions, 41
- FKINSPGMR, 45
- FKINSPGMRSETJAC, 47
- FKINSPGMRSETPSET, 45
- FKINSPGMRSETPSOL, 45
- FKJTIMES, 45
- FKPSET, 45
- FKPSOL, 45
- FNVFREEP, 46
- FNVFREES, 46
- FNVINITP, 44
- FNVINITS, 44
- generic linear solvers
 - SPGMR, 61
 - use in KINSOL, 14
- GMRES method, 20, 61
- half-bandwidths, 39
- header files, 38
- Inexact Newton iteration
 - definition, 9
- IOPT, 42, 43
- Jacobian-vector product approximation
 - difference quotient, 28
 - use in FKINSOL, 45
 - user-supplied, 29, 35
- KIN_ETACHOICE1, 25
- KIN_ETACHOICE2, 25
- KIN_ETACONSTANT, 25
- KIN_ILL_INPUT, 19, 21, 23–28
- KIN_INEXACT_NEWTON, **20**
- KIN_INITIAL_GUESS_OK, 21
- KIN_LINESEARCH, **20**
- KIN_LINESEARCH_BCFAIL, 21
- KIN_LINESEARCH_NONCONV, 21
- KIN_LINIT_FAIL, 21
- KIN_LINSOLV_NO_RECOVERY, 21
- KIN_LSETUP_FAIL, 21
- KIN_LSOLVE_FAIL, 21
- KIN_MAXITER_REACHED, 21
- KIN_MEM_FAIL, 19
- KIN_MEM_NULL, 19, 21, 23–28, 31, 32
- KIN_MXNEWT_5X_EXCEEDED, 21
- KIN_NO_MALLOC, 21
- KIN_PDATA_NULL, 40, 41
- KIN_STEP_LT_STPTOL, 21
- KIN_SUCCESS, 19, 21, 23–28, 31, 32, 40, 41
- KINBBDPRE preconditioner
 - optional output, 40–41
 - usage, 38–39
 - user-callable functions, 39
 - user-supplied functions, 37–38
- KINBBDPrecAlloc, **39**
- KINBBDPrecFree, **40**
- KINBBDPrecGetNumGfnEvals, **41**
- KINBBDPrecGetWorkSpace, **40**
- KINBBDSpgmr, 39, **40**
- KINCreate, **19**
- KINFree, **19**
- KINGetFuncNorm, **32**
- KINGetNumBacktrackOps, **32**
- KINGetNumBetaCondFails, **31**
- KINGetNumFuncEvals, **31**
- KINGetNumNonlinSolvIters, **31**
- KINGetStepLength, **32**
- KINGetWorkSpace, **31**
- KINMalloc, **19**

- KINSetConstraints, **27**
- KINSetErrFile, **23**
- KINSetEtaConstValue, **25**
- KINSetEtaForm, **25**
- KINSetEtaParams, **25**
- KINSetFdata, **24**
- KINSetFuncNormTol, **27**
- KINSetInfoFile, **23**
- KINSetMaxNewtonStep, **26**
- KINSetMaxPrecCalls, **24**
- KINSetNoMinEps, **26**
- KINSetNoPrecInit, **24**
- KINSetNumMaxIters, **24**
- KINSetPrintLevel, **23**
- KINSetRelErrFunc, **26**
- KINSetScaledStepTol, **27**
- KINSetSysFunc, **28**
- KINSOL
 - brief description of, **1**
 - motivation for writing in C, **1**
 - package structure, **11**
 - relationship to NKSOL, **1**
- KINSOL linear solvers
 - built on generic solvers, **20**
 - header files, **16**
 - implementation details, **11–14**
 - KINSPGMR, **20**
 - list of, **11**
- KINSol, **20**
- kinsol.h, **16**
- KINSOLKINSOL linear solvers
 - selecting, **20**
- KINSPGMR linear solver
 - Jacobian approximation used by, **28**
 - memory requirements, **32–33**
 - optional input, **28**
 - optional output, **32–35**
 - preconditioner setup function, **28, 36**
 - preconditioner solve function, **28, 36**
- KINSPGMR linear solver
 - selection of, **20**
- KINSpgrmr, **20, 20**
- kinspgmr.h, **16**
- KINSPGMR_ILL_INPUT, **20, 28, 40**
- KINSPGMR_LMEM_NULL, **28–30, 33, 34**
- KINSPGMR_MEM_FAIL, **20, 40**
- KINSPGMR_MEM_NULL, **20, 28–30, 32–34, 40**
- KINSPGMR_SUCCESS, **20, 28–30, 32–34, 40**
- KINSpgrmrDQJtimes, **28**
- KINSpgrmrGetLastFlag, **34**
- KINSpgrmrGetNumConvFails, **33**
- KINSpgrmrGetNumJtimesEvals, **34**
- KINSpgrmrGetNumLinIters, **33**
- KINSpgrmrGetNumPrecEvals, **33**
- KINSpgrmrGetNumPrecSolves, **33**
- KINSpgrmrGetNumRhsEvals, **34**
- KINSpgrmrGetWorkSpace, **32**
- KINSpgrmrJacTimesVecFn, **35**
- KINSpgrmrPrecSetupFn, **36**
- KINSpgrmrPrecSolveFn, **36**
- KINSpgrmrSetJacData, **30**
- KINSpgrmrSetJacTimesVecFn, **29**
- KINSpgrmrSetMaxRestarts, **28**
- KINSpgrmrSetPrecData, **29**
- KINSpgrmrSetPrecSetupFn, **29**
- KINSpgrmrSetPrecSolveFn, **29**
- KINSysFn, **19, 35**
- linit, **59**
- maxl, **40**
- memory requirements
 - KINBBDPRE preconditioner, **40**
 - KINSOL solver, **30**
 - KINSPGMR linear solver, **32–33**
- MPI, **2**
- N_VCloneEmpty_Parallel, **57**
- N_VCloneEmpty_Serial, **54**
- N_VCloneVectorArray, **50**
- N_VDestroyVectorArray, **50**
- N_VDestroyVectorArray_Parallel, **57**
- N_VDestroyVectorArray_Serial, **55**
- N_Vector, **16, 49, 49**
- N_VMake_Parallel, **57**
- N_VMake_Serial, **54**
- N_VNew_Parallel, **56**
- N_VNew_Serial, **54**
- N_VNewEmpty_Parallel, **56**
- N_VNewEmpty_Serial, **54**
- N_VNewVectorArray_Parallel, **57**
- N_VNewVectorArray_Serial, **54**
- N_VNewVectorArrayEmpty_Parallel, **57**
- N_VNewVectorArrayEmpty_Serial, **54**
- N_VPrint_Parallel, **57**
- N_VPrint_Serial, **55**
- nonlinear system
 - definition, **9**
- NV_COMM_P, **56**
- NV_CONTENT_P, **55**
- NV_CONTENT_S, **53**
- NV_DATA_P, **56**
- NV_DATA_S, **54**
- NV_GLOBLENGTH_P, **56**
- NV_Ith_P, **56**
- NV_Ith_S, **54**
- NV_LENGTH_S, **54**
- NV_LOCLENGTH_P, **56**
- NV_OWN_DATA_P, **56**

NV_OWN_DATA_S, 54
NVECTOR module, 49
nvector.h, 16
nvector_parallel.h, 16
nvector_serial.h, 16

optional input
 FKINSOL, 42
 iterative linear solver, 28
 solver, 22

optional output
 band-block-diagonal preconditioner, 40–41
 FKINBBD, 47
 FKINSOL, 42
 iterative linear solver, 32–35
 solver, 30

portability, 15
 FORTRAN, 42

preconditioning
 setup and solve phases, 11
 user-supplied, 28–29, 36

problem-defining function, 35

RCNST, 15
realtype, 15
ROPT, 42, 43

SMALL_REAL, 15

SPGMR generic linear solver
 description of, 61
 functions, 61
 support functions, 61

SUNDIALS_CASE_LOWER, 42
SUNDIALS_CASE_UPPER, 42
SUNDIALS_UNDERSCORE_NONE, 42
SUNDIALS_UNDERSCORE_ONE, 42
SUNDIALS_UNDERSCORE_TWO, 42
sundialstypes.h, 15, 16

UNIT_ROUNDOFF, 15

User main program
 FCVBBD usage, 47
 FKINSOL usage, 42
 KINBBDPRE usage, 38
 KINSOL usage, 16

