

# User Documentation for IDA v4.0.0-dev.2 (SUNDIALS v4.0.0-dev.2)

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

September 28, 2018



UCRL-SM-208112

## **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 Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed 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 Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

# Contents

<b>List of Tables</b>	<b>vii</b>
<b>List of Figures</b>	<b>ix</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Changes from previous versions . . . . .	2
1.2 Reading this User Guide . . . . .	9
1.3 SUNDIALS Release License . . . . .	9
1.3.1 Copyright Notices . . . . .	10
1.3.1.1 SUNDIALS Copyright . . . . .	10
1.3.1.2 ARKode Copyright . . . . .	10
1.3.2 BSD License . . . . .	10
<b>2 Mathematical Considerations</b>	<b>13</b>
2.1 IVP solution . . . . .	13
2.2 Preconditioning . . . . .	17
2.3 Rootfinding . . . . .	17
<b>3 Code Organization</b>	<b>19</b>
3.1 SUNDIALS organization . . . . .	19
3.2 IDA organization . . . . .	19
<b>4 Using IDA for C Applications</b>	<b>25</b>
4.1 Access to library and header files . . . . .	25
4.2 Data types . . . . .	26
4.2.1 Floating point types . . . . .	26
4.2.2 Integer types used for vector and matrix indices . . . . .	26
4.3 Header files . . . . .	27
4.4 A skeleton of the user's main program . . . . .	28
4.5 User-callable functions . . . . .	31
4.5.1 IDA initialization and deallocation functions . . . . .	32
4.5.2 IDA tolerance specification functions . . . . .	32
4.5.3 Linear solver interface functions . . . . .	34
4.5.4 Nonlinear solver interface function . . . . .	36
4.5.5 Initial condition calculation function . . . . .	36
4.5.6 Rootfinding initialization function . . . . .	37
4.5.7 IDA solver function . . . . .	38
4.5.8 Optional input functions . . . . .	39
4.5.8.1 Main solver optional input functions . . . . .	39
4.5.8.2 Linear solver interface optional input functions . . . . .	45
4.5.8.3 Initial condition calculation optional input functions . . . . .	48
4.5.8.4 Rootfinding optional input functions . . . . .	50
4.5.9 Interpolated output function . . . . .	51

4.5.10	Optional output functions . . . . .	51
4.5.10.1	SUNDIALS version information . . . . .	53
4.5.10.2	Main solver optional output functions . . . . .	53
4.5.10.3	Initial condition calculation optional output functions . . . . .	59
4.5.10.4	Rootfinding optional output functions . . . . .	59
4.5.10.5	IDALS linear solver interface optional output functions . . . . .	60
4.5.11	IDA reinitialization function . . . . .	64
4.6	User-supplied functions . . . . .	65
4.6.1	Residual function . . . . .	65
4.6.2	Error message handler function . . . . .	66
4.6.3	Error weight function . . . . .	66
4.6.4	Rootfinding function . . . . .	67
4.6.5	Jacobian construction (matrix-based linear solvers) . . . . .	67
4.6.6	Jacobian-vector product (matrix-free linear solvers) . . . . .	69
4.6.7	Jacobian-vector product setup (matrix-free linear solvers) . . . . .	70
4.6.8	Preconditioner solve (iterative linear solvers) . . . . .	71
4.6.9	Preconditioner setup (iterative linear solvers) . . . . .	71
4.7	A parallel band-block-diagonal preconditioner module . . . . .	72
<b>5</b>	<b>FIDA, an Interface Module for FORTRAN Applications</b>	<b>79</b>
5.1	Important note on portability . . . . .	79
5.2	Fortran Data Types . . . . .	79
5.3	FIDA routines . . . . .	80
5.4	Usage of the FIDA interface module . . . . .	82
5.5	FIDA optional input and output . . . . .	89
5.6	Usage of the FIDAROOT interface to rootfinding . . . . .	92
5.7	Usage of the FIDABBD interface to IDABBDPRE . . . . .	93
<b>6</b>	<b>Description of the NVECTOR module</b>	<b>97</b>
6.1	The NVECTOR_SERIAL implementation . . . . .	106
6.2	The NVECTOR_PARALLEL implementation . . . . .	108
6.3	The NVECTOR_OPENMP implementation . . . . .	111
6.4	The NVECTOR_PTHREADS implementation . . . . .	114
6.5	The NVECTOR_PARHYP implementation . . . . .	116
6.6	The NVECTOR_PETSC implementation . . . . .	118
6.7	The NVECTOR_CUDA implementation . . . . .	119
6.8	The NVECTOR_RAJA implementation . . . . .	122
6.9	NVECTOR Examples . . . . .	125
6.10	NVECTOR functions used by IDA . . . . .	128
<b>7</b>	<b>Description of the SUNMatrix module</b>	<b>131</b>
7.1	The SUNMatrix_Dense implementation . . . . .	134
7.2	The SUNMatrix_Band implementation . . . . .	137
7.3	The SUNMatrix_Sparse implementation . . . . .	141
7.4	SUNMatrix Examples . . . . .	147
7.5	SUNMatrix functions used by IDA . . . . .	148
<b>8</b>	<b>Description of the SUNLinearSolver module</b>	<b>151</b>
8.0.1	SUNLinearSolver core functions . . . . .	152
8.0.2	SUNLinearSolver set functions . . . . .	153
8.0.3	SUNLinearSolver get functions . . . . .	154
8.0.4	Functions provided by SUNDIALS packages . . . . .	156
8.0.5	SUNLinearSolver return codes . . . . .	157
8.0.6	The generic SUNLinearSolver module . . . . .	158
8.1	Compatibility of SUNLinearSolver modules . . . . .	159

8.2	Implementing a custom <code>SUNLinearSolver</code> module . . . . .	159
8.3	The <code>SUNLinearSolver_Dense</code> implementation . . . . .	160
8.3.1	<code>SUNLINSOL_DENSE</code> usage . . . . .	160
8.3.2	<code>SUNLINSOL_DENSE</code> description . . . . .	161
8.4	The <code>SUNLinearSolver_Band</code> implementation . . . . .	162
8.4.1	<code>SUNLINSOL_BAND</code> usage . . . . .	162
8.4.2	<code>SUNLINSOL_BAND</code> description . . . . .	163
8.5	The <code>SUNLinearSolver_LapackDense</code> implementation . . . . .	164
8.5.1	<code>SUNLINSOL_LAPACKDENSE</code> usage . . . . .	164
8.5.2	<code>SUNLINSOL_LAPACKDENSE</code> description . . . . .	165
8.6	The <code>SUNLinearSolver_LapackBand</code> implementation . . . . .	166
8.6.1	<code>SUNLINSOL_LAPACKBAND</code> usage . . . . .	166
8.6.2	<code>SUNLINSOL_LAPACKBAND</code> description . . . . .	167
8.7	The <code>SUNLinearSolver_KLU</code> implementation . . . . .	168
8.7.1	<code>SUNLINSOL_KLU</code> usage . . . . .	168
8.7.2	<code>SUNLINSOL_KLU</code> description . . . . .	172
8.8	The <code>SUNLinearSolver_SuperLUMT</code> implementation . . . . .	173
8.8.1	<code>SUNLINSOL_SUPERLUMT</code> usage . . . . .	173
8.8.2	<code>SUNLINSOL_SUPERLUMT</code> description . . . . .	176
8.9	The <code>SUNLinearSolver_SPGMR</code> implementation . . . . .	177
8.9.1	<code>SUNLINSOL_SPGMR</code> usage . . . . .	178
8.9.2	<code>SUNLINSOL_SPGMR</code> description . . . . .	181
8.10	The <code>SUNLinearSolver_SPGFMR</code> implementation . . . . .	183
8.10.1	<code>SUNLINSOL_SPGFMR</code> usage . . . . .	184
8.10.2	<code>SUNLINSOL_SPGFMR</code> description . . . . .	187
8.11	The <code>SUNLinearSolver_SPBCGS</code> implementation . . . . .	189
8.11.1	<code>SUNLINSOL_SPBCGS</code> usage . . . . .	190
8.11.2	<code>SUNLINSOL_SPBCGS</code> description . . . . .	193
8.12	The <code>SUNLinearSolver_SPTFQMR</code> implementation . . . . .	194
8.12.1	<code>SUNLINSOL_SPTFQMR</code> usage . . . . .	194
8.12.2	<code>SUNLINSOL_SPTFQMR</code> description . . . . .	197
8.13	The <code>SUNLinearSolver_PCG</code> implementation . . . . .	199
8.13.1	<code>SUNLINSOL_PCG</code> usage . . . . .	200
8.13.2	<code>SUNLINSOL_PCG</code> description . . . . .	203
8.14	<code>SUNLinearSolver</code> Examples . . . . .	204
8.15	<code>SUNLinearSolver</code> functions used by IDA . . . . .	205
<b>9</b>	<b>Description of the <code>SUNNonlinearSolver</code> module</b> . . . . .	<b>207</b>
9.1	The <code>SUNNonlinearSolver</code> API . . . . .	207
9.1.1	<code>SUNNonlinearSolver</code> core functions . . . . .	207
9.1.2	<code>SUNNonlinearSolver</code> set functions . . . . .	209
9.1.3	<code>SUNNonlinearSolver</code> get functions . . . . .	210
9.1.4	Functions provided by <code>SUNDIALS</code> integrators . . . . .	211
9.1.5	<code>SUNNonlinearSolver</code> return codes . . . . .	213
9.1.6	The generic <code>SUNNonlinearSolver</code> module . . . . .	213
9.1.7	Usage with sensitivity enabled integrators . . . . .	214
9.1.8	Implementing a Custom <code>SUNNonlinearSolver</code> Module . . . . .	215
9.2	The <code>SUNNonlinearSolver_Newton</code> implementation . . . . .	216
9.2.1	<code>SUNNonlinearSolver_Newton</code> description . . . . .	216
9.2.2	<code>SUNNonlinearSolver_Newton</code> functions . . . . .	217
9.2.3	<code>SUNNonlinearSolver_Newton</code> content . . . . .	218
9.2.4	<code>SUNNonlinearSolver_Newton</code> Fortran interface . . . . .	218
9.3	The <code>SUNNonlinearSolver_FixedPoint</code> implementation . . . . .	218
9.3.1	<code>SUNNonlinearSolver_FixedPoint</code> description . . . . .	219

9.3.2	SUNNonlinearSolver_FixedPoint functions . . . . .	219
9.3.3	SUNNonlinearSolver_FixedPoint content . . . . .	221
9.3.4	SUNNonlinearSolver_FixedPoint Fortran interface . . . . .	222
<b>A</b>	<b>SUNDIALS Package Installation Procedure</b>	<b>223</b>
A.1	CMake-based installation . . . . .	224
A.1.1	Configuring, building, and installing on Unix-like systems . . . . .	224
A.1.2	Configuration options (Unix/Linux) . . . . .	226
A.1.3	Configuration examples . . . . .	232
A.1.4	Working with external Libraries . . . . .	233
A.1.5	Testing the build and installation . . . . .	235
A.2	Building and Running Examples . . . . .	235
A.3	Configuring, building, and installing on Windows . . . . .	236
A.4	Installed libraries and exported header files . . . . .	236
<b>B</b>	<b>IDA Constants</b>	<b>241</b>
B.1	IDA input constants . . . . .	241
B.2	IDA output constants . . . . .	241
	<b>Bibliography</b>	<b>243</b>
	<b>Index</b>	<b>245</b>

# List of Tables

4.1	SUNDIALS linear solver interfaces and vector implementations that can be used for each.	31
4.2	Optional inputs for IDA and IDALS . . . . .	40
4.3	Optional outputs from IDA and IDALS . . . . .	52
5.1	Keys for setting FIDA optional inputs . . . . .	90
5.2	Description of the FIDA optional output arrays <b>IOUT</b> and <b>ROUT</b> . . . . .	91
6.1	Vector Identifications associated with vector kernels supplied with SUNDIALS. . . . .	99
6.2	Description of the <b>NVECTOR</b> operations . . . . .	100
6.3	Description of the <b>NVECTOR</b> fused operations . . . . .	103
6.4	Description of the <b>NVECTOR</b> vector array operations . . . . .	104
6.5	List of vector functions usage by IDA code modules . . . . .	128
7.1	Identifiers associated with matrix kernels supplied with SUNDIALS. . . . .	132
7.2	Description of the <b>SUNMatrix</b> operations . . . . .	132
7.3	SUNDIALS matrix interfaces and vector implementations that can be used for each. . .	133
7.4	List of matrix functions usage by IDA code modules . . . . .	148
8.1	Description of the <b>SUNLinearSolver</b> error codes . . . . .	157
8.2	SUNDIALS matrix-based linear solvers and matrix implementations that can be used for each. . . . .	159
8.3	List of linear solver function usage by IDA code modules . . . . .	206
9.1	Description of the <b>SUNNonlinearSolver</b> return codes . . . . .	213
A.1	SUNDIALS libraries and header files . . . . .	236





# List of Figures

3.1	High-level diagram of the SUNDIALS suite . . . . .	20
3.2	Organization of the SUNDIALS suite . . . . .	21
3.3	Overall structure diagram of the IDA package . . . . .	22
7.1	Diagram of the storage for a SUNMATRIX_BAND object . . . . .	138
7.2	Diagram of the storage for a compressed-sparse-column matrix . . . . .	144
A.1	Initial <i>ccmake</i> configuration screen . . . . .	225
A.2	Changing the <i>instdir</i> . . . . .	226



# Chapter 1

## Introduction

IDA is part of a software family called SUNDIALS: SUite of Nonlinear and Differential/ALgebraic equation Solvers [23]. This suite consists of CVODE, ARKODE, KINSOL, and IDA, and variants of these with sensitivity analysis capabilities, CVODES and IDAS.

IDA is a general purpose solver for the initial value problem (IVP) for systems of differential-algebraic equations (DAEs). The name IDA stands for Implicit Differential-Algebraic solver. IDA is based on DASPK [7, 8], but is written in ANSI-standard C rather than FORTRAN77. Its most notable features are that, (1) in the solution of the underlying nonlinear system at each time step, it offers a choice of Newton/direct methods and a choice of Inexact Newton/Krylov (iterative) methods; and (2) it is written in a *data-independent* manner in that it acts on generic vectors and matrices without any assumptions on the underlying organization of the data. Thus IDA shares significant modules previously written within CASC at LLNL to support the ordinary differential equation (ODE) solvers CVODE [24, 13] and PVODE [11, 12], and also the nonlinear system solver KINSOL [14].

At present, IDA may utilize a variety of Krylov methods provided in SUNDIALS that can be used in conjunction with Newton iteration: these include the GMRES (Generalized Minimal RESidual) [33], FGMRES (Flexible Generalized Minimum RESidual) [32], Bi-CGStab (Bi-Conjugate Gradient Stabilized) [34], TFQMR (Transpose-Free Quasi-Minimal Residual) [20], and PCG (Preconditioned Conjugate Gradient) [21] linear iterative methods. As Krylov methods, these require little matrix storage for solving the Newton equations as compared to direct methods. However, the algorithms allow for a user-supplied preconditioner matrix, and, for most problems, preconditioning is essential for an efficient solution.

For very large DAE systems, the Krylov methods are preferable over direct linear solver methods, and are often the only feasible choice. Among the Krylov methods in SUNDIALS, we recommend GMRES as the best overall choice. However, users are encouraged to compare all options, especially if encountering convergence failures with GMRES. Bi-CGStab and TFQMR have an advantage in storage requirements, in that the number of workspace vectors they require is fixed, while that number for GMRES depends on the desired Krylov subspace size. FGMRES has an advantage in that it is designed to support preconditioners that vary between iterations (e.g. iterative methods). PCG exhibits rapid convergence and minimal workspace vectors, but only works for symmetric linear systems.

There are several motivations for choosing the C language for IDA. First, a general movement away from FORTRAN and toward C in scientific computing was 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 IDA 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 extended FORTRAN.

## 1.1 Changes from previous versions

### Changes in v4.0.0-dev.2

IDA's previous direct and iterative linear solver interfaces, IDADLS and IDASPILS, have been merged into a single unified linear solver interface, IDALS, to support any valid SUNLINSOL module. The user interface for the new IDALS module is very similar to the previous IDADLS and IDASPILS interfaces; however to minimize challenges in user migration to the new names, the previous C and FORTRAN routine names may still be used; these will be deprecated in future releases, so we recommend that users migrate to the new names soon. We do note that FORTRAN users, however, may need to enlarge their `iout` array of optional integer outputs, and update the indices that they query for certain linear-solver-related statistics.

The names of all constructor routines for SUNDIALS-provided SUNLINSOL implementations have been updated to `SUNLinSol_Band`, `SUNLinSol_Dense`, `SUNLinSol_KLU`, `SUNLinSol_LapackBand`, `SUNLinSol_LapackDense`, `SUNLinSol_PCG`, `SUNLinSol_SPBCGS`, `SUNLinSol_SPGMR`, `SUNLinSol_SPGMR`, `SUNLinSol_SPTFQMR`, and `SUNLinSol_SuperLUMT`. Solver-specific “set” routine names have been similarly standardized. While the previous routine names may still be used, these will be deprecated in future releases, so we recommend that users migrate to the new names soon.

### Changes in v4.0.0-dev.1

An API for encapsulating the nonlinear solvers used in SUNDIALS implicit integrators has been introduced. The goal of this API is to ease the introduction of new nonlinear solver options in SUNDIALS integrators and allow for external or user-supplied nonlinear solvers. The `SUNNONLINSOL` API and provided `SUNNONLINSOL` modules are described in Chapter 9 and follow the same object oriented design and implementation used by the `NVECTOR`, `SUNMATRIX`, and `SUNLINSOL` modules.

`SUNNONLINSOL` modules are intended to solve nonlinear systems formulated as either a rootfinding problem  $F(y) = 0$  or a fixed-point problem  $G(y) = y$ . Currently two `SUNNONLINSOL` implementations are provided, `SUNNONLINSOL_NEWTON` and `SUNNONLINSOL_FIXEDPOINT`. These replicate the previous integrator specific implementations of a Newton iteration and a fixed-point iteration (previously referred to as a functional iteration), respectively. Additionally, the fixed-point iteration can use Anderson's method to accelerate convergence. Example programs using each of these nonlinear solver modules in a standalone manner have been added and all IDA example programs have been updated to use generic `SUNNONLINSOL` modules.

By default IDA uses the `SUNNONLINSOL_NEWTON` module implementation of a Newton iteration. Since IDA previously only used an internal implementation of a Newton iteration for nonlinear solves, no changes are required to user programs and functions for setting the nonlinear solver options (e.g., `IDASetMaxNonlinIters`) or getting nonlinear solver statistics (e.g., `IDAGetNumNonlinSolvIters`) remain unchanged and internally call generic `SUNNONLINSOL` functions.

While SUNDIALS includes a fixed-point nonlinear solver module, it is not currently supported in IDA. However, user-supplied nonlinear solver modules adhering to the `SUNNONLINSOL` API for solving  $F(y) = 0$  can be used with IDA. To use a non-default nonlinear solver with IDA a user must do the following:

1. include the header of the desired nonlinear solver module,
2. create a nonlinear solver object by calling the constructor defined by the nonlinear solver module. For example, SUNDIALS provided nonlinear solver modules define constructors of the form

```
NLS = SUNNonlinSol_***(...);
```

where `NLS` is the nonlinear solver object of type `SUNNonlinearSolver`, `***` is the name of the nonlinear solver, and the inputs `...` are nonlinear solver specific.

3. after initializing IDA with `IDAInit`, call

```
retval = IDASetNonlinearSolver(ida_mem, NLS);
```

to attach the nonlinear solver object to the IDA memory structure.

The example program `idaRoberts_dns.c` explicitly creates and attaches the `SUNNONLINSOL_NEWTON` module to demonstrate this process (this is not necessary in general as IDA uses the `SUNNONLINSOL_NEWTON` module by default).

## Changes in v4.0.0-dev

Three fused vector operations and seven vector array operations have been added to the NVECTOR API. These *optional* operations are intended to increase data reuse in vector operations, reduce parallel communication on distributed memory systems, and lower the number of kernel launches on systems with accelerators. The new operations are `N_VLinearCombination`, `N_VScaleAddMulti`, `N_VDotProdMulti`, `N_VLinearCombinationVectorArray`, `N_VScaleVectorArray`, `N_VConstVectorArray`, `N_VWrmsNormVectorArray`, `N_VWrmsNormMaskVectorArray`, `N_VScaleAddMultiVectorArray`, and `N_VLinearCombinationVectorArray`. If any of these operations are defined as `NULL` in an NVECTOR implementation the NVECTOR interface will automatically call standard NVECTOR operations as necessary. Details on the new operations can be found in Chapter 6.

## Changes in v3.2.0

Fixed a problem with setting `sunindextype` which would occur with some compilers (e.g. `armclang`) that did not define `__STDC_VERSION__`.

Added hybrid MPI/CUDA and MPI/RAJA vectors to allow use of more than one MPI rank when using a GPU system. The vectors assume one GPU device per MPI rank.

Changed the name of the RAJA NVECTOR library to `libsundials_nveccudaraja.lib` from `libsundials_nvecraja.lib` to better reflect that we only support CUDA as a backend for RAJA currently.

Several changes were made to the build system:

- CMake 3.1.3 is now the minimum required CMake version.
- Deprecate the behavior of the `SUNDIALS_INDEX_TYPE` CMake option and added the `SUNDIALS_INDEX_SIZE` CMake option to select the `sunindextype` integer size.
- The native CMake FindMPI module is now used to locate an MPI installation.
- If MPI is enabled and MPI compiler wrappers are not set, the build system will check if `CMAKE_  
<language>_COMPILER` can compile MPI programs before trying to locate and use an MPI installation.
- The previous options for setting MPI compiler wrappers and the executable for running MPI programs have been deprecated. The new options that align with those used in native CMake FindMPI module are `MPI_C_COMPILER`, `MPI_CXX_COMPILER`, `MPI_Fortran_COMPILER`, and `MPIEXEC_EXECUTABLE`.
- When a Fortran name-mangling scheme is needed (e.g., `LAPACK_ENABLE` is `ON`) the build system will infer the scheme from the Fortran compiler. If a Fortran compiler is not available or the inferred or default scheme needs to be overridden, the advanced options `SUNDIALS_F77_FUNC_CASE` and `SUNDIALS_F77_FUNC_UNDERSCORES` can be used to manually set the name-mangling scheme and bypass trying to infer the scheme.
- Parts of the main `CMakeLists.txt` file were moved to new files in the `src` and `example` directories to make the CMake configuration file structure more modular.

## Changes in v3.1.2

The changes in this minor release include the following:

- Updated the minimum required version of CMake to 2.8.12 and enabled using rpath by default to locate shared libraries on OSX.
- Fixed Windows specific problem where `sunindextype` was not correctly defined when using 64-bit integers for the SUNDIALS index type. On Windows `sunindextype` is now defined as the MSVC basic type `__int64`.
- Added sparse SUNMatrix “Reallocate” routine to allow specification of the nonzero storage.
- Updated the KLU SUNLinearSolver module to set constants for the two reinitialization types, and fixed a bug in the full reinitialization approach where the sparse SUNMatrix pointer would go out of scope on some architectures.
- Updated the “ScaleAdd” and “ScaleAddI” implementations in the sparse SUNMatrix module to more optimally handle the case where the target matrix contained sufficient storage for the sum, but had the wrong sparsity pattern. The sum now occurs in-place, by performing the sum backwards in the existing storage. However, it is still more efficient if the user-supplied Jacobian routine allocates storage for the sum  $I + \gamma J$  manually (with zero entries if needed).
- Changed the LICENSE install path to `instdir/include/sundials`.

## Changes in v3.1.1

The changes in this minor release include the following:

- Fixed a potential memory leak in the SPGMR and SPFGMR linear solvers: if “Initialize” was called multiple times then the solver memory was reallocated (without being freed).
- Updated KLU SUNLINEARSOLVER module to use a `typedef` for the precision-specific solve function to be used (to avoid compiler warnings).
- Added missing typecasts for some `(void*)` pointers (again, to avoid compiler warnings).
- Bugfix in `sunmatrix_sparse.c` where we had used `int` instead of `sunindextype` in one location.
- Added missing `#include <stdio.h>` in NVECTOR and SUNMATRIX header files.
- Added missing prototype for `IDASpilsGetNumJTSetupEvals`.
- Fixed an indexing bug in the CUDA NVECTOR implementation of `N_VWrmsNormMask` and revised the RAJA NVECTOR implementation of `N_VWrmsNormMask` to work with mask arrays using values other than zero or one. Replaced `double` with `realtype` in the RAJA vector test functions.
- Fixed compilation issue with GCC 7.3.0 and Fortran programs that do not require a SUNMATRIX module (e.g., iterative linear solvers).

In addition to the changes above, minor corrections were also made to the example programs, build system, and user documentation.

## Changes in v3.1.0

Added NVECTOR print functions that write vector data to a specified file (e.g., `N_VPrintFile_Serial`).

Added `make test` and `make test_install` options to the build system for testing SUNDIALS after building with `make` and installing with `make install` respectively.

## Changes in v3.0.0

All interfaces to matrix structures and linear solvers have been reworked, and all example programs have been updated. The goal of the redesign of these interfaces was to provide more encapsulation and to ease interfacing of custom linear solvers and interoperability with linear solver libraries. Specific changes include:

- Added generic **SUNMATRIX** module with three provided implementations: dense, banded and sparse. These replicate previous SUNDIALS Dls and SlS matrix structures in a single object-oriented API.
- Added example problems demonstrating use of generic **SUNMATRIX** modules.
- Added generic **SUNLinearSolver** module with eleven provided implementations: SUNDIALS native dense, SUNDIALS native banded, LAPACK dense, LAPACK band, KLU, SuperLU\_MT, SPGMR, SPBCGS, SPTFQMR, SPFGMR, and PCG. These replicate previous SUNDIALS generic linear solvers in a single object-oriented API.
- Added example problems demonstrating use of generic **SUNLinearSolver** modules.
- Expanded package-provided direct linear solver (Dls) interfaces and scaled, preconditioned, iterative linear solver (Spils) interfaces to utilize generic **SUNMATRIX** and **SUNLinearSolver** objects.
- Removed package-specific, linear solver-specific, solver modules (e.g. **CVDENSE**, **KINBAND**, **IDAKLU**, **ARKSPGMR**) since their functionality is entirely replicated by the generic Dls/Spils interfaces and **SUNLinearSolver**/**SUNMATRIX** modules. The exception is **CVDIAG**, a diagonal approximate Jacobian solver available to **CVODE** and **CVODES**.
- Converted all SUNDIALS example problems and files to utilize the new generic **SUNMATRIX** and **SUNLinearSolver** objects, along with updated Dls and Spils linear solver interfaces.
- Added Spils interface routines to **ARKODE**, **CVODE**, **CVODES**, **IDA**, and **IDAS** to allow specification of a user-provided "JTSetup" routine. This change supports users who wish to set up data structures for the user-provided Jacobian-times-vector ("JTTimes") routine, and where the cost of one JTSetup setup per Newton iteration can be amortized between multiple JTTimes calls.

Two additional **NVECTOR** implementations were added – one for **CUDA** and one for **RAJA** vectors. These vectors are supplied to provide very basic support for running on GPU architectures. Users are advised that these vectors both move all data to the GPU device upon construction, and speedup will only be realized if the user also conducts the right-hand-side function evaluation on the device. In addition, these vectors assume the problem fits on one GPU. Further information about **RAJA**, users are referred to the web site, <https://software.llnl.gov/RAJA/>. These additions are accompanied by additions to various interface functions and to user documentation.

All indices for data structures were updated to a new **sunindextype** that can be configured to be a 32- or 64-bit integer data index type. **sunindextype** is defined to be **int32\_t** or **int64\_t** when portable types are supported, otherwise it is defined as **int** or **long int**. The Fortran interfaces continue to use **long int** for indices, except for their sparse matrix interface that now uses the new **sunindextype**. This new flexible capability for index types includes interfaces to PETSc, hypre, SuperLU\_MT, and KLU with either 32-bit or 64-bit capabilities depending how the user configures SUNDIALS.

To avoid potential namespace conflicts, the macros defining **boolean** values **TRUE** and **FALSE** have been changed to **SUNTRUE** and **SUNFALSE** respectively.

Temporary vectors were removed from preconditioner setup and solve routines for all packages. It is assumed that all necessary data for user-provided preconditioner operations will be allocated and stored in user-provided data structures.

The file **include/sundials\_fconfig.h** was added. This file contains SUNDIALS type information for use in Fortran programs.

The build system was expanded to support many of the xSDK-compliant keys. The xSDK is a movement in scientific software to provide a foundation for the rapid and efficient production of high-quality, sustainable extreme-scale scientific applications. More information can be found at, <https://xsdk.info>.

Added functions `SUNDIALSGetVersion` and `SUNDIALSGetVersionNumber` to get SUNDIALS release version information at runtime.

In addition, numerous changes were made to the build system. These include the addition of separate `BLAS_ENABLE` and `BLAS_LIBRARIES` CMake variables, additional error checking during CMake configuration, minor bug fixes, and renaming CMake options to enable/disable examples for greater clarity and an added option to enable/disable Fortran 77 examples. These changes included changing `EXAMPLES_ENABLE` to `EXAMPLES_ENABLE_C`, changing `CXX_ENABLE` to `EXAMPLES_ENABLE_CXX`, changing `F90_ENABLE` to `EXAMPLES_ENABLE_F90`, and adding an `EXAMPLES_ENABLE_F77` option.

A bug fix was done to add a missing prototype for `IDASSetMaxBacksIC` in `ida.h`.

Corrections and additions were made to the examples, to installation-related files, and to the user documentation.

## Changes in v2.9.0

Two additional NVECTOR implementations were added – one for Hypre (parallel) ParVector vectors, and one for PETSc vectors. These additions are accompanied by additions to various interface functions and to user documentation.

Each NVECTOR module now includes a function, `N_VGetVectorID`, that returns the NVECTOR module name.

An optional input function was added to set a maximum number of linesearch backtracks in the initial condition calculation. Also, corrections were made to three Fortran interface functions.

For each linear solver, the various solver performance counters are now initialized to 0 in both the solver specification function and in solver `init` function. This ensures that these solver counters are initialized upon linear solver instantiation as well as at the beginning of the problem solution.

A memory leak was fixed in the banded preconditioner interface. In addition, updates were done to return integers from linear solver and preconditioner 'free' functions.

The Krylov linear solver Bi-CGstab was enhanced by removing a redundant dot product. Various additions and corrections were made to the interfaces to the sparse solvers KLU and SuperLU\_MT, including support for CSR format when using KLU.

New examples were added for use of the OpenMP vector.

Minor corrections and additions were made to the IDA solver, to the Fortran interfaces, to the examples, to installation-related files, and to the user documentation.

## Changes in v2.8.0

Two major additions were made to the linear system solvers that are available for use with the IDA solver. First, in the serial case, an interface to the sparse direct solver KLU was added. Second, an interface to SuperLU\_MT, the multi-threaded version of SuperLU, was added as a thread-parallel sparse direct solver option, to be used with the serial version of the NVECTOR module. As part of these additions, a sparse matrix (CSC format) structure was added to IDA.

Otherwise, only relatively minor modifications were made to IDA:

In `IDARootfind`, a minor bug was corrected, where the input array `rootdir` was ignored, and a line was added to break out of root-search loop if the initial interval size is below the tolerance `ttol`.

In `IDALapackBand`, the line `smu = MIN(N-1,mu+m1)` was changed to `smu = mu + m1` to correct an illegal input error for `DGBTRF/DGBTRS`.

A minor bug was fixed regarding the testing of the input `tstop` on the first call to `IDASolve`.

In order to avoid possible name conflicts, the mathematical macro and function names `MIN`, `MAX`, `SQR`, `RAbs`, `RSqrt`, `RExp`, `RPowerI`, and `RPowerR` were changed to `SUNMIN`, `SUNMAX`, `SUNSQR`, `SUNRAbs`, `SUNRSqrt`, `SUNRExp`, `SUNRpowerI`, and `SUNRpowerR`, respectively. These names occur in both the solver and in various example programs.



In the FIDA optional input routines `FIDASETIIN`, `FIDASETRIN`, and `FIDASETVIN`, the optional fourth argument `key_length` was removed, with hardcoded key string lengths passed to all `strncmp` tests.

In all FIDA examples, integer declarations were revised so that those which must match a C type `long int` are declared `INTEGER*8`, and a comment was added about the type match. All other integer declarations are just `INTEGER`. Corresponding minor corrections were made to the user guide.

Two new `NVECTOR` modules have been added for thread-parallel computing environments — one for OpenMP, denoted `NVECTOR_OPENMP`, and one for Pthreads, denoted `NVECTOR_PTHREADS`.

With this version of SUNDIALS, support and documentation of the Autotools mode of installation is being dropped, in favor of the CMake mode, which is considered more widely portable.

## Changes in v2.7.0

One significant design change was made with this release: The problem size and its relatives, bandwidth parameters, related internal indices, pivot arrays, and the optional output `lsflag` have all been changed from type `int` to type `long int`, except for the problem size and bandwidths in user calls to routines specifying BLAS/LAPACK routines for the dense/band linear solvers. The function `NewIntArray` is replaced by a pair `NewIntArray/NewLintArray`, for `int` and `long int` arrays, respectively.

A large number of minor errors have been fixed. Among these are the following: After the solver memory is created, it is set to zero before being filled. To be consistent with IDAS, IDA uses the function `IDAGetDky` for optional output retrieval. In each linear solver interface function, the linear solver memory is freed on an error return, and the `**Free` function now includes a line setting to NULL the main memory pointer to the linear solver memory. A memory leak was fixed in two of the `IDASp***Free` functions. In the rootfinding functions `IDARcheck1`/`IDARcheck2`, when an exact zero is found, the array `glo` of  $g$  values at the left endpoint is adjusted, instead of shifting the  $t$  location `tlo` slightly. In the installation files, we modified the treatment of the macro `SUNDIALS_USE_GENERIC_MATH`, so that the parameter `GENERIC_MATH_LIB` is either defined (with no value) or not defined.

## Changes in v2.6.0

Two new features were added in this release: (a) a new linear solver module, based on BLAS and LAPACK for both dense and banded matrices, and (b) option to specify which direction of zero-crossing is to be monitored while performing rootfinding.

The user interface has been further refined. Some of the API changes involve: (a) a reorganization of all linear solver modules into two families (besides the already present family of scaled preconditioned iterative linear solvers, the direct solvers, including the new LAPACK-based ones, were also organized into a *direct* family); (b) maintaining a single pointer to user data, optionally specified through a `Set`-type function; (c) a general streamlining of the band-block-diagonal preconditioner module distributed with the solver.

## Changes in v2.5.0

The main changes in this release involve a rearrangement of the entire SUNDIALS source tree (see §3.1). At the user interface level, the main impact is in the mechanism of including SUNDIALS header files which must now include the relative path (e.g. `#include <cvode/cvode.h>`). Additional changes were made to the build system: all exported header files are now installed in separate subdirectories of the installation *include* directory.

A bug was fixed in the internal difference-quotient dense and banded Jacobian approximations, related to the estimation of the perturbation (which could have led to a failure of the linear solver when zero components with sufficiently small absolute tolerances were present).

The user interface to the consistent initial conditions calculations was modified. The `IDACalcIC` arguments `t0`, `yy0`, and `yp0` were removed and a new function, `IDAGetconsistentIC` is provided (see

§4.5.5 and §4.5.10.3 for details).

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

## Changes in v2.4.0

FIDA, a FORTRAN-C interface module, was added (for details see Chapter 5).

IDASPCBG and IDASPTFQMR modules have been added to interface with the Scaled Preconditioned Bi-CGstab (SPBCGS) and Scaled Preconditioned Transpose-Free Quasi-Minimal Residual (SPTFQMR) linear solver modules, respectively (for details see Chapter 4). At the same time, function type names for Scaled Preconditioned Iterative Linear Solvers were added for the user-supplied Jacobian-times-vector and preconditioner setup and solve functions.

The rootfinding feature was added, whereby the roots of a set of given functions may be computed during the integration of the DAE system.

A user-callable routine was added to access the estimated local error vector.

The deallocation functions now take as arguments the address of the respective memory block pointer.

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

## Changes in v2.3.0

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

## Changes in v2.2.2

Minor corrections and improvements were made to the build system. A new chapter in the User Guide was added — with constants that appear in the user interface.

## Changes in v2.2.1

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

## Changes in v2.2.0

The major changes from the previous version involve a redesign of the user interface across the entire SUNDIALS suite. We have eliminated the mechanism of providing optional inputs and extracting optional statistics from the solver through the `iopt` and `ropt` arrays. Instead, IDA now provides a set of routines (with prefix `IDASet`) to change the default values for various quantities controlling the solver and a set of extraction routines (with prefix `IDAGet`) 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 §4.5.8 and §4.5.10.

Additionally, the interfaces to several user-supplied routines (such as those providing Jacobians 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 `Get`-type functions.

Installation of IDA (and all of SUNDIALS) has been completely redesigned and is now based on configure scripts.

## 1.2 Reading this User Guide

The structure of this document is as follows:

- In Chapter 2, we give short descriptions of the numerical methods implemented by IDA for the solution of initial value problems for systems of DAEs, along with short descriptions of preconditioning (§2.2) and rootfinding (§2.3).
- The following chapter describes the structure of the SUNDIALS suite of solvers (§3.1) and the software organization of the IDA solver (§3.2).
- Chapter 4 is the main usage document for IDA for C applications. It includes a complete description of the user interface for the integration of DAE initial value problems.
- In Chapter 5, we describe FIDA, an interface module for the use of IDA with FORTRAN applications.
- Chapter 6 gives a brief overview of the generic NVECTOR module shared among the various components of SUNDIALS, as well as details on the NVECTOR implementations provided with SUNDIALS.
- Chapter 7 gives a brief overview of the generic SUNMATRIX module shared among the various components of SUNDIALS, and details on the SUNMATRIX implementations provided with SUNDIALS: a dense implementation (§7.1), a banded implementation (§7.2) and a sparse implementation (§7.3).
- Chapter 8 gives a brief overview of the generic SUNLINSOL module shared among the various components of SUNDIALS. This chapter contains details on the SUNLINSOL implementations provided with SUNDIALS. The chapter also contains details on the SUNLINSOL implementations provided with SUNDIALS that interface with external linear solver libraries.
- Chapter 9 describes the SUNNONLINSOL API and nonlinear solver implementations shared among the various components of SUNDIALS.
- Finally, in the appendices, we provide detailed instructions for the installation of IDA, within the structure of SUNDIALS (Appendix A), as well as a list of all the constants used for input to and output from IDA functions (Appendix B).

Finally, the reader should be aware of the following notational conventions in this user guide: program listings and identifiers (such as `IDAInit`) within textual explanations appear in typewriter type style; fields in C structures (such as *content*) appear in italics; and packages or modules, such as IDADLS, are written in all capitals. Usage and installation instructions that constitute important warnings are marked with a triangular symbol in the margin.



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

## 1.3 SUNDIALS Release License

The SUNDIALS packages are released open source, under a BSD license. The only requirements of the BSD license are preservation of copyright and a standard disclaimer of liability. Our Copyright notice is below along with the license.

**\*\*PLEASE NOTE\*\*** If you are using SUNDIALS with any third party libraries linked in (e.g.,



LaPACK, KLU, SuperLU\_MT, PETsc, or *hypre*), be sure to review the respective license of the package as that license may have more restrictive terms than the SUNDIALS license. For example, if someone builds SUNDIALS with a statically linked KLU, the build is subject to terms of the LGPL license (which is what KLU is released with) and *not* the SUNDIALS BSD license anymore.

### 1.3.1 Copyright Notices

All SUNDIALS packages except ARKode are subject to the following Copyright notice.

#### 1.3.1.1 SUNDIALS Copyright

Copyright (c) 2002-2016, Lawrence Livermore National Security. Produced at the Lawrence Livermore National Laboratory. Written by A.C. Hindmarsh, D.R. Reynolds, R. Serban, C.S. Woodward, S.D. Cohen, A.G. Taylor, S. Peles, L.E. Banks, and D. Shumaker.

UCRL-CODE-155951 (CVODE)

UCRL-CODE-155950 (CVODES)

UCRL-CODE-155952 (IDA)

UCRL-CODE-237203 (IDAS)

LLNL-CODE-665877 (KINSOL)

All rights reserved.

#### 1.3.1.2 ARKode Copyright

ARKode is subject to the following joint Copyright notice. Copyright (c) 2015-2016, Southern Methodist University and Lawrence Livermore National Security Written by D.R. Reynolds, D.J. Gardner, A.C. Hindmarsh, C.S. Woodward, and J.M. Sexton.

LLNL-CODE-667205 (ARKODE)

All rights reserved.

### 1.3.2 BSD License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the disclaimer below.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the disclaimer (as noted below) in the documentation and/or other materials provided with the distribution.
3. Neither the name of the LLNS/LLNL nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL LAWRENCE LIVERMORE NATIONAL SECURITY, LLC, THE U.S. DEPARTMENT OF ENERGY OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Additional BSD Notice

1. This notice is required to be provided under our contract with the U.S. Department of Energy (DOE). This work was produced at Lawrence Livermore National Laboratory under Contract No. DE-AC52-07NA27344 with the DOE.
2. Neither the United States Government nor Lawrence Livermore National Security, LLC nor any of their employees, makes any warranty, express or implied, or assumes any 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.
3. Also, reference herein to any specific commercial products, process, or services by trade name, trademark, manufacturer or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.



## Chapter 2

# Mathematical Considerations

IDA solves the initial-value problem (IVP) for a DAE system of the general form

$$F(t, y, \dot{y}) = 0, \quad y(t_0) = y_0, \quad \dot{y}(t_0) = \dot{y}_0, \quad (2.1)$$

where  $y$ ,  $\dot{y}$ , and  $F$  are vectors in  $\mathbf{R}^N$ ,  $t$  is the independent variable,  $\dot{y} = dy/dt$ , and initial values  $y_0$ ,  $\dot{y}_0$  are given. (Often  $t$  is time, but it certainly need not be.)

### 2.1 IVP solution

Prior to integrating a DAE initial-value problem, an important requirement is that the pair of vectors  $y_0$  and  $\dot{y}_0$  are both initialized to satisfy the DAE residual  $F(t_0, y_0, \dot{y}_0) = 0$ . For a class of problems that includes so-called semi-explicit index-one systems, IDA provides a routine that computes consistent initial conditions from a user's initial guess [8]. For this, the user must identify sub-vectors of  $y$  (not necessarily contiguous), denoted  $y_d$  and  $y_a$ , which are its differential and algebraic parts, respectively, such that  $F$  depends on  $\dot{y}_d$  but not on any components of  $\dot{y}_a$ . The assumption that the system is “index one” means that for a given  $t$  and  $y_d$ , the system  $F(t, y, \dot{y}) = 0$  defines  $y_a$  uniquely. In this case, a solver within IDA computes  $y_a$  and  $\dot{y}_d$  at  $t = t_0$ , given  $y_d$  and an initial guess for  $y_a$ . A second available option with this solver also computes all of  $y(t_0)$  given  $\dot{y}(t_0)$ ; this is intended mainly for quasi-steady-state problems, where  $\dot{y}(t_0) = 0$  is given. In both cases, IDA solves the system  $F(t_0, y_0, \dot{y}_0) = 0$  for the unknown components of  $y_0$  and  $\dot{y}_0$ , using Newton iteration augmented with a line search global strategy. In doing this, it makes use of the existing machinery that is to be used for solving the linear systems during the integration, in combination with certain tricks involving the step size (which is set artificially for this calculation). For problems that do not fall into either of these categories, the user is responsible for passing consistent values, or risks failure in the numerical integration.

The integration method used in IDA is the variable-order, variable-coefficient BDF (Backward Differentiation Formula), in fixed-leading-coefficient form [4]. The method order ranges from 1 to 5, with the BDF of order  $q$  given by the multistep formula

$$\sum_{i=0}^q \alpha_{n,i} y_{n-i} = h_n \dot{y}_n, \quad (2.2)$$

where  $y_n$  and  $\dot{y}_n$  are the computed approximations to  $y(t_n)$  and  $\dot{y}(t_n)$ , respectively, and the step size is  $h_n = t_n - t_{n-1}$ . The coefficients  $\alpha_{n,i}$  are uniquely determined by the order  $q$ , and the history of the step sizes. The application of the BDF (2.2) to the DAE system (2.1) results in a nonlinear algebraic system to be solved at each step:

$$G(y_n) \equiv F \left( t_n, y_n, h_n^{-1} \sum_{i=0}^q \alpha_{n,i} y_{n-i} \right) = 0. \quad (2.3)$$

By default IDA solves (2.3) with a Newton iteration but IDA also allows for user-defined nonlinear solvers (see Chapter 9). Each Newton iteration requires the solution of a linear system of the form

$$J[y_{n(m+1)} - y_{n(m)}] = -G(y_{n(m)}), \quad (2.4)$$

where  $y_{n(m)}$  is the  $m$ -th approximation to  $y_n$ . Here  $J$  is some approximation to the system Jacobian

$$J = \frac{\partial G}{\partial y} = \frac{\partial F}{\partial y} + \alpha \frac{\partial F}{\partial \dot{y}}, \quad (2.5)$$

where  $\alpha = \alpha_{n,0}/h_n$ . The scalar  $\alpha$  changes whenever the step size or method order changes.

For the solution of the linear systems within the Newton iteration, IDA provides several choices, including the option of a user-supplied linear solver module (see Chapter 8). The linear solver modules distributed with SUNDIALS are organized in two families, a *direct* family comprising direct linear solvers for dense, banded, or sparse matrices and a *spils* family comprising scaled preconditioned iterative (Krylov) linear solvers. The methods offered through these modules are as follows:

- dense direct solvers, using either an internal implementation or a BLAS/LAPACK implementation (serial or threaded vector modules only),
- band direct solvers, using either an internal implementation or a BLAS/LAPACK implementation (serial or threaded vector modules only),
- sparse direct solver interfaces, using either the KLU sparse solver library [15, 1], or the thread-enabled SuperLU\_MT sparse solver library [29, 17, 2] (serial or threaded vector modules only) [Note that users will need to download and install the KLU or SUPERLUMT packages independent of IDA],
- SPGMR, a scaled preconditioned GMRES (Generalized Minimal Residual method) solver without restarts,
- SPFGMR, a scaled preconditioned FGMRES (Flexible Generalized Minimal Residual method) solver,
- SPBCGS, a scaled preconditioned Bi-CGStab (Bi-Conjugate Gradient Stable method) solver,
- SPTFQMR, a scaled preconditioned TFQMR (Transpose-Free Quasi-Minimal Residual method) solver, or
- PCG, a scaled preconditioned CG (Conjugate Gradient method) solver.

For large stiff systems, where direct methods are not feasible, the combination of a BDF integrator and any of the preconditioned Krylov methods (SPGMR, SPBCGS, or SPTFQMR) yields a powerful tool because it combines established methods for stiff integration, nonlinear iteration, and Krylov (linear) iteration with a problem-specific treatment of the dominant source of stiffness, in the form of the user-supplied preconditioner matrix [6]. For the *spils* linear solvers, preconditioning is allowed only on the left (see §2.2). Note that the dense, band, and sparse direct linear solvers can only be used with serial and threaded vector representations.

In the process of controlling errors at various levels, IDA uses a weighted root-mean-square norm, denoted  $\|\cdot\|_{\text{WRMS}}$ , for all error-like quantities. The multiplicative weights used are based on the current solution and on the relative and absolute tolerances input by the user, namely

$$W_i = 1/[\text{RTOL} \cdot |y_i| + \text{ATOL}_i]. \quad (2.6)$$

Because  $1/W_i$  represents a tolerance in the component  $y_i$ , a vector whose norm is 1 is regarded as “small.” For brevity, we will usually drop the subscript WRMS on norms in what follows.

In the case of a matrix-based linear solver, the default Newton iteration is a Modified Newton iteration, in that the Jacobian  $J$  is fixed (and usually out of date) throughout the nonlinear iterations, with a coefficient  $\bar{\alpha}$  in place of  $\alpha$  in  $J$ . However, in the case that a matrix-free iterative linear solver is



used, the default Newton iteration is an Inexact Newton iteration, in which  $J$  is applied in a matrix-free manner, with matrix-vector products  $Jv$  obtained by either difference quotients or a user-supplied routine. In this case, the linear residual  $J\Delta y + G$  is nonzero but controlled. With the default Newton iteration, the matrix  $J$  and preconditioner matrix  $P$  are updated as infrequently as possible to balance the high costs of matrix operations against other costs. Specifically, this matrix update occurs when:

- starting the problem,
- the value  $\bar{\alpha}$  at the last update is such that  $\alpha/\bar{\alpha} < 3/5$  or  $\alpha/\bar{\alpha} > 5/3$ , or
- a non-fatal convergence failure occurred with an out-of-date  $J$  or  $P$ .

The above strategy balances the high cost of frequent matrix evaluations and preprocessing with the slow convergence due to infrequent updates. To reduce storage costs on an update, Jacobian information is always reevaluated from scratch.

The default stopping test for nonlinear solver iterations in IDA ensures that the iteration error  $y_n - y_{n(m)}$  is small relative to  $y$  itself. For this, we estimate the linear convergence rate at all iterations  $m > 1$  as

$$R = \left( \frac{\delta_m}{\delta_1} \right)^{\frac{1}{m-1}},$$

where the  $\delta_m = y_{n(m)} - y_{n(m-1)}$  is the correction at iteration  $m = 1, 2, \dots$ . The nonlinear solver iteration is halted if  $R > 0.9$ . The convergence test at the  $m$ -th iteration is then

$$S\|\delta_m\| < 0.33, \quad (2.7)$$

where  $S = R/(R-1)$  whenever  $m > 1$  and  $R \leq 0.9$ . The user has the option of changing the constant in the convergence test from its default value of 0.33. The quantity  $S$  is set to  $S = 20$  initially and whenever  $J$  or  $P$  is updated, and it is reset to  $S = 100$  on a step with  $\alpha \neq \bar{\alpha}$ . Note that at  $m = 1$ , the convergence test (2.7) uses an old value for  $S$ . Therefore, at the first nonlinear solver iteration, we make an additional test and stop the iteration if  $\|\delta_1\| < 0.33 \cdot 10^{-4}$  (since such a  $\delta_1$  is probably just noise and therefore not appropriate for use in evaluating  $R$ ). We allow only a small number (default value 4) of nonlinear iterations. If convergence fails with  $J$  or  $P$  current, we are forced to reduce the step size  $h_n$ , and we replace  $h_n$  by  $h_n/4$ . The integration is halted after a preset number (default value 10) of convergence failures. Both the maximum number of allowable nonlinear iterations and the maximum number of nonlinear convergence failures can be changed by the user from their default values.

When an iterative method is used to solve the linear system, to minimize the effect of linear iteration errors on the nonlinear and local integration error controls, we require the preconditioned linear residual to be small relative to the allowed error in the nonlinear iteration, i.e.,  $\|P^{-1}(Jx+G)\| < 0.05 \cdot 0.33$ . The safety factor 0.05 can be changed by the user.

When the Jacobian is stored using either dense or band SUNMATRIX objects, the Jacobian  $J$  defined in (2.5) can be either supplied by the user or have IDA compute one internally by difference quotients. In the latter case, we use the approximation

$$J_{ij} = [F_i(t, y + \sigma_j e_j, \dot{y} + \alpha \sigma_j e_j) - F_i(t, y, \dot{y})]/\sigma_j, \text{ with} \\ \sigma_j = \sqrt{U} \max\{|y_j|, |h\dot{y}_j|, 1/W_j\} \text{sign}(h\dot{y}_j),$$

where  $U$  is the unit roundoff,  $h$  is the current step size, and  $W_j$  is the error weight for the component  $y_j$  defined by (2.6). We note that with sparse and user-supplied SUNMATRIX objects, the Jacobian *must* be supplied by a user routine.

In the case of an iterative linear solver, if a routine for  $Jv$  is not supplied, such products are approximated by

$$Jv = [F(t, y + \sigma v, \dot{y} + \alpha \sigma v) - F(t, y, \dot{y})]/\sigma,$$

where the increment  $\sigma$  is  $1/\|v\|$ . As an option, the user can specify a constant factor that is inserted into this expression for  $\sigma$ .

During the course of integrating the system, IDA computes an estimate of the local truncation error, LTE, at the  $n$ -th time step, and requires this to satisfy the inequality

$$\|\text{LTE}\|_{\text{WRMS}} \leq 1.$$

Asymptotically, LTE varies as  $h^{q+1}$  at step size  $h$  and order  $q$ , as does the predictor-corrector difference  $\Delta_n \equiv y_n - y_{n(0)}$ . Thus there is a constant  $C$  such that

$$\text{LTE} = C\Delta_n + O(h^{q+2}),$$

and so the norm of LTE is estimated as  $|C| \cdot \|\Delta_n\|$ . In addition, IDA requires that the error in the associated polynomial interpolant over the current step be bounded by 1 in norm. The leading term of the norm of this error is bounded by  $\bar{C}\|\Delta_n\|$  for another constant  $\bar{C}$ . Thus the local error test in IDA is

$$\max\{|C|, \bar{C}\}\|\Delta_n\| \leq 1. \quad (2.8)$$

A user option is available by which the algebraic components of the error vector are omitted from the test (2.8), if these have been so identified.

In IDA, the local error test is tightly coupled with the logic for selecting the step size and order. First, there is an initial phase that is treated specially; for the first few steps, the step size is doubled and the order raised (from its initial value of 1) on every step, until (a) the local error test (2.8) fails, (b) the order is reduced (by the rules given below), or (c) the order reaches 5 (the maximum). For step and order selection on the general step, IDA uses a different set of local error estimates, based on the asymptotic behavior of the local error in the case of fixed step sizes. At each of the orders  $q'$  equal to  $q$ ,  $q-1$  (if  $q > 1$ ),  $q-2$  (if  $q > 2$ ), or  $q+1$  (if  $q < 5$ ), there are constants  $C(q')$  such that the norm of the local truncation error at order  $q'$  satisfies

$$\text{LTE}(q') = C(q')\|\phi(q'+1)\| + O(h^{q'+2}),$$

where  $\phi(k)$  is a modified divided difference of order  $k$  that is retained by IDA (and behaves asymptotically as  $h^k$ ). Thus the local truncation errors are estimated as  $\text{ELTE}(q') = C(q')\|\phi(q'+1)\|$  to select step sizes. But the choice of order in IDA is based on the requirement that the scaled derivative norms,  $\|h^k y^{(k)}\|$ , are monotonically decreasing with  $k$ , for  $k$  near  $q$ . These norms are again estimated using the  $\phi(k)$ , and in fact

$$\|h^{q'+1} y^{(q'+1)}\| \approx T(q') \equiv (q'+1)\text{ELTE}(q').$$

The step/order selection begins with a test for monotonicity that is made even *before* the local error test is performed. Namely, the order is reset to  $q' = q-1$  if (a)  $q = 2$  and  $T(1) \leq T(2)/2$ , or (b)  $q > 2$  and  $\max\{T(q-1), T(q-2)\} \leq T(q)$ ; otherwise  $q' = q$ . Next the local error test (2.8) is performed, and if it fails, the step is redone at order  $q \leftarrow q'$  and a new step size  $h'$ . The latter is based on the  $h^{q+1}$  asymptotic behavior of  $\text{ELTE}(q)$ , and, with safety factors, is given by

$$\eta = h'/h = 0.9/[2\text{ELTE}(q)]^{1/(q+1)}.$$

The value of  $\eta$  is adjusted so that  $0.25 \leq \eta \leq 0.9$  before setting  $h \leftarrow h' = \eta h$ . If the local error test fails a second time, IDA uses  $\eta = 0.25$ , and on the third and subsequent failures it uses  $q = 1$  and  $\eta = 0.25$ . After 10 failures, IDA returns with a give-up message.

As soon as the local error test has passed, the step and order for the next step may be adjusted. No such change is made if  $q' = q-1$  from the prior test, if  $q = 5$ , or if  $q$  was increased on the previous step. Otherwise, if the last  $q+1$  steps were taken at a constant order  $q < 5$  and a constant step size, IDA considers raising the order to  $q+1$ . The logic is as follows: (a) If  $q = 1$ , then reset  $q = 2$  if  $T(2) < T(1)/2$ . (b) If  $q > 1$  then

- reset  $q \leftarrow q-1$  if  $T(q-1) \leq \min\{T(q), T(q+1)\}$ ;
- else reset  $q \leftarrow q+1$  if  $T(q+1) < T(q)$ ;

- leave  $q$  unchanged otherwise [then  $T(q-1) > T(q) \leq T(q+1)$ ].

In any case, the new step size  $h'$  is set much as before:

$$\eta = h'/h = 1/[2 \text{ELTE}(q)]^{1/(q+1)}.$$

The value of  $\eta$  is adjusted such that (a) if  $\eta > 2$ ,  $\eta$  is reset to 2; (b) if  $\eta \leq 1$ ,  $\eta$  is restricted to  $0.5 \leq \eta \leq 0.9$ ; and (c) if  $1 < \eta < 2$  we use  $\eta = 1$ . Finally  $h$  is reset to  $h' = \eta h$ . Thus we do not increase the step size unless it can be doubled. See [4] for details.

IDA permits the user to impose optional inequality constraints on individual components of the solution vector  $y$ . Any of the following four constraints can be imposed:  $y_i > 0$ ,  $y_i < 0$ ,  $y_i \geq 0$ , or  $y_i \leq 0$ . The constraint satisfaction is tested after a successful nonlinear system solution. If any constraint fails, we declare a convergence failure of the nonlinear iteration and reduce the step size. Rather than cutting the step size by some arbitrary factor, IDA estimates a new step size  $h'$  using a linear approximation of the components in  $y$  that failed the constraint test (including a safety factor of 0.9 to cover the strict inequality case). These additional constraints are also imposed during the calculation of consistent initial conditions.

Normally, IDA takes steps until a user-defined output value  $t = t_{\text{out}}$  is overtaken, and then computes  $y(t_{\text{out}})$  by interpolation. However, a “one step” mode option is available, where control returns to the calling program after each step. There are also options to force IDA not to integrate past a given stopping point  $t = t_{\text{stop}}$ .

## 2.2 Preconditioning

When using a nonlinear solver that requires the solution of a linear system of the form  $J\Delta y = -G$  (e.g., the default Newton iteration), IDA makes repeated use of a linear solver. If this linear system solve is done with one of the scaled preconditioned iterative linear solvers supplied with SUNDIALS, these solvers are rarely successful if used without preconditioning; it is generally necessary to precondition the system in order to obtain acceptable efficiency. A system  $Ax = b$  can be preconditioned on the left, on the right, or on both sides. The Krylov method is then applied to a system with the matrix  $P^{-1}A$ , or  $AP^{-1}$ , or  $P_L^{-1}AP_R^{-1}$ , instead of  $A$ . However, within IDA, preconditioning is allowed *only* on the left, so that the iterative method is applied to systems  $(P^{-1}J)\Delta y = -P^{-1}G$ . Left preconditioning is required to make the norm of the linear residual in the nonlinear iteration meaningful; in general,  $\|J\Delta y + G\|$  is meaningless, since the weights used in the WRMS-norm correspond to  $y$ .

In order to improve the convergence of the Krylov iteration, the preconditioner matrix  $P$  should in some sense approximate the system matrix  $A$ . Yet at the same time, in order to be cost-effective, the matrix  $P$  should be reasonably efficient to evaluate and solve. Finding a good point in this tradeoff between rapid convergence and low cost can be very difficult. Good choices are often problem-dependent (for example, see [6] for an extensive study of preconditioners for reaction-transport systems).

Typical preconditioners used with IDA are based on approximations to the iteration matrix of the systems involved; in other words,  $P \approx \frac{\partial F}{\partial y} + \alpha \frac{\partial F}{\partial \dot{y}}$ , where  $\alpha$  is a scalar inversely proportional to the integration step size  $h$ . Because the Krylov iteration occurs within a nonlinear solver iteration and further also within a time integration, and since each of these iterations has its own test for convergence, the preconditioner may use a very crude approximation, as long as it captures the dominant numerical feature(s) of the system. We have found that the combination of a preconditioner with the Newton-Krylov iteration, using even a fairly poor approximation to the Jacobian, can be surprisingly superior to using the same matrix without Krylov acceleration (i.e., a modified Newton iteration), as well as to using the Newton-Krylov method with no preconditioning.

## 2.3 Rootfinding

The IDA solver has been augmented to include a rootfinding feature. This means that, while integrating the Initial Value Problem (2.1), IDA can also find the roots of a set of user-defined functions  $g_i(t, y, \dot{y})$  that depend on  $t$ , the solution vector  $y = y(t)$ , and its  $t$ -derivative  $\dot{y}(t)$ . The number of these root

functions is arbitrary, and if more than one  $g_i$  is found to have a root in any given interval, the various root locations are found and reported in the order that they occur on the  $t$  axis, in the direction of integration.

Generally, this rootfinding feature finds only roots of odd multiplicity, corresponding to changes in sign of  $g_i(t, y(t), \dot{y}(t))$ , denoted  $g_i(t)$  for short. If a user root function has a root of even multiplicity (no sign change), it will probably be missed by IDA. If such a root is desired, the user should reformulate the root function so that it changes sign at the desired root.

The basic scheme used is to check for sign changes of any  $g_i(t)$  over each time step taken, and then (when a sign change is found) to home in on the root (or roots) with a modified secant method [22]. In addition, each time  $g$  is computed, IDA checks to see if  $g_i(t) = 0$  exactly, and if so it reports this as a root. However, if an exact zero of any  $g_i$  is found at a point  $t$ , IDA computes  $g$  at  $t + \delta$  for a small increment  $\delta$ , slightly further in the direction of integration, and if any  $g_i(t + \delta) = 0$  also, IDA stops and reports an error. This way, each time IDA takes a time step, it is guaranteed that the values of all  $g_i$  are nonzero at some past value of  $t$ , beyond which a search for roots is to be done.

At any given time in the course of the time-stepping, after suitable checking and adjusting has been done, IDA has an interval  $(t_{lo}, t_{hi}]$  in which roots of the  $g_i(t)$  are to be sought, such that  $t_{hi}$  is further ahead in the direction of integration, and all  $g_i(t_{lo}) \neq 0$ . The endpoint  $t_{hi}$  is either  $t_n$ , the end of the time step last taken, or the next requested output time  $t_{out}$  if this comes sooner. The endpoint  $t_{lo}$  is either  $t_{n-1}$ , or the last output time  $t_{out}$  (if this occurred within the last step), or the last root location (if a root was just located within this step), possibly adjusted slightly toward  $t_n$  if an exact zero was found. The algorithm checks  $g$  at  $t_{hi}$  for zeros and for sign changes in  $(t_{lo}, t_{hi})$ . If no sign changes are found, then either a root is reported (if some  $g_i(t_{hi}) = 0$ ) or we proceed to the next time interval (starting at  $t_{hi}$ ). If one or more sign changes were found, then a loop is entered to locate the root to within a rather tight tolerance, given by

$$\tau = 100 * U * (|t_n| + |h|) \quad (U = \text{unit roundoff}) .$$

Whenever sign changes are seen in two or more root functions, the one deemed most likely to have its root occur first is the one with the largest value of  $|g_i(t_{hi})|/|g_i(t_{hi}) - g_i(t_{lo})|$ , corresponding to the closest to  $t_{lo}$  of the secant method values. At each pass through the loop, a new value  $t_{mid}$  is set, strictly within the search interval, and the values of  $g_i(t_{mid})$  are checked. Then either  $t_{lo}$  or  $t_{hi}$  is reset to  $t_{mid}$  according to which subinterval is found to have the sign change. If there is none in  $(t_{lo}, t_{mid})$  but some  $g_i(t_{mid}) = 0$ , then that root is reported. The loop continues until  $|t_{hi} - t_{lo}| < \tau$ , and then the reported root location is  $t_{hi}$ .

In the loop to locate the root of  $g_i(t)$ , the formula for  $t_{mid}$  is

$$t_{mid} = t_{hi} - (t_{hi} - t_{lo})g_i(t_{hi})/[g_i(t_{hi}) - \alpha g_i(t_{lo})] ,$$

where  $\alpha$  a weight parameter. On the first two passes through the loop,  $\alpha$  is set to 1, making  $t_{mid}$  the secant method value. Thereafter,  $\alpha$  is reset according to the side of the subinterval (low vs high, i.e. toward  $t_{lo}$  vs toward  $t_{hi}$ ) in which the sign change was found in the previous two passes. If the two sides were opposite,  $\alpha$  is set to 1. If the two sides were the same,  $\alpha$  is halved (if on the low side) or doubled (if on the high side). The value of  $t_{mid}$  is closer to  $t_{lo}$  when  $\alpha < 1$  and closer to  $t_{hi}$  when  $\alpha > 1$ . If the above value of  $t_{mid}$  is within  $\tau/2$  of  $t_{lo}$  or  $t_{hi}$ , it is adjusted inward, such that its fractional distance from the endpoint (relative to the interval size) is between .1 and .5 (.5 being the midpoint), and the actual distance from the endpoint is at least  $\tau/2$ .

# Chapter 3

## Code Organization

### 3.1 SUNDIALS organization

The family of solvers referred to as SUNDIALS consists of the solvers CVODE and ARKODE (for ODE systems), KINSOL (for nonlinear algebraic systems), and IDA (for differential-algebraic systems). In addition, SUNDIALS also includes variants of CVODE and IDA with sensitivity analysis capabilities (using either forward or adjoint methods), called CVODES and IDAS, respectively.

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 Figs. 3.1 and 3.2). The following is a list of the solver packages presently available, and the basic functionality of each:

- CVODE, a solver for stiff and nonstiff ODE systems  $dy/dt = f(t, y)$  based on Adams and BDF methods;
- CVODES, a solver for stiff and nonstiff ODE systems with sensitivity analysis capabilities;
- ARKODE, a solver for ODE systems  $Mdy/dt = f_E(t, y) + f_I(t, y)$  based on additive Runge-Kutta methods;
- IDA, a solver for differential-algebraic systems  $F(t, y, \dot{y}) = 0$  based on BDF methods;
- IDAS, a solver for differential-algebraic systems with sensitivity analysis capabilities;
- KINSOL, a solver for nonlinear algebraic systems  $F(u) = 0$ .

### 3.2 IDA organization

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

The overall organization of the IDA package is shown in Figure 3.3. The central integration module, implemented in the files `ida.h`, `ida_impl.h`, and `ida.c`, deals with the evaluation of integration coefficients, estimation of local error, selection of stepsize and order, and interpolation to user output points, among other issues.

IDA utilizes generic linear and nonlinear solver modules defined by the SUNLINSOL API (see Chapter 8) and SUNNONLINSOL API (see Chapter 9) respectively. As such, IDA has no knowledge of the method being used to solve the linear and nonlinear systems that arise in each time step. For any given user problem, there exists a single nonlinear solver interface and, if necessary, a linear system solver interface is specified, and invoked as needed during the integration. While SUNDIALS includes a fixed-point nonlinear solver module, it is not currently supported in IDA (note the fixed-point module is listed in Figure 3.1 but not Figure 3.3).

IDA now has a single unified linear solver interface, IDALS, supporting both direct and iterative linear solvers built using the generic SUNLINSOL API (see Chapter 8). These solvers may utilize a

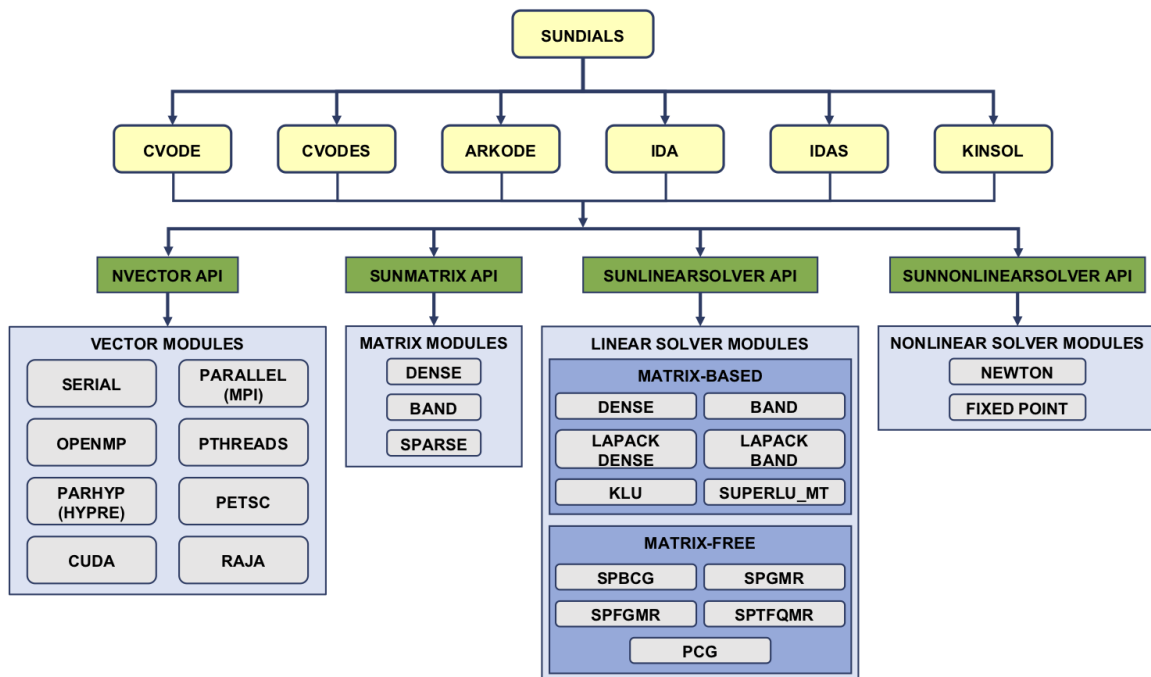


Figure 3.1: High-level diagram of the SUNDIALS suite

SUNMATRIX object (see Chapter 7) for storing Jacobian information, or they may be matrix-free. Since IDA can operate on any valid SUNLINSOL implementation, the set of linear solver modules available to IDA will expand as new SUNLINSOL modules are developed.

For users employing dense or banded Jacobian matrices, IDALS includes algorithms for their approximation through difference quotients, but the user also has the option of supplying the Jacobian (or an approximation to it) directly. This user-supplied routine is required when using sparse or user-supplied Jacobian matrices.

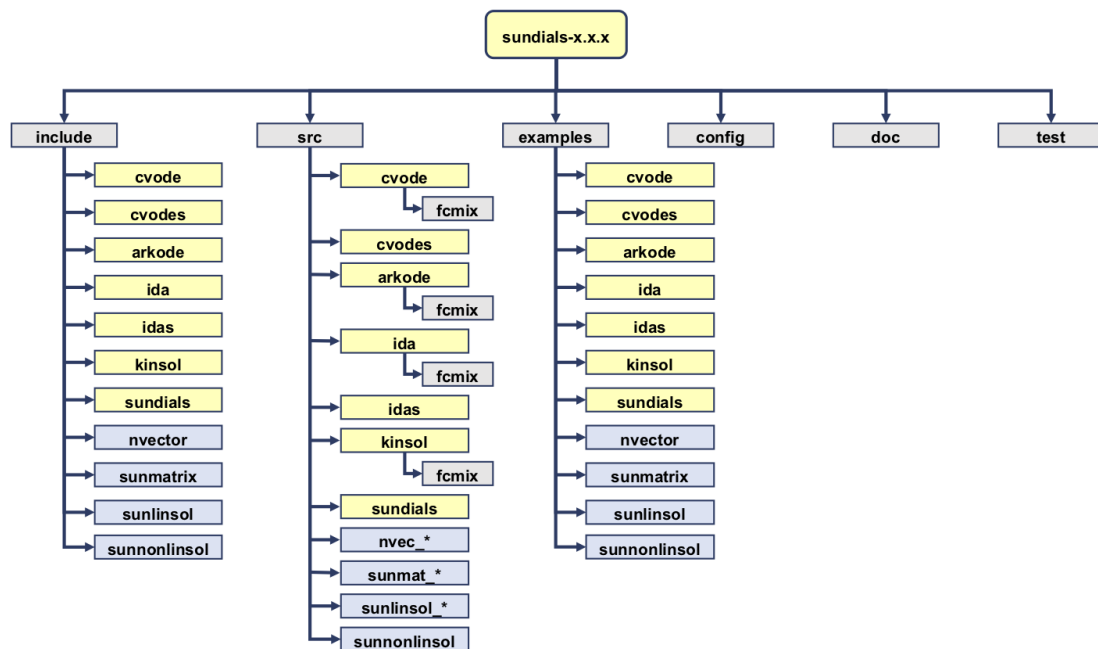
For users employing matrix-free iterative linear solvers, IDALS includes an algorithm for the approximation by difference quotients of the product between the Jacobian matrix and a vector,  $Jv$ . Again, the user has the option of providing routines for this operation, in two phases: setup (preprocessing of Jacobian data) and multiplication.

For preconditioned iterative methods, the preconditioning must be supplied by the user, again in two phases: setup and solve. While there is no default choice of preconditioner analogous to the difference-quotient approximation in the direct case, the references [6, 10], together with the example and demonstration programs included with IDA, offer considerable assistance in building preconditioners.

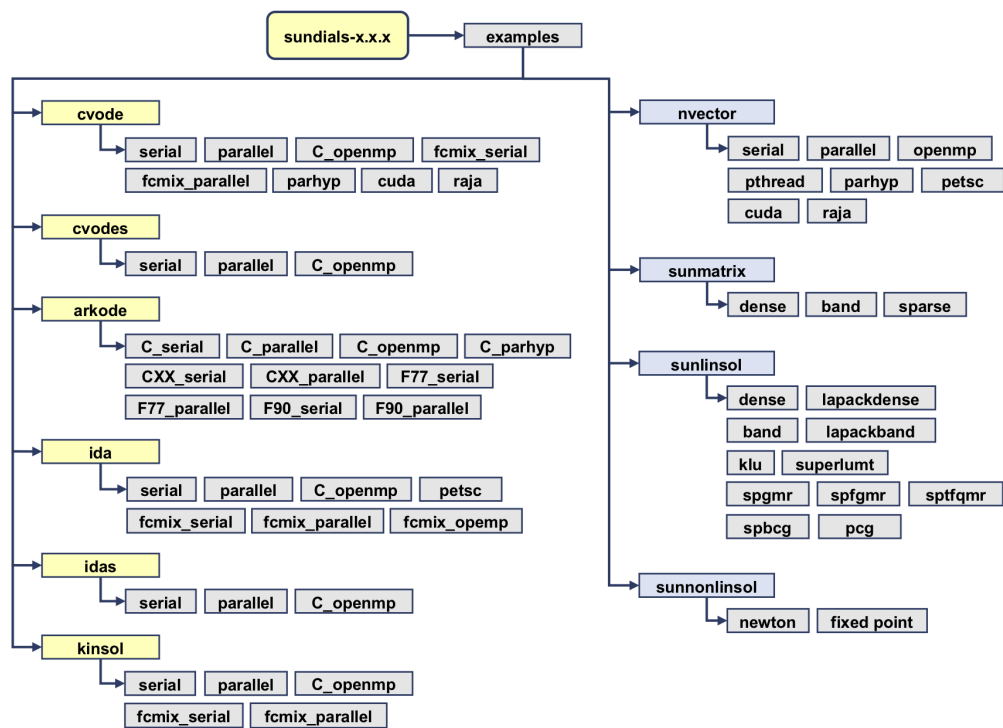
IDA's linear solver interface consists of four primary routines, devoted to (1) memory allocation and initialization, (2) setup of the matrix data involved, (3) solution of the system, and (4) freeing of memory. The setup and solution phases are separate because the evaluation of Jacobians and preconditioners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

IDA also provides a preconditioner module, IDABBDPRE for use with any of the Krylov iterative linear solvers. It works in conjunction with NVECTOR\_PARALLEL and generates a preconditioner that is a block-diagonal matrix with each block being a banded matrix.

All state information used by IDA 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 IDA package, and so, in this



(a) Directory structure of the SUNDIALS source tree



(b) Directory structure of the SUNDIALS examples

Figure 3.2: Organization of the SUNDIALS suite

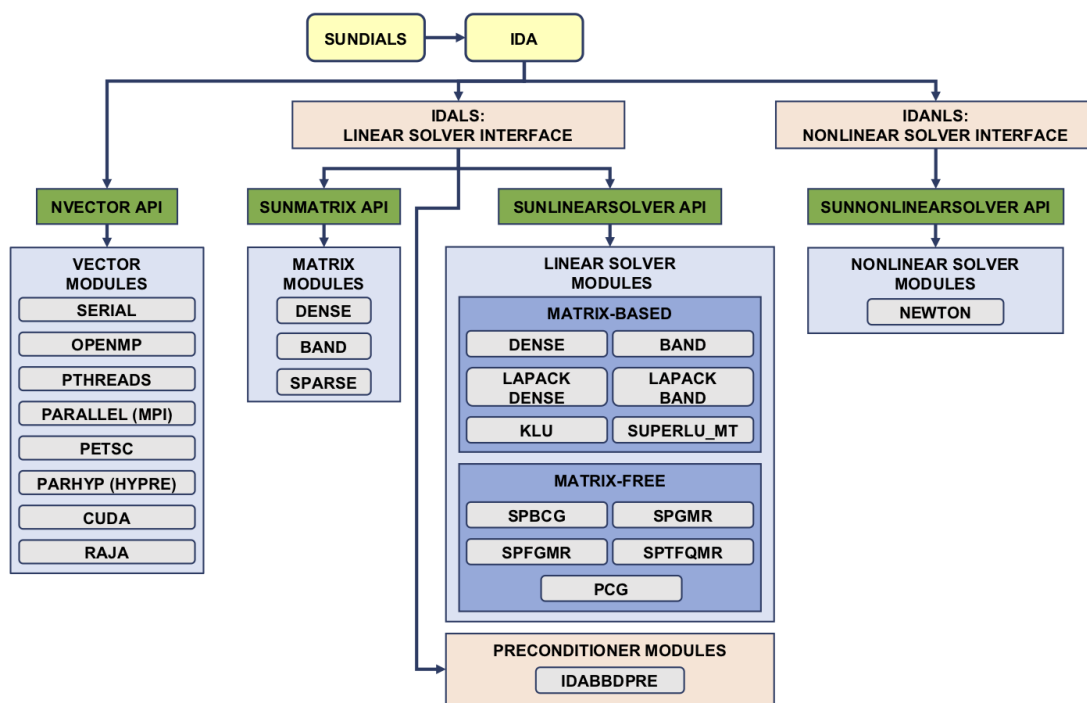


Figure 3.3: Overall structure diagram of the IDA package. Modules specific to IDA begin with “IDA” (IDALS, IDABBDPRE, and IDANLS), all other items correspond to generic solver and auxiliary modules. Note also that the LAPACK, KLU and SUPERLUMT support is through interfaces to external packages. Users will need to download and compile those packages independently.



respect, it is reentrant. State information specific to the linear solver is saved in a separate structure, a pointer to which resides in the IDA memory structure. The reentrancy of IDA was motivated by the situation where two or more problems are solved by intermixed calls to the package from one user program.



## Chapter 4

# Using IDA for C Applications

This chapter is concerned with the use of IDA for the integration of DAEs in a C language setting. The following sections treat the header files, the layout of the user's main program, description of the IDA user-callable functions, and description of user-supplied functions.

The sample programs described in the companion document [25] may also be helpful. Those codes may be used as templates (with the removal of some lines involved in testing), and are included in the IDA package.

Users with applications written in FORTRAN should see Chapter 5, which describes the FORTRAN/C interface module.

The user should be aware that not all SUNLINSOL and SUNMATRIX modules are compatible with all NVECTOR implementations. Details on compatibility are given in the documentation for each SUNMATRIX module (Chapter 7) and each SUNLINSOL module (Chapter 8). For example, NVECTOR\_PARALLEL is not compatible with the dense, banded, or sparse SUNMATRIX types, or with the corresponding dense, banded, or sparse SUNLINSOL modules. Please check Chapters 7 and 8 to verify compatibility between these modules. In addition to that documentation, we note that the preconditioner module IDABBDPRE can only be used with NVECTOR\_PARALLEL. It is not recommended to use a threaded vector module with SuperLU\_MT unless it is the NVECTOR\_OPENMP module, and SuperLU\_MT is also compiled with OpenMP.

IDA uses various constants for both input and output. These are defined as needed in this chapter, but for convenience are also listed separately in Appendix B.

### 4.1 Access to library and header files

At this point, it is assumed that the installation of IDA, following the procedure described in Appendix A, has been completed successfully.

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

- *libdir/libsundials\_ida.lib*,
- *libdir/libsundials\_nvec\*.lib*,

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

- *incdir/include/ida*
- *incdir/include/sundials*
- *incdir/include/nvector*

- `incdir/include/sunmatrix`
- `incdir/include/sunlinsol`
- `incdir/include/sunnonlinsol`

The directories `libdir` and `incdir` are the install library and include directories, respectively. For a default installation, these are `instdir/lib` and `instdir/include`, respectively, where `instdir` is the directory where SUNDIALS was installed (see Appendix A).

Note that an application cannot link to both the IDA and IDAS libraries because both contain user-callable functions with the same names (to ensure that IDAS is backward compatible with IDA). Therefore, applications that contain both DAE problems and DAEs with sensitivity analysis, should use IDAS.

## 4.2 Data types

The `sundials_types.h` file contains the definition of the type `realtype`, which is used by the SUNDIALS solvers for all floating-point data, the definition of the integer type `sunindextype`, which is used for vector and matrix indices, and `booleantype`, which is used for certain logic operations within SUNDIALS.

### 4.2.1 Floating point types

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 §A.1.2).

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

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

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

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

A user program which uses the type `realtype` and the `RCONST` macro to handle floating-point constants is precision-independent except for any calls to precision-specific standard math library functions. (Our example programs use both `realtype` and `RCONST`.) Users can, however, use the type `double`, `float`, or `long double` in their code (assuming that this usage is consistent with the typedef for `realtype`). 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 §A.1.2).

### 4.2.2 Integer types used for vector and matrix indices

The type `sunindextype` can be either a 32- or 64-bit *signed* integer. The default is the portable `int64_t` type, and the user can change it to `int32_t` at the configuration stage. The configuration system will detect if the compiler does not support portable types, and will replace `int32_t` and `int64_t` with `int` and `long int`, respectively, to ensure use of the desired sizes on Linux, Mac OS X,

and Windows platforms. SUNDIALS currently does not support *unsigned* integer types for vector and matrix indices, although these could be added in the future if there is sufficient demand.

A user program which uses `sunindextype` to handle vector and matrix indices will work with both index storage types except for any calls to index storage-specific external libraries. (Our C and C++ example programs use `sunindextype`.) Users can, however, use any one of `int`, `long int`, `int32_t`, `int64_t` or `long long int` in their code, assuming that this usage is consistent with the typedef for `sunindextype` on their architecture). Thus, a previously existing piece of ANSI C code can use SUNDIALS without modifying the code to use `sunindextype`, so long as the SUNDIALS libraries use the appropriate index storage type (for details see §A.1.2).

## 4.3 Header files

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

- `ida/ida.h`, the header file for IDA, which defines the several types and various constants, and includes function prototypes. This includes the header file for IDALS, `ida/ida_ls.h`.

Note that `ida.h` includes `sundials_types.h`, which defines the types `realtype`, `sunindextype`, and `booleantype` and the constants `SUNFALSE` and `SUNTRUE`.

The calling program must also include a `NVECTOR` implementation header file, of the form `nvector/nvector_***.h`. See Chapter 6 for the appropriate name. This file in turn includes the header file `sundials_nvector.h` which defines the abstract `N_Vector` data type.

If using a non-default nonlinear solver module, or when interacting with a `SUNNONLINSOL` module directly, the calling program must also include a `SUNNONLINSOL` implementation header file, of the form `sunnonlinsol/sunnonlinsol_***.h` where `***` is the name of the nonlinear solver module (see Chapter 9 for more information). This file in turn includes the header file `sundials_nonlinearsolver.h` which defines the abstract `SUNNonlinearSolver` data type.

If using a nonlinear solver that requires the solution of a linear system of the form (2.4) (e.g., the default Newton iteration), a linear solver module header file is also required. The header files corresponding to the various linear solver modules available for use with IDA are:

- Direct linear solvers:
  - `sunlinsol/sunlinsol_dense.h`, which is used with the dense linear solver module, `SUNLINSOL_DENSE`;
  - `sunlinsol/sunlinsol_band.h`, which is used with the banded linear solver module, `SUNLINSOL_BAND`;
  - `sunlinsol/sunlinsol_lapackdense.h`, which is used with the LAPACK dense linear solver interface module, `SUNLINSOL_LAPACKDENSE`;
  - `sunlinsol/sunlinsol_lapackband.h`, which is used with the LAPACK banded linear solver interface module, `SUNLINSOL_LAPACKBAND`;
  - `sunlinsol/sunlinsol_klu.h`, which is used with the KLU sparse linear solver interface module, `SUNLINSOL_KLU`;
  - `sunlinsol/sunlinsol_superlunt.h`, which is used with the SUPERLUNT sparse linear solver interface module, `SUNLINSOL_SUPERLUNT`;
- Iterative linear solvers:
  - `sunlinsol/sunlinsol_spgmr.h`, which is used with the scaled, preconditioned GMRES Krylov linear solver module, `SUNLINSOL_SPGMR`;
  - `sunlinsol/sunlinsol_spfgmr.h`, which is used with the scaled, preconditioned FGMRES Krylov linear solver module, `SUNLINSOL_SPFGMR`;

- `sunlinsol/sunlinsol.spbcgs.h`, which is used with the scaled, preconditioned Bi-CGStab Krylov linear solver module, `SUNLINSOL_SPBCGS`;
- `sunlinsol/sunlinsol.sptfqmr.h`, which is used with the scaled, preconditioned TFQMR Krylov linear solver module, `SUNLINSOL_SPTFQMR`;
- `sunlinsol/sunlinsol.pcg.h`, which is used with the scaled, preconditioned CG Krylov linear solver module, `SUNLINSOL_PCG`;

The header files for the `SUNLINSOL_DENSE` and `SUNLINSOL_LAPACKDENSE` linear solver modules include the file `sunmatrix/sunmatrix.dense.h`, which defines the `SUNMATRIX_DENSE` matrix module, as well as various functions and macros acting on such matrices.

The header files for the `SUNLINSOL_BAND` and `SUNLINSOL_LAPACKBAND` linear solver modules include the file `sunmatrix/sunmatrix.band.h`, which defines the `SUNMATRIX_BAND` matrix module, as well as various functions and macros acting on such matrices.

The header files for the `SUNLINSOL_KLU` and `SUNLINSOL_SUPERLUMT` sparse linear solvers include the file `sunmatrix/sunmatrix.sparse.h`, which defines the `SUNMATRIX_SPARSE` matrix module, as well as various functions and macros acting on such matrices.

The header files for the Krylov iterative solvers include the file `sundials/sundials.iterative.h`, which enumerates the kind of preconditioning, and (for the `SPGMR` and `SPFGMR` solvers) the choices for the Gram-Schmidt process.

Other headers may be needed, according to the choice of preconditioner, etc. For example, in the `idaFoodWeb_kry_p` example (see [25]), preconditioning is done with a block-diagonal matrix. For this, even though the `SUNLINSOL_SPGMR` linear solver is used, the header `sundials/sundials.dense.h` is included for access to the underlying generic dense matrix arithmetic routines.

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

The following is a skeleton of the user's main program (or calling program) for the integration of a DAE IVP. Most of the steps are independent of the `NVECTOR`, `SUNMATRIX`, `SUNLINSOL`, and `SUNNONLINSOL` implementations used. For the steps that are not, refer to Chapter 6, 7, 8, and 9 for the specific name of the function to be called or macro to be referenced.

### 1. Initialize parallel or multi-threaded environment, if appropriate

For example, call `MPI_Init` to initialize MPI if used, or set `num_threads`, the number of threads to use within the threaded vector functions, if used.

### 2. Set problem dimensions etc.

This generally includes the problem size `N`, and may include the local vector length `Nlocal`.

Note: The variables `N` and `Nlocal` should be of type `sunindextype`.

### 3. Set vectors of initial values

To set the vectors `y0` and `yp0` to initial values for  $y$  and  $\dot{y}$ , use the appropriate functions defined by the particular `NVECTOR` implementation.

For native `SUNDIALS` vector implementations (except the `CUDA` and `RAJA`-based ones), use a call of the form `y0 = N_VMake_***(..., ydata)` if the `realtype` array `ydata` containing the initial values of  $y$  already exists. Otherwise, create a new vector by making a call of the form `y0 = N_VNew_***(...)`, and then set its elements by accessing the underlying data with a call of the form `ydata = N_VGetArrayPointer(y0)`. See §6.1-6.4 for details.

For the `hypr` and `PETSc` vector wrappers, first create and initialize the underlying vector and then create an `NVECTOR` wrapper with a call of the form `y0 = N_VMake_***(yvec)`, where `yvec` is a `hypr` or `PETSc` vector. Note that calls like `N_VNew_***(...)` and `N_VGetArrayPointer(...)` are not available for these vector wrappers. See §6.5 and §6.6 for details.

If using either the CUDA- or RAJA-based vector implementations use a call of the form `y0 = N_VMake_***(..., c)` where `c` is a pointer to a `suncudavec` or `sunrajavec` vector class if this class already exists. Otherwise, create a new vector by making a call of the form `y0 = N_VNew_***(...)`, and then set its elements by accessing the underlying data where it is located with a call of the form `N_VGetDeviceArrayPointer_***` or `N_VGetHostArrayPointer_***`. Note that the vector class will allocate memory on both the host and device when instantiated. See §6.7-6.8 for details.

Set the vector `yp0` of initial conditions for  $y$  similarly.

#### 4. Create IDA object

Call `ida_mem = IDACreate()` to create the IDA memory block. `IDACreate` returns a pointer to the IDA memory structure. See §4.5.1 for details. This `void *` pointer must then be passed as the first argument to all subsequent IDA function calls.

#### 5. Initialize IDA solver

Call `IDAInit(...)` to provide required problem specifications (residual function, initial time, and initial conditions), allocate internal memory for IDA, and initialize IDA. `IDAInit` returns an error flag to indicate success or an illegal argument value. See §4.5.1 for details.

#### 6. Specify integration tolerances

Call `IDASStolerances(...)` or `IDASvtolerances(...)` to specify, respectively, a scalar relative tolerance and scalar absolute tolerance, or a scalar relative tolerance and a vector of absolute tolerances. Alternatively, call `IDAWFtolerances` to specify a function which sets directly the weights used in evaluating WRMS vector norms. See §4.5.2 for details.

#### 7. Create matrix object

If a nonlinear solver requiring a linear solver will be used (e.g., the default Newton iteration) and the linear solver will be a direct linear solver, then a template Jacobian matrix must be created by calling the appropriate constructor function defined by the particular SUNMATRIX implementation.

For the SUNDIALS-supplied SUNMATRIX implementations, the matrix object may be created using a call of the form

```
SUNMatrix J = SUNBandMatrix(...);
```

or

```
SUNMatrix J = SUNDenseMatrix(...);
```

or

```
SUNMatrix J = SUNSparseMatrix(...);
```

NOTE: The dense, banded, and sparse matrix objects are usable only in a serial or threaded environment.

#### 8. Create linear solver object

If a nonlinear solver requiring a linear solver is chosen (e.g., the default Newton iteration), then the desired linear solver object must be created by calling the appropriate constructor function defined by the particular SUNLINSOL implementation.

For any of the SUNDIALS-supplied SUNLINSOL implementations, the linear solver object may be created using a call of the form

```
SUNLinearSolver LS = SUNLinSol_*(...);
```

where `*` can be replaced with “Dense”, “SPGMR”, or other options, as discussed in §4.5.3 and Chapter 8.

#### 9. Set linear solver optional inputs

Call **\*Set\*** functions from the selected linear solver module to change optional inputs specific to that linear solver. See the documentation for each SUNLINSOL module in Chapter 8 for details.

#### 10. Attach linear solver module

If a nonlinear solver requiring a linear solver is chosen (e.g., the default Newton iteration), then initialize the IDALS linear solver interface by attaching the linear solver object (and matrix object, if applicable) with the following call (for details see §4.5.3):

```
ier = IDASetLinearSolver(...);
```

#### 11. Set optional inputs

Optionally, call **IDASet\*** functions to change from their default values any optional inputs that control the behavior of IDA. See §4.5.8.1 for details.

#### 12. Create nonlinear solver object (*optional*)

If using a non-default nonlinear solver (see §4.5.4), then create the desired nonlinear solver object by calling the appropriate constructor function defined by the particular SUNNONLINSOL implementation (e.g., `NLS = SUNNonlinSol_***(...)`; where **\*\*\*** is the name of the nonlinear solver (see Chapter 9 for details).

#### 13. Attach nonlinear solver module (*optional*)

If using a non-default nonlinear solver, then initialize the nonlinear solver interface by attaching the nonlinear solver object by calling `ier = IDASetNonlinearSolver(ida_mem, NLS)`; (see §4.5.4 for details).

#### 14. Set nonlinear solver optional inputs (*optional*)

Call the appropriate set functions for the selected nonlinear solver module to change optional inputs specific to that nonlinear solver. These *must* be called after **IDAInit** if using the default nonlinear solver or after attaching a new nonlinear solver to IDA, otherwise the optional inputs will be overridden by IDA defaults. See Chapter 9 for more information on optional inputs.

#### 15. Correct initial values

Optionally, call **IDACalcIC** to correct the initial values `y0` and `yp0` passed to **IDAInit**. See §4.5.5. Also see §4.5.8.3 for relevant optional input calls.

#### 16. Specify rootfinding problem

Optionally, call **IDARootInit** to initialize a rootfinding problem to be solved during the integration of the DAE system. See §4.5.6 for details, and see §4.5.8.4 for relevant optional input calls.

#### 17. Advance solution in time

For each point at which output is desired, call `flag = IDASolve(ida_mem, tout, &tret, yret, ypret, itask)`. Here `itask` specifies the return mode. The vector `yret` (which can be the same as the vector `y0` above) will contain  $y(t)$ , while the vector `ypret` (which can be the same as the vector `yp0` above) will contain  $\dot{y}(t)$ . See §4.5.7 for details.

#### 18. Get optional outputs

Call **IDA\*Get\*** functions to obtain optional output. See §4.5.10 for details.

#### 19. Deallocate memory for solution vectors

Upon completion of the integration, deallocate memory for the vectors `yret` and `ypret` (or `y` and `yp`) by calling the appropriate destructor function defined by the NVECTOR implementation:

```
N_VDestroy(yret);
```

and similarly for `ypret`.



## 20. Free solver memory

IDAFree(&ida\_mem) to free the memory allocated for IDA.

## 21. Free nonlinear solver memory (*optional*)

If a non-default nonlinear solver was used, then call SUNNonlinSolFree(NLS) to free any memory allocated for the SUNNONLINSOL object.

## 22. Free linear solver and matrix memory

Call SUNLinSolFree and SUNMatDestroy to free any memory allocated for the linear solver and matrix objects created above.

## 23. Finalize MPI, if used

Call MPI\_Finalize() to terminate MPI.

SUNDIALS provides some linear solvers only as a means for users to get problems running and not as highly efficient solvers. For example, if solving a dense system, we suggest using the LAPACK solvers if the size of the linear system is  $> 50,000$ . (Thanks to A. Nicolai for his testing and recommendation.) Table 4.1 shows the linear solver interfaces available as SUNLINSOL modules and the vector implementations required for use. As an example, one cannot use the dense direct solver interfaces with the MPI-based vector implementation. However, as discussed in Chapter 8 the SUNDIALS packages operate on generic SUNLINSOL objects, allowing a user to develop their own solvers should they so desire.

Table 4.1: SUNDIALS linear solver interfaces and vector implementations that can be used for each.

Linear Solver	Serial	Parallel (MPI)	OpenMP	pThreads	hypr	PETSc	CUDA	RAJA	User Supp.
Dense	✓		✓	✓					✓
Band	✓		✓	✓					✓
LapackDense	✓		✓	✓					✓
LapackBand	✓		✓	✓					✓
KLU	✓		✓	✓					✓
SUPERLUMT	✓		✓	✓					✓
SPGMR	✓	✓	✓	✓	✓	✓	✓	✓	✓
SPFGMR	✓	✓	✓	✓	✓	✓	✓	✓	✓
SPBCGS	✓	✓	✓	✓	✓	✓	✓	✓	✓
SPTFQMR	✓	✓	✓	✓	✓	✓	✓	✓	✓
PCG	✓	✓	✓	✓	✓	✓	✓	✓	✓
User Supp.	✓	✓	✓	✓	✓	✓	✓	✓	✓

## 4.5 User-callable functions

This section describes the IDA functions that are called by the user to set up and solve a DAE. Some of these are required. However, starting with §4.5.8, the functions listed involve optional inputs/outputs or restarting, and those paragraphs can be skipped for a casual use of IDA. In any case, refer to §4.4 for the correct order of these calls.

On an error, each user-callable function returns a negative value and sends an error message to the error handler routine, which prints the message on `stderr` by default. However, the user can set a file as error output or can provide his own error handler function (see §4.5.8.1).

### 4.5.1 IDA 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 DAE solution is complete, as it frees the IDA memory block created and allocated by the first two calls.

#### IDACreate

Call `ida_mem = IDACreate();`

Description The function `IDACreate` instantiates an IDA solver object.

Arguments `IDACreate` has no arguments.

Return value If successful, `IDACreate` returns a pointer to the newly created IDA memory block (of type `void *`). Otherwise it returns `NULL`.

#### IDAInit

Call `flag = IDAInit(ida_mem, res, t0, y0, yp0);`

Description The function `IDAInit` provides required problem and solution specifications, allocates internal memory, and initializes IDA.

Arguments `ida_mem` (`void *`) pointer to the IDA memory block returned by `IDACreate`.

`res` (`IDAResFn`) is the C function which computes the residual function  $F$  in the DAE. This function has the form `res(t, yy, yp, resval, user_data)`. For full details see §4.6.1.

`t0` (`realtype`) is the initial value of  $t$ .

`y0` (`N_Vector`) is the initial value of  $y$ .

`yp0` (`N_Vector`) is the initial value of  $\dot{y}$ .

Return value The return value `flag` (of type `int`) will be one of the following:

`IDA_SUCCESS` The call to `IDAInit` was successful.

`IDA_MEM_NULL` The IDA memory block was not initialized through a previous call to `IDACreate`.

`IDA_MEM_FAIL` A memory allocation request has failed.

`IDA_ILL_INPUT` An input argument to `IDAInit` has an illegal value.

Notes If an error occurred, `IDAInit` also sends an error message to the error handler function.

#### IDAFree

Call `IDAFree(&ida_mem);`

Description The function `IDAFree` frees the pointer allocated by a previous call to `IDACreate`.

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

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

### 4.5.2 IDA tolerance specification functions

One of the following three functions must be called to specify the integration tolerances (or directly specify the weights used in evaluating WRMS vector norms). Note that this call must be made after the call to `IDAInit`.

**IDASStolerances**

**Call** `flag = IDASStolerances(ida_mem, reltol, abstol);`

**Description** The function `IDASStolerances` specifies scalar relative and absolute tolerances.

**Arguments** `ida_mem` (`void *`) pointer to the IDA memory block returned by `IDACreate`.  
`reltol` (`realtype`) is the scalar relative error tolerance.  
`abstol` (`realtype`) is the scalar absolute error tolerance.

**Return value** The return value `flag` (of type `int`) will be one of the following:

- `IDA_SUCCESS` The call to `IDASStolerances` was successful.
- `IDA_MEM_NULL` The IDA memory block was not initialized through a previous call to `IDACreate`.
- `IDA_NO_MALLOC` The allocation function `IDAINIT` has not been called.
- `IDA_ILL_INPUT` One of the input tolerances was negative.

**IDASVtolerances**

**Call** `flag = IDASVtolerances(ida_mem, reltol, abstol);`

**Description** The function `IDASVtolerances` specifies scalar relative tolerance and vector absolute tolerances.

**Arguments** `ida_mem` (`void *`) pointer to the IDA memory block returned by `IDACreate`.  
`reltol` (`realtype`) is the scalar relative error tolerance.  
`abstol` (`N_Vector`) is the vector of absolute error tolerances.

**Return value** The return value `flag` (of type `int`) will be one of the following:

- `IDA_SUCCESS` The call to `IDASVtolerances` was successful.
- `IDA_MEM_NULL` The IDA memory block was not initialized through a previous call to `IDACreate`.
- `IDA_NO_MALLOC` The allocation function `IDAINIT` has not been called.
- `IDA_ILL_INPUT` The relative error tolerance was negative or the absolute tolerance had a negative component.

**Notes** This choice of tolerances is important when the absolute error tolerance needs to be different for each component of the state vector  $y$ .

**IDAWFtolerances**

**Call** `flag = IDAWFtolerances(ida_mem, efun);`

**Description** The function `IDAWFtolerances` specifies a user-supplied function `efun` that sets the multiplicative error weights  $W_i$  for use in the weighted RMS norm, which are normally defined by Eq. (2.6).

**Arguments** `ida_mem` (`void *`) pointer to the IDA memory block returned by `IDACreate`.  
`efun` (`IDAwtFn`) is the C function which defines the `ewt` vector (see §4.6.3).

**Return value** The return value `flag` (of type `int`) will be one of the following:

- `IDA_SUCCESS` The call to `IDAWFtolerances` was successful.
- `IDA_MEM_NULL` The IDA memory block was not initialized through a previous call to `IDACreate`.
- `IDA_NO_MALLOC` The allocation function `IDAINIT` has not been called.

**General advice on choice of tolerances.** For many users, the appropriate choices for tolerance values in `reltol` and `abstol` are a concern. The following pieces of advice are relevant.

(1) The scalar relative tolerance `reltol` is to be set to control relative errors. So `reltol`= $10^{-4}$  means that errors are controlled to .01%. We do not recommend using `reltol` larger than  $10^{-3}$ .

On the other hand, `reltol` should not be so small that it is comparable to the unit roundoff of the machine arithmetic (generally around  $10^{-15}$ ).

(2) The absolute tolerances `abstol` (whether scalar or vector) need to be set to control absolute errors when any components of the solution vector `y` may be so small that pure relative error control is meaningless. For example, if `y[i]` starts at some nonzero value, but in time decays to zero, then pure relative error control on `y[i]` makes no sense (and is overly costly) after `y[i]` is below some noise level. Then `abstol` (if scalar) or `abstol[i]` (if a vector) needs to be set to that noise level. If the different components have different noise levels, then `abstol` should be a vector. See the example `idaRoberts_dns` in the IDA package, and the discussion of it in the IDA Examples document [25]. In that problem, the three components vary between 0 and 1, and have different noise levels; hence the `abstol` vector. It is impossible to give any general advice on `abstol` values, because the appropriate noise levels are completely problem-dependent. The user or modeler hopefully has some idea as to what those noise levels are.

(3) Finally, it is important to pick all the tolerance values conservatively, because they control the error committed on each individual time step. The final (global) errors are a sort of accumulation of those per-step errors. A good rule of thumb is to reduce the tolerances by a factor of .01 from the actual desired limits on errors. So if you want .01% accuracy (globally), a good choice is `reltol`=  $10^{-6}$ . But in any case, it is a good idea to do a few experiments with the tolerances to see how the computed solution values vary as tolerances are reduced.

**Advice on controlling unphysical negative values.** In many applications, some components in the true solution are always positive or non-negative, though at times very small. In the numerical solution, however, small negative (hence unphysical) values can then occur. In most cases, these values are harmless, and simply need to be controlled, not eliminated. The following pieces of advice are relevant.

(1) The way to control the size of unwanted negative computed values is with tighter absolute tolerances. Again this requires some knowledge of the noise level of these components, which may or may not be different for different components. Some experimentation may be needed.

(2) If output plots or tables are being generated, and it is important to avoid having negative numbers appear there (for the sake of avoiding a long explanation of them, if nothing else), then eliminate them, but only in the context of the output medium. Then the internal values carried by the solver are unaffected. Remember that a small negative value in `yret` returned by IDA, with magnitude comparable to `abstol` or less, is equivalent to zero as far as the computation is concerned.

(3) The user's residual routine `res` should never change a negative value in the solution vector `yy` to a non-negative value, as a "solution" to this problem. This can cause instability. If the `res` routine cannot tolerate a zero or negative value (e.g. because there is a square root or log of it), then the offending value should be changed to zero or a tiny positive number in a temporary variable (not in the input `yy` vector) for the purposes of computing  $F(t, y, \dot{y})$ .

(4) IDA provides the option of enforcing positivity or non-negativity on components. Also, such constraints can be enforced by use of the recoverable error return feature in the user-supplied residual function. However, because these options involve some extra overhead cost, they should only be exercised if the use of absolute tolerances to control the computed values is unsuccessful.

### 4.5.3 Linear solver interface functions

As previously explained, if the nonlinear solver requires the solution of linear systems of the form (2.4) (e.g., the default Newton iteration), then solution of these linear systems is handled with the IDALS linear solver interface. This interface supports all valid SUNLINSOL modules. Here, matrix-based SUNLINSOL modules utilize SUNMATRIX objects to store the Jacobian matrix  $J = \partial F / \partial y + \alpha \partial F / \partial \dot{y}$  and factorizations used throughout the solution process. Conversely, matrix-free SUNLINSOL modules instead use iterative methods to solve the linear systems of equations, and only require the *action* of the Jacobian on a vector,  $Jv$ .

With most iterative linear solvers, preconditioning can be done on the left only, on the right only, on both the left and the right, or not at all. The exceptions to this rule are SPFGMR that supports

right preconditioning only and PCG that performs symmetric preconditioning. For the specification of a preconditioner, see the iterative linear solver sections in §4.5.8 and §4.6. A preconditioner matrix  $P$  must approximate the Jacobian  $J$ , at least crudely.

To specify a generic linear solver to IDA, after the call to `IDACreate` but before any calls to `IDASolve`, the user's program must create the appropriate `SUNLINSOL` object and call the function `IDASetLinearSolver`, as documented below. To create the `SUNLinearSolver` object, the user may call one of the SUNDIALS-packaged `SUNLINSOL` module constructor routines via a call of the form

```
SUNLinearSolver LS = SUNLinSol_*(...);
```

The current list of such constructor routines includes `SUNLinSol_Dense`, `SUNLinSol_Band`, `SUNLinSol_LapackDense`, `SUNLinSol_LapackBand`, `SUNLinSol_KLU`, `SUNLinSol_SuperLUMT`, `SUNLinSol_SPGMR`, `SUNLinSol_SPFQMR`, `SUNLinSol_SPBCGS`, `SUNLinSol_SPTFQMR`, and `SUNLinSol_PCG`.

Alternately, a user-supplied `SUNLinearSolver` module may be created and used instead. The use of each of the generic linear solvers involves certain constants, functions and possibly some macros, that are likely to be needed in the user code. These are available in the corresponding header file associated with the specific `SUNMATRIX` or `SUNLINSOL` module in question, as described in Chapters 7 and 8.

Once this solver object has been constructed, the user should attach it to IDA via a call to `IDASetLinearSolver`. The first argument passed to this function is the IDA memory pointer returned by `IDACreate`; the second argument is the desired `SUNLINSOL` object to use for solving systems. The third argument is an optional `SUNMATRIX` object to accompany matrix-based `SUNLINSOL` inputs (for matrix-free linear solvers, the third argument should be `NULL`). A call to this function initializes the IDALS linear solver interface, linking it to the main IDA integrator, and allows the user to specify additional parameters and routines pertinent to their choice of linear solver.

#### `IDASetLinearSolver`

Call	<code>flag = IDASetLinearSolver(ida_mem, LS, J);</code>
Description	The function <code>IDASetLinearSolver</code> attaches a generic <code>SUNLINSOL</code> object <code>LS</code> and corresponding template Jacobian <code>SUNMATRIX</code> object <code>J</code> (if applicable) to IDA, initializing the IDALS linear solver interface.
Arguments	<p><code>ida_mem</code> (void *) pointer to the IDA memory block.</p> <p><code>LS</code> (<code>SUNLinearSolver</code>) <code>SUNLINSOL</code> object to use for solving linear systems of the form (2.4).</p> <p><code>J</code> (<code>SUNMatrix</code>) <code>SUNMATRIX</code> object for used as a template for the Jacobian (or <code>NULL</code> if not applicable).</p>
Return value	<p>The return value <code>flag</code> (of type <code>int</code>) is one of</p> <p><code>IDALS_SUCCESS</code> The IDALS initialization was successful.</p> <p><code>IDALS_MEM_NULL</code> The <code>ida_mem</code> pointer is <code>NULL</code>.</p> <p><code>IDALS_ILL_INPUT</code> The IDALS interface is not compatible with the <code>LS</code> or <code>J</code> input objects or is incompatible with the current <code>NVECTOR</code> module.</p> <p><code>IDALS_SUNLS_FAIL</code> A call to the <code>LS</code> object failed.</p> <p><code>IDALS_MEM_FAIL</code> A memory allocation request failed.</p>
Notes	<p>If <code>LS</code> is a matrix-based linear solver, then the template Jacobian matrix <code>J</code> will be used in the solve process, so if additional storage is required within the <code>SUNMATRIX</code> object (e.g. for factorization of a banded matrix), ensure that the input object is allocated with sufficient size (see the documentation of the particular <code>SUNMATRIX</code> type in Chapter 7 for further information).</p> <p>The previous routines <code>IDADlsSetLinearSolver</code> and <code>IDASpilsSetLinearSolver</code> are now wrappers for this routine, and may still be used for backward-compatibility. However, these will be deprecated in future releases, so we recommend that users transition to the new routine name soon.</p>

#### 4.5.4 Nonlinear solver interface function

By default IDA uses the SUNNONLINSOL implementation of Newton's method defined by the SUNNONLINSOL\_NEWTON module (see §9.2). To specify a different nonlinear solver in IDA, the user's program must create a SUNNONLINSOL object by calling the appropriate constructor routine. The user must then attach the SUNNONLINSOL object to IDA by calling `IDASetNonlinearSolver`, as documented below.

When changing the nonlinear solver in IDA, `IDASetNonlinearSolver` must be called after `IDAInit`. If any calls to `IDASolve` have been made, then IDA will need to be reinitialized by calling `IDAReInit` to ensure that the nonlinear solver is initialized correctly before any subsequent calls to `IDASolve`.

The first argument passed to the routine `IDASetNonlinearSolver` is the IDA memory pointer returned by `IDACreate` and the second argument is the SUNNONLINSOL object to use for solving the nonlinear system 2.3. A call to this function attaches the nonlinear solver to the main IDA integrator. We note that at present, the SUNNONLINSOL object *must be of type* `SUNNONLINEARSOLVER_ROOTFIND`.

##### `IDASetNonlinearSolver`

**Call** `flag = IDASetNonlinearSolver(ida_mem, NLS);`

**Description** The function `IDASetNonLinearSolver` attaches a SUNNONLINSOL object (NLS) to IDA.

**Arguments** `ida_mem` (void \*) pointer to the IDA memory block.  
**NLS** (`SUNNonlinearSolver`) SUNNONLINSOL object to use for solving nonlinear systems.

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

- `IDA_SUCCESS` The nonlinear solver was successfully attached.
- `IDA_MEM_NULL` The `ida_mem` pointer is NULL.
- `IDA_ILL_INPUT` The SUNNONLINSOL object is NULL, does not implement the required nonlinear solver operations, is not of the correct type, or the residual function, convergence test function, or maximum number of nonlinear iterations could not be set.

#### 4.5.5 Initial condition calculation function

`IDACalcIC` calculates corrected initial conditions for the DAE system for certain index-one problems including a class of systems of semi-implicit form. (See §2.1 and Ref. [8].) It uses Newton iteration combined with a linesearch algorithm. Calling `IDACalcIC` is optional. It is only necessary when the initial conditions do not satisfy the given system. Thus if `y0` and `yp0` are known to satisfy  $F(t_0, y_0, \dot{y}_0) = 0$ , then a call to `IDACalcIC` is generally *not* necessary.

A call to the function `IDACalcIC` must be preceded by successful calls to `IDACreate` and `IDAInit` (or `IDARInit`), and by a successful call to the linear system solver specification function. The call to `IDACalcIC` should precede the call(s) to `IDASolve` for the given problem.

##### `IDACalcIC`

**Call** `flag = IDACalcIC(ida_mem, icopt, tout1);`

**Description** The function `IDACalcIC` corrects the initial values `y0` and `yp0` at time `t0`.

**Arguments** `ida_mem` (void \*) pointer to the IDA memory block.  
**icopt** (`int`) is one of the following two options for the initial condition calculation.  
`icopt=IDA_YA_YDP_INIT` directs `IDACalcIC` to compute the algebraic components of  $y$  and differential components of  $\dot{y}$ , given the differential components of  $y$ . This option requires that the `N_Vector id` was set through `IDASetId`, specifying the differential and algebraic components.  
`icopt=IDA_Y_INIT` directs `IDACalcIC` to compute all components of  $y$ , given  $\dot{y}$ . In this case, `id` is not required.

tout1	(realtype) is the first value of $t$ at which a solution will be requested (from <code>IDASolve</code> ). This value is needed here only to determine the direction of integration and rough scale in the independent variable $t$ .
Return value	The return value <code>flag</code> (of type <code>int</code> ) will be one of the following:
<code>IDA_SUCCESS</code>	<code>IDASolve</code> succeeded.
<code>IDA_MEM_NULL</code>	The argument <code>ida_mem</code> was <code>NULL</code> .
<code>IDA_NO_MALLOC</code>	The allocation function <code>IDAInit</code> has not been called.
<code>IDA_ILL_INPUT</code>	One of the input arguments was illegal.
<code>IDA_LSETUP_FAIL</code>	The linear solver's setup function failed in an unrecoverable manner.
<code>IDA_LINIT_FAIL</code>	The linear solver's initialization function failed.
<code>IDA_LSOLVE_FAIL</code>	The linear solver's solve function failed in an unrecoverable manner.
<code>IDA_BAD_EWT</code>	Some component of the error weight vector is zero (illegal), either for the input value of <code>y0</code> or a corrected value.
<code>IDA_FIRST_RES_FAIL</code>	The user's residual function returned a recoverable error flag on the first call, but <code>IDACalcIC</code> was unable to recover.
<code>IDA_RES_FAIL</code>	The user's residual function returned a nonrecoverable error flag.
<code>IDA_NO_RECOVERY</code>	The user's residual function, or the linear solver's setup or solve function had a recoverable error, but <code>IDACalcIC</code> was unable to recover.
<code>IDA_CONSTR_FAIL</code>	<code>IDACalcIC</code> was unable to find a solution satisfying the inequality constraints.
<code>IDA_LINESEARCH_FAIL</code>	The linesearch algorithm failed to find a solution with a step larger than <code>steptol</code> in weighted RMS norm, and within the allowed number of backtracks.
<code>IDA_CONV_FAIL</code>	<code>IDACalcIC</code> failed to get convergence of the Newton iterations.
Notes	<p>All failure return values are negative and therefore a test <code>flag &lt; 0</code> will trap all <code>IDACalcIC</code> failures.</p> <p>Note that <code>IDACalcIC</code> will correct the values of <math>y(t_0)</math> and <math>\dot{y}(t_0)</math> which were specified in the previous call to <code>IDAInit</code> or <code>IDARInit</code>. To obtain the corrected values, call <code>IDAGetconsistentIC</code> (see §4.5.10.3).</p>

#### 4.5.6 Rootfinding initialization function

While integrating the IVP, IDA has the capability of finding the roots of a set of user-defined functions. To activate the rootfinding algorithm, call the following function. This is normally called only once, prior to the first call to `IDASolve`, but if the rootfinding problem is to be changed during the solution, `IDARootInit` can also be called prior to a continuation call to `IDASolve`.

##### `IDARootInit`

Call	<code>flag = IDARootInit(ida_mem, nrtfn, g);</code>
Description	The function <code>IDARootInit</code> specifies that the roots of a set of functions $g_i(t, y, \dot{y})$ are to be found while the IVP is being solved.
Arguments	<p><code>ida_mem</code> (<code>void *</code>) pointer to the IDA memory block returned by <code>IDACreate</code>.</p> <p><code>nrtfn</code> (<code>int</code>) is the number of root functions <math>g_i</math>.</p> <p><code>g</code> (<code>IDARootFn</code>) is the C function which defines the <code>nrtfn</code> functions <math>g_i(t, y, \dot{y})</math> whose roots are sought. See §4.6.4 for details.</p>
Return value	The return value <code>flag</code> (of type <code>int</code> ) is one of



	<b>IDA_SUCCESS</b>	The call to <code>IDARootInit</code> was successful.
	<b>IDA_MEM_NULL</b>	The <code>ida_mem</code> argument was NULL.
	<b>IDA_MEM_FAIL</b>	A memory allocation failed.
	<b>IDA_ILL_INPUT</b>	The function <code>g</code> is NULL, but <code>nrtfn</code> > 0.
Notes		If a new IVP is to be solved with a call to <code>IDAReInit</code> , where the new IVP has no rootfinding problem but the prior one did, then call <code>IDARootInit</code> with <code>nrtfn</code> = 0.

#### 4.5.7 IDA solver function

This is the central step in the solution process, the call to perform the integration of the DAE. One of the input arguments (`itask`) specifies one of two modes as to where IDA is to return a solution. But these modes are modified if the user has set a stop time (with `IDASetStopTime`) or requested rootfinding.

##### **IDASolve**

Call	<code>flag = IDASolve(ida_mem, tout, &amp;tret, yret, ypret, itask);</code>
Description	The function <code>IDASolve</code> integrates the DAE over an interval in $t$ .
Arguments	<p><code>ida_mem</code> (void *) pointer to the IDA memory block.</p> <p><code>tout</code> (realtype) the next time at which a computed solution is desired.</p> <p><code>tret</code> (realtype) the time reached by the solver (output).</p> <p><code>yret</code> (N_Vector) the computed solution vector <math>y</math>.</p> <p><code>ypret</code> (N_Vector) the computed solution vector <math>\dot{y}</math>.</p> <p><code>itask</code> (int) a flag indicating the job of the solver for the next user step. The <code>IDA_NORMAL</code> task is to have the solver take internal steps until it has reached or just passed the user specified <code>tout</code> parameter. The solver then interpolates in order to return approximate values of <math>y(\text{tout})</math> and <math>\dot{y}(\text{tout})</math>. The <code>IDA_ONE_STEP</code> option tells the solver to just take one internal step and return the solution at the point reached by that step.</p>
Return value	<p><code>IDASolve</code> returns vectors <code>yret</code> and <code>ypret</code> and a corresponding independent variable value <math>t = \text{tret}</math>, such that (<code>yret</code>, <code>ypret</code>) are the computed values of (<math>y(t)</math>, <math>\dot{y}(t)</math>).</p> <p>In <code>IDA_NORMAL</code> mode with no errors, <code>tret</code> will be equal to <code>tout</code> and <code>yret</code> = <math>y(\text{tout})</math>, <code>ypret</code> = <math>\dot{y}(\text{tout})</math>.</p> <p>The return value <code>flag</code> (of type <code>int</code>) will be one of the following:</p> <p><b>IDA_SUCCESS</b> <code>IDASolve</code> succeeded.</p> <p><b>IDA_TSTOP_RETURN</b> <code>IDASolve</code> succeeded by reaching the stop point specified through the optional input function <code>IDASetStopTime</code>.</p> <p><b>IDA_ROOT_RETURN</b> <code>IDASolve</code> succeeded and found one or more roots. In this case, <code>tret</code> is the location of the root. If <code>nrtfn</code> &gt; 1, call <code>IDAGetRootInfo</code> to see which <math>g_i</math> were found to have a root. See §4.5.10.4 for more information.</p> <p><b>IDA_MEM_NULL</b> The <code>ida_mem</code> argument was NULL.</p> <p><b>IDA_ILL_INPUT</b> One of the inputs to <code>IDASolve</code> was illegal, or some other input to the solver was either illegal or missing. The latter category includes the following situations: (a) The tolerances have not been set. (b) A component of the error weight vector became zero during internal time-stepping. (c) The linear solver initialization function (called by the user after calling <code>IDACreate</code>) failed to set the linear solver-specific <code>lsolve</code> field in <code>ida_mem</code>. (d) A root of one of the root functions was found both at a point <math>t</math> and also very near <math>t</math>. In any case, the user should see the printed error message for details.</p>



	IDA_TOO_MUCH_WORK	The solver took <code>mxstep</code> internal steps but could not reach <code>tout</code> . The default value for <code>mxstep</code> is <code>MXSTEP_DEFAULT = 500</code> .
	IDA_TOO_MUCH_ACC	The solver could not satisfy the accuracy demanded by the user for some internal step.
	IDA_ERR_FAIL	Error test failures occurred too many times ( <code>MXNEF = 10</code> ) during one internal time step or occurred with $ h  = h_{min}$ .
	IDA_CONV_FAIL	Convergence test failures occurred too many times ( <code>MXNCF = 10</code> ) during one internal time step or occurred with $ h  = h_{min}$ .
	IDA_LINIT_FAIL	The linear solver's initialization function failed.
	IDA_LSETUP_FAIL	The linear solver's setup function failed in an unrecoverable manner.
	IDA_LSOLVE_FAIL	The linear solver's solve function failed in an unrecoverable manner.
	IDA_CONSTR_FAIL	The inequality constraints were violated and the solver was unable to recover.
	IDA_REP_RES_ERR	The user's residual function repeatedly returned a recoverable error flag, but the solver was unable to recover.
	IDA_RES_FAIL	The user's residual function returned a nonrecoverable error flag.
	IDA_RTFUNC_FAIL	The rootfinding function failed.
Notes		The vector <code>yret</code> can occupy the same space as the vector <code>y0</code> of initial conditions that was passed to <code>IDAInit</code> , and the vector <code>ypret</code> can occupy the same space as <code>yp0</code> .
		In the <code>IDA.ONE_STEP</code> mode, <code>tout</code> is used on the first call only, and only to get the direction and rough scale of the independent variable.
		All failure return values are negative and therefore a test <code>flag &lt; 0</code> will trap all <code>IDASolve</code> failures.
		On any error return in which one or more internal steps were taken by <code>IDASolve</code> , the returned values of <code>tret</code> , <code>yret</code> , and <code>ypret</code> correspond to the farthest point reached in the integration. On all other error returns, these values are left unchanged from the previous <code>IDASolve</code> return.

### 4.5.8 Optional input functions

There are numerous optional input parameters that control the behavior of the IDA solver. IDA provides functions that can be used to change these optional input parameters from their default values. Table 4.2 lists all optional input functions in IDA which are then described in detail in the remainder of this section. For the most casual use of IDA, the reader can skip to §4.6.

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

#### 4.5.8.1 Main solver optional input functions

The calls listed here can be executed in any order. However, if the user's program calls either `IDASetErrFile` or `IDASetErrHandlerFn`, then that call should appear first, in order to take effect for any later error message.

##### `IDASetErrFile`

Call `flag = IDASetErrFile(ida_mem, errfp);`

Description The function `IDASetErrFile` specifies the pointer to the file where all IDA messages should be directed when the default IDA error handler function is used.

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.

Table 4.2: Optional inputs for IDA and IDALS

Optional input	Function name	Default
<b>IDA main solver</b>		
Pointer to an error file	IDASetErrFile	stderr
Error handler function	IDASetErrHandlerFn	internal fn.
User data	IDASetUserData	NULL
Maximum order for BDF method	IDASetMaxOrd	5
Maximum no. of internal steps before $t_{\text{out}}$	IDASetMaxNumSteps	500
Initial step size	IDASetInitStep	estimated
Maximum absolute step size	IDASetMaxStep	$\infty$
Value of $t_{\text{stop}}$	IDASetStopTime	$\infty$
Maximum no. of error test failures	IDASetMaxErrTestFails	10
Maximum no. of nonlinear iterations	IDASetMaxNonlinIters	4
Maximum no. of convergence failures	IDASetMaxConvFails	10
Maximum no. of error test failures	IDASetMaxErrTestFails	7
Coeff. in the nonlinear convergence test	IDASetNonlinConvCoef	0.33
Suppress alg. vars. from error test	IDASetSuppressAlg	SUNFALSE
Variable types (differential/algebraic)	IDASetId	NULL
Inequality constraints on solution	IDASetConstraints	NULL
Direction of zero-crossing	IDASetRootDirection	both
Disable rootfinding warnings	IDASetNoInactiveRootWarn	none
<b>IDA initial conditions calculation</b>		
Coeff. in the nonlinear convergence test	IDASetNonlinConvCoefIC	0.0033
Maximum no. of steps	IDASetMaxNumStepsIC	5
Maximum no. of Jacobian/precond. evals.	IDASetMaxNumJacsIC	4
Maximum no. of Newton iterations	IDASetMaxNumItersIC	10
Max. linesearch backtracks per Newton iter.	IDASetMaxBacksIC	100
Turn off linesearch	IDASetLineSearchOffIC	SUNFALSE
Lower bound on Newton step	IDASetStepToleranceIC	around <sup>2/3</sup>
<b>IDALS linear solver interface</b>		
Jacobian function	IDASetJacFn	DQ
Jacobian-times-vector function	IDASetJacTimes	NULL, DQ
Preconditioner functions	IDASetPreconditioner	NULL, NULL
Ratio between linear and nonlinear tolerances	IDASetEpsLin	0.05
Increment factor used in DQ $Jv$ approx.	IDASetIncrementFactor	1.0

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

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

**IDA\_SUCCESS** The optional value has been successfully set.

**IDA\_MEM\_NULL** The **ida\_mem** pointer is NULL.

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

Passing a value NULL disables all future error message output (except for the case in which the IDA memory pointer is NULL). This use of **IDASetErrFile** is strongly discouraged.

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



#### **IDASetErrHandlerFn**

Call **flag** = **IDASetErrHandlerFn**(**ida\_mem**, **ehfun**, **eh\_data**);

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

Arguments **ida\_mem** (void \*) pointer to the IDA memory block.

**ehfun** (**IDAErrorHandlerFn**) is the user's C error handler function (see §4.6.2).

**eh\_data** (void \*) pointer to user data passed to **ehfun** every time it is called.

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

**IDA\_SUCCESS** The function **ehfun** and data pointer **eh\_data** have been successfully set.

**IDA\_MEM\_NULL** The **ida\_mem** pointer is NULL.

Notes Error messages indicating that the IDA solver memory is NULL will always be directed to **stderr**.

#### **IDASetUserData**

Call **flag** = **IDASetUserData**(**ida\_mem**, **user\_data**);

Description The function **IDASetUserData** specifies the user data block **user\_data** and attaches it to the main IDA memory block.

Arguments **ida\_mem** (void \*) pointer to the IDA memory block.

**user\_data** (void \*) pointer to the user data.

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

**IDA\_SUCCESS** The optional value has been successfully set.

**IDA\_MEM\_NULL** The **ida\_mem** pointer is NULL.

Notes If specified, the pointer to **user\_data** is passed to all user-supplied functions that have it as an argument. Otherwise, a NULL pointer is passed.

If **user\_data** is needed in user linear solver or preconditioner functions, the call to **IDASetUserData** must be made *before* the call to specify the linear solver.



#### **IDASetMaxOrd**

Call **flag** = **IDASetMaxOrd**(**ida\_mem**, **maxord**);

Description The function **IDASetMaxOrd** specifies the maximum order of the linear multistep method.

Arguments **ida\_mem** (void \*) pointer to the IDA memory block.

**maxord** (int) value of the maximum method order. This must be positive.

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

	IDA_SUCCESS	The optional value has been successfully set.
	IDA_MEM_NULL	The <code>ida_mem</code> pointer is NULL.
	IDA_ILL_INPUT	The input value <code>maxord</code> is $\leq 0$ , or larger than its previous value.
Notes		The default value is 5. If the input value exceeds 5, the value 5 will be used. Since <code>maxord</code> affects the memory requirements for the internal IDA memory block, its value cannot be increased past its previous value.

#### IDASetMaxNumSteps

Call	<code>flag = IDASetMaxNumSteps(ida_mem, mxsteps);</code>
Description	The function <code>IDASetMaxNumSteps</code> specifies the maximum number of steps to be taken by the solver in its attempt to reach the next output time.
Arguments	<code>ida_mem</code> (void *) pointer to the IDA memory block. <code>mxsteps</code> (long int) maximum allowed number of steps.
Return value	The return value <code>flag</code> (of type <code>int</code> ) is one of  IDA_SUCCESS The optional value has been successfully set. IDA_MEM_NULL The <code>ida_mem</code> pointer is NULL.
Notes	Passing <code>mxsteps = 0</code> results in IDA using the default value (500). Passing <code>mxsteps &lt; 0</code> disables the test ( <i>not recommended</i> ).

#### IDASetInitStep

Call	<code>flag = IDASetInitStep(ida_mem, hin);</code>
Description	The function <code>IDASetInitStep</code> specifies the initial step size.
Arguments	<code>ida_mem</code> (void *) pointer to the IDA memory block. <code>hin</code> (realtype) value of the initial step size to be attempted. Pass 0.0 to have IDA use the default value.
Return value	The return value <code>flag</code> (of type <code>int</code> ) is one of  IDA_SUCCESS The optional value has been successfully set. IDA_MEM_NULL The <code>ida_mem</code> pointer is NULL.
Notes	By default, IDA estimates the initial step as the solution of $\ h\dot{y}\ _{\text{WRMS}} = 1/2$ , with an added restriction that $ h  \leq .001 \text{tout} - \text{t0} $ .

#### IDASetMaxStep

Call	<code>flag = IDASetMaxStep(ida_mem, hmax);</code>
Description	The function <code>IDASetMaxStep</code> specifies the maximum absolute value of the step size.
Arguments	<code>ida_mem</code> (void *) pointer to the IDA memory block. <code>hmax</code> (realtype) maximum absolute value of the step size.
Return value	The return value <code>flag</code> (of type <code>int</code> ) is one of  IDA_SUCCESS The optional value has been successfully set. IDA_MEM_NULL The <code>ida_mem</code> pointer is NULL. IDA_ILL_INPUT Either <code>hmax</code> is not positive or it is smaller than the minimum allowable step.
Notes	Pass <code>hmax= 0</code> to obtain the default value $\infty$ .

**IDASetStopTime**

Call	<code>flag = IDASetStopTime(ida_mem, tstop);</code>
Description	The function <code>IDASetStopTime</code> specifies the value of the independent variable $t$ past which the solution is not to proceed.
Arguments	<p><code>ida_mem</code> (<code>void *</code>) pointer to the IDA memory block.</p> <p><code>tstop</code> (<code>realtype</code>) value of the independent variable past which the solution should not proceed.</p>
Return value	<p>The return value <code>flag</code> (of type <code>int</code>) is one of</p> <p><code>IDA_SUCCESS</code> The optional value has been successfully set.</p> <p><code>IDA_MEM_NULL</code> The <code>ida_mem</code> pointer is NULL.</p> <p><code>IDA_ILL_INPUT</code> The value of <code>tstop</code> is not beyond the current <math>t</math> value, <math>t_n</math>.</p>
Notes	The default, if this routine is not called, is that no stop time is imposed.

**IDASetMaxErrTestFails**

Call	<code>flag = IDASetMaxErrTestFails(ida_mem, maxnef);</code>
Description	The function <code>IDASetMaxErrTestFails</code> specifies the maximum number of error test failures in attempting one step.
Arguments	<p><code>ida_mem</code> (<code>void *</code>) pointer to the IDA memory block.</p> <p><code>maxnef</code> (<code>int</code>) maximum number of error test failures allowed on one step (<math>&gt; 0</math>).</p>
Return value	<p>The return value <code>flag</code> (of type <code>int</code>) is one of</p> <p><code>IDA_SUCCESS</code> The optional value has been successfully set.</p> <p><code>IDA_MEM_NULL</code> The <code>ida_mem</code> pointer is NULL.</p>
Notes	The default value is 7.

**IDASetMaxNonlinIters**

Call	<code>flag = IDASetMaxNonlinIters(ida_mem, maxcor);</code>
Description	The function <code>IDASetMaxNonlinIters</code> specifies the maximum number of nonlinear solver iterations at one step.
Arguments	<p><code>ida_mem</code> (<code>void *</code>) pointer to the IDA memory block.</p> <p><code>maxcor</code> (<code>int</code>) maximum number of nonlinear solver iterations allowed on one step (<math>&gt; 0</math>).</p>
Return value	<p>The return value <code>flag</code> (of type <code>int</code>) is one of</p> <p><code>IDA_SUCCESS</code> The optional value has been successfully set.</p> <p><code>IDA_MEM_NULL</code> The <code>ida_mem</code> pointer is NULL.</p> <p><code>IDA_MEM_FAIL</code> The <code>SUNNONLINSOL</code> module is NULL.</p>
Notes	The default value is 3.

**IDASetMaxConvFails**

Call	<code>flag = IDASetMaxConvFails(ida_mem, maxncf);</code>
Description	The function <code>IDASetMaxConvFails</code> specifies the maximum number of nonlinear solver convergence failures at one step.
Arguments	<p><code>ida_mem</code> (<code>void *</code>) pointer to the IDA memory block.</p> <p><code>maxncf</code> (<code>int</code>) maximum number of allowable nonlinear solver convergence failures on one step (<math>&gt; 0</math>).</p>

Return value The return value `flag` (of type `int`) is one of  
               `IDA_SUCCESS` The optional value has been successfully set.  
               `IDA_MEM_NULL` The `ida_mem` pointer is NULL.  
 Notes The default value is 10.

#### IDASetNonlinConvCoef

Call `flag = IDASetNonlinConvCoef(ida_mem, nlscoef);`  
 Description The function `IDASetNonlinConvCoef` specifies the safety factor in the nonlinear convergence test; see Chapter 2, Eq. (2.7).  
 Arguments `ida_mem` (`void *`) pointer to the IDA memory block.  
           `nlscoef` (`realtype`) coefficient in nonlinear convergence test ( $> 0.0$ ).  
 Return value The return value `flag` (of type `int`) is one of  
               `IDA_SUCCESS` The optional value has been successfully set.  
               `IDA_MEM_NULL` The `ida_mem` pointer is NULL.  
               `IDA_ILL_INPUT` The value of `nlscoef` is  $\leq 0.0$ .  
 Notes The default value is 0.33.

#### IDASetSuppressAlg

Call `flag = IDASetSuppressAlg(ida_mem, suppressalg);`  
 Description The function `IDASetSuppressAlg` indicates whether or not to suppress algebraic variables in the local error test.  
 Arguments `ida_mem` (`void *`) pointer to the IDA memory block.  
           `suppressalg` (`boolean_t`) indicates whether to suppress (`SUNTRUE`) or not (`SUNFALSE`) the algebraic variables in the local error test.  
 Return value The return value `flag` (of type `int`) is one of  
               `IDA_SUCCESS` The optional value has been successfully set.  
               `IDA_MEM_NULL` The `ida_mem` pointer is NULL.  
 Notes The default value is `SUNFALSE`.  
       If `suppressalg=SUNTRUE` is selected, then the `id` vector must be set (through `IDASetId`) to specify the algebraic components.  
       In general, the use of this option (with `suppressalg = SUNTRUE`) is *discouraged* when solving DAE systems of index 1, whereas it is generally *encouraged* for systems of index 2 or more. See pp. 146-147 of Ref. [4] for more on this issue.

#### IDASetId

Call `flag = IDASetId(ida_mem, id);`  
 Description The function `IDASetId` specifies algebraic/differential components in the  $y$  vector.  
 Arguments `ida_mem` (`void *`) pointer to the IDA memory block.  
           `id` (`N_Vector`) state vector. A value of 1.0 indicates a differential variable, while 0.0 indicates an algebraic variable.  
 Return value The return value `flag` (of type `int`) is one of  
               `IDA_SUCCESS` The optional value has been successfully set.  
               `IDA_MEM_NULL` The `ida_mem` pointer is NULL.  
 Notes The vector `id` is required if the algebraic variables are to be suppressed from the local error test (see `IDASetSuppressAlg`) or if `IDACalcIC` is to be called with `icopt = IDA_YA_YDP_INIT` (see §4.5.5).

**IDASetConstraints**

Call	<code>flag = IDASetConstraints(ida_mem, constraints);</code>
Description	The function <code>IDASetConstraints</code> specifies a vector defining inequality constraints for each component of the solution vector $y$ .
Arguments	<p><code>ida_mem</code> (<code>void *</code>) pointer to the IDA memory block.</p> <p><code>constraints</code> (<code>N_Vector</code>) vector of constraint flags. If <code>constraints[i]</code> is</p> <ul style="list-style-type: none"> <li>0.0 then no constraint is imposed on <math>y_i</math>.</li> <li>1.0 then <math>y_i</math> will be constrained to be <math>y_i \geq 0.0</math>.</li> <li>-1.0 then <math>y_i</math> will be constrained to be <math>y_i \leq 0.0</math>.</li> <li>2.0 then <math>y_i</math> will be constrained to be <math>y_i &gt; 0.0</math>.</li> <li>-2.0 then <math>y_i</math> will be constrained to be <math>y_i &lt; 0.0</math>.</li> </ul>
Return value	<p>The return value <code>flag</code> (of type <code>int</code>) is one of</p> <p><code>IDA_SUCCESS</code> The optional value has been successfully set.</p> <p><code>IDA_MEM_NULL</code> The <code>ida_mem</code> pointer is NULL.</p> <p><code>IDA_ILL_INPUT</code> The constraints vector contains illegal values.</p>
Notes	The presence of a non-NULL constraints vector that is not 0.0 in all components will cause constraint checking to be performed. However, a call with 0.0 in all components of <code>constraints</code> will result in an illegal input return.

**4.5.8.2 Linear solver interface optional input functions**

The mathematical explanation of the linear solver methods available to IDA is provided in §2.1. We group the user-callable routines into four categories: general routines concerning the overall IDALS linear solver interface, optional inputs for matrix-based linear solvers, optional inputs for matrix-free linear solvers, and optional inputs for iterative linear solvers. We note that the matrix-based and matrix-free groups are mutually exclusive, whereas the “iterative” tag can apply to either case.

When using matrix-based linear solver modules, the IDALS solver interface needs a function to compute an approximation to the Jacobian matrix  $J(t, y, \dot{y})$ . This function must be of type `IDALsJacFn`. The user can supply a Jacobian function, or if using a dense or banded matrix  $J$  can use the default internal difference quotient approximation that comes with the IDALS solver. To specify a user-supplied Jacobian function `jac`, IDALS provides the function `IDASetJacFn`. The IDALS interface passes the pointer `user_data` to the Jacobian function. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied Jacobian function, without using global data in the program. The pointer `user_data` may be specified through `IDASetUserData`.

**IDASetJacFn**

Call	<code>flag = IDASetJacFn(ida_mem, jac);</code>
Description	The function <code>IDASetJacFn</code> specifies the Jacobian approximation function to be used for a matrix-based solver within the IDALS interface.
Arguments	<p><code>ida_mem</code> (<code>void *</code>) pointer to the IDA memory block.</p> <p><code>jac</code> (<code>IDALsJacFn</code>) user-defined Jacobian approximation function.</p>
Return value	<p>The return value <code>flag</code> (of type <code>int</code>) is one of</p> <p><code>IDALS_SUCCESS</code> The optional value has been successfully set.</p> <p><code>IDALS_MEM_NULL</code> The <code>ida_mem</code> pointer is NULL.</p> <p><code>IDALS_LMEM_NULL</code> The IDALS linear solver interface has not been initialized.</p>

Notes	<p>This function must be called <i>after</i> the IDALS linear solver interface has been initialized through a call to <code>IDASetLinearSolver</code>.</p> <p>By default, IDALS uses an internal difference quotient function for dense and band matrices. If <code>NULL</code> is passed to <code>jac</code>, this default function is used. An error will occur if no <code>jac</code> is supplied when using other matrix types.</p> <p>The function type <code>IDALsJacFn</code> is described in §4.6.5.</p> <p>The previous routine <code>IDADlsSetJacFn</code> is now a wrapper for this routine, and may still be used for backward-compatibility. However, this will be deprecated in future releases, so we recommend that users transition to the new routine name soon.</p>
-------	--

When using matrix-free linear solver modules, the IDALS solver interface requires a function to compute an approximation to the product between the Jacobian matrix  $J(t, y)$  and a vector  $v$ . The user can supply a Jacobian-times-vector approximation function, or use the default internal difference quotient function that comes with the IDALS solver interface. A user-defined Jacobian-vector function must be of type `IDALsJacTimesVecFn` and can be specified through a call to `IDASetJacTimes` (see §4.6.6 for specification details). The evaluation and processing of any Jacobian-related data needed by the user's Jacobian-times-vector function may be done in the optional user-supplied function `jtsetup` (see §4.6.7 for specification details). The pointer `user_data` received through `IDASetUserData` (or a pointer to `NULL` if `user_data` was not specified) is passed to the Jacobian-times-vector setup and product functions, `jtsetup` and `jtimes`, each time they are called. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied functions without using global data in the program.

#### IDASetJacTimes

Call	<code>flag = IDASetJacTimes(ida_mem, jsetup, jtimes);</code>
Description	The function <code>IDASetJacTimes</code> specifies the Jacobian-vector setup and product functions.
Arguments	<p><code>ida_mem</code> (<code>void *</code>) pointer to the IDA memory block.</p> <p><code>jtsetup</code> (<code>IDALsJacTimesSetupFn</code>) user-defined function to set up the Jacobian-vector product. Pass <code>NULL</code> if no setup is necessary.</p> <p><code>jtimes</code> (<code>IDALsJacTimesVecFn</code>) user-defined Jacobian-vector product function.</p>
Return value	<p>The return value <code>flag</code> (of type <code>int</code>) is one of</p> <p><code>IDALS_SUCCESS</code> The optional value has been successfully set.</p> <p><code>IDALS_MEM_NULL</code> The <code>ida_mem</code> pointer is <code>NULL</code>.</p> <p><code>IDALS_LMEM_NULL</code> The IDALS linear solver has not been initialized.</p> <p><code>IDALS_SUNLS_FAIL</code> An error occurred when setting up the system matrix-times-vector routines in the <code>SUNLINSOL</code> object used by the IDALS interface.</p>
Notes	<p>The default is to use an internal finite difference quotient for <code>jtimes</code> and to omit <code>jtsetup</code>. If <code>NULL</code> is passed to <code>jtimes</code>, these defaults are used. A user may specify non-<code>NULL</code> <code>jtimes</code> and <code>NULL</code> <code>jtsetup</code> inputs.</p> <p>This function must be called <i>after</i> the IDALS linear solver interface has been initialized through a call to <code>IDASetLinearSolver</code>.</p> <p>The function type <code>IDALsJacTimesSetupFn</code> is described in §4.6.7.</p> <p>The function type <code>IDALsJacTimesVecFn</code> is described in §4.6.6.</p> <p>The previous routine <code>IDASpilsSetJacTimes</code> is now a wrapper for this routine, and may still be used for backward-compatibility. However, this will be deprecated in future releases, so we recommend that users transition to the new routine name soon.</p>

When using an iterative linear solver, the user may supply a preconditioning operator to aid in solution of the system. This operator consists of two user-supplied functions, `psetup` and `psolve`,



that are supplied to IDA using the function `IDASetPreconditioner`. The `psetup` function supplied to this routine should handle evaluation and preprocessing of any Jacobian data needed by the user's preconditioner solve function, `psolve`. Both of these functions are fully specified in §4.6. The user data pointer received through `IDASetUserData` (or a pointer to `NULL` if user data was not specified) is passed to the `psetup` and `psolve` functions. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied preconditioner functions without using global data in the program.

Also, as described in §2.1, the IDALS interface requires that iterative linear solvers stop when the norm of the preconditioned residual satisfies

$$\|r\| \leq \frac{\epsilon_L \epsilon}{10}$$

where  $\epsilon$  is the nonlinear solver tolerance, and the default  $\epsilon_L = 0.05$ ; this value may be modified by the user through the `IDASetEpsLin` function.

#### `IDASetPreconditioner`

Call	<code>flag = IDASetPreconditioner(ida_mem, psetup, psolve);</code>
Description	The function <code>IDASetPreconditioner</code> specifies the preconditioner setup and solve functions.
Arguments	<p><code>ida_mem</code> (<code>void *</code>) pointer to the IDA memory block.</p> <p><code>psetup</code> (<code>IDALsPrecSetupFn</code>) user-defined function to set up the preconditioner. Pass <code>NULL</code> if no setup is necessary.</p> <p><code>psolve</code> (<code>IDALsPrecSolveFn</code>) user-defined preconditioner solve function.</p>
Return value	<p>The return value <code>flag</code> (of type <code>int</code>) is one of</p> <p><code>IDALS_SUCCESS</code> The optional values have been successfully set.</p> <p><code>IDALS_MEM_NULL</code> The <code>ida_mem</code> pointer is <code>NULL</code>.</p> <p><code>IDALS_LMEM_NULL</code> The IDALS linear solver has not been initialized.</p> <p><code>IDALS_SUNLS_FAIL</code> An error occurred when setting up preconditioning in the <code>SUNLINSOL</code> object used by the IDALS interface.</p>
Notes	<p>The default is <code>NULL</code> for both arguments (i.e., no preconditioning).</p> <p>This function must be called <i>after</i> the IDALS linear solver interface has been initialized through a call to <code>IDASetLinearSolver</code>.</p> <p>The function type <code>IDALsPrecSolveFn</code> is described in §4.6.8.</p> <p>The function type <code>IDALsPrecSetupFn</code> is described in §4.6.9.</p> <p>The previous routine <code>IDASpilsSetPreconditioner</code> is now a wrapper for this routine, and may still be used for backward-compatibility. However, this will be deprecated in future releases, so we recommend that users transition to the new routine name soon.</p>

#### `IDASetEpsLin`

Call	<code>flag = IDASetEpsLin(ida_mem, eplifac);</code>
Description	The function <code>IDASetEpsLin</code> specifies the factor by which the Krylov linear solver's convergence test constant is reduced from the nonlinear iteration test constant.
Arguments	<p><code>ida_mem</code> (<code>void *</code>) pointer to the IDA memory block.</p> <p><code>eplifac</code> (<code>realtype</code>) linear convergence safety factor (<math>\geq 0.0</math>).</p>
Return value	<p>The return value <code>flag</code> (of type <code>int</code>) is one of</p> <p><code>IDALS_SUCCESS</code> The optional value has been successfully set.</p> <p><code>IDALS_MEM_NULL</code> The <code>ida_mem</code> pointer is <code>NULL</code>.</p>

	IDALS_LMEM_NULL The IDALS linear solver has not been initialized.
	IDALS_ILL_INPUT The factor <code>eplifac</code> is negative.
Notes	The default value is 0.05.
	This function must be called <i>after</i> the IDALS linear solver interface has been initialized through a call to <code>IDASetLinearSolver</code> .
	If <code>eplifac</code> = 0.0 is passed, the default value is used.
	The previous routine <code>IDASpilsSetEpsLin</code> is now a wrapper for this routine, and may still be used for backward-compatibility. However, this will be deprecated in future releases, so we recommend that users transition to the new routine name soon.

#### 4.5.8.3 Initial condition calculation optional input functions

The following functions can be called just prior to calling `IDACalcIC` to set optional inputs controlling the initial condition calculation.

##### `IDASetNonlinConvCoefIC`

Call	<code>flag = IDASetNonlinConvCoefIC(ida_mem, epiccon);</code>
Description	The function <code>IDASetNonlinConvCoefIC</code> specifies the positive constant in the Newton iteration convergence test within the initial condition calculation.
Arguments	<code>ida_mem</code> (void *) pointer to the IDA memory block. <code>epiccon</code> (realtype) coefficient in the Newton convergence test (> 0).
Return value	The return value <code>flag</code> (of type <code>int</code> ) is one of  <code>IDA_SUCCESS</code> The optional value has been successfully set. <code>IDA_MEM_NULL</code> The <code>ida_mem</code> pointer is NULL. <code>IDA_ILL_INPUT</code> The <code>epiccon</code> factor is $\leq 0.0$ .
Notes	The default value is $0.01 \cdot 0.33$ .  This test uses a weighted RMS norm (with weights defined by the tolerances). For new initial value vectors $y$ and $\dot{y}$ to be accepted, the norm of $J^{-1}F(t_0, y, \dot{y})$ must be $\leq$ <code>epiccon</code> , where $J$ is the system Jacobian.

##### `IDASetMaxNumStepsIC`

Call	<code>flag = IDASetMaxNumStepsIC(ida_mem, maxnh);</code>
Description	The function <code>IDASetMaxNumStepsIC</code> specifies the maximum number of steps allowed when <code>icopt</code> = <code>IDA_YA_YDP_INIT</code> in <code>IDACalcIC</code> , where $h$ appears in the system Jacobian, $J = \partial F / \partial y + (1/h) \partial F / \partial \dot{y}$ .
Arguments	<code>ida_mem</code> (void *) pointer to the IDA memory block. <code>maxnh</code> (int) maximum allowed number of values for $h$ .
Return value	The return value <code>flag</code> (of type <code>int</code> ) is one of  <code>IDA_SUCCESS</code> The optional value has been successfully set. <code>IDA_MEM_NULL</code> The <code>ida_mem</code> pointer is NULL. <code>IDA_ILL_INPUT</code> <code>maxnh</code> is non-positive.
Notes	The default value is 5.

**IDASetMaxNumJacsIC**

**Call** `flag = IDASetMaxNumJacsIC(ida_mem, maxnj);`

**Description** The function `IDASetMaxNumJacsIC` specifies the maximum number of the approximate Jacobian or preconditioner evaluations allowed when the Newton iteration appears to be slowly converging.

**Arguments** `ida_mem` (`void *`) pointer to the IDA memory block.  
`maxnj` (`int`) maximum allowed number of Jacobian or preconditioner evaluations.

**Return value** The return value `flag` (of type `int`) is one of  
`IDA_SUCCESS` The optional value has been successfully set.  
`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.  
`IDA_ILL_INPUT` `maxnj` is non-positive.

**Notes** The default value is 4.

**IDASetMaxNumItersIC**

**Call** `flag = IDASetMaxNumItersIC(ida_mem, maxnit);`

**Description** The function `IDASetMaxNumItersIC` specifies the maximum number of Newton iterations allowed in any one attempt to solve the initial conditions calculation problem.

**Arguments** `ida_mem` (`void *`) pointer to the IDA memory block.  
`maxnit` (`int`) maximum number of Newton iterations.

**Return value** The return value `flag` (of type `int`) is one of  
`IDA_SUCCESS` The optional value has been successfully set.  
`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.  
`IDA_ILL_INPUT` `maxnit` is non-positive.

**Notes** The default value is 10.

**IDASetMaxBacksIC**

**Call** `flag = IDASetMaxBacksIC(ida_mem, maxbacks);`

**Description** The function `IDASetMaxBacksIC` specifies the maximum number of linesearch backtracks allowed in any Newton iteration, when solving the initial conditions calculation problem.

**Arguments** `ida_mem` (`void *`) pointer to the IDA memory block.  
`maxbacks` (`int`) maximum number of linesearch backtracks per Newton step.

**Return value** The return value `flag` (of type `int`) is one of  
`IDA_SUCCESS` The optional value has been successfully set.  
`IDA_MEM_NULL` The `ida_mem` pointer is `NULL`.  
`IDA_ILL_INPUT` `maxbacks` is non-positive.

**Notes** The default value is 100.

**IDASetLineSearchOffIC**

**Call** `flag = IDASetLineSearchOffIC(ida_mem, lsoff);`

**Description** The function `IDASetLineSearchOffIC` specifies whether to turn on or off the linesearch algorithm.

**Arguments** `ida_mem` (`void *`) pointer to the IDA memory block.  
`lsoff` (`boolean_t`) a flag to turn off (`SUNTRUE`) or keep (`SUNFALSE`) the linesearch algorithm.

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

- IDA\_SUCCESS** The optional value has been successfully set.
- IDA\_MEM\_NULL** The **ida\_mem** pointer is NULL.

Notes The default value is **SUNFALSE**.

#### IDASetsStepToleranceIC

Call **flag = IDASetsStepToleranceIC(ida\_mem, steptol);**

Description The function **IDASetsStepToleranceIC** specifies a positive lower bound on the Newton step.

Arguments **ida\_mem** (**void \***) pointer to the IDA memory block.  
**steptol** (**int**) Minimum allowed WRMS-norm of the Newton step ( $> 0.0$ ).

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

- IDA\_SUCCESS** The optional value has been successfully set.
- IDA\_MEM\_NULL** The **ida\_mem** pointer is NULL.
- IDA\_ILL\_INPUT** The **steptol** tolerance is  $\leq 0.0$ .

Notes The default value is  $(\text{unit roundoff})^{2/3}$ .

#### 4.5.8.4 Rootfinding optional input functions

The following functions can be called to set optional inputs to control the rootfinding algorithm.

#### IDASetRootDirection

Call **flag = IDASetRootDirection(ida\_mem, rootdir);**

Description The function **IDASetRootDirection** specifies the direction of zero-crossings to be located and returned to the user.

Arguments **ida\_mem** (**void \***) pointer to the IDA memory block.  
**rootdir** (**int \***) state array of length **nrtfn**, the number of root functions  $g_i$ , as specified in the call to the function **IDARootInit**. A value of 0 for **rootdir[i]** indicates that crossing in either direction should be reported for  $g_i$ . A value of +1 or -1 indicates that the solver should report only zero-crossings where  $g_i$  is increasing or decreasing, respectively.

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

- IDA\_SUCCESS** The optional value has been successfully set.
- IDA\_MEM\_NULL** The **ida\_mem** pointer is NULL.
- IDA\_ILL\_INPUT** rootfinding has not been activated through a call to **IDARootInit**.

Notes The default behavior is to locate both zero-crossing directions.

#### IDASetNoInactiveRootWarn

Call **flag = IDASetNoInactiveRootWarn(ida\_mem);**

Description The function **IDASetNoInactiveRootWarn** disables issuing a warning if some root function appears to be identically zero at the beginning of the integration.

Arguments **ida\_mem** (**void \***) pointer to the IDA memory block.

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

- IDA\_SUCCESS** The optional value has been successfully set.
- IDA\_MEM\_NULL** The **ida\_mem** pointer is NULL.

Notes IDA will not report the initial conditions as a possible zero-crossing (assuming that one or more components  $g_i$  are zero at the initial time). However, if it appears that some  $g_i$  is identically zero at the initial time (i.e.,  $g_i$  is zero at the initial time and after the first step), IDA will issue a warning which can be disabled with this optional input function.

### 4.5.9 Interpolated output function

An optional function `IDAGetDky` is available to obtain additional output values. This function must be called after a successful return from `IDASolve` and provides interpolated values of  $y$  or its derivatives of order up to the last internal order used for any value of  $t$  in the last internal step taken by IDA.

The call to the `IDAGetDky` function has the following form:

<code>IDAGetDky</code>	
Call	<code>flag = IDAGetDky(ida_mem, t, k, dky);</code>
Description	The function <code>IDAGetDky</code> computes the interpolated values of the $k^{th}$ derivative of $y$ for any value of $t$ in the last internal step taken by IDA. The value of $k$ must be non-negative and smaller than the last internal order used. A value of 0 for $k$ means that the $y$ is interpolated. The value of $t$ must satisfy $t_n - h_u \leq t \leq t_n$ , where $t_n$ denotes the current internal time reached, and $h_u$ is the last internal step size used successfully.
Arguments	<code>ida_mem</code> (void *) pointer to the IDA memory block. <code>t</code> (realtype) time at which to interpolate. <code>k</code> (int) integer specifying the order of the derivative of $y$ wanted. <code>dky</code> (N.Vector) vector containing the interpolated $k^{th}$ derivative of $y(t)$ .
Return value	The return value <code>flag</code> (of type <code>int</code> ) is one of <code>IDA_SUCCESS</code> <code>IDAGetDky</code> succeeded. <code>IDA_MEM_NULL</code> The <code>ida_mem</code> argument was <code>NULL</code> . <code>IDA_BAD_T</code> <code>t</code> is not in the interval $[t_n - h_u, t_n]$ . <code>IDA_BAD_K</code> <code>k</code> is not one of $\{0, 1, \dots, klast\}$ . <code>IDA_BAD_DKY</code> <code>dky</code> is <code>NULL</code> .
Notes	It is only legal to call the function <code>IDAGetDky</code> after a successful return from <code>IDASolve</code> . Functions <code>IDAGetCurrentTime</code> , <code>IDAGetLastStep</code> and <code>IDAGetLastOrder</code> (see §4.5.10.2) can be used to access $t_n$ , $h_u$ and $klast$ .

### 4.5.10 Optional output functions

IDA provides an extensive list of functions that can be used to obtain solver performance information. Table 4.3 lists all optional output functions in IDA, which are then described in detail in the remainder of this section.

Some of the optional outputs, especially the various counters, can be very useful in determining how successful the IDA solver is in doing its job. For example, the counters `nsteps` and `nrevals` provide a rough measure of the overall cost of a given run, and can be compared among runs with differing input options to suggest which set of options is most efficient. The ratio `nniters/nsteps` measures the performance of the nonlinear solver in solving the nonlinear systems at each time step; typical values for this range from 1.1 to 1.8. The ratio `njevals/nniters` (in the case of a matrix-based linear solver), and the ratio `npevals/nniters` (in the case of an iterative linear solver) measure the overall degree of nonlinearity in these systems, and also the quality of the approximate Jacobian or preconditioner being used. Thus, for example, `njevals/nniters` can indicate if a user-supplied Jacobian is inaccurate, if this ratio is larger than for the case of the corresponding internal Jacobian. The ratio `nliters/nniters` measures the performance of the Krylov iterative linear solver, and thus (indirectly) the quality of the preconditioner.

Table 4.3: Optional outputs from IDA and IDALS

Optional output	Function name
<b>IDA main solver</b>	
Size of IDA real and integer workspace	IDAGetWorkSpace
Cumulative number of internal steps	IDAGetNumSteps
No. of calls to residual function	IDAGetNumResEvals
No. of calls to linear solver setup function	IDAGetNumLinSolvSetups
No. of local error test failures that have occurred	IDAGetNumErrTestFails
Order used during the last step	IDAGetLastOrder
Order to be attempted on the next step	IDAGetCurrentOrder
Order reductions due to stability limit detection	IDAGetNumStabLimOrderReds
Actual initial step size used	IDAGetActualInitStep
Step size used for the last step	IDAGetLastStep
Step size to be attempted on the next step	IDAGetCurrentStep
Current internal time reached by the solver	IDAGetCurrentTime
Suggested factor for tolerance scaling	IDAGetTolScaleFactor
Error weight vector for state variables	IDAGetErrWeights
Estimated local errors	IDAGetEstLocalErrors
No. of nonlinear solver iterations	IDAGetNumNonlinSolvIters
No. of nonlinear convergence failures	IDAGetNumNonlinSolvConvFails
Array showing roots found	IDAGetRootInfo
No. of calls to user root function	IDAGetNumGEvals
Name of constant associated with a return flag	IDAGetReturnFlagName
<b>IDA initial conditions calculation</b>	
Number of backtrack operations	IDAGetNumBacktrackops
Corrected initial conditions	IDAGetConsistentIC
<b>IDALS linear solver interface</b>	
Size of real and integer workspace	IDAGetLinWorkSpace
No. of Jacobian evaluations	IDAGetNumJacEvals
No. of residual calls for finite diff. Jacobian[-vector] evals.	IDAGetNumLinResEvals
No. of linear iterations	IDAGetNumLinIters
No. of linear convergence failures	IDAGetNumLinConvFails
No. of preconditioner evaluations	IDAGetNumPrecEvals
No. of preconditioner solves	IDAGetNumPrecSolves
No. of Jacobian-vector setup evaluations	IDAGetNumJTSetupEvals
No. of Jacobian-vector product evaluations	IDAGetNumJtimesEvals
Last return from a linear solver function	IDAGetLastLinFlag
Name of constant associated with a return flag	IDAGetLinReturnFlagName

#### 4.5.10.1 SUNDIALS version information

The following functions provide a way to get SUNDIALS version information at runtime.

##### SUNDIALSGetVersion

Call	<code>flag = SUNDIALSGetVersion(version, len);</code>
Description	The function <code>SUNDIALSGetVersion</code> fills a character array with SUNDIALS version information.
Arguments	<b>version</b> ( <code>char *</code> ) character array to hold the SUNDIALS version information. <b>len</b> ( <code>int</code> ) allocated length of the <b>version</b> character array.
Return value	If successful, <code>SUNDIALSGetVersion</code> returns 0 and <b>version</b> contains the SUNDIALS version information. Otherwise, it returns <code>-1</code> and <b>version</b> is not set (the input character array is too short).
Notes	A string of 25 characters should be sufficient to hold the version information. Any trailing characters in the <b>version</b> array are removed.

##### SUNDIALSGetVersionNumber

Call	<code>flag = SUNDIALSGetVersionNumber(&amp;major, &amp;minor, &amp;patch, label, len);</code>
Description	The function <code>SUNDIALSGetVersionNumber</code> set integers for the SUNDIALS major, minor, and patch release numbers and fills a character array with the release label if applicable.
Arguments	<b>major</b> ( <code>int</code> ) SUNDIALS release major version number. <b>minor</b> ( <code>int</code> ) SUNDIALS release minor version number. <b>patch</b> ( <code>int</code> ) SUNDIALS release patch version number. <b>label</b> ( <code>char *</code> ) character array to hold the SUNDIALS release label. <b>len</b> ( <code>int</code> ) allocated length of the <b>label</b> character array.
Return value	If successful, <code>SUNDIALSGetVersionNumber</code> returns 0 and the <b>major</b> , <b>minor</b> , <b>patch</b> , and <b>label</b> values are set. Otherwise, it returns <code>-1</code> and the values are not set (the input character array is too short).
Notes	A string of 10 characters should be sufficient to hold the label information. If a label is not used in the release version, no information is copied to <b>label</b> . Any trailing characters in the <b>label</b> array are removed.

#### 4.5.10.2 Main solver optional output functions

IDA 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, solver performance statistics, as well as additional data from the IDA memory block (a suggested tolerance scaling factor, the error weight vector, and the vector of estimated local errors). Also provided are functions to extract statistics related to the performance of the SUNNONLINSOL nonlinear solver being used. As a convenience, additional extraction functions provide the optional outputs in groups. These optional output functions are described next.

##### IDAGetWorkSpace

Call	<code>flag = IDAGetWorkSpace(ida_mem, &amp;lenrw, &amp;leniw);</code>
Description	The function <code>IDAGetWorkSpace</code> returns the IDA real and integer workspace sizes.
Arguments	<b>ida_mem</b> ( <code>void *</code> ) pointer to the IDA memory block. <b>lenrw</b> ( <code>long int</code> ) number of real values in the IDA workspace. <b>leniw</b> ( <code>long int</code> ) number of integer values in the IDA workspace.

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

- `IDA_SUCCESS` The optional output value has been successfully set.
- `IDA_MEM_NULL` The `ida_mem` pointer is NULL.

**Notes** In terms of the problem size  $N$ , the maximum method order `maxord`, and the number `nrtfn` of root functions (see §4.5.6), the actual size of the real workspace, in `realtype` words, is given by the following:

- base value:  $\text{lenrw} = 55 + (m + 6) * N_r + 3 * \text{nrtfn}$ ;
- with `IDASVtolerances`:  $\text{lenrw} = \text{lenrw} + N_r$ ;
- with constraint checking (see `IDASetConstraints`):  $\text{lenrw} = \text{lenrw} + N_r$ ;
- with `id` specified (see `IDASetId`):  $\text{lenrw} = \text{lenrw} + N_r$ ;

where  $m = \max(\text{maxord}, 3)$ , and  $N_r$  is the number of real words in one `N_Vector` ( $\approx N$ ). The size of the integer workspace (without distinction between `int` and `long int` words) is given by:

- base value:  $\text{leniw} = 38 + (m + 6) * N_i + \text{nrtfn}$ ;
- with `IDASVtolerances`:  $\text{leniw} = \text{leniw} + N_i$ ;
- with constraint checking:  $\text{leniw} = \text{leniw} + N_i$ ;
- with `id` specified:  $\text{leniw} = \text{leniw} + N_i$ ;

where  $N_i$  is the number of integer words in one `N_Vector` ( $= 1$  for `NVECTOR_SERIAL` and  $2 * \text{npes}$  for `NVECTOR_PARALLEL` on `npes` processors).

For the default value of `maxord`, with no rootfinding, no `id`, no constraints, and with no call to `IDASVtolerances`, these lengths are given roughly by:  $\text{lenrw} = 55 + 11N$ ,  $\text{leniw} = 49$ .

#### IDAGetNumSteps

**Call** `flag = IDAGetNumSteps(ida_mem, &nsteps);`

**Description** The function `IDAGetNumSteps` returns the cumulative number of internal steps taken by the solver (total so far).

**Arguments** `ida_mem` (`void *`) pointer to the IDA memory block.  
`nsteps` (`long int`) number of steps taken by IDA.

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

- `IDA_SUCCESS` The optional output value has been successfully set.
- `IDA_MEM_NULL` The `ida_mem` pointer is NULL.

#### IDAGetNumResEvals

**Call** `flag = IDAGetNumResEvals(ida_mem, &nrevals);`

**Description** The function `IDAGetNumResEvals` returns the number of calls to the user's residual evaluation function.

**Arguments** `ida_mem` (`void *`) pointer to the IDA memory block.  
`nrevals` (`long int`) number of calls to the user's `res` function.

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

- `IDA_SUCCESS` The optional output value has been successfully set.
- `IDA_MEM_NULL` The `ida_mem` pointer is NULL.

**Notes** The `nrevals` value returned by `IDAGetNumResEvals` does not account for calls made to `res` from a linear solver or preconditioner module.



**IDAGetNumLinSolvSetups**

**Call**            `flag = IDAGetNumLinSolvSetups(ida_mem, &nlinsetups);`

**Description**   The function `IDAGetNumLinSolvSetups` returns the cumulative number of calls made to the linear solver's setup function (total so far).

**Arguments**    `ida_mem`    (void \*) pointer to the IDA memory block.  
                  `nlinsetups` (long int) number of calls made to the linear solver setup function.

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

`IDA_SUCCESS`   The optional output value has been successfully set.  
                  `IDA_MEM_NULL`   The `ida_mem` pointer is NULL.

**IDAGetNumErrTestFails**

**Call**            `flag = IDAGetNumErrTestFails(ida_mem, &netfails);`

**Description**   The function `IDAGetNumErrTestFails` returns the cumulative number of local error test failures that have occurred (total so far).

**Arguments**    `ida_mem`    (void \*) pointer to the IDA memory block.  
                  `netfails` (long int) number of error test failures.

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

`IDA_SUCCESS`   The optional output value has been successfully set.  
                  `IDA_MEM_NULL`   The `ida_mem` pointer is NULL.

**IDAGetLastOrder**

**Call**            `flag = IDAGetLastOrder(ida_mem, &klast);`

**Description**   The function `IDAGetLastOrder` returns the integration method order used during the last internal step.

**Arguments**    `ida_mem` (void \*) pointer to the IDA memory block.  
                  `klast`    (int) method order used on the last internal step.

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

`IDA_SUCCESS`   The optional output value has been successfully set.  
                  `IDA_MEM_NULL`   The `ida_mem` pointer is NULL.

**IDAGetCurrentOrder**

**Call**            `flag = IDAGetCurrentOrder(ida_mem, &kcur);`

**Description**   The function `IDAGetCurrentOrder` returns the integration method order to be used on the next internal step.

**Arguments**    `ida_mem` (void \*) pointer to the IDA memory block.  
                  `kcur`    (int) method order to be used on the next internal step.

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

`IDA_SUCCESS`   The optional output value has been successfully set.  
                  `IDA_MEM_NULL`   The `ida_mem` pointer is NULL.

**IDAGetLastStep**

**Call**            `flag = IDAGetLastStep(ida_mem, &hlast);`

**Description**   The function `IDAGetLastStep` returns the integration step size taken on the last internal step.

**Arguments**    `ida_mem` (`void *`) pointer to the IDA memory block.  
                  `hlast`    (`realtype`) step size taken on the last internal step by IDA, or last artificial step size used in `IDACalcIC`, whichever was called last.

**Return value**   The return value `flag` (of type `int`) is one of  
                  `IDA_SUCCESS`   The optional output value has been successfully set.  
                  `IDA_MEM_NULL`   The `ida_mem` pointer is `NULL`.

**IDAGetCurrentStep**

**Call**            `flag = IDAGetCurrentStep(ida_mem, &hcur);`

**Description**   The function `IDAGetCurrentStep` returns the integration step size to be attempted on the next internal step.

**Arguments**    `ida_mem` (`void *`) pointer to the IDA memory block.  
                  `hcur`    (`realtype`) step size to be attempted on the next internal step.

**Return value**   The return value `flag` (of type `int`) is one of  
                  `IDA_SUCCESS`   The optional output value has been successfully set.  
                  `IDA_MEM_NULL`   The `ida_mem` pointer is `NULL`.

**IDAGetActualInitStep**

**Call**            `flag = IDAGetActualInitStep(ida_mem, &hinused);`

**Description**   The function `IDAGetActualInitStep` returns the value of the integration step size used on the first step.

**Arguments**    `ida_mem` (`void *`) pointer to the IDA memory block.  
                  `hinused` (`realtype`) actual value of initial step size.

**Return value**   The return value `flag` (of type `int`) is one of  
                  `IDA_SUCCESS`   The optional output value has been successfully set.  
                  `IDA_MEM_NULL`   The `ida_mem` pointer is `NULL`.

**Notes**           Even if the value of the initial integration step size was specified by the user through a call to `IDASetInitStep`, this value might have been changed by IDA to ensure that the step size is within the prescribed bounds ( $h_{\min} \leq h_0 \leq h_{\max}$ ), or to meet the local error test.

**IDAGetCurrentTime**

**Call**            `flag = IDAGetCurrentTime(ida_mem, &tcur);`

**Description**   The function `IDAGetCurrentTime` returns the current internal time reached by the solver.


**Arguments**    `ida_mem` (`void *`) pointer to the IDA memory block.  
                  `tcur`    (`realtype`) current internal time reached.

**Return value**   The return value `flag` (of type `int`) is one of  
                  `IDA_SUCCESS`   The optional output value has been successfully set.  
                  `IDA_MEM_NULL`   The `ida_mem` pointer is `NULL`.


**IDAGetTolScaleFactor**

- Call** `flag = IDAGetTolScaleFactor(ida_mem, &tolsfac);`
- Description** The function `IDAGetTolScaleFactor` returns a suggested factor by which the user's tolerances should be scaled when too much accuracy has been requested for some internal step.
- Arguments** `ida_mem` (`void *`) pointer to the IDA memory block.  
`tolsfac` (`realtype`) suggested scaling factor for user tolerances.
- Return value** The return value `flag` (of type `int`) is one of  
`IDA_SUCCESS` The optional output value has been successfully set.  
`IDA_MEM_NULL` The `ida_mem` pointer is NULL.

**IDAGetErrWeights**

- Call** `flag = IDAGetErrWeights(ida_mem, eweight);`
- Description** The function `IDAGetErrWeights` returns the solution error weights at the current time. These are the  $W_i$  given by Eq. (2.6) (or by the user's `IDAewtFn`).
- Arguments** `ida_mem` (`void *`) pointer to the IDA memory block.  
`eweight` (`N_Vector`) solution error weights at the current time.
- Return value** The return value `flag` (of type `int`) is one of  
`IDA_SUCCESS` The optional output value has been successfully set.  
`IDA_MEM_NULL` The `ida_mem` pointer is NULL.
- Notes** The user must allocate space for `eweight`. 

**IDAGetEstLocalErrors**

- Call** `flag = IDAGetEstLocalErrors(ida_mem, ele);`
- Description** The function `IDAGetEstLocalErrors` returns the estimated local errors.
- Arguments** `ida_mem` (`void *`) pointer to the IDA memory block.  
`ele` (`N_Vector`) estimated local errors at the current time.
- Return value** The return value `flag` (of type `int`) is one of  
`IDA_SUCCESS` The optional output value has been successfully set.  
`IDA_MEM_NULL` The `ida_mem` pointer is NULL.
- Notes** The user must allocate space for `ele`.  
The values returned in `ele` are only valid if `IDASolve` returned a non-negative value.  
The `ele` vector, together with the `eweight` vector from `IDAGetErrWeights`, can be used to determine how the various components of the system contributed to the estimated local error test. Specifically, that error test uses the RMS norm of a vector whose components are the products of the components of these two vectors. Thus, for example, if there were recent error test failures, the components causing the failures are those with largest values for the products, denoted loosely as `eweight[i]*ele[i]`. 

**IDAGetIntegratorStats**

- Call** `flag = IDAGetIntegratorStats(ida_mem, &nsteps, &nrevals, &nlinsetups,  
&netfails, &klast, &kcur, &hinused,  
&hlast, &hcur, &tcur);`
- Description** The function `IDAGetIntegratorStats` returns the IDA integrator statistics as a group.
- Arguments** `ida_mem` (`void *`) pointer to the IDA memory block.

**nsteps** (long int) cumulative number of steps taken by IDA.  
**nrevals** (long int) cumulative number of calls to the user's **res** function.  
**nlinsetups** (long int) cumulative number of calls made to the linear solver setup function.  
**netfails** (long int) cumulative number of error test failures.  
**klast** (int) method order used on the last internal step.  
**kcur** (int) method order to be used on the next internal step.  
**hinused** (realtype) actual value of initial step size.  
**hlast** (realtype) step size taken on the last internal step.  
**hcur** (realtype) step size to be attempted on the next internal step.  
**tcur** (realtype) current internal time reached.

Return value The return value **flag** (of type **int**) is one of  
     **IDA\_SUCCESS** the optional output values have been successfully set.  
     **IDA\_MEM\_NULL** the **ida\_mem** pointer is NULL.

#### IDAGetNumNonlinSolvIters

Call **flag = IDAGetNumNonlinSolvIters(ida\_mem, &nniters);**  
 Description The function **IDAGetNumNonlinSolvIters** returns the cumulative number of nonlinear iterations performed.  
 Arguments **ida\_mem** (void \*) pointer to the IDA memory block.  
           **nniters** (long int) number of nonlinear iterations performed.  
 Return value The return value **flag** (of type **int**) is one of  
     **IDA\_SUCCESS** The optional output value has been successfully set.  
     **IDA\_MEM\_NULL** The **ida\_mem** pointer is NULL.  
     **IDA\_MEM\_FAIL** The **SUNNONLINSOL** module is NULL.

#### IDAGetNumNonlinSolvConvFails

Call **flag = IDAGetNumNonlinSolvConvFails(ida\_mem, &nncfails);**  
 Description The function **IDAGetNumNonlinSolvConvFails** returns the cumulative number of nonlinear convergence failures that have occurred.  
 Arguments **ida\_mem** (void \*) pointer to the IDA memory block.  
           **nncfails** (long int) number of nonlinear convergence failures.  
 Return value The return value **flag** (of type **int**) is one of  
     **IDA\_SUCCESS** The optional output value has been successfully set.  
     **IDA\_MEM\_NULL** The **ida\_mem** pointer is NULL.

#### IDAGetNonlinSolvStats

Call **flag = IDAGetNonlinSolvStats(ida\_mem, &nniters, &nncfails);**  
 Description The function **IDAGetNonlinSolvStats** returns the IDA nonlinear solver statistics as a group.  
 Arguments **ida\_mem** (void \*) pointer to the IDA memory block.  
           **nniters** (long int) cumulative number of nonlinear iterations performed.  
           **nncfails** (long int) cumulative number of nonlinear convergence failures.  
 Return value The return value **flag** (of type **int**) is one of  
     **IDA\_SUCCESS** The optional output value has been successfully set.

IDA\_MEM\_NULL The `ida_mem` pointer is NULL.

IDA\_MEM\_FAIL The SUNNONLINSOL module is NULL.

#### IDAGetReturnFlagName

Call `name = IDAGetReturnFlagName(flag);`

Description The function `IDAGetReturnFlagName` returns the name of the IDA constant corresponding to `flag`.

Arguments The only argument, of type `int`, is a return flag from an IDA function.

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

#### 4.5.10.3 Initial condition calculation optional output functions

#### IDAGetNumBcktrackOps

Call `flag = IDAGetNumBacktrackOps(ida_mem, &nbacktr);`

Description The function `IDAGetNumBacktrackOps` returns the number of backtrack operations done in the linesearch algorithm in `IDACalcIC`.

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.  
`nbacktr` (`long int`) the cumulative number of backtrack operations.

Return value The return value `flag` (of type `int`) is one of  
 IDA\_SUCCESS The optional output value has been successfully set.  
 IDA\_MEM\_NULL The `ida_mem` pointer is NULL.

#### IDAGetConsistentIC

Call `flag = IDAGetConsistentIC(ida_mem, yy0_mod, yp0_mod);`

Description The function `IDAGetConsistentIC` returns the corrected initial conditions calculated by `IDACalcIC`.

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.  
`yy0_mod` (`N_Vector`) consistent solution vector.  
`yp0_mod` (`N_Vector`) consistent derivative vector.

Return value The return value `flag` (of type `int`) is one of  
 IDA\_SUCCESS The optional output value has been successfully set.  
 IDA\_ILL\_INPUT The function was not called before the first call to `IDASolve`.  
 IDA\_MEM\_NULL The `ida_mem` pointer is NULL.

Notes If the consistent solution vector or consistent derivative vector is not desired, pass NULL for the corresponding argument.

The user must allocate space for `yy0_mod` and `yp0_mod` (if not NULL).



#### 4.5.10.4 Rootfinding optional output functions

There are two optional output functions associated with rootfinding.

**IDAGetRootInfo**

Call	<code>flag = IDAGetRootInfo(ida_mem, rootsfound);</code>
Description	The function <code>IDAGetRootInfo</code> returns an array showing which functions were found to have a root.
Arguments	<p><code>ida_mem</code> (<code>void *</code>) pointer to the IDA memory block.</p> <p><code>rootsfound</code> (<code>int *</code>) array of length <code>nrtfn</code> with the indices of the user functions <math>g_i</math> found to have a root. For <math>i = 0, \dots, \text{nrtfn} - 1</math>, <code>rootsfound[i]</code> <math>\neq 0</math> if <math>g_i</math> has a root, and <math>= 0</math> if not.</p>
Return value	<p>The return value <code>flag</code> (of type <code>int</code>) is one of</p> <p><code>IDA_SUCCESS</code> The optional output values have been successfully set.</p> <p><code>IDA_MEM_NULL</code> The <code>ida_mem</code> pointer is NULL.</p>
Notes	<p>Note that, for the components <math>g_i</math> for which a root was found, the sign of <code>rootsfound[i]</code> indicates the direction of zero-crossing. A value of <math>+1</math> indicates that <math>g_i</math> is increasing, while a value of <math>-1</math> indicates a decreasing <math>g_i</math>.</p> <p>The user must allocate memory for the vector <code>rootsfound</code>.</p>

**IDAGetNumGEvals**

Call	<code>flag = IDAGetNumGEvals(ida_mem, &amp;ngevals);</code>
Description	The function <code>IDAGetNumGEvals</code> returns the cumulative number of calls to the user root function $g$ .
Arguments	<p><code>ida_mem</code> (<code>void *</code>) pointer to the IDA memory block.</p> <p><code>ngevals</code> (<code>long int</code>) number of calls to the user's function <math>g</math> so far.</p>
Return value	<p>The return value <code>flag</code> (of type <code>int</code>) is one of</p> <p><code>IDA_SUCCESS</code> The optional output value has been successfully set.</p> <p><code>IDA_MEM_NULL</code> The <code>ida_mem</code> pointer is NULL.</p>

**4.5.10.5 IDALS linear solver interface optional output functions**

The following optional outputs are available from the IDALS modules: workspace requirements, number of calls to the Jacobian routine, number of calls to the residual routine for finite-difference Jacobian or Jacobian-vector product approximation, number of linear iterations, number of linear convergence failures, number of calls to the preconditioner setup and solve routines, number of calls to the Jacobian-vector setup and product routines, and last return value from an IDALS function. Note that, where the name of an output would otherwise conflict with the name of an optional output from the main solver, a suffix LS (for Linear Solver) has been added (e.g. `lenrwLS`).

**IDAGetLinWorkSpace**

Call	<code>flag = IDAGetLinWorkSpace(ida_mem, &amp;lenrwLS, &amp;leniwLS);</code>
Description	The function <code>IDAGetLinWorkSpace</code> returns the sizes of the real and integer workspaces used by the IDALS linear solver interface.
Arguments	<p><code>ida_mem</code> (<code>void *</code>) pointer to the IDA memory block.</p> <p><code>lenrwLS</code> (<code>long int</code>) the number of real values in the IDALS workspace.</p> <p><code>leniwLS</code> (<code>long int</code>) the number of integer values in the IDALS workspace.</p>
Return value	<p>The return value <code>flag</code> (of type <code>int</code>) is one of</p> <p><code>IDALS_SUCCESS</code> The optional output value has been successfully set.</p> <p><code>IDALS_MEM_NULL</code> The <code>ida_mem</code> pointer is NULL.</p> <p><code>IDALS_LMEM_NULL</code> The IDALS linear solver has not been initialized.</p>

**Notes** The workspace requirements reported by this routine correspond only to memory allocated within this interface and to memory allocated by the SUNLINSOL object attached to it. The template Jacobian matrix allocated by the user outside of IDALS is not included in this report.

The previous routines `IDADlsGetWorkspace` and `IDASpilsGetWorkspace` are now wrappers for this routine, and may still be used for backward-compatibility. However, these will be deprecated in future releases, so we recommend that users transition to the new routine name soon.

#### `IDAGetNumJacEvals`

**Call** `flag = IDAGetNumJacEvals(ida_mem, &njevals);`

**Description** The function `IDAGetNumJacEvals` returns the cumulative number of calls to the IDALS Jacobian approximation function.

**Arguments** `ida_mem` (`void *`) pointer to the IDA memory block.  
`njevals` (`long int`) the cumulative number of calls to the Jacobian function (total so far).

**Return value** The return value `flag` (of type `int`) is one of  
`IDALS_SUCCESS` The optional output value has been successfully set.  
`IDALS_MEM_NULL` The `ida_mem` pointer is `NULL`.  
`IDALS_LMEM_NULL` The IDALS linear solver has not been initialized.

**Notes** The previous routine `IDADlsGetNumJacEvals` is now a wrapper for this routine, and may still be used for backward-compatibility. However, this will be deprecated in future releases, so we recommend that users transition to the new routine name soon.

#### `IDAGetNumLinResEvals`

**Call** `flag = IDAGetNumLinResEvals(ida_mem, &nrevalsLS);`

**Description** The function `IDAGetNumLinResEvals` returns the cumulative number of calls to the user residual function due to the finite difference Jacobian approximation or finite difference Jacobian-vector product approximation.

**Arguments** `ida_mem` (`void *`) pointer to the IDA memory block.  
`nrevalsLS` (`long int`) the cumulative number of calls to the user residual function.

**Return value** The return value `flag` (of type `int`) is one of  
`IDALS_SUCCESS` The optional output value has been successfully set.  
`IDALS_MEM_NULL` The `ida_mem` pointer is `NULL`.  
`IDALS_LMEM_NULL` The IDALS linear solver has not been initialized.

**Notes** The value `nrevalsLS` is incremented only if one of the default internal difference quotient functions is used.

The previous routines `IDADlsGetNumRhsEvals` and `IDASpilsGetNumRhsEvals` are now wrappers for this routine, and may still be used for backward-compatibility. However, these will be deprecated in future releases, so we recommend that users transition to the new routine name soon.

#### `IDAGetNumLinIters`

**Call** `flag = IDAGetNumLinIters(ida_mem, &nliters);`

**Description** The function `IDAGetNumLinIters` returns the cumulative number of linear iterations.

**Arguments** `ida_mem` (`void *`) pointer to the IDA memory block.

`nliters` (`long int`) the current number of linear iterations.

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

`IDALS_SUCCESS` The optional output value has been successfully set.

`IDALS_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDALS_LMEM_NULL` The IDALS linear solver has not been initialized.

Notes The previous routine `IDASpilsGetNumLinIters` is now a wrapper for this routine, and may still be used for backward-compatibility. However, this will be deprecated in future releases, so we recommend that users transition to the new routine name soon.

#### `IDAGetNumLinConvFails`

Call `flag = IDAGetNumLinConvFails(ida_mem, &nlcfails);`

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

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.

`nlcfails` (`long int`) the current number of linear convergence failures.

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

`IDALS_SUCCESS` The optional output value has been successfully set.

`IDALS_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDALS_LMEM_NULL` The IDALS linear solver has not been initialized.

Notes The previous routine `IDASpilsGetNumConvFails` is now a wrapper for this routine, and may still be used for backward-compatibility. However, this will be deprecated in future releases, so we recommend that users transition to the new routine name soon.

#### `IDAGetNumPrecEvals`

Call `flag = IDAGetNumPrecEvals(ida_mem, &npevals);`

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

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.

`npevals` (`long int`) the cumulative number of calls to `psetup`.

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

`IDALS_SUCCESS` The optional output value has been successfully set.

`IDALS_MEM_NULL` The `ida_mem` pointer is `NULL`.

`IDALS_LMEM_NULL` The IDALS linear solver has not been initialized.

Notes The previous routine `IDASpilsGetNumPrecEvals` is now a wrapper for this routine, and may still be used for backward-compatibility. However, this will be deprecated in future releases, so we recommend that users transition to the new routine name soon.

#### `IDAGetNumPrecSolves`

Call `flag = IDAGetNumPrecSolves(ida_mem, &npsolves);`

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

Arguments `ida_mem` (`void *`) pointer to the IDA memory block.

`npsolves` (`long int`) the cumulative number of calls to `psolve`.

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

`IDALS_SUCCESS` The optional output value has been successfully set.



IDALS\_MEM\_NULL The `ida_mem` pointer is NULL.  
 IDALS\_LMEM\_NULL The IDALS linear solver has not been initialized.

Notes The previous routine `IDASpilsGetNumPrecSolves` is now a wrapper for this routine, and may still be used for backward-compatibility. However, this will be deprecated in future releases, so we recommend that users transition to the new routine name soon.

#### IDAGetNumJTSetupEvals

Call `flag = IDAGetNumJTSetupEvals(ida_mem, &njtsetup);`

Description The function `IDAGetNumJTSetupEvals` returns the cumulative number of calls made to the Jacobian-vector setup function `jtsetup`.

Arguments `ida_mem` (void \*) pointer to the IDA memory block.  
`njtsetup` (long int) the current number of calls to `jtsetup`.

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

IDALS\_SUCCESS The optional output value has been successfully set.  
 IDALS\_MEM\_NULL The `ida_mem` pointer is NULL.  
 IDALS\_LMEM\_NULL The IDALS linear solver has not been initialized.

Notes The previous routine `IDASpilsGetNumJTSetupEvals` is now a wrapper for this routine, and may still be used for backward-compatibility. However, this will be deprecated in future releases, so we recommend that users transition to the new routine name soon.

#### IDAGetNumJtimesEvals

Call `flag = IDAGetNumJtimesEvals(ida_mem, &njvevals);`

Description The function `IDAGetNumJtimesEvals` returns the cumulative number of calls made to the Jacobian-vector function, `jtimes`.

Arguments `ida_mem` (void \*) pointer to the IDA memory block.  
`njvevals` (long int) the cumulative number of calls to `jtimes`.

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

IDALS\_SUCCESS The optional output value has been successfully set.  
 IDALS\_MEM\_NULL The `ida_mem` pointer is NULL.  
 IDALS\_LMEM\_NULL The IDALS linear solver has not been initialized.

Notes The previous routine `IDASpilsGetNumJtimesEvals` is now a wrapper for this routine, and may still be used for backward-compatibility. However, this will be deprecated in future releases, so we recommend that users transition to the new routine name soon.

#### IDAGetLastLinFlag

Call `flag = IDAGetLastLinFlag(ida_mem, &lsflag);`

Description The function `IDAGetLastLinFlag` returns the last return value from an IDALS routine.

Arguments `ida_mem` (void \*) pointer to the IDA memory block.  
`lsflag` (long int) the value of the last return flag from an IDALS function.

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

IDALS\_SUCCESS The optional output value has been successfully set.  
 IDALS\_MEM\_NULL The `ida_mem` pointer is NULL.  
 IDALS\_LMEM\_NULL The IDALS linear solver has not been initialized.

Notes	<p>If the IDALS setup function failed (i.e., <code>IDASolve</code> returned <code>IDA_LSETUP_FAIL</code>) when using the <code>SUNLINSOL_DENSE</code> or <code>SUNLINSOL_BAND</code> modules, then the value of <code>lsflag</code> is equal to the column index (numbered from one) at which a zero diagonal element was encountered during the LU factorization of the (dense or banded) Jacobian matrix.</p> <p>If the IDALS setup function failed when using another <code>SUNLINSOL</code> module, then <code>lsflag</code> will be <code>SUNLS_PSET_FAIL_UNREC</code>, <code>SUNLS_ASET_FAIL_UNREC</code>, or <code>SUNLS_PACKAGE_FAIL_UNREC</code>.</p> <p>If the IDALS solve function failed (<code>IDASolve</code> returned <code>IDA_LSOLVE_FAIL</code>), <code>lsflag</code> contains the error return flag from the <code>SUNLINSOL</code> object, which will be one of: <code>SUNLS_MEM_NULL</code>, indicating that the <code>SUNLINSOL</code> memory is <code>NULL</code>; <code>SUNLS_ATIMES_FAIL_UNREC</code>, indicating an unrecoverable failure in the <math>J*v</math> function; <code>SUNLS_PSOLVE_FAIL_UNREC</code>, indicating that the preconditioner solve function <code>psolve</code> failed unrecoverably; <code>SUNLS_GS_FAIL</code>, indicating a failure in the Gram-Schmidt procedure (generated only in <code>SPGMR</code> or <code>SPFGMR</code>); <code>SUNLS_QRSOL_FAIL</code>, indicating that the matrix <math>R</math> was found to be singular during the QR solve phase (<code>SPGMR</code> and <code>SPFGMR</code> only); or <code>SUNLS_PACKAGE_FAIL_UNREC</code>, indicating an unrecoverable failure in an external iterative linear solver package.</p> <p>The previous routines <code>IDADlsGetLastFlag</code> and <code>IDASpilsGetLastFlag</code> are now wrappers for this routine, and may still be used for backward-compatibility. However, these will be deprecated in future releases, so we recommend that users transition to the new routine name soon.</p>
-------	--

<b>IDAGetLinReturnFlagName</b>
--------------------------------

Call	<code>name = IDAGetLinReturnFlagName(lsflag);</code>
Description	The function <code>IDAGetLinReturnFlagName</code> returns the name of the IDALS constant corresponding to <code>lsflag</code> .
Arguments	The only argument, of type <code>long int</code> , is a return flag from an IDALS function.
Return value	The return value is a string containing the name of the corresponding constant. If $1 \leq \text{lsflag} \leq N$ (LU factorization failed), this function returns “NONE”.
Notes	The previous routines <code>IDADlsGetReturnFlagName</code> and <code>IDASpilsGetReturnFlagName</code> are now wrappers for this routine, and may still be used for backward-compatibility. However, these will be deprecated in future releases, so we recommend that users transition to the new routine name soon.

#### 4.5.11 IDA reinitialization function

The function `IDAREInit` reinitializes the main IDA solver for the solution of a new problem, where a prior call to `IDAInit` has been made. The new problem must have the same size as the previous one. `IDAREInit` performs the same input checking and initializations that `IDAInit` does, but does no memory allocation, as it assumes that the existing internal memory is sufficient for the new problem. A call to `IDAREInit` deletes the solution history that was stored internally during the previous integration. Following a successful call to `IDAREInit`, call `IDASolve` again for the solution of the new problem.

The use of `IDAREInit` requires that the maximum method order, `maxord`, is no larger for the new problem than for the problem specified in the last call to `IDAInit`. In addition, the same `NVECTOR` module set for the previous problem will be reused for the new problem.

If there are changes to the linear solver specifications, make the appropriate calls to either the linear solver objects themselves, or to the IDALS interface routines, as described in §4.5.3.

If there are changes to any optional inputs, make the appropriate `IDASet***` calls, as described in §4.5.8. Otherwise, all solver inputs set previously remain in effect.

One important use of the `IDAREInit` function is in the treating of jump discontinuities in the residual function. Except in cases of fairly small jumps, it is usually more efficient to stop at each point

of discontinuity and restart the integrator with a readjusted DAE model, using a call to `IDAReInit`. To stop when the location of the discontinuity is known, simply make that location a value of `tout`. To stop when the location of the discontinuity is determined by the solution, use the rootfinding feature. In either case, it is critical that the residual function *not* incorporate the discontinuity, but rather have a smooth extension over the discontinuity, so that the step across it (and subsequent rootfinding, if used) can be done efficiently. Then use a switch within the residual function (communicated through `user_data`) that can be flipped between the stopping of the integration and the restart, so that the restarted problem uses the new values (which have jumped). Similar comments apply if there is to be a jump in the dependent variable vector.

#### **IDAReInit**

Call	<code>flag = IDAReInit(ida_mem, t0, y0, yp0);</code>
Description	The function <code>IDAReInit</code> provides required problem specifications and reinitializes IDA.
Arguments	<code>ida_mem</code> (void *) pointer to the IDA memory block. <code>t0</code> (realtype) is the initial value of $t$ . <code>y0</code> (N_Vector) is the initial value of $y$ . <code>yp0</code> (N_Vector) is the initial value of $\dot{y}$ .
Return value	The return value <code>flag</code> (of type <code>int</code> ) will be one of the following: <ul style="list-style-type: none"> <li><code>IDA_SUCCESS</code> The call to <code>IDAReInit</code> was successful.</li> <li><code>IDA_MEM_NULL</code> The IDA memory block was not initialized through a previous call to <code>IDACreate</code>.</li> <li><code>IDA_NO_MALLOC</code> Memory space for the IDA memory block was not allocated through a previous call to <code>IDAInit</code>.</li> <li><code>IDA_ILL_INPUT</code> An input argument to <code>IDAReInit</code> has an illegal value.</li> </ul>
Notes	If an error occurred, <code>IDAReInit</code> also sends an error message to the error handler function.

## 4.6 User-supplied functions

The user-supplied functions consist of one function defining the DAE residual, (optionally) a function that handles error and warning messages, (optionally) a function that provides the error weight vector, (optionally) one or two functions that provide Jacobian-related information for the linear solver, and (optionally) one or two functions that define the preconditioner for use in any of the Krylov iteration algorithms.

### 4.6.1 Residual function

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

#### **IDAResFn**

Definition	<code>typedef int (*IDAResFn)(realtype tt, N_Vector yy, N_Vector yp, N_Vector rr, void *user_data);</code>
Purpose	This function computes the problem residual for given values of the independent variable $t$ , state vector $y$ , and derivative $\dot{y}$ .
Arguments	<code>tt</code> is the current value of the independent variable. <code>yy</code> is the current value of the dependent variable vector, $y(t)$ . <code>yp</code> is the current value of $\dot{y}(t)$ . <code>rr</code> is the output residual vector $F(t, y, \dot{y})$ .

`user_data` is a pointer to user data, the same as the `user_data` parameter passed to `IDASetUserData`.

**Return value** An `IDAResFn` function type should return a value of 0 if successful, a positive value if a recoverable error occurred (e.g. `yy` has an illegal value), or a negative value if a nonrecoverable error occurred. In the last case, the integrator halts. If a recoverable error occurred, the integrator will attempt to correct and retry.

**Notes** A recoverable failure error return from the `IDAResFn` is typically used to flag a value of the dependent variable `y` that is “illegal” in some way (e.g., negative where only a non-negative value is physically meaningful). If such a return is made, IDA will attempt to recover (possibly repeating the nonlinear solve, or reducing the step size) in order to avoid this recoverable error return.

For efficiency reasons, the DAE residual function is not evaluated at the converged solution of the nonlinear solver. Therefore, in general, a recoverable error in that converged value cannot be corrected. (It may be detected when the right-hand side function is called the first time during the following integration step, but a successful step cannot be undone.)

Allocation of memory for `yp` is handled within IDA.

#### 4.6.2 Error message handler function

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

**IDAErrorHandlerFn**

**Definition**

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

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

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

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

**Notes** `error_code` is negative for errors and positive (`IDA_WARNING`) for warnings. If a function that returns a pointer to memory encounters an error, it sets `error_code` to 0.

#### 4.6.3 Error weight function

As an alternative to providing the relative and absolute tolerances, the user may provide a function of type `IDAWEtFn` to compute a vector `ewt` containing the multiplicative weights  $W_i$  used in the WRMS norm  $\|v\|_{\text{WRMS}} = \sqrt{(1/N) \sum_1^N (W_i \cdot v_i)^2}$ . These weights will be used in place of those defined by Eq. (2.6). The function type `IDAWEtFn` is defined as follows:

**IDAWEtFn**

**Definition**

```
typedef int (*IDAWEtFn)(N_Vector y, N_Vector ewt, void *user_data);
```

**Purpose** This function computes the WRMS error weights for the vector `y`.

Arguments	$y$	is the value of the dependent variable vector at which the weight vector is to be computed.
	<code>ewt</code>	is the output vector containing the error weights.
	<code>user_data</code>	is a pointer to user data, the same as the <code>user_data</code> parameter passed to <code>IDASetUserData</code> .
Return value	An <code>IDAEwtFn</code> function type must return 0 if it successfully set the error weights and $-1$ otherwise.	
Notes	Allocation of memory for <code>ewt</code> is handled within IDA.  The error weight vector must have all components positive. It is the user's responsibility to perform this test and return $-1$ if it is not satisfied.	



#### 4.6.4 Rootfinding function

If a rootfinding problem is to be solved during the integration of the DAE system, the user must supply a C function of type `IDARootFn`, defined as follows:

##### `IDARootFn`

Definition	<pre>typedef int (*IDARootFn)(realtype t, N_Vector y, N_Vector yp,                         realtype *gout, void *user_data);</pre>	
Purpose	This function computes a vector-valued function $g(t, y, \dot{y})$ such that the roots of the <code>nrtfn</code> components $g_i(t, y, \dot{y})$ are to be found during the integration.	
Arguments	$t$	is the current value of the independent variable.
	$y$	is the current value of the dependent variable vector, $y(t)$ .
	$yp$	is the current value of $\dot{y}(t)$ , the $t$ -derivative of $y$ .
	<code>gout</code>	is the output array, of length <code>nrtfn</code> , with components $g_i(t, y, \dot{y})$ .
	<code>user_data</code>	is a pointer to user data, the same as the <code>user_data</code> parameter passed to <code>IDASetUserData</code> .
Return value	An <code>IDARootFn</code> should return 0 if successful or a non-zero value if an error occurred (in which case the integration is halted and <code>IDASolve</code> returns <code>IDA_RTFUNC_FAIL</code> ).	
Notes	Allocation of memory for <code>gout</code> is handled within IDA.	

#### 4.6.5 Jacobian construction (matrix-based linear solvers)

If a matrix-based linear solver module is used (i.e. a non-NULL `SUNMATRIX` object was supplied to `IDASetLinearSolver`), the user may provide a function of type `IDALsJacFn` defined as follows:

##### `IDALsJacFn`

Definition	<pre>typedef int (*IDALsJacFn)(realtype tt, realtype cj,                           N_Vector yy, N_Vector yp, N_Vector rr,                           SUNMatrix Jac, void *user_data,                           N_Vector tmp1, N_Vector tmp2, N_Vector tmp3);</pre>	
Purpose	This function computes the Jacobian matrix $J$ of the DAE system (or an approximation to it), defined by Eq. (2.5).	
Arguments	$tt$	is the current value of the independent variable $t$ .
	$cj$	is the scalar in the system Jacobian, proportional to the inverse of the step size ( $\alpha$ in Eq. (2.5)).
	$yy$	is the current value of the dependent variable vector, $y(t)$ .
	$yp$	is the current value of $\dot{y}(t)$ .
	$rr$	is the current value of the residual vector $F(t, y, \dot{y})$ .

- Jac** is the output (approximate) Jacobian matrix (of type `SUNMatrix`),  $J = \partial F / \partial y + c_j \partial F / \partial \dot{y}$ .
- user\_data** is a pointer to user data, the same as the `user_data` parameter passed to `IDASSetUserData`.
- tmp1**  
**tmp2**  
**tmp3** are pointers to memory allocated for variables of type `N_Vector` which can be used by `IDALsJacFn` function as temporary storage or work space.
- Return value** An `IDALsJacFn` should return 0 if successful, a positive value if a recoverable error occurred, or a negative value if a nonrecoverable error occurred.
- In the case of a recoverable error return, the integrator will attempt to recover by reducing the stepsize, and hence changing  $\alpha$  in (2.5).
- Notes** Information regarding the structure of the specific `SUNMATRIX` structure (e.g. number of rows, upper/lower bandwidth, sparsity type) may be obtained through using the implementation-specific `SUNMATRIX` interface functions (see Chapter 7 for details).
- Prior to calling the user-supplied Jacobian function, the Jacobian matrix  $J(t, y)$  is zeroed out, so only nonzero elements need to be loaded into **Jac**.
- If the user's `IDALsJacFn` function uses difference quotient approximations, it may need to access quantities not in the call list. These quantities may include the current stepsize, the error weights, etc. To obtain these, the user will need to add a pointer to `ida_mem` to `user_data` and then use the `IDAGet*` functions described in §4.5.10.2. The unit roundoff can be accessed as `UNIT_ROUNDOFF` defined in `sundials_types.h`.

#### **dense:**

A user-supplied dense Jacobian function must load the  $\text{Neq} \times \text{Neq}$  dense matrix **Jac** with an approximation to the Jacobian matrix  $J(t, y, \dot{y})$  at the point `(tt, yy, yp)`. The accessor macros `SM_ELEMENT_D` and `SM_COLUMN_D` allow the user to read and write dense matrix elements without making explicit references to the underlying representation of the `SUNMATRIX_DENSE` type. `SM_ELEMENT_D(J, i, j)` references the  $(i, j)$ -th element of the dense matrix **Jac** (with  $i, j = 0 \dots N - 1$ ). This macro is meant for small problems for which efficiency of access is not a major concern. Thus, in terms of the indices  $m$  and  $n$  ranging from 1 to  $N$ , the Jacobian element  $J_{m,n}$  can be set using the statement `SM_ELEMENT_D(J, m-1, n-1) = Jm,n`. Alternatively, `SM_COLUMN_D(J, j)` returns a pointer to the first element of the  $j$ -th column of **Jac** (with  $j = 0 \dots N - 1$ ), and the elements of the  $j$ -th column can then be accessed using ordinary array indexing. Consequently,  $J_{m,n}$  can be loaded using the statements `col_n = SM_COLUMN_D(J, n-1); col_n[m-1] = Jm,n`. For large problems, it is more efficient to use `SM_COLUMN_D` than to use `SM_ELEMENT_D`. Note that both of these macros number rows and columns starting from 0. The `SUNMATRIX_DENSE` type and accessor macros are documented in §7.1.

#### **banded:**

A user-supplied banded Jacobian function must load the  $\text{Neq} \times \text{Neq}$  banded matrix **Jac** with an approximation to the Jacobian matrix  $J(t, y, \dot{y})$  at the point `(tt, yy, yp)`. The accessor macros `SM_ELEMENT_B`, `SM_COLUMN_B`, and `SM_COLUMN_ELEMENT_B` allow the user to read and write banded matrix elements without making specific references to the underlying representation of the `SUNMATRIX_BAND` type. `SM_ELEMENT_B(J, i, j)` references the  $(i, j)$ -th element of the banded matrix **Jac**, counting from 0. This macro is meant for use in small problems for which efficiency of access is not a major concern. Thus, in terms of the indices  $m$  and  $n$  ranging from 1 to  $N$  with  $(m, n)$  within the band defined by `mupper` and `mlower`, the Jacobian element  $J_{m,n}$  can be loaded using the statement `SM_ELEMENT_B(J, m-1, n-1) = Jm,n`. The elements within the band are those with  $-\text{mupper} \leq m-n \leq \text{mlower}$ . Alternatively, `SM_COLUMN_B(J, j)` returns a pointer to the diagonal element of the  $j$ -th column of **Jac**, and if we

assign this address to `realtype *col_j`, then the  $i$ -th element of the  $j$ -th column is given by `SM_COLUMN_ELEMENT_B(col_j, i, j)`, counting from 0. Thus, for  $(m, n)$  within the band,  $J_{m,n}$  can be loaded by setting `col_n = SM_COLUMN_B(J, n-1)`; and `SM_COLUMN_ELEMENT_B(col_n, m-1, n-1) = J_{m,n}`. The elements of the  $j$ -th column can also be accessed via ordinary array indexing, but this approach requires knowledge of the underlying storage for a band matrix of type `SUNMATRIX_BAND`. The array `col_n` can be indexed from `-mupper` to `mlower`. For large problems, it is more efficient to use `SM_COLUMN_B` and `SM_COLUMN_ELEMENT_B` than to use the `SM_ELEMENT_B` macro. As in the dense case, these macros all number rows and columns starting from 0. The `SUNMATRIX_BAND` type and accessor macros are documented in §7.2.

sparse:

A user-supplied sparse Jacobian function must load the  $\text{Neq} \times \text{Neq}$  compressed-sparse-column or compressed-sparse-row matrix `Jac` with an approximation to the Jacobian matrix  $J(t, y, \dot{y})$  at the point `(tt, yy, yp)`. Storage for `Jac` already exists on entry to this function, although the user should ensure that sufficient space is allocated in `Jac` to hold the nonzero values to be set; if the existing space is insufficient the user may reallocate the data and index arrays as needed. The amount of allocated space in a `SUNMATRIX_SPARSE` object may be accessed using the macro `SM_NNZ_S` or the routine `SUNSparseMatrix_NNZ`. The `SUNMATRIX_SPARSE` type and accessor macros are documented in §7.3.

The previous function type `IDADlsJacFn` is identical to `IDALsJacFn`, and may still be used for backward-compatibility. However, this will be deprecated in future releases, so we recommend that users transition to the new function type name soon.

#### 4.6.6 Jacobian-vector product (matrix-free linear solvers)

If a matrix-free linear solver is to be used (i.e., a NULL-valued `SUNMATRIX` was supplied to `IDASetLinearSolver`), the user may provide a function of type `IDALsJacTimesVecFn` in the following form, to compute matrix-vector products  $Jv$ . If such a function is not supplied, the default is a difference quotient approximation to these products.

## IDALsJacTimesVecFn

[illegible]

Purpose	This function computes the product $J\mathbf{v}$ of the DAE system Jacobian $J$ (or an approximation to it) and a given vector $\mathbf{v}$ , where $J$ is defined by Eq. (2.5).
---------	--

Arguments	tt	is the current value of the independent variable.
	yy	is the current value of the dependent variable vector, $y(t)$ .
	yp	is the current value of $\dot{y}(t)$ .
	rr	is the current value of the residual vector $F(t, y, \dot{y})$ .
	v	is the vector by which the Jacobian must be multiplied to the right.
	Jv	is the computed output vector.
	cj	is the scalar in the system Jacobian, proportional to the inverse of the step size ( $\alpha$ in Eq. (2.5) ).
	user_data	is a pointer to user data, the same as the <code>user_data</code> parameter passed to <code>IDASetUserData</code> .
	tmp1	

	<code>tmp2</code>	are pointers to memory allocated for variables of type <code>N_Vector</code> which can be used by <code>IDALsJacTimesVecFn</code> as temporary storage or work space.
Return value	The value returned by the Jacobian-times-vector function should be 0 if successful. A nonzero value indicates that a nonrecoverable error occurred.	
Notes	<p>This function must return a value of <math>J * v</math> that uses the <i>current</i> value of <math>J</math>, i.e. as evaluated at the current <math>(t, y, \dot{y})</math>.</p> <p>If the user's <code>IDALsJacTimesVecFn</code> function uses difference quotient approximations, it may need to access quantities not in the call list. These include the current stepsize, the error weights, etc. To obtain these, the user will need to add a pointer to <code>ida_mem</code> to <code>user_data</code> and then use the <code>IDAGet*</code> functions described in §4.5.10.2. The unit roundoff can be accessed as <code>UNIT_ROUNDOFF</code> defined in <code>sundials.types.h</code>.</p> <p>The previous function type <code>IDASpilsJacTimesVecFn</code> is identical to <code>IDALsJacTimesVecFn</code>, and may still be used for backward-compatibility. However, this will be deprecated in future releases, so we recommend that users transition to the new function type name soon.</p>	

#### 4.6.7 Jacobian-vector product setup (matrix-free linear solvers)

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

**IDALsJacTimesSetupFn**

Definition	<pre>typedef int (*IDALsJacTimesSetupFn)(realtype tt, N_Vector yy,                                      N_Vector yp, N_Vector rr,                                      realtype cj, void *user_data);</pre>	
Purpose	This function preprocesses and/or evaluates Jacobian data needed by the Jacobian-times-vector routine.	
Arguments	<code>tt</code>	is the current value of the independent variable.
	<code>yy</code>	is the current value of the dependent variable vector, $y(t)$ .
	<code>yp</code>	is the current value of $\dot{y}(t)$ .
	<code>rr</code>	is the current value of the residual vector $F(t, y, \dot{y})$ .
	<code>cj</code>	is the scalar in the system Jacobian, proportional to the inverse of the step size ( $\alpha$ in Eq. (2.5)).
	<code>user_data</code>	is a pointer to user data, the same as the <code>user_data</code> parameter passed to <code>IDASetUserData</code> .
Return value	The value returned by the Jacobian-vector setup function should be 0 if successful, positive for a recoverable error (in which case the step will be retried), or negative for an unrecoverable error (in which case the integration is halted).	
Notes	<p>Each call to the Jacobian-vector setup function is preceded by a call to the <code>IDAResFn</code> user function with the same <math>(t, y, yp)</math> arguments. Thus, the setup function can use any auxiliary data that is computed and saved during the evaluation of the DAE residual.</p> <p>If the user's <code>IDALsJacTimesVecFn</code> function uses difference quotient approximations, it may need to access quantities not in the call list. These include the current stepsize, the error weights, etc. To obtain these, the user will need to add a pointer to <code>ida_mem</code> to <code>user_data</code> and then use the <code>IDAGet*</code> functions described in §4.5.10.2. The unit roundoff can be accessed as <code>UNIT_ROUNDOFF</code> defined in <code>sundials.types.h</code>.</p> <p>The previous function type <code>IDASpilsJacTimesSetupFn</code> is identical to <code>IDALsJacTimesSetupFn</code>, and may still be used for backward-compatibility. However, this will be deprecated in</p>	





Purpose	This function evaluates and/or preprocesses Jacobian-related data needed by the preconditioner.	
Arguments	<b>tt</b>	is the current value of the independent variable.
	<b>yy</b>	is the current value of the dependent variable vector, $y(t)$ .
	<b>yp</b>	is the current value of $\dot{y}(t)$ .
	<b>rr</b>	is the current value of the residual vector $F(t, y, \dot{y})$ .
	<b>cj</b>	is the scalar in the system Jacobian, proportional to the inverse of the step size ( $\alpha$ in Eq. (2.5)).
	<b>user_data</b>	is a pointer to user data, the same as the <b>user_data</b> parameter passed to the function <b>IDASetUserData</b> .
Return value	The value 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 (in which case the step will be retried), negative for an unrecoverable error (in which case the integration is halted).	
Notes	The operations performed by this function might include forming a crude approximate Jacobian, and performing an LU factorization on the resulting approximation.	
	Each call to the preconditioner setup function is preceded by a call to the <b>IDAResFn</b> user function with the same ( <b>tt</b> , <b>yy</b> , <b>yp</b> ) arguments. Thus the preconditioner setup function can use any auxiliary data that is computed and saved during the evaluation of the DAE residual.	
	This function is not called in advance of every call to the preconditioner solve function, but rather is called only as often as needed to achieve convergence in the nonlinear solver.	
	If the user's <b>IDALsPrecSetupFn</b> function uses difference quotient approximations, it may need to access quantities not in the call list. These include the current stepsize, the error weights, etc. To obtain these, the user will need to add a pointer to <b>ida_mem</b> to <b>user_data</b> and then use the <b>IDAGet*</b> functions described in §4.5.10.2. The unit roundoff can be accessed as <b>UNIT_ROUNDOFF</b> defined in <b>sundials.types.h</b> .	
	The previous function type <b>IDASpilsPrecSetupFn</b> is identical to <b>IDALsPrecSetupFn</b> , and may still be used for backward-compatibility. However, this will be deprecated in future releases, so we recommend that users transition to the new function type name soon.	

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

A principal reason for using a parallel DAE solver such as IDA lies in the solution of partial differential equations (PDEs). Moreover, the use of a Krylov iterative method for the solution of many such problems is motivated by the nature of the underlying linear system of equations (2.4) that must be solved at each time step. The linear algebraic system is large, sparse, and structured. However, if a Krylov iterative method is to be effective in this setting, then a nontrivial preconditioner needs to be used. Otherwise, the rate of convergence of the Krylov iterative method is usually unacceptably slow. Unfortunately, an effective preconditioner tends to be problem-specific.

However, we have developed one type of preconditioner that treats a rather broad class of PDE-based problems. It has been successfully used for several realistic, large-scale problems [27] and is included in a software module within the IDA package. This module works with the parallel vector module **NVECTOR\_PARALLEL** and generates a preconditioner that is a block-diagonal matrix with each block being a band matrix. The blocks need not have the same number of super- and sub-diagonals, and these numbers may vary from block to block. This Band-Block-Diagonal Preconditioner module is called **IDABBDPRE**.

One way to envision these preconditioners is to think of the domain of the computational PDE problem as being subdivided into  $M$  non-overlapping sub-domains. Each of these sub-domains is then

$$G(t, y, \dot{y}) = [G_1(t, \bar{y}_1, \dot{\bar{y}}_1), G_2(t, \bar{y}_2, \dot{\bar{y}}_2), \dots, G_M(t, \bar{y}_M, \dot{\bar{y}}_M)]^T, \quad (4.1)$$

The preconditioner associated with this decomposition has the form

$$P = \text{diag}[P_1, P_2, \dots, P_M] \quad (4.2)$$

$$P_m \approx \partial G_m / \partial y_m + \alpha \partial G_m / \partial \dot{y}_m \quad (4.3)$$

The solution of the complete linear system

$$Px = b \tag{4.4}$$

$$P_m x_m = b_m \quad (4.5)$$

The IDABBDPRE module calls two user-provided functions to construct  $P$ : a required function **Gres** (of type **IDABBDLocalFn**) which approximates the residual function  $G(t, y, \dot{y}) \approx F(t, y, \dot{y})$  and which is computed locally, and an optional function **Gcomm** (of type **IDABBDCommFn**) which performs all inter-process communication necessary to evaluate the approximate residual  $G$ . These are in addition to the user-supplied residual function **res**. Both functions take as input the same pointer **user\_data** as passed by the user to **IDASetUserData** and passed to the user's function **res**. The user is responsible for providing space (presumably within **user\_data**) for components of **yy** and **yp** that are communicated by **Gcomm** from the other processors, and that are then used by **Gres**, which should not do any communication.

[illegible]

Purpose	This <b>Gres</b> function computes $G(t, y, \dot{y})$ . It loads the vector <b>gval</b> as a function of <b>tt</b> , <b>yy</b> , and <b>yp</b> .
Arguments	<b>Nlocal</b> is the local vector length. <b>tt</b> is the value of the independent variable. <b>yy</b> is the dependent variable. <b>yp</b> is the derivative of the dependent variable. <b>gval</b> is the output vector. <b>user_data</b> is a pointer to user data, the same as the <b>user_data</b> parameter passed to <b>IDASetUserData</b> .
Return value	An <b>IDABBDLocalFn</b> function type should return 0 to indicate success, 1 for a recoverable error, or -1 for a non-recoverable error.
Notes	This function must assume that all inter-processor communication of data needed to calculate <b>gval</b> has already been done, and this data is accessible within <b>user_data</b> . The case where $G$ is mathematically identical to $F$ is allowed.

IDABBDCommFn

Definition	typedef int (*IDABBDCommFn)(sunindextype Nlocal, realtype tt, N_Vector yy, N_Vector yp, void *user_data);
Purpose	This <b>Gcomm</b> function performs all inter-processor communications necessary for the execution of the <b>Gres</b> function above, using the input vectors <b>yy</b> and <b>yp</b> .
Arguments	<b>Nlocal</b> is the local vector length. <b>tt</b> is the value of the independent variable. <b>yy</b> is the dependent variable. <b>yp</b> is the derivative of the dependent variable. <b>user_data</b> is a pointer to user data, the same as the <b>user_data</b> parameter passed to <b>IDASetUserData</b> .
Return value	An <b>IDABBDCommFn</b> function type should return 0 to indicate success, 1 for a recoverable error, or -1 for a non-recoverable error.
Notes	The <b>Gcomm</b> function is expected to save communicated data in space defined within the structure <b>user_data</b> . Each call to the <b>Gcomm</b> function is preceded by a call to the residual function <b>res</b> with the same ( <b>tt</b> , <b>yy</b> , <b>yp</b> ) arguments. Thus <b>Gcomm</b> can omit any communications done by <b>res</b> if relevant to the evaluation of <b>Gres</b> . If all necessary communication was done in <b>res</b> , then <b>Gcomm</b> = NULL can be passed in the call to <b>IDABBDPrecInit</b> (see below).

Besides the header files required for the integration of the DAE problem (see §4.3), to use the **IDABBDPRE** module, the main program must include the header file **ida\_bbdpre.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 §4.4 are grayed-out.

1. Initialize MPI
2. Set problem dimensions etc.
3. Set vectors of initial values
4. Create IDA object
5. Initialize IDA solver

## 6. Specify integration tolerances

## 7. Create linear solver object

When creating the iterative linear solver object, specify the use of left preconditioning (`PREC_LEFT`) as IDA only supports left preconditioning.

## 8. Set linear solver optional inputs

## 9. Attach linear solver module

## 10. Set optional inputs

Note that the user should not overwrite the preconditioner setup function or solve function through calls to `idIDASetPreconditioner` optional input function.

11. Initialize the `IDABBDPRE` preconditioner module

Specify the upper and lower bandwidths `mudq`, `mldq` and `mukeep`, `mlkeep` and call

```
flag = IDABBDPrecInit(ida_mem, Nlocal, mudq, mldq,
                      mukeep, mlkeep, dq_rel_yy, Gres, Gcomm);
```

to allocate memory and initialize the internal preconditioner data. The last two arguments of `IDABBDPrecInit` are the two user-supplied functions described above.

## 12. Create nonlinear solver object

## 13. Attach nonlinear solver module

## 14. Set nonlinear solver optional inputs

## 15. Correct initial values

## 16. Specify rootfinding problem

## 17. Advance solution in time

## 18. Get optional outputs

Additional optional outputs associated with `IDABBDPRE` are available by way of two routines described below, `IDABBDPrecGetWorkSpace` and `IDABBDPrecGetNumGfnEvals`.

## 19. Deallocate memory for solution vectors

## 20. Free solver memory

## 21. Free nonlinear solver memory

## 22. Free linear solver memory

## 23. Finalize MPI

The user-callable functions that initialize (step 11 above) or re-initialize the `IDABBDPRE` preconditioner module are described next.

<code>IDABBDPrecInit</code>
-----------------------------

Call            `flag = IDABBDPrecInit(ida_mem, Nlocal, mudq, mldq, mukeep, mlkeep, dq_rel_yy, Gres, Gcomm);`

Description    The function `IDABBDPrecInit` initializes and allocates (internal) memory for the `IDABBDPRE` preconditioner.

Arguments	<b>ida_mem</b>	(void *) pointer to the IDA memory block.
	<b>Nlocal</b>	(sunindextype) local vector dimension.
	<b>mudq</b>	(sunindextype) upper half-bandwidth to be used in the difference-quotient Jacobian approximation.
	<b>mldq</b>	(sunindextype) lower half-bandwidth to be used in the difference-quotient Jacobian approximation.
	<b>mukeep</b>	(sunindextype) upper half-bandwidth of the retained banded approximate Jacobian block.
	<b>mlkeep</b>	(sunindextype) lower half-bandwidth of the retained banded approximate Jacobian block.
	<b>dq_rel_yy</b>	(realtype) the relative increment in components of <b>y</b> used in the difference quotient approximations. The default is $\text{dq\_rel\_yy} = \sqrt{\text{unit roundoff}}$ , which can be specified by passing <b>dq_rel_yy</b> = 0.0.
	<b>Gres</b>	(IDABBDLocalFn) the C function which computes the local residual approximation $G(t, y, \dot{y})$ .
	<b>Gcomm</b>	(IDABBDCommFn) the optional C function which performs all inter-process communication required for the computation of $G(t, y, \dot{y})$ .

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

<b>IDALS_SUCCESS</b>	The call to <b>IDABBDPrecInit</b> was successful.
<b>IDALS_MEM_NULL</b>	The <b>ida_mem</b> pointer was <b>NULL</b> .
<b>IDALS_MEM_FAIL</b>	A memory allocation request has failed.
<b>IDALS_LMEM_NULL</b>	An IDALS linear solver memory was not attached.
<b>IDALS_ILL_INPUT</b>	The supplied vector implementation was not compatible with the block band preconditioner.

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

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

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

For all four half-bandwidths, the values need not be the same on every processor.

The IDABBDPRE module also provides a reinitialization function to allow for a sequence of problems of the same size, with the same linear solver choice, provided there is no change in **local\_N**, **mukeep**, or **mlkeep**. After solving one problem, and after calling **IDAReInit** to re-initialize IDA for a subsequent problem, a call to **IDABBDPrecReInit** can be made to change any of the following: the half-bandwidths **mudq** and **mldq** used in the difference-quotient Jacobian approximations, the relative increment **dq\_rel\_yy**, or one of the user-supplied functions **Gres** and **Gcomm**. If there is a change in any of the linear solver inputs, an additional call to the “Set” routines provided by the SUNLINSOL module, and/or one or more of the corresponding **IDASet\*\*\*** functions, must also be made (in the proper order).

#### **IDABBDPrecReInit**

Call **flag** = **IDABBDPrecReInit**(**ida\_mem**, **mudq**, **mldq**, **dq\_rel\_yy**);

Description The function **IDABBDPrecReInit** reinitializes the IDABBDPRE preconditioner.

Arguments	<b>ida_mem</b>	(void *) pointer to the IDA memory block.
	<b>mudq</b>	(sunindextype) upper half-bandwidth to be used in the difference-quotient Jacobian approximation.

**mldq** (sunindextype) lower half-bandwidth to be used in the difference-quotient Jacobian approximation.

**dq\_rel\_yy** (realtype) the relative increment in components of **y** used in the difference quotient approximations. The default is  $\text{dq\_rel\_yy} = \sqrt{\text{unit roundoff}}$ , which can be specified by passing  $\text{dq\_rel\_yy} = 0.0$ .

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

**IDALS\_SUCCESS** The call to **IDABBDPrecReInit** was successful.

**IDALS\_MEM\_NULL** The **ida\_mem** pointer was **NULL**.

**IDALS\_LMEM\_NULL** An **IDALS** linear solver memory was not attached.

**IDALS\_PMEM\_NULL** The function **IDABBDPrecInit** was not previously called.

**Notes** If one of the half-bandwidths **mudq** or **mldq** is negative or exceeds the value **Nlocal**−1, it is replaced by 0 or **Nlocal**−1, accordingly.

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

#### **IDABBDPrecGetWorkSpace**

**Call** **flag** = **IDABBDPrecGetWorkSpace**(**ida\_mem**, &**lenrwBBDP**, &**leniwBBDP**);

**Description** The function **IDABBDPrecGetWorkSpace** returns the local sizes of the **IDABBDPRE** real and integer workspaces.

**Arguments** **ida\_mem** (void \*) pointer to the IDA memory block.

**lenrwBBDP** (long int) local number of real values in the **IDABBDPRE** workspace.

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

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

**IDALS\_SUCCESS** The optional output value has been successfully set.

**IDALS\_MEM\_NULL** The **ida\_mem** pointer was **NULL**.

**IDALS\_PMEM\_NULL** The **IDABBDPRE** preconditioner has not been initialized.

**Notes** The workspace requirements reported by this routine correspond only to memory allocated within the **IDABBDPRE** module (the banded matrix approximation, banded **SUNLINSOL** object, temporary vectors). These values are local to each process.

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

#### **IDABBDPrecGetNumGfnEvals**

**Call** **flag** = **IDABBDPrecGetNumGfnEvals**(**ida\_mem**, &**ngevalsBBDP**);

**Description** The function **IDABBDPrecGetNumGfnEvals** returns the cumulative number of calls to the user **Gres** function due to the finite difference approximation of the Jacobian blocks used within **IDABBDPRE**'s preconditioner setup function.

**Arguments** **ida\_mem** (void \*) pointer to the IDA memory block.

**ngevalsBBDP** (long int) the cumulative number of calls to the user **Gres** function.

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

**IDALS\_SUCCESS** The optional output value has been successfully set.

**IDALS\_MEM\_NULL** The **ida\_mem** pointer was **NULL**.

**IDALS\_PMEM\_NULL** The **IDABBDPRE** preconditioner has not been initialized.

In addition to the **ngevalsBBDP** **Gres** evaluations, the costs associated with **IDABBDPRE** also include **nlinsetups** LU factorizations, **nlinsetups** calls to **Gcomm**, **npsolves** banded backsolve calls, and **nrevalsLS** residual function evaluations, where **nlinsetups** is an optional IDA output (see §4.5.10.2), and **npsolves** and **nrevalsLS** are linear solver optional outputs (see §4.5.10.5).





## Chapter 5

# FIDA, an Interface Module for FORTRAN Applications

The FIDA interface module is a package of C functions which support the use of the IDA solver, for the solution of DAE systems, in a mixed FORTRAN/C setting. While IDA 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 IDA for all supplied serial and parallel NVECTOR implementations.

### 5.1 Important note on portability

In this package, the names of the interface functions, and the names of the FORTRAN user routines called by them, appear as dummy names which are mapped to actual values by a series of definitions in the header files. By default, those mapping definitions depend in turn on the C macro `F77_FUNC` defined in the header file `sundials_config.h`. The mapping defined by `F77_FUNC` in turn transforms the C interface names to match the name-mangling approach used by the supplied Fortran compiler.

By “name-mangling”, we mean that due to the case-independent nature of the FORTRAN language, FORTRAN compilers convert all subroutine and object names to use either all lower-case or all upper-case characters, and append either zero, one or two underscores as a prefix or suffix to the name. For example, the FORTRAN subroutine `MyFunction()` will be changed to one of `myfunction`, `MYFUNCTION`, `myfunction_`, `MYFUNCTION_`, and so on, depending on the FORTRAN compiler used.

SUNDIALS determines this name-mangling scheme at configuration time (see [Appendix A](#)).

### 5.2 Fortran Data Types

Throughout this documentation, we will refer to data types according to their usage in C. The equivalent types to these may vary, depending on your computer architecture and on how SUNDIALS was compiled (see [Appendix A](#)). A FORTRAN user should first determine the equivalent types for their architecture and compiler, and then take care that all arguments passed through this FORTRAN/C interface are declared of the appropriate type.

**Integers:** While SUNDIALS uses the configurable `sunindextype` type as the integer type for vector and matrix indices for its C code, the FORTRAN interfaces are more restricted. The `sunindextype` is only used for index values and pointers when filling sparse matrices. As for C, the `sunindextype` can be configured to be a 32- or 64-bit signed integer by setting the variable `SUNDIALS_INDEX_TYPE` at compile time (See [Appendix A](#)). The default value is `int64_t`. A FORTRAN user should set this variable based on the integer type used for vector and matrix indices in their FORTRAN code. The corresponding FORTRAN types are:

- `int32_t` – equivalent to an `INTEGER` or `INTEGER*4` in FORTRAN

- `int64_t` – equivalent to an `INTEGER*8` in FORTRAN

In general, for the FORTRAN interfaces in SUNDIALS, flags of type `int`, vector and matrix lengths, counters, and arguments to `*SETIN()` functions all have `long int` type, and `sunindextype` is only used for index values and pointers when filling sparse matrices. Note that if an F90 (or higher) user wants to find out the value of `sunindextype`, they can include `sundials_fconfig.h`.

**Real numbers:** As discussed in Appendix A, at compilation SUNDIALS allows the configuration option `SUNDIALS_PRECISION`, that accepts values of `single`, `double` or `extended` (the default is `double`). This choice dictates the size of a `realtype` variable. The corresponding FORTRAN types for these `realtype` sizes are:

- `single` – equivalent to a `REAL` or `REAL*4` in FORTRAN
- `double` – equivalent to a `DOUBLE PRECISION` or `REAL*8` in FORTRAN
- `extended` – equivalent to a `REAL*16` in FORTRAN

### 5.3 FIDA routines

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

- Interface to the NVECTOR modules
  - `FNVINITS` (defined by `NVECTOR_SERIAL`) interfaces to `N_VNewEmpty_Serial`.
  - `FNVINITP` (defined by `NVECTOR_PARALLEL`) interfaces to `N_VNewEmpty_Parallel`.
  - `FNVINITOMP` (defined by `NVECTOR_OPENMP`) interfaces to `N_VNewEmpty_OpenMP`.
  - `FNVINITPTS` (defined by `NVECTOR_PTHREADS`) interfaces to `N_VNewEmpty_Pthreads`.
- Interface to the SUNMATRIX modules
  - `FSUNBANDMATINIT` (defined by `SUNMATRIX_BAND`) interfaces to `SUNBandMatrix`.
  - `FSUNDENSEMATINIT` (defined by `SUNMATRIX_DENSE`) interfaces to `SUNDenseMatrix`.
  - `FSUNSPARSEMATINIT` (defined by `SUNMATRIX_SPARSE`) interfaces to `SUNSparseMatrix`.
- Interface to the SUNLINSOL modules
  - `FSUNBANDLINSOLINIT` (defined by `SUNLINSOL_BAND`) interfaces to `SUNLinSol_Band`.
  - `FSUNDENSELINSOLINIT` (defined by `SUNLINSOL_DENSE`) interfaces to `SUNLinSol_Dense`.
  - `FSUNKLUINIT` (defined by `SUNLINSOL_KLU`) interfaces to `SUNLinSol_KLU`.
  - `FSUNKLUREINIT` (defined by `SUNLINSOL_KLU`) interfaces to `SUNLinSol_KLUReinit`.
  - `FSUNLAPACKBANDINIT` (defined by `SUNLINSOL_LAPACKBAND`) interfaces to `SUNLinSol_LapackBand`.
  - `FSUNLAPACKDENSEINIT` (defined by `SUNLINSOL_LAPACKDENSE`) interfaces to `SUNLinSol_LapackDense`.
  - `FSUNPCGINIT` (defined by `SUNLINSOL_PCG`) interfaces to `SUNLinSol_PCG`.
  - `FSUNSPBCGSINIT` (defined by `SUNLINSOL_SPBCGS`) interfaces to `SUNLinSol_SPBCGS`.
  - `FSUNSPFGMRINIT` (defined by `SUNLINSOL_SPFGMR`) interfaces to `SUNLinSol_SPFGMR`.
  - `FSUNSPGMRINIT` (defined by `SUNLINSOL_SPGMR`) interfaces to `SUNLinSol_SPGMR`.
  - `FSUNSPTFQMRINIT` (defined by `SUNLINSOL_SPTFQMR`) interfaces to `SUNLinSol_SPTFQMR`.
  - `FSUNSUPERLUMTINIT` (defined by `SUNLINSOL_SUPERLUMT`) interfaces to `SUNLinSol_SuperLUMT`.

- 
- Interface to the main IDA module
  - FIDAMALLOC interfaces to IDACreate, IDASetUserData, IDAInit, IDASStolerances, and IDASVtolerances.
  - FIDAREINIT interfaces to IDAReInit and IDASStolerances/IDASVtolerances.
  - FIDASETIIN, FIDASETVIN, and FIDASETRIN interface to IDASet\* functions.
  - FIDATOLREINIT interfaces to IDASStolerances/IDASVtolerances.
  - FIDACALCIC interfaces to IDACalcIC.
  - FIDAEWTSET interfaces to IDAWFtolerances.
  - FIDASOLVE interfaces to IDASolve, IDAGet\* functions, and to the optional output functions for the selected linear solver module.
  - FIDAGETDKY interfaces to IDAGetDky.
  - FIDAGETERRWEIGHTS interfaces to IDAGetErrWeights.
  - FIDAGETESTLOCALERR interfaces to IDAGetEstLocalErrors.
  - FIDAFREE interfaces to IDAFree.
- Interface to the IDALS module
  - FIDALSINIT interfaces to IDASetLinearSolver.
  - FIDALSSETEPSLIN interfaces to IDASetEpsLin
  - FIDALSSETJAC interfaces to IDASetJacTimes.
  - FIDALSSETPREC interfaces to IDASetPreconditioner.
  - FIDADENSESETJAC interfaces to IDASetJacFn.
  - FIDABANDSETJAC interfaces to IDASetJacFn.
  - FIDASPARSESETJAC interfaces to IDASetJacFn.

The user-supplied functions, each listed with the corresponding internal interface function which calls it (and its type within IDA), are as follows:

FIDA routine (FORTRAN, user-supplied)	IDA function (C, interface)	IDA type of interface function
FIDARESFUN	FIDaresfn	IDAResFn
FIDAEWT	FIDAEwtSet	IDAEwtFn
FIDADJAC	FIDADenseJac	IDALsJacFn
FIDABJAC	FIDABandJac	IDALsJacFn
FIDASPJAC	FIDASparseJac	IDALsJacFn
FIDAPSOL	FIDAPSol	IDALsPrecSolveFn
FIDAPSET	FIDAPSet	IDALsPrecSetupFn
FIDAJTIMES	FIDAJtimes	IDALsJacTimesVecFn
FIDAJTSETUP	FIDAJTSetup	IDALsJacTimesSetupFn

In contrast to the case of direct use of IDA, and of most FORTRAN DAE solvers, the names of all user-supplied routines here are fixed, in order to maximize portability for the resulting mixed-language program.

## 5.4 Usage of the FIDA interface module

The usage of FIDA requires calls to a variety of 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 IDA 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 FIDA for rootfinding, and usage of FIDA with preconditioner modules, are each described in later sections.

### 1. Residual function specification

The user must, in all cases, supply the following FORTRAN routine

```
SUBROUTINE FIDARESFUN (T, Y, YP, R, IPAR, RPAR, IER)
  DIMENSION Y(*), YP(*), R(*), IPAR(*), RPAR(*)
```

It must set the R array to  $F(t, y, \dot{y})$ , the residual function of the DAE system, as a function of  $T = t$  and the arrays  $Y = y$  and  $YP = \dot{y}$ . The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FIDAMALLOC. It should return  $IER = 0$  if it was successful,  $IER = 1$  if it had a recoverable failure, or  $IER = -1$  if it had a non-recoverable failure.

### 2. NVECTOR module initialization

If using one of the NVECTOR modules supplied with SUNDIALS, the user must make a call of the form

```
CALL FNVINIT***(...)
```

in which the name and call sequence are as described in the appropriate section of Chapter 6.

### 3. SUNMATRIX module initialization

In the case of a stiff system, the implicit BDF method involves the solution of linear systems related to the Jacobian of the DAE system. If using a Newton iteration with the direct SUNLINSOL linear solver module and one of the SUNMATRIX modules supplied with SUNDIALS, the user must make a call of the form

```
CALL FSUN***MATINIT(...)
```

in which the name and call sequence are as described in the appropriate section of Chapter 7. Note that the dense, band, or sparse matrix options are usable only in a serial or multi-threaded environment.

### 4. SUNLINSOL module initialization

If using a Newton iteration with one of the SUNLINSOL linear solver modules supplied with SUNDIALS, the user must make a call of the form

```
CALL FSUNBANDLINSOLINIT(...)
CALL FSUNDENSELINSOLINIT(...)
CALL FSUNKLUINIT(...)
CALL FSUNLAPACKBANDINIT(...)
CALL FSUNLAPACKDENSEINIT(...)
CALL FSUNPCGINIT(...)
CALL FSUNSPBCGSINIT(...)
CALL FSUNSPFGMRINIT(...)
```

```
CALL FSUNSPGMRINIT(...)
CALL FSUNSPTFQMRINIT(...)
CALL FSUNSUPERLUMTINIT(...)
```

in which the call sequence is as described in the appropriate section of Chapter 8. Note that the dense, band, or sparse solvers are usable only in a serial or multi-threaded environment.

Once one of these solvers has been initialized, its solver parameters may be modified using a call to the functions

```
CALL FSUNKLUSETORDERING(...)
CALL FSUNSUPERLUMTSETORDERING(...)
CALL FSUNPCGSETPRECTYPE(...)
CALL FSUNPCGSETMAXL(...)
CALL FSUNSPBCGSSETPRECTYPE(...)
CALL FSUNSPBCGSSETMAXL(...)
CALL FSUNSPFGMRSETGSTYPE(...)
CALL FSUNSPFGMRSETPRECTYPE(...)
CALL FSUNSPGMRSETGSTYPE(...)
CALL FSUNSPGMRSETPRECTYPE(...)
CALL FSUNSPTFQMRSETPRECTYPE(...)
CALL FSUNSPTFQMRSETMAXL(...)
```

where again the call sequences are described in the appropriate sections of Chapter 8.

## 5. Problem specification

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

### **FIDAMALLOC**

Call	CALL FIDAMALLOC(TO, YO, YPO, IATOL, RTOL, ATOL, & IOUT, ROUT, IPAR, RPAR, IER)
Description	This function provides required problem and solution specifications, specifies optional inputs, allocates internal memory, and initializes IDA.
Arguments	<p>TO is the initial value of <math>t</math>.</p> <p>YO is an array of initial conditions for <math>y</math>.</p> <p>YPO is an array of initial conditions for <math>\dot{y}</math>.</p> <p>IATOL specifies the type for absolute tolerance ATOL: 1 for scalar or 2 for array. If IATOL= 3, the arguments RTOL and ATOL are ignored and the user is expected to subsequently call FIDAEWTSET and provide the function FIDAEWT.</p> <p>RTOL is the relative tolerance (scalar).</p> <p>ATOL is the absolute tolerance (scalar or array).</p> <p>IOUT is an integer array of length at least 21 for integer optional outputs.</p> <p>ROUT is a real array of length at least 6 for real optional outputs.</p> <p>IPAR is an integer array of user data which will be passed unmodified to all user-provided routines.</p> <p>RPAR is a real array of user data which will be passed unmodified to all user-provided routines.</p>
Return value	IER is a return completion flag. Values are 0 for successful return and -1 otherwise. See printed message for details in case of failure.
Notes	The user integer data arrays IOUT and IPAR must be declared as INTEGER*4 or INTEGER*8 according to the C type long int.

Modifications to the user data arrays `IPAR` and `RPAR` inside a user-provided routine will be propagated to all subsequent calls to such routines.

The optional outputs associated with the main IDA integrator are listed in Table 5.2.

As an alternative to providing tolerances in the call to `FIDAMALLOC`, the user may provide a routine to compute the error weights used in the WRMS norm evaluations. If supplied, it must have the following form:

```
SUBROUTINE FIDAEWT (Y, EWT, IPAR, RPAR, IER)
  DIMENSION Y(*), EWT(*), IPAR(*), RPAR(*)
```

It must set the positive components of the error weight vector `EWT` for the calculation of the WRMS norm of `Y`. On return, set `IER` = 0 if `FIDAEWT` was successful, and nonzero otherwise. The arrays `IPAR` (of integers) and `RPAR` (of reals) contain user data and are the same as those passed to `FIDAMALLOC`.

If the `FIDAEWT` routine is provided, then, following the call to `FIDAMALLOC`, the user must make the call:

```
CALL FIDAEWTSET (FLAG, IER)
```

with `FLAG`  $\neq$  0 to specify use of the user-supplied error weight routine. The argument `IER` is an error return flag, which is 0 for success or non-zero if an error occurred.

## 6. Set optional inputs

Call `FIDASETIIN`, `FIDASETRIN`, and/or `FIDASETVIN` to set desired optional inputs, if any. See §5.5 for details.

## 7. Linear solver interface specification

The variable-order, variable-coefficient BDF method used by IDA involves the solution of linear systems related to the system Jacobian  $J = \partial F / \partial y + \alpha \partial F / \partial \dot{y}$ . See Eq. (2.4). To attach the linear solver (and optionally the matrix) objects initialized in steps 3 and 4 above, the user of FIDA must initialize the IDALS linear solver interface. To attach any `SUNLINSOL` object (and optional `SUNMATRIX` object) to IDA, then following calls to initialize the `SUNLINSOL` (and `SUNMATRIX`) object(s) in steps 3 and 4 above, the user must make the call:

```
CALL FIDALSINIT(IER)
```

`IER` is an error return flag set on 0 on success or  $-1$  if a memory failure occurred.

The previous routines `FIDADLSINIT` and `FIDASPILSINIT` are now wrappers for this routine, and may still be used for backward-compatibility. However, these will be deprecated in future releases, so we recommend that users transition to the new routine name soon.

### IDALS with dense Jacobian matrix

As an option when using the IDALS interface with the `SUNLINSOL_DENSE` or `SUNLINSOL_LAPACKDENSE` linear solvers, the user may supply a routine that computes a dense approximation of the system Jacobian  $J$ . If supplied, it must have the following form:

```
SUBROUTINE FIDADJAC (NEQ, T, Y, YP, R, DJAC, CJ, EWT, H,
&                    IPAR, RPAR, WK1, WK2, WK3, IER)
  DIMENSION Y(*), YP(*), R(*), EWT(*), DJAC(NEQ,*),
&                    IPAR(*), RPAR(*), WK1(*), WK2(*), WK3(*)
```

This routine must compute the Jacobian and store it columnwise in `DJAC`. The vectors `WK1`, `WK2`, and `WK3` of length `NEQ` are provided as work space for use in `FIDADJAC`. The input arguments `T`, `Y`,

YP, R, and CJ are the current values of  $t$ ,  $y$ ,  $\dot{y}$ ,  $F(t, y, \dot{y})$ , and  $\alpha$ , respectively. The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FIDAMALLOC. NOTE: The argument NEQ has a type consistent with C type `long int` even in the case when the LAPACK dense solver is to be used.

If the user's FIDADJAC uses difference quotient approximations, it may need to use the error weight array EWT and current stepsize H in the calculation of suitable increments. It may also need the unit roundoff, which can be obtained as the optional output ROUT(6), passed from the calling program to this routine using COMMON.

If the FIDADJAC routine is provided, then, following the call to FIDALSINIT the user must make the call:

```
CALL FIDADENSESETJAC (FLAG, IER)
```

with `FLAG`  $\neq 0$  to specify use of the user-supplied Jacobian approximation. The argument `IER` is an error return flag, which is 0 for success or non-zero if an error occurred.

#### IDLALS with band Jacobian matrix

As an option when using the IDALS interface with the SUNLINSOL\_BAND or SUNLINSOL\_LAPACKBAND linear solvers, the user may supply a routine that computes a band approximation of the system Jacobian  $J$ . If supplied, it must have the following form:

```
SUBROUTINE FIDABJAC(NEQ, MU, ML, MDIM, T, Y, YP, R, CJ, BJAC,
&                  EWT, H, IPAR, RPAR, WK1, WK2, WK3, IER)
  DIMENSION Y(*), YP(*), R(*), EWT(*), BJAC(MDIM,*),
&          IPAR(*), RPAR(*), WK1(*), WK2(*), WK3(*)
```

This routine must load the MDIM by NEQ array BJAC with the Jacobian matrix at the current  $(t, y, \dot{y})$  in band form. Store in  $BJAC(k, j)$  the Jacobian element  $J_{i,j}$  with  $k = i - j + MU + 1$  ( $k = 1 \cdots ML + MU + 1$ ) and  $j = 1 \cdots N$ . The vectors WK1, WK2, and WK3 of length NEQ are provided as work space for use in FIDABJAC. The input arguments T, Y, YP, R, and CJ are the current values of  $t$ ,  $y$ ,  $\dot{y}$ ,  $F(t, y, \dot{y})$ , and  $\alpha$ , respectively. The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FIDAMALLOC. NOTE: The arguments NEQ, MU, ML, and MDIM have a type consistent with C type `long int` even in the case when the LAPACK band solver is to be used.

If the user's FIDABJAC uses difference quotient approximations, it may need to use the error weight array EWT and current stepsize H in the calculation of suitable increments. It may also need the unit roundoff, which can be obtained as the optional output ROUT(6), passed from the calling program to this routine using COMMON.

If the FIDABJAC routine is provided, then, following the call to FIDALSINIT, the user must make the call:

```
CALL FIDABANDSETJAC (FLAG, IER)
```

with `FLAG`  $\neq 0$  to specify use of the user-supplied Jacobian approximation. The argument `IER` is an error return flag, which is 0 for success or non-zero if an error occurred.

#### IDLALS with sparse Jacobian matrix

When using the IDALS interface with the SUNLINSOL\_KLU or SUNLINSOL\_SUPERLUMT linear solvers, the user must supply the FIDASPJAC routine that computes a compressed-sparse-column (CSC) or compressed-sparse-row (CSR) approximation of the system Jacobian  $J = \partial F / \partial y + c_j \partial F / \partial \dot{y}$ . If supplied, it must have the following form:

```
SUBROUTINE FIDASPJAC(T, CJ, Y, YP, R, N, NNZ, JDATA, JINDEXVALS,
&                  JINDEXPTRS, H, IPAR, RPAR, WK1, WK2, WK3, IER)
```

It must load the  $N$  by  $N$  compressed sparse column [or compressed sparse row] matrix with storage for  $NNZ$  nonzeros, stored in the arrays `JDATA` (nonzero values), `JINDEXVALS` (row [or column] indices for each nonzero), `JINDEXPTRS` (indices for start of each column [or row]), with the Jacobian matrix at the current  $(t, y)$  in CSC [or CSR] form (see `summatrix.sparse.h` for more information). The arguments are `T`, the current time; `CJ`, scalar in the system proportional to the inverse step size; `Y`, an array containing state variables; `YP`, an array containing state derivatives; `R`, an array containing the system nonlinear residual; `N`, the number of matrix rows/columns in the Jacobian; `NNZ`, allocated length of nonzero storage; `JDATA`, nonzero values in the Jacobian (of length `NNZ`); `JINDEXVALS`, row [or column] indices for each nonzero in Jacobian (of length `NNZ`); `JINDEXPTRS`, pointers to each Jacobian column [or row] in the two preceding arrays (of length  $N+1$ ); `H`, the current step size; `IPAR`, an array containing integer user data that was passed to `FIDAMALLOC`; `RPAR`, an array containing real user data that was passed to `FIDAMALLOC`; `WK*`, work arrays containing temporary workspace of same size as `Y`; and `IER`, error return code (0 if successful,  $> 0$  if a recoverable error occurred, or  $< 0$  if an unrecoverable error occurred.)

To indicate that the `FIDASPJAC` routine has been provided, then following the call to `FIDALSINIT`, the following call must be made

```
CALL FIDAPARSESETJAC (IER)
```

The int return flag `IER` is an error return flag which is 0 for success or nonzero for an error.

#### IDALS with Jacobian-vector product

As an option when using the IDALS linear solver interface, the user may supply a routine that computes the product of the system Jacobian  $J = \partial F / \partial y + \alpha \partial F / \partial \dot{y}$  and a given vector  $v$ . If supplied, it must have the following form:

```
SUBROUTINE FIDAJTIMES(T, Y, YP, R, V, FJV, CJ, EWT, H,
&                      IPAR, RPAR, WK1, WK2, IER)
  DIMENSION Y(*), YP(*), R(*), V(*), FJV(*), EWT(*),
&          IPAR(*), RPAR(*), WK1(*), WK2(*)
```

This routine must compute the product vector  $Jv$ , where the vector  $v$  is stored in `V`, and store the product in `FJV`. On return, set `IER` = 0 if `FIDAJTIMES` was successful, and nonzero otherwise. The vectors `WK1` and `WK2`, of length `NEQ`, are provided as work space for use in `FIDAJTIMES`. The input arguments `T`, `Y`, `YP`, `R`, and `CJ` are the current values of  $t$ ,  $y$ ,  $\dot{y}$ ,  $F(t, y, \dot{y})$ , and  $\alpha$ , respectively. The arrays `IPAR` (of integers) and `RPAR` (of reals) contain user data and are the same as those passed to `FIDAMALLOC`.

If the user's `FIDAJTIMES` uses difference quotient approximations, it may need to use the error weight array `EWT` and current stepsize `H` in the calculation of suitable increments. It may also need the unit roundoff, which can be obtained as the optional output `ROUT(6)`, passed from the calling program to this routine using `COMMON`.

If the user's Jacobian-times-vector product routine requires that any Jacobian related data be evaluated or preprocessed, then the following routine can be used for the evaluation and preprocessing of this data:

```
SUBROUTINE FIDAJTSETUP (T, Y, YP, R, CJ, EWT, H, IPAR, RPAR, IER)
  DIMENSION Y(*), YP(*), R(*), EWT(*), IPAR(*), RPAR(*)
```

Typically this routine will use only `T`, `Y`, and `IDAYP`. It should compute any necessary data for subsequent calls to `FIDAJTIMES`. On return, set `IER` = 0 if `FIDAJTSETUP` was successful, and nonzero otherwise. The arrays `IPAR` (of integers) and `RPAR` (of reals) contain user data and are the same as those passed to `FIDAMALLOC`.

To indicate that the `FIDAJTIMES` and `FIDAJTSETUP` routines have been provided, then following the call to `FIDALSINIT`, the following call must be made



```
CALL FIDALSSETJAC (FLAG, IER)
```

with  $\text{FLAG} \neq 0$ . The return flag  $\text{IER}$  is 0 if successful, or negative if a memory error occurred.

The previous routine `FIDASPILSETJAC` is now a wrapper for this routine, and may still be used for backward-compatibility. However, this will be deprecated in future releases, so we recommend that users transition to the new routine name soon.

If the user calls `FIDALSSETJAC`, the routine `FIDAJTSETUP` must be provided, even if it is not needed, and it must return  $\text{IER}=0$ .



#### IDALS with preconditioning

If user-supplied preconditioning is to be performed, the following routine must be supplied for solution of the preconditioner linear system:

```
SUBROUTINE FIDAPSOL(T, Y, YP, R, RV, ZV, CJ, DELTA, EWT,
&                  IPAR, RPAR, IER)
  DIMENSION Y(*), YP(*), R(*), RV(*), ZV(*), EWT(*),
&          IPAR(*), RPAR(*)
```

It must solve the preconditioner linear system  $Pz = r$ , where  $r = \text{RV}$  is input, and store the solution  $z$  in  $\text{ZV}$ . Here  $P$  is the left preconditioner. The input arguments  $\text{T}$ ,  $\text{Y}$ ,  $\text{YP}$ ,  $\text{R}$ , and  $\text{CJ}$  are the current values of  $t$ ,  $y$ ,  $\dot{y}$ ,  $F(t, y, \dot{y})$ , and  $\alpha$ , respectively. On return, set  $\text{IER} = 0$  if `FIDAPSOL` was successful, set  $\text{IER}$  positive if a recoverable error occurred, and set  $\text{IER}$  negative if a non-recoverable error occurred.

The arguments  $\text{EWT}$  and  $\text{DELTA}$  are input and provide the error weight array and a scalar tolerance, respectively, for use by `FIDAPSOL` if it uses an iterative method in its solution. In that case, the residual vector  $\rho = r - Pz$  of the system should be made less than  $\text{DELTA}$  in weighted  $\ell_2$  norm, i.e.  $\sqrt{\sum (\rho_i * \text{EWT}[i])^2} < \text{DELTA}$ . The arrays  $\text{IPAR}$  (of integers) and  $\text{RPAR}$  (of reals) contain user data and are the same as those passed to `FIDAMALLOC`.

If the user's preconditioner requires that any Jacobian-related data be evaluated or preprocessed, then the following routine is to be used for the evaluation and preprocessing of the preconditioner:

```
SUBROUTINE FIDAPSET(T, Y, YP, R, CJ, EWT, H,
&                  IPAR, RPAR, IER)
  DIMENSION Y(*), YP(*), R(*), EWT(*),
&          IPAR(*), RPAR(*)
```

It must perform any evaluation of Jacobian-related data and preprocessing needed for the solution of the preconditioner linear systems by `FIDAPSOL`. The input arguments  $\text{T}$ ,  $\text{Y}$ ,  $\text{YP}$ ,  $\text{R}$ , and  $\text{CJ}$  are the current values of  $t$ ,  $y$ ,  $\dot{y}$ ,  $F(t, y, \dot{y})$ , and  $\alpha$ , respectively. On return, set  $\text{IER} = 0$  if `FIDAPSET` was successful, set  $\text{IER}$  positive if a recoverable error occurred, and set  $\text{IER}$  negative if a non-recoverable error occurred. The arrays  $\text{IPAR}$  (of integers) and  $\text{RPAR}$  (of reals) contain user data and are the same as those passed to `FIDAMALLOC`.

If the user's `FIDAPSET` uses difference quotient approximations, it may need to use the error weight array  $\text{EWT}$  and current stepsize  $\text{H}$  in the calculation of suitable increments. It may also need the unit roundoff, which can be obtained as the optional output  $\text{ROUT}(6)$ , passed from the calling program to this routine using `COMMON`.

To indicate that the `FIDAPSET` and `FIDAPSOL` routines are supplied, then following the call to `FIDALSINIT`, the user must call

```
CALL FIDALSSETPREC (FLAG, IER)
```

with `FLAG`  $\neq 0$ . The return flag `IER` is 0 if successful, or negative if a memory error occurred. In addition, the user must supply preconditioner routines `FIDAPSET` and `FIDAPSOL`.

The previous routine `FIDASPILSETPREC` is now a wrapper for this routine, and may still be used for backward-compatibility. However, this will be deprecated in future releases, so we recommend that users transition to the new routine name soon.



If the user calls `FIDALSSETPREC`, the subroutine `FIDAPSET` must be provided, even if it is not needed, and it must return `IER = 0`.

## 8. Correct initial values

Optionally, to correct the initial values  $y$  and/or  $\dot{y}$ , make the call

```
CALL FIDACALCIC (ICOPT, TOUT1, IER)
```

(See §2.1 for details.) The arguments are as follows: `ICOPT` is 1 for initializing the algebraic components of  $y$  and differential components of  $\dot{y}$ , or 2 for initializing all of  $y$ . `IER` is an error return flag, which is 0 for success, or negative for a failure (see `IDACalcIC` return values).

## 9. Problem solution

Carrying out the integration is accomplished by making calls as follows:

```
CALL FIDASOLVE (TOUT, T, Y, YP, ITASK, IER)
```

The arguments are as follows. `TOUT` specifies the next value of  $t$  at which a solution is desired (input). `T` is the value of  $t$  reached by the solver on output. `Y` is an array containing the computed solution vector  $y$  on output. `YP` is an array containing the computed solution vector  $\dot{y}$  on output. `ITASK` is a task indicator and should be set to 1 for normal mode (overshoot `TOUT` and interpolate), or to 2 for one-step mode (return after each internal step taken). `IER` is a completion flag and will be set to a positive value upon successful return or to a negative value if an error occurred. These values correspond to the `IDASolve` returns (see §4.5.7 and §B.2). The current values of the optional outputs are available in `IOUT` and `ROUT` (see Table 5.2).

## 10. Additional solution output

After a successful return from `FIDASOLVE`, the routine `FIDAGETDKY` may be called to get interpolated values of  $y$  or any derivative  $d^k y/dt^k$  for  $k$  not exceeding the current method order, and for any value of  $t$  in the last internal step taken by IDA. The call is as follows:

```
CALL FIDAGETDKY (T, K, DKY, IER)
```

where `T` is the input value of  $t$  at which solution derivative is desired, `K` is the derivative order, and `DKY` is an array containing the computed vector  $y^{(K)}(t)$  on return. The value of `T` must lie between `TCUR - HLAST` and `TCUR`. The value of `K` must satisfy  $0 \leq K \leq \text{QLAST}$ . (See the optional outputs for `TCUR`, `HLAST`, and `QLAST`.) The return flag `IER` is set to 0 upon successful return, or to a negative value to indicate an illegal input.

## 11. Problem reinitialization

To re-initialize the IDA solver for the solution of a new problem of the same size as one already solved, make the following call:

```
CALL FIDAREINIT (T0, Y0, YP0, IATOL, RTOL, ATOL, IER)
```

The arguments have the same names and meanings as those of `FIDAMALLOC`. `FIDAREINIT` performs the same initializations as `FIDAMALLOC`, but does no memory allocation, using instead the existing internal memory created by the previous `FIDAMALLOC` call.

Following this call, if the choice of linear solver is being changed then a user must make a call to create the alternate SUNLINSOL module and then attach it to the IDALS interface, as shown above. If only linear solver parameters are being modified, then these calls may be made without re-attaching to the IDALS interface.

## 12. Memory deallocation

To free the internal memory created by the call to FIDAMALLOC, FIDALSINIT, FNVINIT\* and FSUN\*\*\*MATINIT, make the call

```
CALL FIDAFREE
```

## 5.5 FIDA optional input and output

In order to keep the number of user-callable FIDA interface routines to a minimum, optional inputs to the IDA solver are passed through only three routines: FIDASETIIN for integer optional inputs, FIDASETRIN for real optional inputs, and FIDASETVIN for real vector (array) optional inputs. These functions should be called as follows:

```
CALL FIDASETIIN(KEY, IVAL, IER)
CALL FIDASETRIN(KEY, RVAL, IER)
CALL FIDASETVIN(KEY, VVAL, IER)
```

where KEY is a quoted string indicating which optional input is set (see Table 5.1), IVAL is the input integer value, RVAL is the input real value (scalar), VVAL is the input real array, and IER is an integer return flag which is set to 0 on success and a negative value if a failure occurred. IVAL should be declared so as to match C type `long int`.

When using FIDASETVIN to specify the variable types (KEY = 'ID\_VEC') the components in the array VVAL must be 1.0 to indicate a differential variable, or 0.0 to indicate an algebraic variable. Note that this array is required only if FIDACALCIC is to be called with ICOPT = 1, or if algebraic variables are suppressed from the error test (indicated using FIDASETIIN with KEY = 'SUPPRESS\_ALG'). When using FIDASETVIN to specify optional constraints on the solution vector (KEY = 'CONSTR\_VEC') the components in the array VVAL should be one of -2.0, -1.0, 0.0, 1.0, or 2.0. See the description of IDASetConstraints (§4.5.8.1) for details.

The optional outputs from the IDA solver are accessed not through individual functions, but rather through a pair of arrays, IOUT (integer type) of dimension at least 21, and ROUT (real type) of dimension at least 6. These arrays are owned (and allocated) by the user and are passed as arguments to FIDAMALLOC. Table 5.2 lists the entries in these two arrays and specifies the optional variable as well as the IDA function which is actually called to extract the optional output.

For more details on the optional inputs and outputs, see §4.5.8 and §4.5.10.

In addition to the optional inputs communicated through FIDASET\* calls and the optional outputs extracted from IOUT and ROUT, the following user-callable routines are available:

To reset the tolerances at any time, make the following call:

```
CALL FIDATOLREINIT (IATOL, RTOL, ATOL, IER)
```

The tolerance arguments have the same names and meanings as those of FIDAMALLOC. The error return flag IER is 0 if successful, and negative if there was a memory failure or illegal input.

To obtain the error weight array EWT, containing the multiplicative error weights used the WRMS norms, make the following call:

```
CALL FIDAGETERRWEIGHTS (EWT, IER)
```

This computes the EWT array, normally defined by Eq. (2.6). The array EWT, of length NEQ or NLOCAL, must already have been declared by the user. The error return flag IER is zero if successful, and negative if there was a memory error.

Table 5.1: Keys for setting FIDA optional inputs

Integer optional inputs (FIDASETIIN)		
Key	Optional input	Default value
MAX_ORD	Maximum LMM method order	5
MAX_NSTEPS	Maximum no. of internal steps before $t_{\text{out}}$	500
MAX_ERRFAIL	Maximum no. of error test failures	10
MAX_NITERS	Maximum no. of nonlinear iterations	4
MAX_CONVFAIL	Maximum no. of convergence failures	10
SUPPRESS_ALG	Suppress alg. vars. from error test (1 = SUNTRUE)	0 (= SUNFALSE)
MAX_NSTEPS_IC	Maximum no. of steps for IC calc.	5
MAX_NITERS_IC	Maximum no. of Newton iterations for IC calc.	10
MAX_NJE_IC	Maximum no. of Jac. evals fo IC calc.	4
LS_OFF_IC	Turn off line search (1 = SUNTRUE)	0 (= SUNFALSE)

Real optional inputs (FIDASETRIN)		
Key	Optional input	Default value
INIT_STEP	Initial step size	estimated
MAX_STEP	Maximum absolute step size	$\infty$
STOP_TIME	Value of $t_{\text{stop}}$	undefined
NLCONV_COEF	Coeff. in the nonlinear conv. test	0.33
NLCONV_COEF_IC	Coeff. in the nonlinear conv. test for IC calc.	0.0033
STEP_TOL_IC	Lower bound on Newton step for IC calc.	around <sup>2/3</sup>

Real vector optional inputs (FIDASETVIN)		
Key	Optional input	Default value
ID_VEC	Differential/algebraic component types	undefined
CONSTR_VEC	Inequality constraints on solution	undefined

Table 5.2: Description of the FIDA optional output arrays IOUT and ROUT

Integer output array IOUT		
Index	Optional output	IDA function
IDA main solver		
1	LENRW	IDAGetWorkSpace
2	LENIW	IDAGetWorkSpace
3	NST	IDAGetNumSteps
4	NRE	IDAGetNumResEvals
5	NETF	IDAGetNumErrTestFails
6	NNCFAILS	IDAGetNonlinSolvConvFails
7	NNI	IDAGetNumNonlinSolvIters
8	NSETUPS	IDAGetNumLinSolvSetups
9	QLAST	IDAGetLastOrder
10	QCUR	IDAGetCurrentOrder
11	NBCKTRKOPS	IDAGetNumBacktrackOps
12	NGE	IDAGetNumGEvals
IDALS linear solver interface		
13	LENRWLS	IDAGetLinWorkSpace
14	LENIWLS	IDAGetLinWorkSpace
15	LS_FLAG	IDAGetLastLinFlag
16	NRELS	IDAGetNumLinResEvals
17	NJE	IDAGetNumJacEvals
18	NJTS	IDAGetNumJTSetupEvals
19	NJT	IDAGetNumJtimesEvals
20	NPE	IDAGetNumPrecEvals
21	NPS	IDAGetNumPrecSolves
22	NLI	IDAGetNumLinIters
23	NCFL	IDAGetNumLinConvFails

Real output array ROUT		
Index	Optional output	IDA function
1	H0_USED	IDAGetActualInitStep
2	HLAST	IDAGetLastStep
3	HCUR	IDAGetCurrentStep
4	TCUR	IDAGetCurrentTime
5	TOLFACT	IDAGetTolScaleFactor
6	UROUND	unit roundoff

To obtain the estimated local errors, following a successful call to `FIDASOLVE`, make the following call:

```
CALL FIDAGETESTLOCALERR (ELE, IER)
```

This computes the `ELE` array of estimated local errors as of the last step taken. The array `ELE` must already have been declared by the user. The error return flag `IER` is zero if successful, and negative if there was a memory error.

## 5.6 Usage of the FIDAROOT interface to rootfinding

The `FIDAROOT` interface package allows programs written in FORTRAN to use the rootfinding feature of the IDA solver module. The user-callable functions in `FIDAROOT`, with the corresponding IDA functions, are as follows:

- `FIDAROOTINIT` interfaces to `IDARootInit`.
- `FIDAROOTINFO` interfaces to `IDAGetRootInfo`.
- `FIDAROOTFREE` interfaces to `IDARootFree`.

Note that, at this time `FIDAROOT` does not provide support to specify the direction of zero-crossing that is to be monitored. Instead, all roots are considered. However, the actual direction of zero-crossing is reported (through the sign of the non-zero elements in the array `INFO` returned by `FIDAROOTINFO`).

In order to use the rootfinding feature of IDA, the following call must be made, after calling `FIDAMALLOC` but prior to calling `FIDASOLVE`, to allocate and initialize memory for the `FIDAROOT` module:

```
CALL FIDAROOTINIT (NRTFN, IER)
```

The arguments are as follows: `NRTFN` is the number of root functions. `IER` is a return completion flag; its values are 0 for success, -1 if the IDA memory was NULL, and -14 if a memory allocation failed.

To specify the functions whose roots are to be found, the user must define the following routine:

```
SUBROUTINE FIDAROOTFN (T, Y, YP, G, IPAR, RPAR, IER)
  DIMENSION Y(*), YP(*), G(*), IPAR(*), RPAR(*)
```

It must set the `G` array, of length `NRTFN`, with components  $g_i(t, y, \dot{y})$ , as a function of  $T = t$  and the arrays  $Y = y$  and  $YP = \dot{y}$ . The arrays `IPAR` (of integers) and `RPAR` (of reals) contain user data and are the same as those passed to `FIDAMALLOC`. Set `IER` on 0 if successful, or on a non-zero value if an error occurred.

When making calls to `FIDASOLVE` to solve the DAE system, the occurrence of a root is flagged by the return value `IER = 2`. In that case, if `NRTFN > 1`, the functions  $g_i$  which were found to have a root can be identified by making the following call:

```
CALL FIDAROOTINFO (NRTFN, INFO, IER)
```

The arguments are as follows: `NRTFN` is the number of root functions. `INFO` is an integer array of length `NRTFN` with root information. `IER` is a return completion flag; its values are 0 for success, negative if there was a memory failure. The returned values of `INFO(i)` ( $i = 1, \dots, NRTFN$ ) are 0 or  $\pm 1$ , such that `INFO(i) = +1` if  $g_i$  was found to have a root and  $g_i$  is increasing, `INFO(i) = -1` if  $g_i$  was found to have a root and  $g_i$  is decreasing, and `INFO(i) = 0` otherwise.

The total number of calls made to the root function `FIDAROOTFN`, denoted `NGE`, can be obtained from `IOUT(12)`. If the FIDA/IDA memory block is reinitialized to solve a different problem via a call to `FIDAREINIT`, then the counter `NGE` is reset to zero.

To free the memory resources allocated by a prior call to `FIDAROOTINIT`, make the following call:

```
CALL FIDAROOTFREE
```

See §4.5.6 for additional information on the rootfinding feature.

## 5.7 Usage of the FIDABBD interface to IDABBDPRE

The FIDABBD interface sub-module is a package of C functions which, as part of the FIDA interface module, support the use of the IDA solver with the parallel NVECTOR\_PARALLEL module, in a combination of any of the Krylov iterative solver modules with the IDABBDPRE preconditioner module (see §4.7).

The user-callable functions in this package, with the corresponding IDA and IDABBDPRE functions, are as follows:

- FIDABBDINIT interfaces to IDABBDPrecAlloc.
- FIDABBDREINIT interfaces to IDABBDPrecReInit.
- FIDABBDOPT interfaces to IDABBDPRE optional output functions.
- FIDABBDFREE interfaces to IDABBDPrecFree.

In addition to the FORTRAN residual function FIDARESFUN, the user-supplied functions used by this package, are listed below, each with the corresponding interface function which calls it (and its type within IDABBDPRE or IDA):

FIDABBD routine (FORTRAN)	IDA function (C)	IDA function type
FIDAGLOCFN	FIDAgloc	IDABBDLocalFn
FIDACOMMFN	FIDAcfn	IDABBDCommFn
FIDAJTIMES	FIDAJtimes	IDALsJacTimesVecFn
FIDAJTSETUP	FIDAJTSetup	IDALsJacTimesSetupFn

As with the rest of the FIDA routines, the names of all user-supplied routines here are fixed, in order to maximize portability for the resulting mixed-language program. Additionally, based on flags discussed above in §5.3, the names of the user-supplied routines are mapped to actual values through a series of definitions in the header file `fidabbd.h`.

The following is a summary of the usage of this module. Steps that are unchanged from the main program described in §5.4 are grayed-out.

1. Residual function specification
2. NVECTOR module initialization
3. SUNLINSOL module initialization

Initialize one of the iterative SUNLINSOL modules, by calling one of FSUNPCGINIT, FSUNSPBCGSINIT, FSUNSPFGMRINIT, FSUNSPGMRINIT or FSUNSPTFQMRINIT.

4. Problem specification
5. Set optional inputs
6. Linear solver interface specification

Initialize the IDALS iterative linear solver interface by calling FIDALSINIT.

7. BBD preconditioner initialization

To initialize the IDABBDPRE preconditioner, make the following call:

```
CALL FIDABBDINIT (NLOCAL, MUDQ, MLDQ, MU, ML, DQRELY, IER)
```

The arguments are as follows. NLOCAL is the local size of vectors on this processor. MUDQ and MLDQ are the upper and lower half-bandwidths to be used in the computation of the local Jacobian blocks by difference quotients. These may be smaller than the true half-bandwidths of the Jacobian of

the local block of  $G$ , when smaller values may provide greater efficiency. MU and ML are the upper and lower half-bandwidths of the band matrix that is retained as an approximation of the local Jacobian block. These may be smaller than MUDQ and MLDQ. DQRELY is the relative increment factor in  $y$  for difference quotients (optional). A value of 0.0 indicates the default,  $\sqrt{\text{unit roundoff}}$ . 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.

#### 8. Correct initial values

#### 9. Problem solution

#### 10. Additional solution output

#### 11. IDABBDPRE Optional outputs

Optional outputs specific to the SPGMR, SPBCGS, or SPTFQMR solver are listed in Table 5.2. To obtain the optional outputs associated with the IDABBDPRE module, make the following call:

```
CALL FIDABBDOPT (LENRWBBD, LENIWBBBD, NGEBBBD)
```

The arguments should be consistent with C type `long int`. Their returned values are as follows: LENRWBBD is the length of real preconditioner work space, in `realtype` words. LENIWBBBD is the length of integer preconditioner work space, in integer words. Both of these sizes are local to the current processor. NGEBBBD is the number of  $G(t, y, \dot{y})$  evaluations (calls to FIDALOCFN) so far.

#### 12. Problem reinitialization

If a sequence of problems of the same size is being solved using the same linear solver in combination with the IDABBDPRE preconditioner, then the IDA package can be re-initialized for the second and subsequent problems by calling FIDAREINIT, following which a call to FIDABBDINIT may or may not be needed. If the input arguments are the same, no FIDABBDINIT call is needed. If there is a change in input arguments other than MU or ML, then the user program should make the call

```
CALL FIDABBDREINIT (NLOCAL, MUDQ, MLDQ, DQRELY, IER)
```

This reinitializes the IDABBDPRE preconditioner, but without reallocating its memory. The arguments of the FIDABBDREINIT routine have the same names and meanings as those of FIDABBDINIT. If the value of MU or ML is being changed, then a call to FIDABBDINIT must be made. Finally, if there is a change in any of the linear solver inputs, then a call to one of FSUN\*\*\*\*INIT, followed by a call to FIDALSINIT must also be made; in this case the linear solver memory is reallocated.

#### 13. Memory deallocation

(The memory allocated for the FIDABBD module is deallocated automatically by FIDAFREE.)

#### 14. User-supplied routines

The following two routines must be supplied for use with the IDABBDPRE module:

```
SUBROUTINE FIDAGLOCFN (NLOC, T, YLOC, YPLOC, GLOC, IPAR, RPAR, IER)
  DIMENSION YLOC(*), YPLOC(*), GLOC(*), IPAR(*), RPAR(*)
```

This routine is to evaluate the function  $G(t, y, \dot{y})$  approximating  $F$  (possibly identical to  $F$ ), in terms of  $T = t$ , and the arrays YLOC and YPLOC (of length NLOC), which are the sub-vectors of  $y$  and  $\dot{y}$  local to this processor. The resulting (local) sub-vector is to be stored in the array GLOC. IER is a return flag that should be set to 0 if successful, to 1 (for a recoverable error), or to -1 (for a non-recoverable error). The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FIDAMALLOC.



```
SUBROUTINE FIDACOMMFN (NLOC, T, YLOC, YPLOC, IPAR, RPAR, IER)
  DIMENSION YLOC(*), YPLOC(*), IPAR(*), RPAR(*)
```

This routine is to perform the inter-processor communication necessary for the FIDAGLOCFN routine. Each call to FIDACOMMFN is preceded by a call to the residual routine FIDARESFUN with the same arguments T, YLOC, and YPLOC. Thus FIDACOMMFN can omit any communications done by FIDARESFUN if relevant to the evaluation of GLOC. The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FIDAMALLOC. IER is a return flag that should be set to 0 if successful, to 1 (for a recoverable error), or to -1 (for a non-recoverable error).

The subroutine FIDACOMMFN must be supplied even if it is empty, and it must return IER = 0.

Optionally, the user can supply routines FIDAJTIMES and FIDAJTSETUP for the evaluation of Jacobian-vector products, as described above in step 7 in §5.4.





## 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 the implementations provided with SUNDIALS. The generic operations are described below and the implementations provided with SUNDIALS are described in the following sections.

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_ID (*nvgetvectorid)(N_Vector);  
    N_Vector (*nvclone)(N_Vector);  
    N_Vector (*nvcloneempty)(N_Vector);  
    void (*nvdestroy)(N_Vector);  
    void (*nvspace)(N_Vector, sunindextype *, sunindextype *);  
    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    (*nvlinorm)(N_Vector);
void        (*nvcompare)(realtype, N_Vector, N_Vector);
boolean_t   (*nvinvtest)(N_Vector, N_Vector);
boolean_t   (*nvconstrmask)(N_Vector, N_Vector, N_Vector);
realtype    (*nvminquotient)(N_Vector, N_Vector);
int         (*nvlinearcombination)(int, realtype*, N_Vector*, N_Vector);
int         (*nvscaleaddmulti)(int, realtype*, N_Vector, N_Vector*, N_Vector*);
int         (*nvdotprodmulti)(int, N_Vector, N_Vector*, realtype*);
int         (*nvlinearsumvectorarray)(int, realtype, N_Vector*, realtype,
                                     N_Vector*, N_Vector*);
int         (*nvscalevectorarray)(int, realtype*, N_Vector*, N_Vector*);
int         (*nvconstvectorarray)(int, realtype, N_Vector*);
int         (*nvwrmsnomrvectorarray)(int, N_Vector*, N_Vector*, realtype*);
int         (*nvwrmsnomrmaskvectorarray)(int, N_Vector*, N_Vector*, N_Vector,
                                     realtype*);
int         (*nvscaleaddmultivectorarray)(int, int, realtype*, N_Vector*,
                                     N_Vector**, N_Vector**);
int         (*nvlinearcombinationvectorarray)(int, int, realtype*, N_Vector**,
                                     N_Vector*);
};

```

The generic NVECTOR module defines and implements the vector operations acting on an `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 6.2 contains a complete list of all standard vector operations defined by the generic NVECTOR module. Tables 6.3 and 6.4 list *optional* fused and vector array operations respectively. These operations are intended to increase data reuse, reduce parallel communication on distributed memory systems, and lower the number of kernel launches on systems with accelerators. If a particular NVECTOR implementation defines one of the fused or vector array operations as `NULL`, the NVECTOR interface will call one of the standard vector operations as necessary.

Finally, note that the generic NVECTOR module defines the functions `N_VCloneVectorArray` and `N_VCloneVectorArrayEmpty`. Both functions create (by cloning) an array of `count` variables of type `N_Vector`, each of the same type as an existing `N_Vector`. Their prototypes are

```

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

```

and their definitions are based on the implementation-specific `N_VClone` and `N_VCloneEmpty` operations, respectively.

An array of variables of type `N_Vector` can be destroyed by calling `N_VDestroyVectorArray`, whose prototype is

```

void N_VDestroyVectorArray(N_Vector *vs, int count);

```

and whose definition is based on the implementation-specific `N_VDestroy` operation.

A particular implementation of the NVECTOR module must:

Table 6.1: Vector Identifications associated with vector kernels supplied with SUNDIALS.

Vector ID	Vector type	ID Value
SUNDIALS_NVEC_SERIAL	Serial	0
SUNDIALS_NVEC_PARALLEL	Distributed memory parallel (MPI)	1
SUNDIALS_NVEC_OPENMP	OpenMP shared memory parallel	2
SUNDIALS_NVEC_PTHREADS	PThreads shared memory parallel	3
SUNDIALS_NVEC_PARHYP	<i>hypre</i> ParHyp parallel vector	4
SUNDIALS_NVEC_PETSC	PETSc parallel vector	5
SUNDIALS_NVEC_CUSTOM	User-provided custom vector	6

- 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**.

Each NVECTOR implementation included in SUNDIALS has a unique identifier specified in enumeration and shown in Table 6.1. It is recommended that a user-supplied NVECTOR implementation use the SUNDIALS\_NVEC\_CUSTOM identifier.

Table 6.2: Description of the NVECTOR operations

Name	Usage and Description
<code>N_VGetVectorID</code>	<code>id = N_VGetVectorID(w);</code> Returns the vector type identifier for the vector <code>w</code> . It is used to determine the vector implementation type (e.g. serial, parallel, ...) from the abstract <code>N_Vector</code> interface. Returned values are given in Table 6.1.
<code>N_VClone</code>	<code>v = N_VClone(w);</code> Creates a new <code>N_Vector</code> of the same type as an existing vector <code>w</code> and sets the <code>ops</code> field. It does not copy the vector, but rather allocates storage for the new vector.
<code>N_VCloneEmpty</code>	<code>v = N_VCloneEmpty(w);</code> Creates a new <code>N_Vector</code> of the same type as an existing vector <code>w</code> and sets the <code>ops</code> field. It does not allocate storage for data.
<code>N_VDestroy</code>	<code>N_VDestroy(v);</code> Destroys the <code>N_Vector</code> <code>v</code> and frees memory allocated for its internal data.
<code>N_VSpace</code>	<code>N_VSpace(nvSpec, &amp;lrw, &amp;liw);</code> Returns storage requirements for one <code>N_Vector</code> . <code>lrw</code> contains the number of realtype words and <code>liw</code> contains the number of integer words. This function is advisory only, for use in determining a user's total space requirements; it could be a dummy function in a user-supplied NVECTOR module if that information is not of interest.
<code>N_VGetArrayPointer</code>	<code>vdata = N_VGetArrayPointer(v);</code> Returns a pointer to a <code>realtype</code> array from the <code>N_Vector</code> <code>v</code> . Note that this assumes that the internal data in <code>N_Vector</code> is a contiguous array of <code>realtype</code> . This routine is only used in the solver-specific interfaces to the dense and banded (serial) linear solvers, the sparse linear solvers (serial and threaded), and in the interfaces to the banded (serial) and band-block-diagonal (parallel) preconditioner modules provided with SUNDIALS.
<code>N_VSetArrayPointer</code>	<code>N_VSetArrayPointer(vdata, v);</code> Overwrites the data in an <code>N_Vector</code> with a given array of <code>realtype</code> . Note that this assumes that the internal data in <code>N_Vector</code> is a contiguous array of <code>realtype</code> . This routine is only used in the interfaces to the dense (serial) linear solver, hence need not exist in a user-supplied NVECTOR module for a parallel environment.
<i>continued on next page</i>	

<i>continued from last page</i>	
Name	Usage and Description
N_VLinearSum	<code>N_VLinearSum(a, x, b, y, z);</code> Performs the operation $z = ax + by$ , where $a$ and $b$ are <b>realtype</b> scalars and $x$ and $y$ are of type <b>N_Vector</b> : $z_i = ax_i + by_i$ , $i = 0, \dots, n-1$ .
N_VConst	<code>N_VConst(c, z);</code> Sets all components of the <b>N_Vector</b> $z$ to <b>realtype</b> $c$ : $z_i = c$ , $i = 0, \dots, n-1$ .
N_VProd	<code>N_VProd(x, y, z);</code> Sets the <b>N_Vector</b> $z$ to be the component-wise product of the <b>N_Vector</b> inputs $x$ and $y$ : $z_i = x_i y_i$ , $i = 0, \dots, n-1$ .
N_VDiv	<code>N_VDiv(x, y, z);</code> Sets the <b>N_Vector</b> $z$ to be the component-wise ratio of the <b>N_Vector</b> 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 a $y$ that is guaranteed to have all nonzero components.
N_VScale	<code>N_VScale(c, x, z);</code> Scales the <b>N_Vector</b> $x$ by the <b>realtype</b> scalar $c$ and returns the result in $z$ : $z_i = cx_i$ , $i = 0, \dots, n-1$ .
N_VAbs	<code>N_VAbs(x, z);</code> Sets the components of the <b>N_Vector</b> $z$ to be the absolute values of the components of the <b>N_Vector</b> $x$ : $y_i =  x_i $ , $i = 0, \dots, n-1$ .
N_VInv	<code>N_VInv(x, z);</code> Sets the components of the <b>N_Vector</b> $z$ to be the inverses of the components of the <b>N_Vector</b> $x$ : $z_i = 1.0/x_i$ , $i = 0, \dots, n-1$ . This routine may not check for division by 0. It should be called only with an $x$ which is guaranteed to have all nonzero components.
N_VAddConst	<code>N_VAddConst(x, b, z);</code> Adds the <b>realtype</b> scalar $b$ to all components of $x$ and returns the result in the <b>N_Vector</b> $z$ : $z_i = x_i + b$ , $i = 0, \dots, n-1$ .
N_VDotProd	<code>d = N_VDotProd(x, y);</code> Returns the value of the ordinary dot product of $x$ and $y$ : $d = \sum_{i=0}^{n-1} x_i y_i$ .
N_VMaxNorm	<code>m = N_VMaxNorm(x);</code> Returns the maximum norm of the <b>N_Vector</b> $x$ : $m = \max_i  x_i $ .
<i>continued on next page</i>	

<i>continued from last page</i>	
Name	Usage and Description
N_VWrmsNorm	$\mathbf{m} = \text{N\_VWrmsNorm}(\mathbf{x}, \mathbf{w})$ Returns the weighted root-mean-square norm of the N_Vector $\mathbf{x}$ with <b>realtype</b> weight vector $\mathbf{w}$ : $m = \sqrt{\left(\sum_{i=0}^{n-1} (x_i w_i)^2\right) / n}$ .
N_VWrmsNormMask	$\mathbf{m} = \text{N\_VWrmsNormMask}(\mathbf{x}, \mathbf{w}, \text{id});$ Returns the weighted root mean square norm of the N_Vector $\mathbf{x}$ with <b>realtype</b> weight vector $\mathbf{w}$ built using only the elements of $\mathbf{x}$ corresponding to positive elements of the N_Vector $\text{id}$ : $m = \sqrt{\left(\sum_{i=0}^{n-1} (x_i w_i H(\text{id}_i))^2\right) / n}, \text{ where } H(\alpha) = \begin{cases} 1 & \alpha > 0 \\ 0 & \alpha \leq 0 \end{cases}$
N_VMin	$\mathbf{m} = \text{N\_VMin}(\mathbf{x});$ Returns the smallest element of the N_Vector $\mathbf{x}$ : $m = \min_i x_i$ .
N_VWL2Norm	$\mathbf{m} = \text{N\_VWL2Norm}(\mathbf{x}, \mathbf{w});$ Returns the weighted Euclidean $\ell_2$ norm of the N_Vector $\mathbf{x}$ with <b>realtype</b> weight vector $\mathbf{w}$ : $m = \sqrt{\sum_{i=0}^{n-1} (x_i w_i)^2}$ .
N_VL1Norm	$\mathbf{m} = \text{N\_VL1Norm}(\mathbf{x});$ Returns the $\ell_1$ norm of the N_Vector $\mathbf{x}$ : $m = \sum_{i=0}^{n-1}  x_i $ .
N_VCompare	$\text{N\_VCompare}(\mathbf{c}, \mathbf{x}, \mathbf{z});$ Compares the components of the N_Vector $\mathbf{x}$ to the <b>realtype</b> scalar $\mathbf{c}$ and returns an N_Vector $\mathbf{z}$ such that: $z_i = 1.0$ if $ x_i  \geq c$ and $z_i = 0.0$ otherwise.
N_VInvTest	$\mathbf{t} = \text{N\_VInvTest}(\mathbf{x}, \mathbf{z});$ Sets the components of the N_Vector $\mathbf{z}$ to be the inverses of the components of the N_Vector $\mathbf{x}$ , with prior testing for zero values: $z_i = 1.0/x_i, i = 0, \dots, n-1$ . This routine returns a boolean assigned to <b>SUNTRUE</b> if all components of $\mathbf{x}$ are nonzero (successful inversion) and returns <b>SUNFALSE</b> otherwise.
N_VConstrMask	$\mathbf{t} = \text{N\_VConstrMask}(\mathbf{c}, \mathbf{x}, \mathbf{m});$ 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 a boolean assigned to <b>SUNFALSE</b> if any element failed the constraint test and assigned to <b>SUNTRUE</b> if all passed. It also sets a mask vector $\mathbf{m}$ , with elements equal to 1.0 where the constraint test failed, and 0.0 where the test passed. This routine is used only for constraint checking.
<i>continued on next page</i>	



<i>continued from last page</i>	
Name	Usage and Description
N_VMinQuotient	<pre>minq = N_VMinQuotient(num, denom);</pre> <p>This routine returns the minimum of the quotients obtained by term-wise dividing <math>\text{num}_i</math> by <math>\text{denom}_i</math>. A zero element in <b>denom</b> will be skipped. If no such quotients are found, then the large value <b>BIG_REAL</b> (defined in the header file <b>sundials_types.h</b>) is returned.</p>

Table 6.3: Description of the NVECTOR fused operations

Name	Usage and Description
N_VLinearCombination	<pre>ier = N_VLinearCombination(nv, c, X, z);</pre> <p>This routine computes the linear combination of <math>n_v</math> vectors with <math>n</math> elements:</p> $z_i = \sum_{j=0}^{n_v-1} c_j x_{j,i}, \quad i = 0, \dots, n-1,$ <p>where <math>c</math> is an array of <math>n_v</math> scalars (type <b>realtype*</b>), <math>X</math> is an array of <math>n_v</math> vectors (type <b>N_Vector*</b>), and <math>z</math> is the output vector (type <b>N_Vector</b>). If the output vector <math>z</math> is one of the vectors in <math>X</math>, then it <i>must</i> be the first vector in the vector array. The operation returns 0 for success and a non-zero value otherwise.</p>
N_VScaleAddMulti	<pre>ier = N_VScaleAddMulti(nv, c, x, Y, Z);</pre> <p>This routine scales and adds one vector to <math>n_v</math> vectors with <math>n</math> elements:</p> $z_{j,i} = c_j x_i + y_{j,i}, \quad j = 0, \dots, n_v-1 \quad i = 0, \dots, n-1,$ <p>where <math>c</math> is an array of <math>n_v</math> scalars (type <b>realtype*</b>), <math>x</math> is the vector (type <b>N_Vector</b>) to be scaled and added to each vector in the vector array of <math>n_v</math> vectors <math>Y</math> (type <b>N_Vector*</b>), and <math>Z</math> (type <b>N_Vector*</b>) is a vector array of <math>n_v</math> output vectors. The operation returns 0 for success and a non-zero value otherwise.</p>
<i>continued on next page</i>	

<i>continued from last page</i>	
Name	Usage and Description
N_VDotProdMulti	<p><b>ier</b> = N_VDotProdMulti(<b>nv</b>, <b>x</b>, <b>Y</b>, <b>d</b>);</p> <p>This routine computes the dot product of a vector with <math>n_v</math> other vectors:</p> $d_j = \sum_{i=0}^{n-1} x_i y_{j,i}, \quad j = 0, \dots, n_v - 1,$ <p>where <math>d</math> (type <b>realtype*</b>) is an array of <math>n_v</math> scalars containing the dot products of the vector <math>x</math> (type <b>N_Vector</b>) with each of the <math>n_v</math> vectors in the vector array <math>Y</math> (type <b>N_Vector*</b>). The operation returns 0 for success and a non-zero value otherwise.</p>

Table 6.4: Description of the NVECTOR vector array operations

Name	Usage and Description
N_VLinearSumVectorArray	<p><b>ier</b> = N_VLinearSumVectorArray(<b>nv</b>, <b>a</b>, <b>X</b>, <b>b</b>, <b>Y</b>, <b>Z</b>);</p> <p>This routine computes the linear sum of two vector arrays containing <math>n_v</math> vectors of <math>n</math> elements:</p> $z_{j,i} = ax_{j,i} + by_{j,i}, \quad i = 0, \dots, n-1 \quad j = 0, \dots, n_v - 1,$ <p>where <math>a</math> and <math>b</math> are <b>realtype</b> scalars and <math>X</math>, <math>Y</math>, and <math>Z</math> are arrays of <math>n_v</math> vectors (type <b>N_Vector*</b>). The operation returns 0 for success and a non-zero value otherwise.</p>
N_VScaleVectorArray	<p><b>ier</b> = N_VScaleVectorArray(<b>nv</b>, <b>c</b>, <b>X</b>, <b>Z</b>);</p> <p>This routine scales each vector of <math>n</math> elements in a vector array of <math>n_v</math> vectors by a potentially different constant:</p> $z_{j,i} = c_j x_{j,i}, \quad i = 0, \dots, n-1 \quad j = 0, \dots, n_v - 1,$ <p>where <math>c</math> is an array of <math>n_v</math> scalars (type <b>realtype*</b>) and <math>X</math> and <math>Z</math> are arrays of <math>n_v</math> vectors (type <b>N_Vector*</b>). The operation returns 0 for success and a non-zero value otherwise.</p>
<i>continued on next page</i>	

<i>continued from last page</i>	
Name	Usage and Description
N_VConstVectorArray	<pre>ier = N_VConstVectorArray(nv, c, X);</pre> <p>This routine sets each element in a vector of <math>n</math> elements in a vector array of <math>n_v</math> vectors to the same value:</p> $z_{j,i} = c, \quad i = 0, \dots, n-1 \quad j = 0, \dots, n_v-1,$ <p>where <math>c</math> is a <b>realtype</b> scalar and <math>X</math> is an array of <math>n_v</math> vectors (type <b>N_Vector*</b>). The operation returns 0 for success and a non-zero value otherwise.</p>
N_VWrmsNormVectorArray	<pre>ier = N_VWrmsNormVectorArray(nv, X, W, m);</pre> <p>This routine computes the weighted root mean square norm of <math>n_v</math> vectors with <math>n</math> elements:</p> $m_j = \left( \frac{1}{n} \sum_{i=0}^{n-1} (x_{j,i} w_{j,i})^2 \right)^{1/2}, \quad j = 0, \dots, n_v-1,$ <p>where <math>m</math> (type <b>realtype*</b>) contains the <math>n_v</math> norms of the vectors in the vector array <math>X</math> (type <b>N_Vector*</b>) with corresponding weight vectors <math>W</math> (type <b>N_Vector*</b>). The operation returns 0 for success and a non-zero value otherwise.</p>
N_VWrmsNormMaskVectorArray	<pre>ier = N_VWrmsNormMaskVectorArray(nv, X, W, id, m);</pre> <p>This routine computes the masked weighted root mean square norm of <math>n_v</math> vectors with <math>n</math> elements:</p> $m_j = \left( \frac{1}{n} \sum_{i=0}^{n-1} (x_{j,i} w_{j,i} H(id_i))^2 \right)^{1/2}, \quad j = 0, \dots, n_v-1,$ <p><math>H(id_i) = 1</math> for <math>id_i &gt; 0</math> and is zero otherwise, <math>m</math> (type <b>realtype*</b>) contains the <math>n_v</math> norms of the vectors in the vector array <math>X</math> (type <b>N_Vector*</b>) with corresponding weight vectors <math>W</math> (type <b>N_Vector*</b>) and mask vector <math>id</math> (type <b>N_Vector</b>). The operation returns 0 for success and a non-zero value otherwise.</p>
<i>continued on next page</i>	

continued from last page	
Name	Usage and Description
N_VScaleAddMultiVectorArray	<p><code>ier = N_VScaleAddMultiVectorArray(nv, ns, c, X, YY, ZZ);</code></p> <p>This routine scales and adds a vector in a vector array of <math>n_v</math> vectors to the corresponding vector in <math>n_s</math> vector arrays:</p> $z_{j,i} = \sum_{k=0}^{n_s-1} c_k x_{k,j,i}, \quad i = 0, \dots, n-1 \quad j = 0, \dots, n_v-1,$ <p>where <math>c</math> is an array of <math>n_s</math> scalars (type <code>realtype*</code>), <math>X</math> is a vector array of <math>n_v</math> vectors (type <code>idN_Vector*</code>) to be scaled and added to the corresponding vector in each of the <math>n_s</math> vector arrays in the array of vector arrays <math>YY</math> (type <code>N_Vector**</code>) and stored in the output array of vector arrays <math>ZZ</math> (type <code>N_Vector**</code>). The operation returns 0 for success and a non-zero value otherwise.</p>
N_VLinearCombinationVectorArray	<p><code>ier = N_VLinearCombinationVectorArray(nv, ns, c, XX, Z);</code></p> <p>This routine computes the linear combination of <math>n_s</math> vector arrays containing <math>n_v</math> vectors with <math>n</math> elements:</p> $z_{j,i} = \sum_{k=0}^{n_s-1} c_k x_{k,j,i}, \quad i = 0, \dots, n-1 \quad j = 0, \dots, n_v-1,$ <p>where <math>c</math> is an array of <math>n_s</math> scalars (type <code>realtype*</code>), <math>XX</math> (type <code>N_Vector**</code>) is an array of <math>n_s</math> vector arrays each containing <math>n_v</math> vectors to be summed into the output vector array of <math>n_v</math> vectors <math>Z</math> (type <code>N_Vector*</code>). If the output vector array <math>Z</math> is one of the vector arrays in <math>XX</math>, then it <i>must</i> be the first vector array in <math>XX</math>. The operation returns 0 for success and a non-zero value otherwise.</p>

## 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 {
    sunindextype length;
    booleantype own_data;
    realtype *data;
};
```

The header file to include when using this module is `nvector_serial.h`. The installed module library to link to is `libsundials_nvecserial.lib` where *.lib* is typically *.so* for shared libraries and *.a* for static libraries.

The following macros are provided to access the content of an NVECTOR\_SERIAL vector. The suffix *\_S* in the names denotes the 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 Tables 6.2, 6.3, and 6.4. Their names are obtained from those in Tables 6.2, 6.3, and 6.4. by appending the suffix `_Serial` (e.g. `N_VDestroy_Serial`). The module `NVECTOR_SERIAL` provides the following additional user-callable routines:

- `N_VNew_Serial`

This function creates and allocates memory for a serial `N_Vector`. Its only argument is the vector length.

```
N_Vector N_VNew_Serial(sunindextype vec_length);
```

- `N_VNewEmpty_Serial`

This function creates a new serial `N_Vector` with an empty (NULL) data array.

```
N_Vector N_VNewEmpty_Serial(sunindextype vec_length);
```

- `N_VMake_Serial`

This function creates and allocates memory for a serial vector with user-provided data array. (This function does *not* allocate memory for `v_data` itself.)

```
N_Vector N_VMake_Serial(sunindextype vec_length, realtype *v_data);
```

- `N_VCloneVectorArray_Serial`

This function creates (by cloning) an array of `count` serial vectors.

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

- `N_VCloneVectorArrayEmpty_Serial`

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

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

- `N_VDestroyVectorArray_Serial`

This function frees memory allocated for the array of `count` variables of type `N_Vector` created with `N_VCloneVectorArray_Serial` or with `N_VCloneVectorArrayEmpty_Serial`.

```
void N_VDestroyVectorArray_Serial(N_Vector *vs, int count);
```

- `N_VGetLength_Serial`

This function returns the number of vector elements.

```
sunindextype N_VGetLength_Serial(N_Vector v);
```

- `N_VPrint_Serial`

This function prints the content of a serial vector to `stdout`.

```
void N_VPrint_Serial(N_Vector v);
```

- `N_VPrintFile_Serial`

This function prints the content of a serial vector to `outfile`.

```
void N_VPrintFile_Serial(N_Vector v, FILE *outfile);
```

## Notes

- When looping over the components of an `N_Vector` `v`, it is more efficient to first obtain the component array via `v_data = NV_DATA_S(v)` and then access `v_data[i]` within the loop than it is to use `NV_Ith_S(v,i)` within the loop.



- `N_VNewEmpty_Serial`, `N_VMake_Serial`, and `N_VCloneVectorArrayEmpty_Serial` set the field `own_data = SUNFALSE`. `N_VDestroy_Serial` and `N_VDestroyVectorArray_Serial` will not attempt to free the pointer `data` for any `N_Vector` with `own_data` set to `SUNFALSE`. 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.

For solvers that include a Fortran interface module, the `NVECTOR_SERIAL` module also includes a Fortran-callable function `FNVINITS(code, NEQ, IER)`, to initialize this `NVECTOR_SERIAL` module. Here `code` is an input solver id (1 for `CVODE`, 2 for `IDA`, 3 for `KINSOL`, 4 for `ARKODE`); `NEQ` is the problem size (declared so as to match C type `long int`); and `IER` is an error return flag equal 0 for success and -1 for failure.

## 6.2 The NVECTOR\_PARALLEL implementation

The `NVECTOR_PARALLEL` implementation of the `NVECTOR` module provided with `SUNDIALS` is based on `MPI`. It 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, and a boolean flag `own_data` indicating ownership of the data array `data`.

```

struct _N_VectorContent_Parallel {
    sunindextype local_length;
    sunindextype global_length;
    booleantype own_data;
    realtype *data;
    MPI_Comm comm;
};

```

The header file to include when using this module is `nvector_parallel.h`. The installed module library to link to is `libsundials_nvecparallel.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

The following macros are provided to access the content of a NVECTOR\_PARALLEL vector. The suffix `_P` in the names denotes the distributed memory 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_VectorContent_Parallel`.

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 Tables 6.2, 6.3, and 6.4. Their names are obtained from those in Tables 6.2, 6.3, and 6.4 by appending the suffix `_Parallel` (e.g. `N_VDestroy_Parallel`). The module NVECTOR\_PARALLEL provides the following additional user-callable routines:

- `N_VNew_Parallel`

This function creates and allocates memory for a parallel vector.

```
N_Vector N_VNew_Parallel(MPI_Comm comm,
                        sunindextype local_length,
                        sunindextype 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,
                              sunindextype local_length,
                              sunindextype global_length);
```

- `N_VMake_Parallel`

This function creates and allocates memory for a parallel vector with user-provided data array. (This function does *not* allocate memory for `v_data` itself.)

```
N_Vector N_VMake_Parallel(MPI_Comm comm,
                          sunindextype local_length,
                          sunindextype global_length,
                          realtype *v_data);
```

- `N_VCloneVectorArray_Parallel`

This function creates (by cloning) an array of `count` parallel vectors.

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

- `N_VCloneVectorArrayEmpty_Parallel`

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

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

- `N_VDestroyVectorArray_Parallel`

This function frees memory allocated for the array of `count` variables of type `N_Vector` created with `N_VCloneVectorArray_Parallel` or with `N_VCloneVectorArrayEmpty_Parallel`.

```
void N_VDestroyVectorArray_Parallel(N_Vector *vs, int count);
```

- `N_VGetLength_Parallel`

This function returns the number of vector elements (global vector length).

```
sunindextype N_VGetLength_Parallel(N_Vector v);
```

- `N_VGetLocalLength_Parallel`

This function returns the local vector length.

```
sunindextype N_VGetLocalLength_Parallel(N_Vector v);
```



- `N_VPrint_Parallel`

This function prints the local content of a parallel vector to `stdout`.

```
void N_VPrint_Parallel(N_Vector v);
```

- `N_VPrintFile_Parallel`

This function prints the local content of a parallel vector to `outfile`.

```
void N_VPrintFile_Parallel(N_Vector v, FILE *outfile);
```

### Notes

- When looping over the components of an `N_Vector` `v`, it is more efficient to first obtain the local component array via `v_data = NV_DATA_P(v)` and then access `v_data[i]` within the loop than it is to use `NV_Ith_P(v,i)` within the loop.
- `N_VNewEmpty_Parallel`, `N_VMake_Parallel`, and `N_VCloneVectorArrayEmpty_Parallel` set the field `own_data = SUNFALSE`. `N_VDestroy_Parallel` and `N_VDestroyVectorArray_Parallel` will not attempt to free the pointer `data` for any `N_Vector` with `own_data` set to `SUNFALSE`. 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.



For solvers that include a Fortran interface module, the `NVECTOR_PARALLEL` module also includes a Fortran-callable function `FNVINITP(COMM, code, NLOCAL, NGLOBAL, IER)`, to initialize this `NVECTOR_PARALLEL` module. Here `COMM` is the MPI communicator, `code` is an input solver id (1 for `CVODE`, 2 for `IDA`, 3 for `KINSOL`, 4 for `ARKODE`); `NLOCAL` and `NGLOBAL` are the local and global vector sizes, respectively (declared so as to match C type `long int`); and `IER` is an error return flag equal 0 for success and -1 for failure. NOTE: If the header file `sundials_config.h` defines `SUNDIALS_MPI_COMM_F2C` to be 1 (meaning the MPI implementation used to build `SUNDIALS` includes the `MPI_Comm_f2c` function), then `COMM` can be any valid MPI communicator. Otherwise, `MPI_COMM_WORLD` will be used, so just pass an integer value as a placeholder.



## 6.3 The NVECTOR\_OPENMP implementation

In situations where a user has a multi-core processing unit capable of running multiple parallel threads with shared memory, `SUNDIALS` provides an implementation of `NVECTOR` using OpenMP, called `NVECTOR_OPENMP`, and an implementation using Pthreads, called `NVECTOR_PTHREADS`. Testing has shown that vectors should be of length at least 100,000 before the overhead associated with creating and using the threads is made up by the parallelism in the vector calculations.

The OpenMP `NVECTOR` implementation provided with `SUNDIALS`, `NVECTOR_OPENMP`, 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, a boolean flag `own_data` which specifies the ownership of `data`, and the number of threads. Operations on the vector are threaded using OpenMP.

```
struct _N_VectorContent_OpenMP {
    sunindextype length;
    booleantype own_data;
    realtype *data;
    int num_threads;
};
```

The header file to include when using this module is `nvector_openmp.h`. The installed module library to link to is `libsundials_nvecopenmp.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

The following macros are provided to access the content of an NVECTOR\_OPENMP vector. The suffix `_OMP` in the names denotes the OpenMP version.

- **NV\_CONTENT\_OMP**

This routine gives access to the contents of the OpenMP vector `N_Vector`.

The assignment `v_cont = NV_CONTENT_OMP(v)` sets `v_cont` to be a pointer to the OpenMP `N_Vector` content structure.

Implementation:

```
#define NV_CONTENT_OMP(v) ( (N_VectorContent_OpenMP)(v->content) )
```

- **NV\_OWN\_DATA\_OMP, NV\_DATA\_OMP, NV\_LENGTH\_OMP, NV\_NUM\_THREADS\_OMP**

These macros give individual access to the parts of the content of a OpenMP `N_Vector`.

The assignment `v_data = NV_DATA_OMP(v)` sets `v_data` to be a pointer to the first component of the data for the `N_Vector` `v`. The assignment `NV_DATA_OMP(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_OMP(v)` sets `v_len` to be the length of `v`. On the other hand, the call `NV_LENGTH_OMP(v) = len_v` sets the length of `v` to be `len_v`.

The assignment `v_num_threads = NV_NUM_THREADS_OMP(v)` sets `v_num_threads` to be the number of threads from `v`. On the other hand, the call `NV_NUM_THREADS_OMP(v) = num_threads_v` sets the number of threads for `v` to be `num_threads_v`.

Implementation:

```
#define NV_OWN_DATA_OMP(v) ( NV_CONTENT_OMP(v)->own_data )
```

```
#define NV_DATA_OMP(v) ( NV_CONTENT_OMP(v)->data )
```

```
#define NV_LENGTH_OMP(v) ( NV_CONTENT_OMP(v)->length )
```

```
#define NV_NUM_THREADS_OMP(v) ( NV_CONTENT_OMP(v)->num_threads )
```

- **NV\_Ith\_OMP**

This macro gives access to the individual components of the data array of an `N_Vector`.

The assignment `r = NV_Ith_OMP(v,i)` sets `r` to be the value of the `i`-th component of `v`. The assignment `NV_Ith_OMP(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_OMP(v,i) ( NV_DATA_OMP(v)[i] )
```

The NVECTOR\_OPENMP module defines OpenMP implementations of all vector operations listed in Tables 6.2, 6.3, and 6.4. Their names are obtained from those in Tables 6.2, 6.3, and 6.4 by appending the suffix `_OpenMP` (e.g. `N_VDestroy_OpenMP`). The module NVECTOR\_OPENMP provides the following additional user-callable routines:

- **N\_VNew\_OpenMP**

This function creates and allocates memory for a OpenMP `N_Vector`. Arguments are the vector length and number of threads.

```
N_Vector N_VNew_OpenMP(sunindextype vec_length, int num_threads);
```

- **N\_VNewEmpty\_OpenMP**

This function creates a new OpenMP `N_Vector` with an empty (NULL) data array.

```
N_Vector N_VNewEmpty_OpenMP(sunindextype vec_length, int num_threads);
```

- `N_VMake_OpenMP`

This function creates and allocates memory for an OpenMP vector with user-provided data array. (This function does *not* allocate memory for `v_data` itself.)

```
N_Vector N_VMake_OpenMP(sunindextype vec_length, realtype *v_data,
                        int num_threads);
```

- `N_VCloneVectorArray_OpenMP`

This function creates (by cloning) an array of `count` OpenMP vectors.

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

- `N_VCloneVectorArrayEmpty_OpenMP`

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

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

- `N_VDestroyVectorArray_OpenMP`

This function frees memory allocated for the array of `count` variables of type `N_Vector` created with `N_VCloneVectorArray_OpenMP` or with `N_VCloneVectorArrayEmpty_OpenMP`.

```
void N_VDestroyVectorArray_OpenMP(N_Vector *vs, int count);
```

- `N_VGetLength_OpenMP`

This function returns number of vector elements.

```
sunindextype N_VGetLength_OpenMP(N_Vector v);
```

- `N_VPrint_OpenMP`

This function prints the content of an OpenMP vector to `stdout`.



```
void N_VPrint_OpenMP(N_Vector v);
```

- `N_VPrintFile_OpenMP`

This function prints the content of an OpenMP vector to `outfile`.

```
void N_VPrintFile_OpenMP(N_Vector v, FILE *outfile);
```

## 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_OMP(v)` and then access `v_data[i]` within the loop than it is to use `NV_Ith_OMP(v,i)` within the loop.
- `N_VNewEmpty_OpenMP`, `N_VMake_OpenMP`, and `N_VCloneVectorArrayEmpty_OpenMP` set the field `own_data = SUNFALSE`. `N_VDestroy_OpenMP` and `N_VDestroyVectorArray_OpenMP` will not attempt to free the pointer `data` for any `N_Vector` with `own_data` set to `SUNFALSE`. In such a case, it is the user's responsibility to deallocate the `data` pointer. 
- To maximize efficiency, vector operations in the `NVECTOR_OPENMP` 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. 

For solvers that include a Fortran interface module, the `NVECTOR_OPENMP` module also includes a Fortran-callable function `FNVINITOMP(code, NEQ, NUMTHREADS, IER)`, to initialize this module. Here `code` is an input solver id (1 for `CVODE`, 2 for `IDA`, 3 for `KINSOL`, 4 for `ARKODE`); `NEQ` is the problem size (declared so as to match C type `long int`); `NUMTHREADS` is the number of threads; and `IER` is an error return flag equal 0 for success and -1 for failure.

## 6.4 The NVECTOR\_PTHREADS implementation

In situations where a user has a multi-core processing unit capable of running multiple parallel threads with shared memory, SUNDIALS provides an implementation of NVECTOR using OpenMP, called NVECTOR\_OPENMP, and an implementation using Pthreads, called NVECTOR\_PTHREADS. Testing has shown that vectors should be of length at least 100,000 before the overhead associated with creating and using the threads is made up by the parallelism in the vector calculations.

The Pthreads NVECTOR implementation provided with SUNDIALS, denoted NVECTOR\_PTHREADS, 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, a boolean flag *own\_data* which specifies the ownership of *data*, and the number of threads. Operations on the vector are threaded using POSIX threads (Pthreads).

```
struct _N_VectorContent_Pthreads {
    sunindextype length;
    booleantype own_data;
    realtype *data;
    int num_threads;
};
```

The header file to include when using this module is `nvector_pthreads.h`. The installed module library to link to is `libsundials_nvecpthreads.lib` where *.lib* is typically *.so* for shared libraries and *.a* for static libraries.

The following macros are provided to access the content of an NVECTOR\_PTHREADS vector. The suffix *\_PT* in the names denotes the Pthreads version.

- NV\_CONTENT\_PT

This routine gives access to the contents of the Pthreads vector *N\_Vector*.

The assignment `v_cont = NV_CONTENT_PT(v)` sets *v\_cont* to be a pointer to the Pthreads *N\_Vector* content structure.

Implementation:

```
#define NV_CONTENT_PT(v) ( (N_VectorContent_Pthreads)(v->content) )
```

- NV\_OWN\_DATA\_PT, NV\_DATA\_PT, NV\_LENGTH\_PT, NV\_NUM\_THREADS\_PT

These macros give individual access to the parts of the content of a Pthreads *N\_Vector*.

The assignment `v_data = NV_DATA_PT(v)` sets *v\_data* to be a pointer to the first component of the data for the *N\_Vector* *v*. The assignment `NV_DATA_PT(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_PT(v)` sets *v\_len* to be the length of *v*. On the other hand, the call `NV_LENGTH_PT(v) = len_v` sets the length of *v* to be *len\_v*.

The assignment `v_num_threads = NV_NUM_THREADS_PT(v)` sets *v\_num\_threads* to be the number of threads from *v*. On the other hand, the call `NV_NUM_THREADS_PT(v) = num_threads_v` sets the number of threads for *v* to be *num\_threads\_v*.

Implementation:

```
#define NV_OWN_DATA_PT(v) ( NV_CONTENT_PT(v)->own_data )
#define NV_DATA_PT(v) ( NV_CONTENT_PT(v)->data )
#define NV_LENGTH_PT(v) ( NV_CONTENT_PT(v)->length )
#define NV_NUM_THREADS_PT(v) ( NV_CONTENT_PT(v)->num_threads )
```

- NV\_Ith\_PT

This macro gives access to the individual components of the data array of an *N\_Vector*.

The assignment `r = NV_Ith_PT(v,i)` sets `r` to be the value of the `i`-th component of `v`. The assignment `NV_Ith_PT(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_PT(v,i) ( NV_DATA_PT(v)[i] )
```

The NVECTOR\_PTHREADS module defines Pthreads implementations of all vector operations listed in Tables 6.2, 6.3, and 6.4. Their names are obtained from those in Tables 6.2, 6.3, and 6.4 by appending the suffix `_Pthreads` (e.g. `N_VDestroy_Pthreads`). The module NVECTOR\_PTHREADS provides the following additional user-callable routines:

- **N\_VNew\_Pthreads**

This function creates and allocates memory for a Pthreads `N_Vector`. Arguments are the vector length and number of threads.

```
N_Vector N_VNew_Pthreads(sunindextype vec_length, int num_threads);
```

- **N\_VNewEmpty\_Pthreads**

This function creates a new Pthreads `N_Vector` with an empty (NULL) data array.

```
N_Vector N_VNewEmpty_Pthreads(sunindextype vec_length, int num_threads);
```

- **N\_VMake\_Pthreads**

This function creates and allocates memory for a Pthreads vector with user-provided data array. (This function does *not* allocate memory for `v_data` itself.)

```
N_Vector N_VMake_Pthreads(sunindextype vec_length, realtype *v_data,
                          int num_threads);
```

- **N\_VCloneVectorArray\_Pthreads**

This function creates (by cloning) an array of `count` Pthreads vectors.

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

- **N\_VCloneVectorArrayEmpty\_Pthreads**

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

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

- **N\_VDestroyVectorArray\_Pthreads**

This function frees memory allocated for the array of `count` variables of type `N_Vector` created with `N_VCloneVectorArray_Pthreads` or with `N_VCloneVectorArrayEmpty_Pthreads`.

```
void N_VDestroyVectorArray_Pthreads(N_Vector *vs, int count);
```

- **N\_VGetLength\_Pthreads**

This function returns the number of vector elements.

```
sunindextype N_VGetLength_Pthreads(N_Vector v);
```

- **N\_VPrint\_Pthreads**

This function prints the content of a Pthreads vector to `stdout`.

```
void N_VPrint_Pthreads(N_Vector v);
```

- **N\_VPrintFile\_Pthreads**

This function prints the content of a Pthreads vector to `outfile`.

```
void N_VPrintFile_Pthreads(N_Vector v, FILE *outfile);
```

## 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_PT(v)` and then access `v_data[i]` within the loop than it is to use `NV_Ith_PT(v,i)` within the loop.



- `N_VNewEmpty_Pthreads`, `N_VMake_Pthreads`, and `N_VCloneVectorArrayEmpty_Pthreads` set the field `own_data = SUNFALSE`. `N_VDestroy_Pthreads` and `N_VDestroyVectorArray_Pthreads` will not attempt to free the pointer `data` for any `N_Vector` with `own_data` set to `SUNFALSE`. In such a case, it is the user's responsibility to deallocate the `data` pointer.



- To maximize efficiency, vector operations in the `NVECTOR_PTHREADS` 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.

For solvers that include a Fortran interface module, the `NVECTOR_PTHREADS` module also includes a Fortran-callable function `FNVINITPTS(code, NEQ, NUMTHREADS, IER)`, to initialize this module. Here `code` is an input solver id (1 for `CVODE`, 2 for `IDA`, 3 for `KINSOL`, 4 for `ARKODE`); `NEQ` is the problem size (declared so as to match C type `long int`); `NUMTHREADS` is the number of threads; and `IER` is an error return flag equal 0 for success and -1 for failure.

## 6.5 The NVECTOR\_PARHYP implementation

The `NVECTOR_PARHYP` implementation of the `NVECTOR` module provided with `SUNDIALS` is a wrapper around `hypr`'s `ParVector` class. Most of the vector kernels simply call `hypr` vector operations. The implementation defines the `content` field of `N_Vector` to be a structure containing the global and local lengths of the vector, a pointer to an object of type `hypr_ParVector`, an MPI communicator, and a boolean flag `own_parvector` indicating ownership of the `hypr` parallel vector object `x`.

```
struct _N_VectorContent_ParHyp {
    sunindextype local_length;
    sunindextype global_length;
    booleantype own_parvector;
    MPI_Comm comm;
    hypr_ParVector *x;
};
```

The header file to include when using this module is `nvector_parhyp.h`. The installed module library to link to is `libsundials_nvecparhyp.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

Unlike native `SUNDIALS` vector types, `NVECTOR_PARHYP` does not provide macros to access its member variables. Note that `NVECTOR_PARHYP` requires `SUNDIALS` to be built with MPI support.

The `NVECTOR_PARHYP` module defines implementations of all vector operations listed in Tables 6.2, 6.3, and 6.4, except for `N_VSetArrayPointer` and `N_VGetArrayPointer`, because accessing raw vector data is handled by low-level `hypr` functions. As such, this vector is not available for use with `SUNDIALS` Fortran interfaces. When access to raw vector data is needed, one should extract the `hypr` vector first, and then use `hypr` methods to access the data. Usage examples of `NVECTOR_PARHYP` are provided in the `cvAdvDiff_non_ph.c` example program for `CVODE` [26] and the `ark_diurnal_kry_ph.c` example program for `ARKODE` [31].

The names of `parhyp` methods are obtained from those in Tables 6.2, 6.3, and 6.4 by appending the suffix `_ParHyp` (e.g. `N_VDestroy_ParHyp`). The module `NVECTOR_PARHYP` provides the following additional user-callable routines:

- `N_VNewEmpty_ParHyp`

This function creates a new `parhyp` `N_Vector` with the pointer to the `hypr` vector set to `NULL`.

```
N_Vector N_VNewEmpty_ParHyp(MPI_Comm comm,
                             sunindextype local_length,
                             sunindextype global_length);
```

- **N\_VMake\_ParHyp**

This function creates an `N_Vector` wrapper around an existing *hypre* parallel vector. It does *not* allocate memory for `x` itself.

```
N_Vector N_VMake_ParHyp(hypre_ParVector *x);
```

- **N\_VGetVector\_ParHyp**

This function returns a pointer to the underlying *hypre* vector.

```
hypre_ParVector *N_VGetVector_ParHyp(N_Vector v);
```

- **N\_VCloneVectorArray\_ParHyp**

This function creates (by cloning) an array of `count` parallel vectors.

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

- **N\_VCloneVectorArrayEmpty\_ParHyp**

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

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

- **N\_VDestroyVectorArray\_ParHyp**

This function frees memory allocated for the array of `count` variables of type `N_Vector` created with `N_VCloneVectorArray_ParHyp` or with `N_VCloneVectorArrayEmpty_ParHyp`.

```
void N_VDestroyVectorArray_ParHyp(N_Vector *vs, int count);
```

- **N\_VPrint\_ParHyp**

This function prints the local content of a parhyp vector to `stdout`.

```
void N_VPrint_ParHyp(N_Vector v);
```

- **N\_VPrintFile\_ParHyp**

This function prints the local content of a parhyp vector to `outfile`.

```
void N_VPrintFile_ParHyp(N_Vector v, FILE *outfile);
```

## Notes

- When there is a need to access components of an `N_Vector_ParHyp`, `v`, it is recommended to extract the *hypre* vector via `x_vec = N_VGetVector_ParHyp(v)` and then access components using appropriate *hypre* functions.
- `N_VNewEmpty_ParHyp`, `N_VMake_ParHyp`, and `N_VCloneVectorArrayEmpty_ParHyp` set the field *own\_parvector* to `SUNFALSE`. `N_VDestroy_ParHyp` and `N_VDestroyVectorArray_ParHyp` will not attempt to delete an underlying *hypre* vector for any `N_Vector` with *own\_parvector* set to `SUNFALSE`. In such a case, it is the user's responsibility to delete the underlying vector.
- To maximize efficiency, vector operations in the NVECTOR\_PARHYP implementation that have more than one `N_Vector` argument do not check for consistent internal representations 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.



## 6.6 The NVECTOR\_PETSC implementation

The NVECTOR\_PETSC module is an NVECTOR wrapper around the PETSc vector. It defines the *content* field of a `N_Vector` to be a structure containing the global and local lengths of the vector, a pointer to the PETSc vector, an MPI communicator, and a boolean flag *own\_data* indicating ownership of the wrapped PETSc vector.

```
struct _N_VectorContent_Petsc {
    sunindextype local_length;
    sunindextype global_length;
    booleantype own_data;
    Vec *pvec;
    MPI_Comm comm;
};
```

The header file to include when using this module is `nvector_petsc.h`. The installed module library to link to is `libsundials_nvecpetsc.lib` where *.lib* is typically *.so* for shared libraries and *.a* for static libraries.

Unlike native SUNDIALS vector types, NVECTOR\_PETSC does not provide macros to access its member variables. Note that NVECTOR\_PETSC requires SUNDIALS to be built with MPI support.

The NVECTOR\_PETSC module defines implementations of all vector operations listed in Tables 6.2, 6.3, and 6.4, except for `N_VGetArrayPointer` and `N_VSetArrayPointer`. As such, this vector cannot be used with SUNDIALS Fortran interfaces. When access to raw vector data is needed, it is recommended to extract the PETSc vector first, and then use PETSc methods to access the data. Usage examples of NVECTOR\_PETSC are provided in example programs for IDA [25].

The names of vector operations are obtained from those in Tables 6.2, 6.3, and 6.4 by appending the suffix `_Petsc` (e.g. `N_VDestroy_Petsc`). The module NVECTOR\_PETSC provides the following additional user-callable routines:

- `N_VNewEmpty_Petsc`

This function creates a new NVECTOR wrapper with the pointer to the wrapped PETSc vector set to (NULL). It is used by the `N_VMake_Petsc` and `N_VClone_Petsc` implementations.

```
N_Vector N_VNewEmpty_Petsc(MPI_Comm comm,
                           sunindextype local_length,
                           sunindextype global_length);
```

- `N_VMake_Petsc`

This function creates and allocates memory for an NVECTOR\_PETSC wrapper around a user-provided PETSc vector. It does *not* allocate memory for the vector `pvec` itself.

```
N_Vector N_VMake_Petsc(Vec *pvec);
```

- `N_VGetVector_Petsc`

This function returns a pointer to the underlying PETSc vector.

```
Vec *N_VGetVector_Petsc(N_Vector v);
```

- `N_VCloneVectorArray_Petsc`

This function creates (by cloning) an array of `count` NVECTOR\_PETSC vectors.

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



- `N_VCloneVectorArrayEmpty_Petsc`

This function creates (by cloning) an array of `count` `NVECTOR_PETSC` vectors, each with pointers to PETSc vectors set to `(NULL)`.

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

- `N_VDestroyVectorArray_Petsc`

This function frees memory allocated for the array of `count` variables of type `N_Vector` created with `N_VCloneVectorArray_Petsc` or with `N_VCloneVectorArrayEmpty_Petsc`.

```
void N_VDestroyVectorArray_Petsc(N_Vector *vs, int count);
```

- `N_VPrint_Petsc`

This function prints the global content of a wrapped PETSc vector to `stdout`.

```
void N_VPrint_Petsc(N_Vector v);
```

- `N_VPrintFile_Petsc`

This function prints the global content of a wrapped PETSc vector to `fname`.

```
void N_VPrintFile_Petsc(N_Vector v, const char fname[]);
```

#### Notes

- When there is a need to access components of an `N_Vector_Petsc`, `v`, it is recommended to extract the PETSc vector via `x_vec = N_VGetVector_Petsc(v)` and then access components using appropriate PETSc functions.
- The functions `N_VNewEmpty_Petsc`, `N_VMake_Petsc`, and `N_VCloneVectorArrayEmpty_Petsc` set the field `own_data` to `SUNFALSE`. `N_VDestroy_Petsc` and `N_VDestroyVectorArray_Petsc` will not attempt to free the pointer `pvec` for any `N_Vector` with `own_data` set to `SUNFALSE`. In such a case, it is the user's responsibility to deallocate the `pvec` pointer.
- To maximize efficiency, vector operations in the `NVECTOR_PETSC` implementation that have more than one `N_Vector` argument do not check for consistent internal representations 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.



## 6.7 The NVECTOR\_CUDA implementation

The `NVECTOR_CUDA` module is an experimental `NVECTOR` implementation in the CUDA language. The module allows for SUNDIALS vector kernels to run on GPU devices. It is intended for users who are already familiar with CUDA and GPU programming. Building this vector module requires a CUDA compiler and, by extension, a C++ compiler. The class `Vector` in namespace `suncudavec` manages vector data layout:

```
template <class T, class I>
class Vector {
    I size_;
    I mem_size_;
    T* h_vec_;
    T* d_vec_;
    ThreadPartitioning<T, I>* partStream_;
    ThreadPartitioning<T, I>* partReduce_;
    bool ownPartitioning_;

    ...
};
```

The class members are vector size (length), size of the vector data memory block, pointers to vector data on the host and the device, pointers to `ThreadPartitioning` implementations that handle thread partitioning for streaming and reduction vector kernels, and a boolean flag that signals if the vector owns the thread partitioning. The class `Vector` inherits from the empty structure

```
struct _N_VectorContent_Cuda {
};
```

to interface the C++ class with the NVECTOR C code. When instantiated, the class `Vector` will allocate memory on both the host and the device. Due to the rapid progress of CUDA development, we expect that the `suncudavec::Vector` class will change frequently in future SUNDIALS releases. The code is structured so that it can tolerate significant changes in the `suncudavec::Vector` class without requiring changes to the user API.

The NVECTOR\_CUDA module can be utilized for single-node parallelism or in a distributed context with MPI. The header file to include when using this module for single-node parallelism is `nvector_cuda.h`. The header file to include when using this module in the distributed case is `nvector_mpicuda.h`. Note that only the NVECTOR\_CUDA constructor signature differs between the two header files. The installed module libraries to link to are `libsundials_nveccuda.lib` in the single-node case, or `libsundials_nvecmpicuda.lib` in the distributed case. Only one of these libraries may be linked to when creating an executable or library. SUNDIALS must be built with MPI support if the distributed library is desired. The extension, `.lib`, is typically `.so` for shared libraries and `.a` for static libraries.

Unlike other native SUNDIALS vector types, NVECTOR\_CUDA does not provide macros to access its member variables. Instead, user should use the accessor functions in the namespace `suncudavec`.

- `getDevData(N_Vector v)`

This function takes an `N_Vector` as an argument and returns a raw pointer to the vector data on the device (GPU). It is the user's responsibility to ensure that the vector argument is of the correct `N_Vector` type.

- `getHostData(N_Vector v)`

This function takes a `N_Vector` as an argument and returns a raw pointer to the vector data on the host (CPU memory). It is the user's responsibility to ensure that the vector argument is of the correct `N_Vector` type.

- `getSize(N_Vector v)`

Returns the vector's local length.

- `getGlobalSize(N_Vector v)`

Returns the vector's global length.

- `getMPIComm(N_Vector v)`

Takes a `N_Vector` as an argument and returns a sundials communicator of type

The NVECTOR\_CUDA module defines implementations of all vector operations listed in Tables 6.2, 6.3, and 6.4, except for `N_VGetArrayPointer` and `N_VSetArrayPointer`. As such, this vector cannot be used with the SUNDIALS Fortran interfaces, nor with the SUNDIALS direct solvers and preconditioners. Instead, the NVECTOR\_CUDA module provides separate functions to access data on the host and on the device. It also provides methods for copying from the host to the device and vice versa. Usage examples of NVECTOR\_CUDA are provided in some example programs for CVODE [26].

The names of vector operations are obtained from those in Tables 6.2, 6.3, and 6.4 by appending the suffix `_Cuda` (e.g. `N_VDestroy_Cuda`). The module NVECTOR\_CUDA provides the following additional user-callable routines:

- **N\_VNew\_Cuda**

*Note: this function signature is defined in the header `nvector_mpicuda.h` and should be used when using this module in a distributed context.* This function creates and allocates memory for a CUDA `N_Vector`. The memory is allocated on both host and device. Its arguments are local and global vector lengths, as well as the MPI communicator. Use this constructor with the `libsundials_nvecmpicuda.lib` library.

```
N_Vector N_VNew_Cuda(MPI_Comm comm,
                     sunindextype local_length,
                     sunindextype global_length);
```

- **N\_VNew\_Cuda**

*Note: this function signature is defined in the header `nvector_cuda.h` and should be used when using this module for single-node parallelism.* This function creates and allocates memory for a CUDA `N_Vector` on a single node. The memory is allocated on both host and device. Its only argument is vector length. Use this constructor with the `libsundials_nveccuda.lib` library.

```
N_Vector N_VNew_Cuda(sunindextype length);
```

- **N\_VNewEmpty\_Cuda**

This function creates a new NVECTOR wrapper with the pointer to the wrapped CUDA vector set to (NULL). It is used by the `N_VNew_Cuda`, `N_VMake_Cuda`, and `N_VClone_Cuda` implementations.

```
N_Vector N_VNewEmpty_Cuda(sunindextype vec_length);
```

- **N\_VMake\_Cuda**

This function creates and allocates memory for an NVECTOR\_CUDA wrapper around a user-provided `suncudavec::Vector` class. Its only argument is of type `N_VectorContent_Cuda`, which is the pointer to the class.

```
N_Vector N_VMake_Cuda(N_VectorContent_Cuda c);
```

- **N\_VGetLength\_Cuda**

This function returns the length of the vector.

```
sunindextype N_VGetLength_Cuda(N_Vector v);
```

- **N\_VGetHostArrayPointer\_Cuda**

This function returns a pointer to the vector data on the host.

```
realtype *N_VGetHostArrayPointer_Cuda(N_Vector v);
```

- **N\_VGetDeviceArrayPointer\_Cuda**

This function returns a pointer to the vector data on the device.

```
realtype *N_VGetDeviceArrayPointer_Cuda(N_Vector v);
```

- **N\_VCopyToDevice\_Cuda**

This function copies host vector data to the device.

```
realtype *N_VCopyToDevice_Cuda(N_Vector v);
```

- **N\_VCopyFromDevice\_Cuda**

This function copies vector data from the device to the host.

```
realtype *N_VCopyFromDevice_Cuda(N_Vector v);
```

- `N_VPrint_Cuda`

This function prints the content of a CUDA vector to `stdout`.

```
void N_VPrint_Cuda(N_Vector v);
```

- `N_VPrintFile_Cuda`

This function prints the content of a CUDA vector to `outfile`.

```
void N_VPrintFile_Cuda(N_Vector v, FILE *outfile);
```

## Notes

- When there is a need to access components of an `N_Vector_Cuda`, `v`, it is recommended to use functions `N_VGetDeviceArrayPointer_Cuda` or `N_VGetHostArrayPointer_Cuda`.



- To maximize efficiency, vector operations in the `NVECTOR_CUDA` implementation that have more than one `N_Vector` argument do not check for consistent internal representations 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.

## 6.8 The NVECTOR\_RAJA implementation

The `NVECTOR_RAJA` module is an experimental `NVECTOR` implementation using the `RAJA` hardware abstraction layer. In this implementation, `RAJA` allows for `SUNDIALS` vector kernels to run on GPU devices. The module is intended for users who are already familiar with `RAJA` and GPU programming. Building this vector module requires a C++11 compliant compiler and a CUDA software development toolkit. Besides the CUDA backend, `RAJA` has other backends such as serial, OpenMP, and OpenAC. These backends are not used in this `SUNDIALS` release. Class `Vector` in namespace `sunrajavec` manages the vector data layout:

```
template <class T, class I>
class Vector {
    I size_;
    I mem_size_;
    T* h_vec_;
    T* d_vec_;
    ...
};
```

The class members are: vector size (length), size of the vector data memory block, and pointers to vector data on the host and on the device. The class `Vector` inherits from an empty structure

```
struct _N_VectorContent_Raja {
};
```

to interface the C++ class with the `NVECTOR` C code. When instantiated, the class `Vector` will allocate memory on both the host and the device. Due to the rapid progress of `RAJA` development, we expect that the `sunrajavec::Vector` class will change frequently in future `SUNDIALS` releases. The code is structured so that it can tolerate significant changes in the `sunrajavec::Vector` class without requiring changes to the user API.

The `NVECTOR_RAJA` module can be utilized for single-node parallelism or in a distributed context with MPI. The header file to include when using this module for single-node parallelism is `nvector_raja.h`. The header file to include when using this module in the distributed case is `nvector_mpiraja.h`. Note that only the `NVECTOR_RAJA` constructor signature differs between the two header files. The installed module libraries to link to are `libsundials_nvecraja.lib` in the single-node case, or `libsundials_nvecmpicudaraja.lib` in the distributed case. Only one of

these libraries may be linked to when creating an executable or library. SUNDIALS must be built with MPI support if the distributed library is desired. The extension, *.lib*, is typically *.so* for shared libraries and *.a* for static libraries.

Unlike other native SUNDIALS vector types, NVECTOR\_RAJA does not provide macros to access its member variables. Instead, user should use the accessor functions in the namespace `sunrajavec`.

- `getDevData(N_Vector v)`

This function takes a `N_Vector` as an argument and returns a raw pointer to the vector data on the device (GPU). It is the user's responsibility to ensure that the vector argument is of the correct `N_Vector` type.

- `getHostData(N_Vector v)`

This function takes a `N_Vector` as an argument and returns a raw pointer to the vector data on the host (CPU memory). It is the user's responsibility to ensure that the vector argument is of the correct `N_Vector` type.

- `getSize(N_Vector v)`

Returns the vector's local length.

- `getGlobalSize(N_Vector v)`

Returns the vector's global length.

- `getMPIComm(N_Vector v)`

Takes a `N_Vector` as an argument and returns a sundials communicator of type `SUNDIALS_Comm`.

The NVECTOR\_RAJA module defines the implementations of all vector operations listed in Tables 6.2, 6.3, and 6.4, except for `N_VDotProdMulti`, `N_VWrmsNormVectorArray`, and `N_VWrmsNormMaskVectorArray` as support for arrays of reduction vectors is not yet supported in RAJA. These function will be added to the NVECTOR\_RAJA implementation in the future. Additionally the vector operations `N_VGetArrayPointer` and `N_VSetArrayPointer` are not implemented by the RAJA vector. As such, this vector cannot be used with the SUNDIALS Fortran interfaces, nor with the SUNDIALS direct solvers and preconditioners. The NVECTOR\_RAJA module provides separate functions to access data on the host and on the device. It also provides methods for copying data from the host to the device and vice versa. Usage examples of NVECTOR\_RAJA are provided in some example programs for CVODE [26].

The names of vector operations are obtained from those in Tables 6.2, 6.3, and 6.4, by appending the suffix `_Raja` (e.g. `N_VDestroy_Raja`). The module NVECTOR\_RAJA provides the following additional user-callable routines:

- `N_VNew_Raja`

*Note: this function signature is defined in the header `nvector_mpiraja.h` and should be used when using this module in a distributed context.* This function creates and allocates memory for a RAJA `N_Vector`. The memory is allocated on both host and device. Its arguments are local and global vector lengths, as well as the MPI communicator. Use this constructor with the `libsundials_nvecmpicudaraja.lib` library.

```
N_Vector N_VNew_Raja(MPI_Comm comm,
                     sunindextype local_length,
                     sunindextype global_length);
```

- `N_VNew_Raja`

*Note: this function signature is defined in the header `nvector_raja.h` and should be used when using this module for single-node parallelism.* This function creates and allocates memory for a RAJA `N_Vector` on a single node. The memory is allocated on both host and device. Its only argument is vector length. Use this constructor with the `libsundials_nveccudaraja.lib` library.

```
N_Vector N_VNew_Raja(sunindextype length);
```

- **N\_VNewEmpty\_Raja**

This function creates a new NVECTOR wrapper with the pointer to the wrapped RAJA vector set to (NULL). It is used by the `N_VNew_Raja`, `N_VMake_Raja`, and `N_VClone_Raja` implementations.

```
N_Vector N_VNewEmpty_Raja(sunindextype vec_length);
```

- **N\_VMake\_Raja**

This function creates and allocates memory for an NVECTOR\_RAJA wrapper around a user-provided `sunrajavec::Vector` class. Its only argument is of type `N_VectorContent_Raja`, which is the pointer to the class.

```
N_Vector N_VMake_Raja(N_VectorContent_Raja c);
```

- **N\_VGetLength\_Raja**

This function returns the length of the vector.

```
sunindextype N_VGetLength_Raja(N_Vector v);
```

- **N\_VGetHostArrayPointer\_Raja**

This function returns a pointer to the vector data on the host.

```
realtype *N_VGetHostArrayPointer_Raja(N_Vector v);
```

- **N\_VGetDeviceArrayPointer\_Raja**

This function returns a pointer to the vector data on the device.

```
realtype *N_VGetDeviceArrayPointer_Raja(N_Vector v);
```

- **N\_VCopyToDevice\_Raja**

This function copies host vector data to the device.

```
realtype *N_VCopyToDevice_Raja(N_Vector v);
```

- **N\_VCopyFromDevice\_Raja**

This function copies vector data from the device to the host.

```
realtype *N_VCopyFromDevice_Raja(N_Vector v);
```

- **N\_VPrint\_Raja**

This function prints the content of a RAJA vector to `stdout`.

```
void N_VPrint_Raja(N_Vector v);
```

- **N\_VPrintFile\_Raja**

This function prints the content of a RAJA vector to `outfile`.

```
void N_VPrintFile_Raja(N_Vector v, FILE *outfile);
```

## Notes

- When there is a need to access components of an `N_Vector_Raja`, `v`, it is recommended to use functions `N_VGetDeviceArrayPointer_Raja` or `N_VGetHostArrayPointer_Raja`.



- To maximize efficiency, vector operations in the NVECTOR\_RAJA implementation that have more than one `N_Vector` argument do not check for consistent internal representations 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.

## 6.9 NVECTOR Examples

There are `NVector` examples that may be installed for the implementations provided with SUNDIALS. Each implementation makes use of the functions in `test_nvector.c`. These example functions show simple usage of the `NVector` family of functions. The input to the examples are the vector length, number of threads (if threaded implementation), and a print timing flag.

The following is a list of the example functions in `test_nvector.c`:

- `Test_N_VClone`: Creates clone of vector and checks validity of clone.
- `Test_N_VCloneEmpty`: Creates clone of empty vector and checks validity of clone.
- `Test_N_VCloneVectorArray`: Creates clone of vector array and checks validity of cloned array.
- `Test_N_VCloneVectorArray`: Creates clone of empty vector array and checks validity of cloned array.
- `Test_N_VGetArrayPointer`: Get array pointer.
- `Test_N_VSetArrayPointer`: Allocate new vector, set pointer to new vector array, and check values.
- `Test_N_VLinearSum` Case 1a: Test  $y = x + y$
- `Test_N_VLinearSum` Case 1b: Test  $y = -x + y$
- `Test_N_VLinearSum` Case 1c: Test  $y = ax + y$
- `Test_N_VLinearSum` Case 2a: Test  $x = x + y$
- `Test_N_VLinearSum` Case 2b: Test  $x = x - y$
- `Test_N_VLinearSum` Case 2c: Test  $x = x + by$
- `Test_N_VLinearSum` Case 3: Test  $z = x + y$
- `Test_N_VLinearSum` Case 4a: Test  $z = x - y$
- `Test_N_VLinearSum` Case 4b: Test  $z = -x + y$
- `Test_N_VLinearSum` Case 5a: Test  $z = x + by$
- `Test_N_VLinearSum` Case 5b: Test  $z = ax + y$
- `Test_N_VLinearSum` Case 6a: Test  $z = -x + by$
- `Test_N_VLinearSum` Case 6b: Test  $z = ax - y$
- `Test_N_VLinearSum` Case 7: Test  $z = a(x + y)$
- `Test_N_VLinearSum` Case 8: Test  $z = a(x - y)$
- `Test_N_VLinearSum` Case 9: Test  $z = ax + by$
- `Test_N_VConst`: Fill vector with constant and check result.
- `Test_N_VProd`: Test vector multiply:  $z = x * y$
- `Test_N_VDiv`: Test vector division:  $z = x / y$
- `Test_N_VScale`: Case 1: scale:  $x = cx$
- `Test_N_VScale`: Case 2: copy:  $z = x$

- **Test\_N\_VScale**: Case 3: negate:  $z = -x$
- **Test\_N\_VScale**: Case 4: combination:  $z = cx$
- **Test\_N\_VAbs**: Create absolute value of vector.
- **Test\_N\_VAddConst**: add constant vector:  $z = c + x$
- **Test\_N\_VDotProd**: Calculate dot product of two vectors.
- **Test\_N\_VMaxNorm**: Create vector with known values, find and validate the max norm.
- **Test\_N\_VWrmsNorm**: Create vector of known values, find and validate the weighted root mean square.
- **Test\_N\_VWrmsNormMask**: Create vector of known values, find and validate the weighted root mean square using all elements except one.
- **Test\_N\_VMin**: Create vector, find and validate the min.
- **Test\_N\_VWL2Norm**: Create vector, find and validate the weighted Euclidean L2 norm.
- **Test\_N\_VL1Norm**: Create vector, find and validate the L1 norm.
- **Test\_N\_VCompare**: Compare vector with constant returning and validating comparison vector.
- **Test\_N\_VInvTest**: Test  $z[i] = 1 / x[i]$
- **Test\_N\_VConstrMask**: Test mask of vector  $x$  with vector  $c$ .
- **Test\_N\_VMinQuotient**: Fill two vectors with known values. Calculate and validate minimum quotient.
- **Test\_N\_VLinearCombination** Case 1a: Test  $x = a x$
- **Test\_N\_VLinearCombination** Case 1b: Test  $z = a x$
- **Test\_N\_VLinearCombination** Case 2a: Test  $x = a x + b y$
- **Test\_N\_VLinearCombination** Case 2b: Test  $z = a x + b y$
- **Test\_N\_VLinearCombination** Case 3a: Test  $x = x + a y + b z$
- **Test\_N\_VLinearCombination** Case 3b: Test  $x = a x + b y + c z$
- **Test\_N\_VLinearCombination** Case 3c: Test  $w = a x + b y + c z$
- **Test\_N\_VScaleAddMulti** Case 1a:  $y = a x + y$
- **Test\_N\_VScaleAddMulti** Case 1b:  $z = a x + y$
- **Test\_N\_VScaleAddMulti** Case 2a:  $Y[i] = c[i] x + Y[i]$ ,  $i = 1,2,3$
- **Test\_N\_VScaleAddMulti** Case 2b:  $Z[i] = c[i] x + Y[i]$ ,  $i = 1,2,3$
- **Test\_N\_VDotProdMulti** Case 1: Calculate the dot product of two vectors
- **Test\_N\_VDotProdMulti** Case 2: Calculate the dot product of one vector with three other vectors in a vector array.
- **Test\_N\_VLinearSumVectorArray** Case 1:  $z = a x + b y$
- **Test\_N\_VLinearSumVectorArray** Case 2a:  $Z[i] = a X[i] + b Y[i]$
- **Test\_N\_VLinearSumVectorArray** Case 2b:  $X[i] = a X[i] + b Y[i]$



- Test\_N\_VLinearSumVectorArray Case 2c:  $Y[i] = a X[i] + b Y[i]$
- Test\_N\_VScaleVectorArray Case 1a:  $y = c y$
- Test\_N\_VScaleVectorArray Case 1b:  $z = c y$
- Test\_N\_VScaleVectorArray Case 2a:  $Y[i] = c[i] Y[i]$
- Test\_N\_VScaleVectorArray Case 2b:  $Z[i] = c[i] Y[i]$
- Test\_N\_VScaleVectorArray Case 1a:  $z = c$
- Test\_N\_VScaleVectorArray Case 1b:  $Z[i] = c$
- Test\_N\_VWrmsNormVectorArray Case 1a: Create a vector of know values, find and validate the weighted root mean square norm.
- Test\_N\_VWrmsNormVectorArray Case 1b: Create a vector array of three vectors of know values, find and validate the weighted root mean square norm of each.
- Test\_N\_VWrmsNormMaskVectorArray Case 1a: Create a vector of know values, find and validate the weighted root mean square norm using all elements except one.
- Test\_N\_VWrmsNormMaskVectorArray Case 1b: Create a vector array of three vectors of know values, find and validate the weighted root mean square norm of each using all elements except one.
- Test\_N\_VScaleAddMultiVectorArray Case 1a:  $y = a x + y$
- Test\_N\_VScaleAddMultiVectorArray Case 1b:  $z = a x + y$
- Test\_N\_VScaleAddMultiVectorArray Case 2a:  $Y[j][0] = a[j] X[0] + Y[j][0]$
- Test\_N\_VScaleAddMultiVectorArray Case 2b:  $Z[j][0] = a[j] X[0] + Y[j][0]$
- Test\_N\_VScaleAddMultiVectorArray Case 3a:  $Y[0][i] = a[0] X[i] + Y[0][i]$
- Test\_N\_VScaleAddMultiVectorArray Case 3b:  $Z[0][i] = a[0] X[i] + Y[0][i]$
- Test\_N\_VScaleAddMultiVectorArray Case 4a:  $Y[j][i] = a[j] X[i] + Y[j][i]$
- Test\_N\_VScaleAddMultiVectorArray Case 4b:  $Z[j][i] = a[j] X[i] + Y[j][i]$
- Test\_N\_VLinearCombinationVectorArray Case 1a:  $x = a x$
- Test\_N\_VLinearCombinationVectorArray Case 1b:  $z = a x$
- Test\_N\_VLinearCombinationVectorArray Case 2a:  $x = a x + b y$
- Test\_N\_VLinearCombinationVectorArray Case 2b:  $z = a x + b y$
- Test\_N\_VLinearCombinationVectorArray Case 3a:  $x = a x + b y + c z$
- Test\_N\_VLinearCombinationVectorArray Case 3b:  $w = a x + b y + c z$
- Test\_N\_VLinearCombinationVectorArray Case 4a:  $X[0][i] = c[0] X[0][i]$
- Test\_N\_VLinearCombinationVectorArray Case 4b:  $Z[i] = c[0] X[0][i]$
- Test\_N\_VLinearCombinationVectorArray Case 5a:  $X[0][i] = c[0] X[0][i] + c[1] X[1][i]$
- Test\_N\_VLinearCombinationVectorArray Case 5b:  $Z[i] = c[0] X[0][i] + c[1] X[1][i]$
- Test\_N\_VLinearCombinationVectorArray Case 6a:  $X[0][i] = X[0][i] + c[1] X[1][i] + c[2] X[2][i]$
- Test\_N\_VLinearCombinationVectorArray Case 6b:  $X[0][i] = c[0] X[0][i] + c[1] X[1][i] + c[2] X[2][i]$
- Test\_N\_VLinearCombinationVectorArray Case 6c:  $Z[i] = c[0] X[0][i] + c[1] X[1][i] + c[2] X[2][i]$

## 6.10 NVECTOR functions used by IDA

In Table 6.5 below, we list the vector functions used in the NVECTOR module used by the IDA package. The table also shows, for each function, which of the code modules uses the function. The IDA column shows function usage within the main integrator module, while the remaining columns show function usage within the IDALS linear solvers interface, the IDABBDPRE preconditioner module, and the FIDA module.

At this point, we should emphasize that the IDA user does not need to know anything about the usage of vector functions by the IDA code modules in order to use IDA. The information is presented as an implementation detail for the interested reader.

Table 6.5: List of vector functions usage by IDA code modules

	IDA	IDALS	IDABBDPRE	FIDA
N_VGetVectorID				
N_VClone	✓	✓	✓	✓
N_VCloneEmpty		1		✓
N_VDestroy	✓	✓	✓	✓
N_VSpace	✓	2		
N_VGetArrayPointer		1	✓	✓
N_VSetArrayPointer		1		✓
N_VLinearSum	✓	✓		
N_VConst	✓	✓		
N_VProd	✓			
N_VDiv	✓			
N_VScale	✓	✓	✓	
N_VAbs	✓			
N_VInv	✓			
N_VAddConst	✓			
N_VDotProd		✓		
N_VMaxNorm	✓			
N_VWrmsNorm	✓			
N_VMin	✓			
N_VMinQuotient	✓			
N_VConstrMask	✓			
N_VWrmsNormMask	✓			
N_VCompare	✓			
N_VLinearCombination	✓			
N_VScaleAddMulti	✓			
N_VDotProdMulti		3		
N_VLinearSumVectorArray	✓			
N_VScaleVectorArray	✓			

Special cases (numbers match markings in table):

1. These routines are only required if an internal difference-quotient routine for constructing dense or band Jacobian matrices is used.
2. This routine is optional, and is only used in estimating space requirements for IDA modules for user feedback.

3. The optional function `N_VDotProdMulti` is only used when Classical Gram-Schmidt is enabled with SPGMR or SPFGMR. The remaining operations from Tables 6.3 and 6.4 not listed above are unused and a user-supplied NVECTOR module for IDA could omit these operations.

Of the functions listed in Table 6.2, `N_VWL2Norm`, `N_VL1Norm`, and `N_VInvTest` are *not* used by IDA. Therefore a user-supplied NVECTOR module for IDA could omit these three functions.



## Chapter 7

# Description of the SUNMatrix module

For problems that involve direct methods for solving linear systems, the SUNDIALS solvers not only operate on generic vectors, but also on generic matrices (of type `SUNMatrix`), through a set of operations defined by the particular SUNMATRIX implementation. Users can provide their own specific implementation of the SUNMATRIX module, particularly in cases where they provide their own `NVECTOR` and/or linear solver modules, and require matrices that are compatible with those implementations. Alternately, we provide three SUNMATRIX implementations: dense, banded, and sparse. The generic operations are described below, and descriptions of the implementations provided with SUNDIALS follow.

The generic `SUNMatrix` type has been modeled after the object-oriented style of the generic `N_Vector` type. Specifically, a generic `SUNMatrix` is a pointer to a structure that has an implementation-dependent *content* field containing the description and actual data of the matrix, and an *ops* field pointing to a structure with generic matrix operations. The type `SUNMatrix` is defined as

```
typedef struct _generic_SUNMatrix *SUNMatrix;
```

```
struct _generic_SUNMatrix {  
    void *content;  
    struct _generic_SUNMatrix_Ops *ops;  
};
```

The `_generic_SUNMatrix_Ops` structure is essentially a list of pointers to the various actual matrix operations, and is defined as

```
struct _generic_SUNMatrix_Ops {  
    SUNMatrix_ID (*getid)(SUNMatrix);  
    SUNMatrix (*clone)(SUNMatrix);  
    void (*destroy)(SUNMatrix);  
    int (*zero)(SUNMatrix);  
    int (*copy)(SUNMatrix, SUNMatrix);  
    int (*scaleadd)(realtype, SUNMatrix, SUNMatrix);  
    int (*scaleaddi)(realtype, SUNMatrix);  
    int (*matvec)(SUNMatrix, N_Vector, N_Vector);  
    int (*space)(SUNMatrix, long int*, long int*);  
};
```

The generic SUNMATRIX module defines and implements the matrix operations acting on `SUNMatrix` objects. These routines are nothing but wrappers for the matrix operations defined by a particular SUNMATRIX implementation, which are accessed through the *ops* field of the `SUNMatrix` structure. To

Table 7.1: Identifiers associated with matrix kernels supplied with SUNDIALS.

Matrix ID	Matrix type	ID Value
SUNMATRIX_DENSE	Dense $M \times N$ matrix	0
SUNMATRIX_BAND	Band $M \times M$ matrix	1
SUNMATRIX_SPARSE	Sparse (CSR or CSC) $M \times N$ matrix	2
SUNMATRIX_CUSTOM	User-provided custom matrix	3

illustrate this point we show below the implementation of a typical matrix operation from the generic SUNMATRIX module, namely `SUNMatZero`, which sets all values of a matrix `A` to zero, returning a flag denoting a successful/failed operation:

```
int SUNMatZero(SUNMatrix A)
{
    return((int) A->ops->zero(A));
}
```

Table 7.2 contains a complete list of all matrix operations defined by the generic SUNMATRIX module. A particular implementation of the SUNMATRIX module must:

- Specify the *content* field of the `SUNMatrix` object.
- Define and implement a minimal subset of the matrix operations. See the documentation for each SUNDIALS solver to determine which SUNMATRIX operations they require.

Note that the names of these routines should be unique to that implementation in order to permit using more than one SUNMATRIX module (each with different `SUNMatrix` internal data representations) in the same code.

- Define and implement user-callable constructor and destructor routines to create and free a `SUNMatrix` with the new *content* field and with *ops* pointing to the new matrix operations.
- Optionally, define and implement additional user-callable routines acting on the newly defined `SUNMatrix` (e.g., a routine to print the content for debugging purposes).
- Optionally, provide accessor macros or functions as needed for that particular implementation to access different parts of the *content* field of the newly defined `SUNMatrix`.

Each SUNMATRIX implementation included in SUNDIALS has a unique identifier specified in enumeration and shown in Table 7.1. It is recommended that a user-supplied SUNMATRIX implementation use the `SUNMATRIX_CUSTOM` identifier.

Table 7.2: Description of the `SUNMatrix` operations

Name	Usage and Description
SUNMatGetID	<code>id = SUNMatGetID(A);</code> Returns the type identifier for the matrix <code>A</code> . It is used to determine the matrix implementation type (e.g. dense, banded, sparse, . . .) from the abstract <code>SUNMatrix</code> interface. This is used to assess compatibility with SUNDIALS-provided linear solver implementations. Returned values are given in the Table 7.1.
<i>continued on next page</i>	

Name	Usage and Description
SUNMatClone	<pre>B = SUNMatClone(A);</pre> <p>Creates a new <b>SUNMatrix</b> of the same type as an existing matrix <b>A</b> and sets the <i>ops</i> field. It does not copy the matrix, but rather allocates storage for the new matrix.</p>
SUNMatDestroy	<pre>SUNMatDestroy(A);</pre> <p>Destroys the <b>SUNMatrix</b> <b>A</b> and frees memory allocated for its internal data.</p>
SUNMatSpace	<pre>ier = SUNMatSpace(A, &amp;lrw, &amp;liw);</pre> <p>Returns the storage requirements for the matrix <b>A</b>. <b>lrw</b> is a <b>long int</b> containing the number of realtype words and <b>liw</b> is a <b>long int</b> containing the number of integer words. The return value is an integer flag denoting success/failure of the operation.</p> <p>This function is advisory only, for use in determining a user's total space requirements; it could be a dummy function in a user-supplied <b>SUNMATRIX</b> module if that information is not of interest.</p>
SUNMatZero	<pre>ier = SUNMatZero(A);</pre> <p>Performs the operation <math>A_{ij} = 0</math> for all entries of the matrix <i>A</i>. The return value is an integer flag denoting success/failure of the operation.</p>
SUNMatCopy	<pre>ier = SUNMatCopy(A,B);</pre> <p>Performs the operation <math>B_{ij} = A_{i,j}</math> for all entries of the matrices <i>A</i> and <i>B</i>. The return value is an integer flag denoting success/failure of the operation.</p>
SUNMatScaleAdd	<pre>ier = SUNMatScaleAdd(c, A, B);</pre> <p>Performs the operation <math>A = cA + B</math>. The return value is an integer flag denoting success/failure of the operation.</p>
SUNMatScaleAddI	<pre>ier = SUNMatScaleAddI(c, A);</pre> <p>Performs the operation <math>A = cA + I</math>. The return value is an integer flag denoting success/failure of the operation.</p>
SUNMatMatvec	<pre>ier = SUNMatMatvec(A, x, y);</pre> <p>Performs the matrix-vector product operation, <math>y = Ax</math>. It should only be called with vectors <b>x</b> and <b>y</b> that are compatible with the matrix <b>A</b> – both in storage type and dimensions. The return value is an integer flag denoting success/failure of the operation.</p>

We note that not all `SUNMATRIX` types are compatible with all `NVECTOR` types provided with `SUNDIALS`. This is primarily due to the need for compatibility within the `SUNMatMatvec` routine; however, compatibility between `SUNMATRIX` and `NVECTOR` implementations is more crucial when considering their interaction within `SUNLINSOL` objects, as will be described in more detail in Chapter 8. More specifically, in Table 7.3 we show the matrix interfaces available as `SUNMATRIX` modules, and the compatible vector implementations.

Table 7.3: SUNDIALS matrix interfaces and vector implementations that can be used for each.

Matrix Interface	Serial	Parallel (MPI)	OpenMP	pThreads	<i>hypr</i> Vec.	PETSc Vec.	CUDA	RAJA	User Suppl.
Dense	✓		✓	✓					✓

*continued on next page*

Matrix Interface	Serial	Parallel (MPI)	OpenMP	pThreads	hybre Vec.	PETSc Vec.	CUDA	RAJA	User Suppl.
Band	✓		✓	✓					✓
Sparse	✓		✓	✓					✓
User supplied	✓	✓	✓	✓	✓	✓	✓	✓	✓

## 7.1 The SUNMatrix\_Dense implementation

The dense implementation of the SUNMATRIX module provided with SUNDIALS, SUNMATRIX\_DENSE, defines the *content* field of **SUNMatrix** to be the following structure:

```
struct _SUNMatrixContent_Dense {
    sunindextype M;
    sunindextype N;
    realtype *data;
    sunindextype ldata;
    realtype **cols;
};
```

These entries of the *content* field contain the following information:

**M** - number of rows

**N** - number of columns

**data** - pointer to a contiguous block of **realtype** variables. The elements of the dense matrix are stored columnwise, i.e. the (i,j)-th element of a dense SUNMATRIX **A** (with  $0 \leq i < M$  and  $0 \leq j < N$ ) may be accessed via **data**[j\*M+i].

**ldata** - length of the data array (= M·N).

**cols** - array of pointers. **cols**[j] points to the first element of the j-th column of the matrix in the array **data**. The (i,j)-th element of a dense SUNMATRIX **A** (with  $0 \leq i < M$  and  $0 \leq j < N$ ) may be accessed via **cols**[j][i].

The header file to include when using this module is **sunmatrix/sunmatrix\_dense.h**. The SUNMATRIX\_DENSE module is accessible from all SUNDIALS solvers *without* linking to the **libsundials\_sunmatrixdense** module library.

The following macros are provided to access the content of a SUNMATRIX\_DENSE matrix. The prefix **SM\_** in the names denotes that these macros are for *SUNMatrix* implementations, and the suffix **\_D** denotes that these are specific to the *dense* version.

- **SM\_CONTENT\_D**

This macro gives access to the contents of the dense **SUNMatrix**.

The assignment **A\_cont = SM\_CONTENT\_D(A)** sets **A\_cont** to be a pointer to the dense **SUNMatrix** content structure.

Implementation:

```
#define SM_CONTENT_D(A)      ( (SUNMatrixContent_Dense)(A->content) )
```

- **SM\_ROWS\_D**, **SM\_COLUMNS\_D**, and **SM\_LDATAL\_D**

These macros give individual access to various lengths relevant to the content of a dense **SUNMatrix**.



These may be used either to retrieve or to set these values. For example, the assignment `A_rows = SM_ROWS_D(A)` sets `A_rows` to be the number of rows in the matrix `A`. Similarly, the assignment `SM_COLUMNS_D(A) = A_cols` sets the number of columns in `A` to equal `A_cols`.

Implementation:

```
#define SM_ROWS_D(A)      ( SM_CONTENT_D(A)->M )
#define SM_COLUMNS_D(A)   ( SM_CONTENT_D(A)->N )
#define SM_LDATA_D(A)     ( SM_CONTENT_D(A)->ldata )
```

- `SM_DATA_D` and `SM_COLS_D`

These macros give access to the `data` and `cols` pointers for the matrix entries.

The assignment `A_data = SM_DATA_D(A)` sets `A_data` to be a pointer to the first component of the data array for the dense `SUNMatrix` `A`. The assignment `SM_DATA_D(A) = A_data` sets the data array of `A` to be `A_data` by storing the pointer `A_data`.

Similarly, the assignment `A_cols = SM_COLS_D(A)` sets `A_cols` to be a pointer to the array of column pointers for the dense `SUNMatrix` `A`. The assignment `SM_COLS_D(A) = A_cols` sets the column pointer array of `A` to be `A_cols` by storing the pointer `A_cols`.

Implementation:

```
#define SM_DATA_D(A)      ( SM_CONTENT_D(A)->data )
#define SM_COLS_D(A)      ( SM_CONTENT_D(A)->cols )
```

- `SM_COLUMN_D` and `SM_ELEMENT_D`

These macros give access to the individual columns and entries of the data array of a dense `SUNMatrix`.

The assignment `col_j = SM_COLUMN_D(A,j)` sets `col_j` to be a pointer to the first entry of the  $j$ -th column of the  $M \times N$  dense matrix `A` (with  $0 \leq j < N$ ). The type of the expression `SM_COLUMN_D(A,j)` is `realtype *`. The pointer returned by the call `SM_COLUMN_D(A,j)` can be treated as an array which is indexed from 0 to  $M - 1$ .

The assignments `SM_ELEMENT_D(A,i,j) = a_ij` and `a_ij = SM_ELEMENT_D(A,i,j)` reference the  $(i,j)$ -th element of the  $M \times N$  dense matrix `A` (with  $0 \leq i < M$  and  $0 \leq j < N$ ).

Implementation:

```
#define SM_COLUMN_D(A,j)  ( (SM_CONTENT_D(A)->cols)[j] )
#define SM_ELEMENT_D(A,i,j) ( (SM_CONTENT_D(A)->cols)[j][i] )
```

The `SUNMATRIX_DENSE` module defines dense implementations of all matrix operations listed in Table 7.2. Their names are obtained from those in Table 7.2 by appending the suffix `_Dense` (e.g. `SUNMatCopy_Dense`). The module `SUNMATRIX_DENSE` provides the following additional user-callable routines:

- `SUNDenseMatrix`

This constructor function creates and allocates memory for a dense `SUNMatrix`. Its arguments are the number of rows,  $M$ , and columns,  $N$ , for the dense matrix.

```
SUNMatrix SUNDenseMatrix(sunindextype M, sunindextype N);
```

- `SUNDenseMatrix.Print`

This function prints the content of a dense `SUNMatrix` to the output stream specified by `outfile`. Note: `stdout` or `stderr` may be used as arguments for `outfile` to print directly to standard output or standard error, respectively.

```
void SUNDenseMatrix_Print(SUNMatrix A, FILE* outfile);
```

- **SUNDenseMatrix.Rows**  
This function returns the number of rows in the dense **SUNMatrix**.  
`sunindextype SUNDenseMatrix_Rows(SUNMatrix A);`
- **SUNDenseMatrix.Columns**  
This function returns the number of columns in the dense **SUNMatrix**.  
`sunindextype SUNDenseMatrix_Columns(SUNMatrix A);`
- **SUNDenseMatrix.LData**  
This function returns the length of the data array for the dense **SUNMatrix**.  
`sunindextype SUNDenseMatrix_LData(SUNMatrix A);`
- **SUNDenseMatrix.Data**  
This function returns a pointer to the data array for the dense **SUNMatrix**.  
`realtype* SUNDenseMatrix_Data(SUNMatrix A);`
- **SUNDenseMatrix.Cols**  
This function returns a pointer to the cols array for the dense **SUNMatrix**.  
`realtype** SUNDenseMatrix_Cols(SUNMatrix A);`
- **SUNDenseMatrix.Column**  
This function returns a pointer to the first entry of the *j*th column of the dense **SUNMatrix**. The resulting pointer should be indexed over the range 0 to *M* − 1.  
`realtype* SUNDenseMatrix_Column(SUNMatrix A, sunindextype j);`

## Notes

- When looping over the components of a dense **SUNMatrix** *A*, the most efficient approaches are to:
  - First obtain the component array via `A_data = SM_DATA_D(A)` or `A_data = SUNDenseMatrix_Data(A)` and then access `A_data[i]` within the loop.
  - First obtain the array of column pointers via `A_cols = SM_COLS_D(A)` or `A_cols = SUNDenseMatrix_Cols(A)`, and then access `A_cols[j][i]` within the loop.
  - Within a loop over the columns, access the column pointer via `A_colj = SUNDenseMatrix_Column(A,j)` and then to access the entries within that column using `A_colj[i]` within the loop.

All three of these are more efficient than using `SM_ELEMENT_D(A,i,j)` within a double loop.



- Within the **SUNMatMatvec\_Dense** routine, internal consistency checks are performed to ensure that the matrix is called with consistent **NVECTOR** implementations. These are currently limited to: **NVECTOR\_SERIAL**, **NVECTOR\_OPENMP**, and **NVECTOR\_PTHREADS**. As additional compatible vector implementations are added to **SUNDIALS**, these will be included within this compatibility check.

For solvers that include a Fortran interface module, the **SUNMATRIX\_DENSE** module also includes the Fortran-callable function **FSUNDenseMatInit**(*code*, *M*, *N*, *ier*) to initialize this **SUNMATRIX\_DENSE** module for a given **SUNDIALS** solver. Here *code* is an integer input solver id (1 for **CVODE**, 2 for **IDA**, 3 for **KINSOL**, 4 for **ARKODE**); *M* and *N* are the corresponding dense matrix construction arguments (declared to match C type **long int**); and *ier* is an error return flag equal to 0 for success and -1 for failure. Both *code* and *ier* are declared to match C type **int**. Additionally, when using **ARKODE** with a non-identity mass matrix, the Fortran-callable function **FSUNDenseMassMatInit**(*M*, *N*, *ier*) initializes this **SUNMATRIX\_DENSE** module for storing the mass matrix.

## 7.2 The SUNMatrix\_Band implementation

The banded implementation of the SUNMATRIX module provided with SUNDIALS, SUNMATRIX\_BAND, defines the *content* field of SUNMatrix to be the following structure:

```
struct _SUNMatrixContent_Band {
    sunindextype M;
    sunindextype N;
    sunindextype mu;
    sunindextype ml;
    sunindextype s_mu;
    sunindextype ldim;
    realtype *data;
    sunindextype ldata;
    realtype **cols;
};
```

A diagram of the underlying data representation in a banded matrix is shown in Figure 7.1. A more complete description of the parts of this *content* field is given below:

**M** - number of rows

**N** - number of columns ( $N = M$ )

**mu** - upper half-bandwidth,  $0 \leq \mu < N$

**ml** - lower half-bandwidth,  $0 \leq ml < N$

**s\_mu** - storage upper bandwidth,  $\mu \leq s\_mu < N$ . The LU decomposition routines in the associated SUNLINSOL\_BAND and SUNLINSOL\_LAPACKBAND modules write the LU factors into the storage for A. The upper triangular factor U, however, may have an upper bandwidth as big as  $\min(N-1, \mu+ml)$  because of partial pivoting. The **s\_mu** field holds the upper half-bandwidth allocated for A.

**ldim** - leading dimension ( $ldim \geq s\_mu+ml+1$ )

**data** - pointer to a contiguous block of **realtype** variables. The elements of the banded matrix are stored columnwise (i.e. columns are stored one on top of the other in memory). Only elements within the specified half-bandwidths are stored. **data** is a pointer to **ldata** contiguous locations which hold the elements within the band of A.

**ldata** - length of the data array ( $= ldim \cdot N$ )

**cols** - array of pointers. **cols**[j] is a pointer to the uppermost element within the band in the j-th column. This pointer may be treated as an array indexed from **s\_mu**-**mu** (to access the uppermost element within the band in the j-th column) to **s\_mu**+**ml** (to access the lowest element within the band in the j-th column). Indices from 0 to **s\_mu**-**mu**-1 give access to extra storage elements required by the LU decomposition function. Finally, **cols**[j][i-j+s\_mu] is the (i,j)-th element with  $j-\mu \leq i \leq j+ml$ .

The header file to include when using this module is **sunmatrix/sunmatrix.band.h**. The SUNMATRIX\_BAND module is accessible from all SUNDIALS solvers *without* linking to the **libsundials\_sunmatrixband** module library.

The following macros are provided to access the content of a SUNMATRIX\_BAND matrix. The prefix **SM\_** in the names denotes that these macros are for *SUNMatrix* implementations, and the suffix **\_B** denotes that these are specific to the *banded* version.

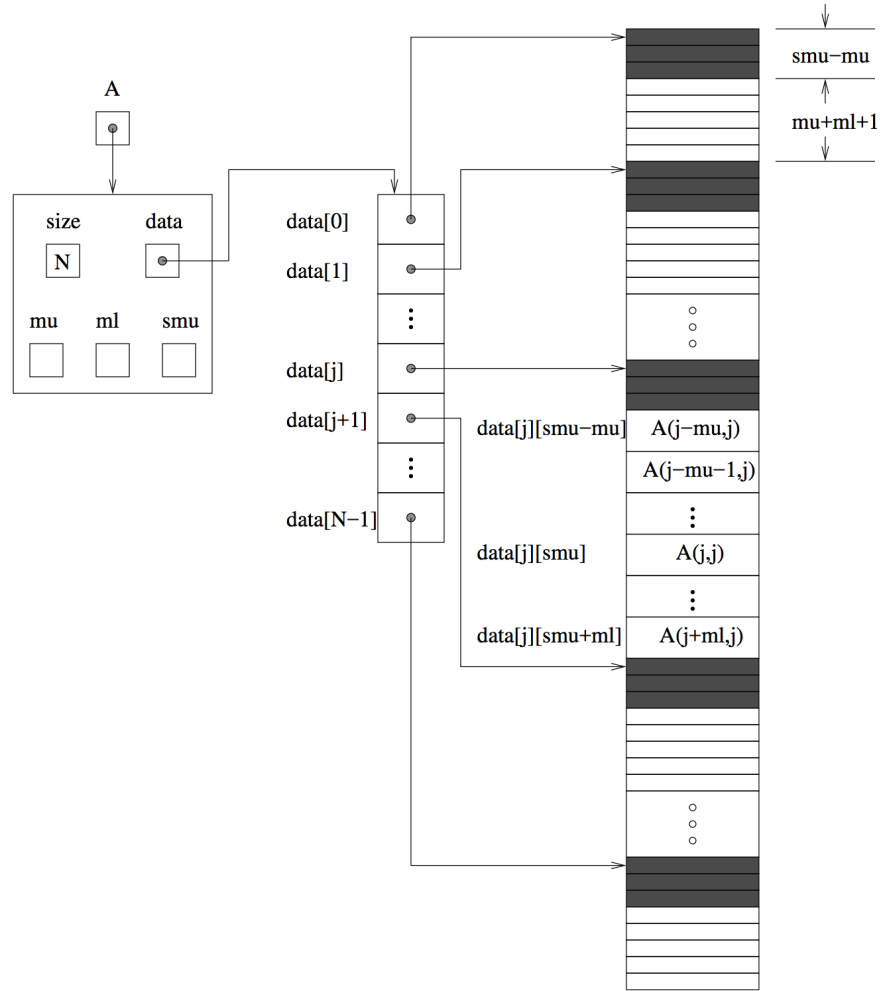


Figure 7.1: Diagram of the storage for the `SUNMATRIX_BAND` module. Here  $A$  is an  $N \times N$  band matrix with upper and lower half-bandwidths  $\mu$  and  $m_l$ , respectively. The rows and columns of  $A$  are numbered from 0 to  $N - 1$  and the  $(i, j)$ -th element of  $A$  is denoted  $A(i, j)$ . The greyed out areas of the underlying component storage are used by the associated `SUNLINSOL_BAND` linear solver.

- **SM\_CONTENT\_B**

This routine gives access to the contents of the banded **SUNMatrix**.

The assignment **A\_cont = SM\_CONTENT\_B(A)** sets **A\_cont** to be a pointer to the banded **SUNMatrix** content structure.

Implementation:

```
#define SM_CONTENT_B(A)      ( (SUNMatrixContent_Band)(A->content) )
```

- **SM\_ROWS\_B**, **SM\_COLUMNS\_B**, **SM\_UBAND\_B**, **SM\_LBAND\_B**, **SM\_SUBAND\_B**, **SM\_LDIM\_B**, and **SM\_LDATA\_B**

These macros give individual access to various lengths relevant to the content of a banded **SUNMatrix**.

These may be used either to retrieve or to set these values. For example, the assignment **A\_rows = SM\_ROWS\_B(A)** sets **A\_rows** to be the number of rows in the matrix **A**. Similarly, the assignment **SM\_COLUMNS\_B(A) = A\_cols** sets the number of columns in **A** to equal **A\_cols**.

Implementation:

```
#define SM_ROWS_B(A)         ( SM_CONTENT_B(A)->M )
#define SM_COLUMNS_B(A)      ( SM_CONTENT_B(A)->N )
#define SM_UBAND_B(A)        ( SM_CONTENT_B(A)->mu )
#define SM_LBAND_B(A)        ( SM_CONTENT_B(A)->m1 )
#define SM_SUBAND_B(A)       ( SM_CONTENT_B(A)->s_mu )
#define SM_LDIM_B(A)         ( SM_CONTENT_B(A)->ldim )
#define SM_LDATA_B(A)        ( SM_CONTENT_B(A)->ldata )
```

- **SM\_DATA\_B** and **SM\_COLS\_B**

These macros give access to the **data** and **cols** pointers for the matrix entries.

The assignment **A\_data = SM\_DATA\_B(A)** sets **A\_data** to be a pointer to the first component of the data array for the banded **SUNMatrix** **A**. The assignment **SM\_DATA\_B(A) = A\_data** sets the data array of **A** to be **A\_data** by storing the pointer **A\_data**.

Similarly, the assignment **A\_cols = SM\_COLS\_B(A)** sets **A\_cols** to be a pointer to the array of column pointers for the banded **SUNMatrix** **A**. The assignment **SM\_COLS\_B(A) = A\_cols** sets the column pointer array of **A** to be **A\_cols** by storing the pointer **A\_cols**.

Implementation:

```
#define SM_DATA_B(A)         ( SM_CONTENT_B(A)->data )
#define SM_COLS_B(A)         ( SM_CONTENT_B(A)->cols )
```

- **SM\_COLUMN\_B**, **SM\_COLUMN\_ELEMENT\_B**, and **SM\_ELEMENT\_B**

These macros give access to the individual columns and entries of the data array of a banded **SUNMatrix**.

The assignments **SM\_ELEMENT\_B(A,i,j) = a\_ij** and **a\_ij = SM\_ELEMENT\_B(A,i,j)** reference the  $(i,j)$ -th element of the  $N \times N$  band matrix **A**, where  $0 \leq i, j \leq N - 1$ . The location  $(i,j)$  should further satisfy  $j - \mu \leq i \leq j + m1$ .

The assignment **col\_j = SM\_COLUMN\_B(A,j)** sets **col\_j** to be a pointer to the diagonal element of the  $j$ -th column of the  $N \times N$  band matrix **A**,  $0 \leq j \leq N - 1$ . The type of the expression **SM\_COLUMN\_B(A,j)** is **realtype \***. The pointer returned by the call **SM\_COLUMN\_B(A,j)** can be treated as an array which is indexed from  $-\mu$  to  $m1$ .

The assignments **SM\_COLUMN\_ELEMENT\_B(col\_j,i,j) = a\_ij** and **a\_ij = SM\_COLUMN\_ELEMENT\_B(col\_j,i,j)** reference the  $(i,j)$ -th entry of the band matrix **A** when used in conjunction with **SM\_COLUMN\_B** to reference the  $j$ -th column through **col\_j**. The index  $(i,j)$  should satisfy  $j - \mu \leq i \leq j + m1$ .

Implementation:

```
#define SM_COLUMN_B(A,j)      ( ((SM_CONTENT_B(A)->cols)[j])+SM_SUBBAND_B(A) )
#define SM_COLUMN_ELEMENT_B(col_j,i,j) (col_j[(i)-(j)])
#define SM_ELEMENT_B(A,i,j)
    ( (SM_CONTENT_B(A)->cols)[j][(i)-(j)+SM_SUBBAND_B(A)] )
```

The `SUNMATRIX_BAND` module defines banded implementations of all matrix operations listed in Table 7.2. Their names are obtained from those in Table 7.2 by appending the suffix `_Band` (e.g. `SUNMatCopy_Band`). The module `SUNMATRIX_BAND` provides the following additional user-callable routines:

- **SUNBandMatrix**

This constructor function creates and allocates memory for a banded **SUNMatrix**. Its arguments are the matrix size, `N`, the upper and lower half-bandwidths of the matrix, `mu` and `ml`, and the stored upper bandwidth, `smu`. When creating a band **SUNMatrix**, this value should be

- at least  $\min(N-1, \mu+ml)$  if the matrix will be used by the `SUNLINSOL_BAND` module;
- exactly equal to  $\mu+ml$  if the matrix will be used by the `SUNLINSOL_LAPACKBAND` module;
- at least  $\mu$  if used in some other manner.

```
SUNMatrix SUNBandMatrix(sunindextype N, sunindextype mu,
                        sunindextype ml, sunindextype smu);
```

- **SUNBandMatrix.Print**

This function prints the content of a banded **SUNMatrix** to the output stream specified by `outfile`. Note: `stdout` or `stderr` may be used as arguments for `outfile` to print directly to standard output or standard error, respectively.

```
void SUNBandMatrix_Print(SUNMatrix A, FILE* outfile);
```

- **SUNBandMatrix.Rows**

This function returns the number of rows in the banded **SUNMatrix**.

```
sunindextype SUNBandMatrix_Rows(SUNMatrix A);
```

- **SUNBandMatrix.Columns**

This function returns the number of columns in the banded **SUNMatrix**.

```
sunindextype SUNBandMatrix_Columns(SUNMatrix A);
```

- **SUNBandMatrix.LowerBandwidth**

This function returns the lower half-bandwidth of the banded **SUNMatrix**.

```
sunindextype SUNBandMatrix_LowerBandwidth(SUNMatrix A);
```

- **SUNBandMatrix.UpperBandwidth**

This function returns the upper half-bandwidth of the banded **SUNMatrix**.

```
sunindextype SUNBandMatrix_UpperBandwidth(SUNMatrix A);
```

- **SUNBandMatrix.StoredUpperBandwidth**

This function returns the stored upper half-bandwidth of the banded **SUNMatrix**.

```
sunindextype SUNBandMatrix_StoredUpperBandwidth(SUNMatrix A);
```

- `SUNBandMatrix_LDim`

This function returns the length of the leading dimension of the banded `SUNMatrix`.

```
sunindextype SUNBandMatrix_LDim(SUNMatrix A);
```

- `SUNBandMatrix_Data`

This function returns a pointer to the data array for the banded `SUNMatrix`.

```
realtype* SUNBandMatrix_Data(SUNMatrix A);
```

- `SUNBandMatrix_Cols`

This function returns a pointer to the cols array for the banded `SUNMatrix`.

```
realtype** SUNBandMatrix_Cols(SUNMatrix A);
```

- `SUNBandMatrix_Column`

This function returns a pointer to the diagonal entry of the  $j$ -th column of the banded `SUNMatrix`. The resulting pointer should be indexed over the range  $-\mu$  to  $m$ .

```
realtype* SUNBandMatrix_Column(SUNMatrix A, sunindextype j);
```

### Notes

- When looping over the components of a banded `SUNMatrix A`, the most efficient approaches are to:
  - First obtain the component array via `A_data = SM_DATA_B(A)` or `A_data = SUNBandMatrix_Data(A)` and then access `A_data[i]` within the loop.
  - First obtain the array of column pointers via `A_cols = SM_COLS_B(A)` or `A_cols = SUNBandMatrix_Cols(A)`, and then access `A_cols[j][i]` within the loop.
  - Within a loop over the columns, access the column pointer via `A_colj = SUNBandMatrix_Column(A,j)` and then to access the entries within that column using `SM_COLUMN_ELEMENT_B(A_colj,i,j)`.

All three of these are more efficient than using `SM_ELEMENT_B(A,i,j)` within a double loop.

- Within the `SUNMatMatvec_Band` routine, internal consistency checks are performed to ensure that the matrix is called with consistent `NVECTOR` implementations. These are currently limited to: `NVECTOR_SERIAL`, `NVECTOR_OPENMP`, and `NVECTOR_PTHREADS`. As additional compatible vector implementations are added to `SUNDIALS`, these will be included within this compatibility check.



For solvers that include a Fortran interface module, the `SUNMATRIX_BAND` module also includes the Fortran-callable function `FSUNBandMatInit(code, N, mu, ml, smu, ier)` to initialize this `SUNMATRIX_BAND` module for a given `SUNDIALS` solver. Here `code` is an integer input solver id (1 for `CVODE`, 2 for `IDA`, 3 for `KINSOL`, 4 for `ARKODE`); `N`, `mu`, `ml` and `smu` are the corresponding band matrix construction arguments (declared to match C type `long int`); and `ier` is an error return flag equal to 0 for success and -1 for failure. Both `code` and `ier` are declared to match C type `int`. Additionally, when using `ARKODE` with a non-identity mass matrix, the Fortran-callable function `FSUNBandMassMatInit(N, mu, ml, smu, ier)` initializes this `SUNMATRIX_BAND` module for storing the mass matrix.

## 7.3 The SUNMatrix\_Sparse implementation

The sparse implementation of the `SUNMATRIX` module provided with `SUNDIALS`, `SUNMATRIX_SPARSE`, is designed to work with either *compressed-sparse-column* (CSC) or *compressed-sparse-row* (CSR) sparse matrix formats. To this end, it defines the *content* field of `SUNMatrix` to be the following structure:

```

struct _SUNMatrixContent_Sparse {
    sunindextype M;
    sunindextype N;
    sunindextype NNZ;
    sunindextype NP;
    realtype *data;
    int sparsetype;
    sunindextype *indexvals;
    sunindextype *indexptrs;
    /* CSC indices */
    sunindextype **rowvals;
    sunindextype **colptrs;
    /* CSR indices */
    sunindextype **colvals;
    sunindextype **rowptrs;
};

```

A diagram of the underlying data representation for a CSC matrix is shown in Figure 7.2 (the CSR format is similar). A more complete description of the parts of this *content* field is given below:

**M** - number of rows

**N** - number of columns

**NNZ** - maximum number of nonzero entries in the matrix (allocated length of **data** and **indexvals** arrays)

**NP** - number of index pointers (e.g. number of column pointers for CSC matrix). For CSC matrices  $NP = N$ , and for CSR matrices  $NP = M$ . This value is set automatically based the input for **sparsetype**.

**data** - pointer to a contiguous block of **realtype** variables (of length **NNZ**), containing the values of the nonzero entries in the matrix

**sparsetype** - type of the sparse matrix (**CSC\_MAT** or **CSR\_MAT**)

**indexvals** - pointer to a contiguous block of **int** variables (of length **NNZ**), containing the row indices (if CSC) or column indices (if CSR) of each nonzero matrix entry held in **data**

**indexptrs** - pointer to a contiguous block of **int** variables (of length  $NP+1$ ). For CSC matrices each entry provides the index of the first column entry into the **data** and **indexvals** arrays, e.g. if **indexptr[3]=7**, then the first nonzero entry in the fourth column of the matrix is located in **data[7]**, and is located in row **indexvals[7]** of the matrix. The last entry contains the total number of nonzero values in the matrix and hence points one past the end of the active data in the **data** and **indexvals** arrays. For CSR matrices, each entry provides the index of the first row entry into the **data** and **indexvals** arrays.

The following pointers are added to the **SlsMat** type for user convenience, to provide a more intuitive interface to the CSC and CSR sparse matrix data structures. They are set automatically when creating a sparse SUNMATRIX, based on the sparse matrix storage type.

**rowvals** - pointer to **indexvals** when **sparsetype** is **CSC\_MAT**, otherwise set to **NULL**.

**colptrs** - pointer to **indexptrs** when **sparsetype** is **CSC\_MAT**, otherwise set to **NULL**.

**colvals** - pointer to **indexvals** when **sparsetype** is **CSR\_MAT**, otherwise set to **NULL**.

**rowptrs** - pointer to **indexptrs** when **sparsetype** is **CSR\_MAT**, otherwise set to **NULL**.



For example, the  $5 \times 4$  CSC matrix

$$\begin{bmatrix} 0 & 3 & 1 & 0 \\ 3 & 0 & 0 & 2 \\ 0 & 7 & 0 & 0 \\ 1 & 0 & 0 & 9 \\ 0 & 0 & 0 & 5 \end{bmatrix}$$

could be stored in this structure as either

```
M = 5;
N = 4;
NNZ = 8;
NP = N;
data = {3.0, 1.0, 3.0, 7.0, 1.0, 2.0, 9.0, 5.0};
sparsetype = CSC_MAT;
indexvals = {1, 3, 0, 2, 0, 1, 3, 4};
indexptrs = {0, 2, 4, 5, 8};
```

or

```
M = 5;
N = 4;
NNZ = 10;
NP = N;
data = {3.0, 1.0, 3.0, 7.0, 1.0, 2.0, 9.0, 5.0, *, *};
sparsetype = CSC_MAT;
indexvals = {1, 3, 0, 2, 0, 1, 3, 4, *, *};
indexptrs = {0, 2, 4, 5, 8};
```

where the first has no unused space, and the second has additional storage (the entries marked with \* may contain any values). Note in both cases that the final value in `indexptrs` is 8, indicating the total number of nonzero entries in the matrix.

Similarly, in CSR format, the same matrix could be stored as

```
M = 5;
N = 4;
NNZ = 8;
NP = N;
data = {3.0, 1.0, 3.0, 2.0, 7.0, 1.0, 9.0, 5.0};
sparsetype = CSR_MAT;
indexvals = {1, 2, 0, 3, 1, 0, 3, 3};
indexptrs = {0, 2, 4, 5, 7, 8};
```

The header file to include when using this module is `sunmatrix/sunmatrix.sparse.h`. The `SUNMATRIX_SPARSE` module is accessible from all SUNDIALS solvers *without* linking to the `libsundials_sunmatrixsparse` module library.

The following macros are provided to access the content of a `SUNMATRIX_SPARSE` matrix. The prefix `SM_` in the names denotes that these macros are for *SUNMatrix* implementations, and the suffix `_S` denotes that these are specific to the *sparse* version.

- `SM_CONTENT_S`

This routine gives access to the contents of the sparse `SUNMatrix`.

The assignment `A_cont = SM_CONTENT_S(A)` sets `A_cont` to be a pointer to the sparse `SUNMatrix` content structure.

Implementation:

```
#define SM_CONTENT_S(A)      ( (SUNMatrixContent_Sparse)(A->content) )
```

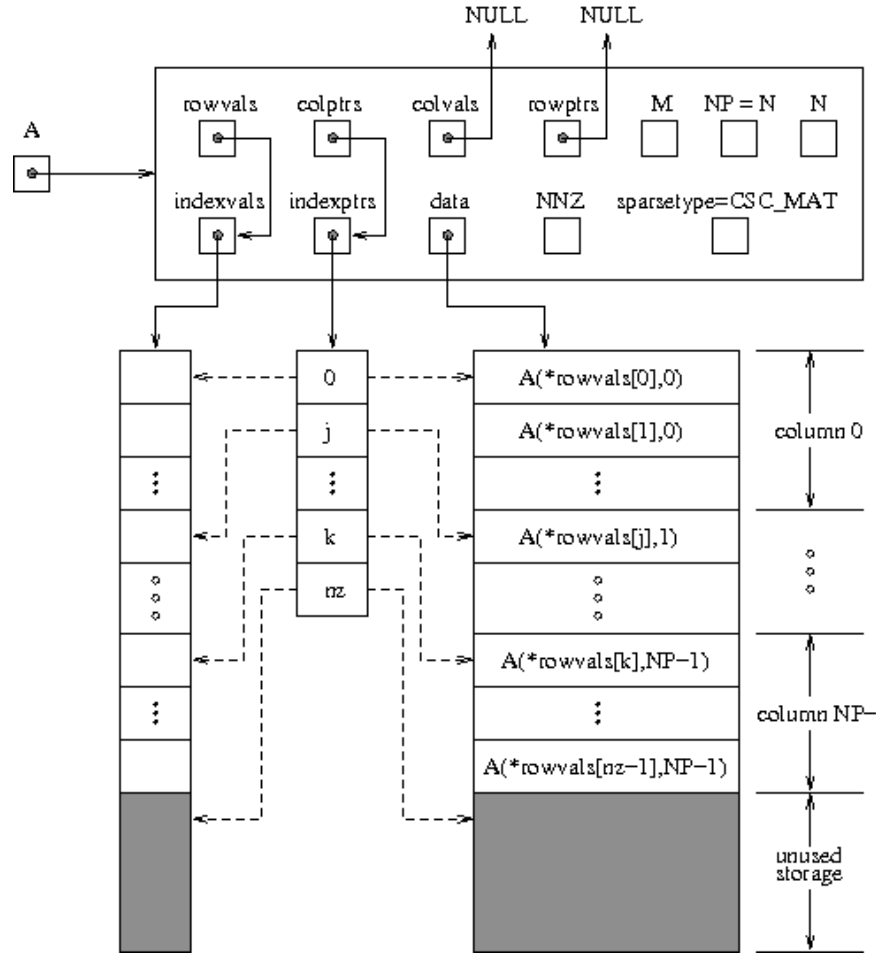


Figure 7.2: Diagram of the storage for a compressed-sparse-column matrix. Here  $A$  is an  $M \times N$  sparse matrix with storage for up to  $NNZ$  nonzero entries (the allocated length of both `data` and `indexvals`). The entries in `indexvals` may assume values from 0 to  $M - 1$ , corresponding to the row index (zero-based) of each nonzero value. The entries in `data` contain the values of the nonzero entries, with the row  $i$ , column  $j$  entry of  $A$  (again, zero-based) denoted as  $A(i, j)$ . The `indexptrs` array contains  $N + 1$  entries; the first  $N$  denote the starting index of each column within the `indexvals` and `data` arrays, while the final entry points one past the final nonzero entry. Here, although  $NNZ$  values are allocated, only `nz` are actually filled in; the greyed-out portions of `data` and `indexvals` indicate extra allocated space.

- `SM_ROWS_S`, `SM_COLUMNS_S`, `SM_NNZ_S`, `SM_NP_S`, and `SM_SPARSETYPE_S`

These macros give individual access to various lengths relevant to the content of a sparse SUNMatrix.

These may be used either to retrieve or to set these values. For example, the assignment `A_rows = SM_ROWS_S(A)` sets `A_rows` to be the number of rows in the matrix `A`. Similarly, the assignment `SM_COLUMNS_S(A) = A_cols` sets the number of columns in `A` to equal `A_cols`.

Implementation:

```
#define SM_ROWS_S(A)      ( SM_CONTENT_S(A)->M )
#define SM_COLUMNS_S(A)   ( SM_CONTENT_S(A)->N )
#define SM_NNZ_S(A)       ( SM_CONTENT_S(A)->NNZ )
#define SM_NP_S(A)        ( SM_CONTENT_S(A)->NP )
#define SM_SPARSETYPE_S(A) ( SM_CONTENT_S(A)->sparsetype )
```

- `SM_DATA_S`, `SM_INDEXVALS_S`, and `SM_INDEXPTRS_S`

These macros give access to the `data` and index arrays for the matrix entries.

The assignment `A_data = SM_DATA_S(A)` sets `A_data` to be a pointer to the first component of the data array for the sparse SUNMatrix `A`. The assignment `SM_DATA_S(A) = A_data` sets the data array of `A` to be `A_data` by storing the pointer `A_data`.

Similarly, the assignment `A_indexvals = SM_INDEXVALS_S(A)` sets `A_indexvals` to be a pointer to the array of index values (i.e. row indices for a CSC matrix, or column indices for a CSR matrix) for the sparse SUNMatrix `A`. The assignment `A_indexptrs = SM_INDEXPTRS_S(A)` sets `A_indexptrs` to be a pointer to the array of index pointers (i.e. the starting indices in the data/indexvals arrays for each row or column in CSR or CSC formats, respectively).

Implementation:

```
#define SM_DATA_S(A)      ( SM_CONTENT_S(A)->data )
#define SM_INDEXVALS_S(A) ( SM_CONTENT_S(A)->indexvals )
#define SM_INDEXPTRS_S(A) ( SM_CONTENT_S(A)->indexptrs )
```

The `SUNMATRIX_SPARSE` module defines sparse implementations of all matrix operations listed in Table 7.2. Their names are obtained from those in Table 7.2 by appending the suffix `_Sparse` (e.g. `SUNMatCopy_Sparse`). The module `SUNMATRIX_SPARSE` provides the following additional user-callable routines:

- `SUNSparseMatrix`

This function creates and allocates memory for a sparse SUNMatrix. Its arguments are the number of rows and columns of the matrix, `M` and `N`, the maximum number of nonzeros to be stored in the matrix, `NNZ`, and a flag `sparsetype` indicating whether to use CSR or CSC format (valid arguments are `CSR_MAT` or `CSC_MAT`).

```
SUNMatrix SUNSparseMatrix(sunindextype M, sunindextype N,
                          sunindextype NNZ, int sparsetype);
```

- `SUNSparseFromDenseMatrix`

This function creates a new sparse matrix from an existing dense matrix by copying all values with magnitude larger than `droptol` into the sparse matrix structure.

Requirements:

- `A` must have type `SUNMATRIX_DENSE`;

- `droptol` must be non-negative;
- `sparsetype` must be either `CSC_MAT` or `CSR_MAT`.

The function returns `NULL` if any requirements are violated, or if the matrix storage request cannot be satisfied.

```
SUNMatrix SUNSparseFromDenseMatrix(SUNMatrix A, realtype droptol,
                                   int sparsetype);
```

- **SUNSparseFromBandMatrix**

This function creates a new sparse matrix from an existing band matrix by copying all values with magnitude larger than `droptol` into the sparse matrix structure.

Requirements:

- `A` must have type `SUNMATRIX_BAND`;
- `droptol` must be non-negative;
- `sparsetype` must be either `CSC_MAT` or `CSR_MAT`.

The function returns `NULL` if any requirements are violated, or if the matrix storage request cannot be satisfied.

```
SUNMatrix SUNSparseFromBandMatrix(SUNMatrix A, realtype droptol,
                                   int sparsetype);
```

- **SUNSparseMatrix\_Realloc**

This function reallocates internal storage arrays in a sparse matrix so that the resulting sparse matrix has no wasted space (i.e. the space allocated for nonzero entries equals the actual number of nonzeros, `indexptrs[NP]`). Returns 0 on success and 1 on failure (e.g. if the input matrix is not sparse).

```
int SUNSparseMatrix_Realloc(SUNMatrix A);
```

- **SUNSparseMatrix\_Reallocate**

This function reallocates internal storage arrays in a sparse matrix so that the resulting sparse matrix has storage for a specified number of nonzeros. Returns 0 on success and 1 on failure (e.g. if the input matrix is not sparse or if `NNZ` is negative).

```
int SUNSparseMatrix_Reallocate(SUNMatrix A, sunindextype NNZ);
```

- **SUNSparseMatrix\_Print**

This function prints the content of a sparse `SUNMatrix` to the output stream specified by `outfile`. Note: `stdout` or `stderr` may be used as arguments for `outfile` to print directly to standard output or standard error, respectively.

```
void SUNSparseMatrix_Print(SUNMatrix A, FILE* outfile);
```

- **SUNSparseMatrix\_Rows**

This function returns the number of rows in the sparse `SUNMatrix`.

```
sunindextype SUNSparseMatrix_Rows(SUNMatrix A);
```

- **SUNSparseMatrix\_Columns**

This function returns the number of columns in the sparse `SUNMatrix`.

```
sunindextype SUNSparseMatrix_Columns(SUNMatrix A);
```

- `SUNSparseMatrix_NNZ`

This function returns the number of entries allocated for nonzero storage for the sparse matrix `SUNMatrix`.

```
sunindextype SUNSparseMatrix_NNZ(SUNMatrix A);
```

- `SUNSparseMatrix_NP`

This function returns the number of columns/rows for the sparse `SUNMatrix`, depending on whether the matrix uses CSC/CSR format, respectively. The `indexptrs` array has `NP+1` entries.

```
sunindextype SUNSparseMatrix_NP(SUNMatrix A);
```

- `SUNSparseMatrix_SparseType`

This function returns the storage type (`CSR_MAT` or `CSC_MAT`) for the sparse `SUNMatrix`.

```
int SUNSparseMatrix_SparseType(SUNMatrix A);
```

- `SUNSparseMatrix_Data`

This function returns a pointer to the data array for the sparse `SUNMatrix`.

```
realtype* SUNSparseMatrix_Data(SUNMatrix A);
```

- `SUNSparseMatrix_IndexValues`

This function returns a pointer to index value array for the sparse `SUNMatrix`: for CSR format this is the column index for each nonzero entry, for CSC format this is the row index for each nonzero entry.

```
sunindextype* SUNSparseMatrix_IndexValues(SUNMatrix A);
```

- `SUNSparseMatrix_IndexPointers`

This function returns a pointer to the index pointer array for the sparse `SUNMatrix`: for CSR format this is the location of the first entry of each row in the `data` and `indexvalues` arrays, for CSC format this is the location of the first entry of each column.

```
sunindextype* SUNSparseMatrix_IndexPointers(SUNMatrix A);
```

Within the `SUNMatMatvec_Sparse` routine, internal consistency checks are performed to ensure that the matrix is called with consistent `NVECTOR` implementations. These are currently limited to: `NVECTOR_SERIAL`, `NVECTOR_OPENMP`, and `NVECTOR_PTHREADS`. As additional compatible vector implementations are added to `SUNDIALS`, these will be included within this compatibility check.

For solvers that include a Fortran interface module, the `SUNMATRIX_SPARSE` module also includes the Fortran-callable function `FSUNSparseMatInit(code, M, N, NNZ, sparsetype, ier)` to initialize this `SUNMATRIX_SPARSE` module for a given `SUNDIALS` solver. Here `code` is an integer input for the solver id (1 for `CVODE`, 2 for `IDA`, 3 for `KINSOL`, 4 for `ARKODE`); `M`, `N` and `NNZ` are the corresponding sparse matrix construction arguments (declared to match C type `long int`); `sparsetype` is an integer flag indicating the sparse storage type (0 for `CSC`, 1 for `CSR`); and `ier` is an error return flag equal to 0 for success and -1 for failure. Each of `code`, `sparsetype` and `ier` are declared so as to match C type `int`. Additionally, when using `ARKODE` with a non-identity mass matrix, the Fortran-callable function `FSUNSparseMassMatInit(M, N, NNZ, sparsetype, ier)` initializes this `SUNMATRIX_SPARSE` module for storing the mass matrix.



## 7.4 SUNMatrix Examples

There are `SUNMatrix` examples that may be installed for each implementation: dense, banded, and sparse. Each implementation makes use of the functions in `test_sunmatrix.c`. These example functions show simple usage of the `SUNMatrix` family of functions. The inputs to the examples depend on the matrix type, and are output to `stdout` if the example is run without the appropriate number of command-line arguments.

The following is a list of the example functions in `test_sunmatrix.c`:

- **Test\_SUNMatGetID**: Verifies the returned matrix ID against the value that should be returned.
- **Test\_SUNMatClone**: Creates clone of an existing matrix, copies the data, and checks that their values match.
- **Test\_SUNMatZero**: Zeros out an existing matrix and checks that each entry equals 0.0.
- **Test\_SUNMatCopy**: Clones an input matrix, copies its data to a clone, and verifies that all values match.
- **Test\_SUNMatScaleAdd**: Given an input matrix  $A$  and an input identity matrix  $I$ , this test clones and copies  $A$  to a new matrix  $B$ , computes  $B = -B + B$ , and verifies that the resulting matrix entries equal 0.0. Additionally, if the matrix is square, this test clones and copies  $A$  to a new matrix  $D$ , clones and copies  $I$  to a new matrix  $C$ , computes  $D = D + I$  and  $C = C + A$  using **SUNMatScaleAdd**, and then verifies that  $C == D$ .
- **Test\_SUNMatScaleAddI**: Given an input matrix  $A$  and an input identity matrix  $I$ , this clones and copies  $I$  to a new matrix  $B$ , computes  $B = -B + I$  using **SUNMatScaleAddI**, and verifies that the resulting matrix entries equal 0.0.
- **Test\_SUNMatMatvec**: Given an input matrix  $A$  and input vectors  $x$  and  $y$  such that  $y = Ax$ , this test has different behavior depending on whether  $A$  is square. If it is square, it clones and copies  $A$  to a new matrix  $B$ , computes  $B = 3B + I$  using **SUNMatScaleAddI**, clones  $y$  to new vectors  $w$  and  $z$ , computes  $z = Bx$  using **SUNMatMatvec**, computes  $w = 3y + x$  using **N\_VLinearSum**, and verifies that  $w == z$ . If  $A$  is not square, it just clones  $y$  to a new vector  $z$ , computes  $z = Ax$  using **SUNMatMatvec**, and verifies that  $y == z$ .
- **Test\_SUNMatSpace** verifies that **SUNMatSpace** can be called, and outputs the results to **stdout**.

## 7.5 SUNMatrix functions used by IDA

In Table 7.4 below, we list the matrix functions in the SUNMATRIX module used within the IDA package. The table also shows, for each function, which of the code modules uses the function. The main IDA integrator does not call any SUNMATRIX functions directly, so the table columns are specific to the IDALS direct solver interface and the IDABBDPRE preconditioner module. We further note that the IDALS interface only utilizes these routines when supplied with a *matrix-based* linear solver, i.e., the SUNMATRIX object passed to **IDASetLinearSolver** was not NULL.

At this point, we should emphasize that the IDA user does not need to know anything about the usage of matrix functions by the IDA code modules in order to use IDA. The information is presented as an implementation detail for the interested reader.

Table 7.4: List of matrix functions usage by IDA code modules

	IDALS	IDABBDPRE
<b>SUNMatGetID</b>	✓	
<b>SUNMatDestroy</b>		✓
<b>SUNMatZero</b>	✓	✓
<b>SUNMatSpace</b>		†

The matrix functions listed in Table 7.2 with a † symbol are optionally used, in that these are only called if they are implemented in the SUNMATRIX module that is being used (i.e. their function pointers are non-NULL). The matrix functions listed in Table 7.2 that are *not* used by IDA are: **SUNMatCopy**,

---

SUNMatClone, SUNMatScaleAdd, SUNMatScaleAddI and SUNMatMatvec. Therefore a user-supplied SUNMATRIX module for IDA could omit these functions.





## Chapter 8

# Description of the SUNLinearSolver module

For problems that involve the solution of linear systems of equations, the SUNDIALS packages operate using generic linear solver modules defined through the SUNLINSOL API. This allows SUNDIALS packages to utilize any valid SUNLINSOL implementation that provides a set of required functions. These functions can be divided into three categories. The first are the core linear solver functions. The second group consists of “set” routines to supply the linear solver object with functions provided by the SUNDIALS package, or for modification of solver parameters. The last group consists of “get” routines for retrieving artifacts (statistics, residual vectors, etc.) from the linear solver. All of these functions are defined in the header file `sundials/sundials_linearsolver.h`.

The implementations provided with SUNDIALS work in coordination with the SUNDIALS generic NVECTOR and SUNMATRIX modules to provide a set of compatible data structures and solvers for the solution of linear systems using direct or matrix-free iterative methods. Moreover, advanced users can provide a customized `SUNLinearSolver` implementation to any SUNDIALS package, particularly in cases where they provide their own NVECTOR and/or SUNMATRIX modules.

Historically, the SUNDIALS packages have been designed to specifically leverage the use of either *direct linear solvers* or matrix-free, *scaled, preconditioned, iterative linear solvers*. However, matrix-based iterative linear solvers are supported.

The iterative linear solvers packaged with SUNDIALS leverage scaling and preconditioning, as applicable, to balance error between solution components and to accelerate convergence of the linear solver. To this end, instead of solving the linear system  $Ax = b$  directly, these apply the underlying iterative algorithm to the transformed system

$$\tilde{A}\tilde{x} = \tilde{b} \tag{8.1}$$

where

$$\begin{aligned} \tilde{A} &= S_1 P_1^{-1} A P_2^{-1} S_2^{-1}, \\ \tilde{b} &= S_1 P_1^{-1} b, \\ \tilde{x} &= S_2 P_2 x, \end{aligned} \tag{8.2}$$

and where

- $P_1$  is the left preconditioner,
- $P_2$  is the right preconditioner,
- $S_1$  is a diagonal matrix of scale factors for  $P_1^{-1}b$ ,
- $S_2$  is a diagonal matrix of scale factors for  $P_2x$ .

SUNDIALS packages request that iterative linear solvers stop based on the 2-norm of the scaled preconditioned residual meeting a prescribed tolerance

$$\|\tilde{b} - \tilde{A}\tilde{x}\|_2 < \text{tol}.$$

When provided an iterative SUNLINSOL implementation that does not support the scaling matrices  $S_1$  and  $S_2$ , SUNDIALS' packages will adjust the value of tol accordingly. In this case, they instead request that iterative linear solvers stop based on the criteria

$$\|P_1^{-1}b - P_1^{-1}Ax\|_2 < \text{tol}.$$

We note that the corresponding adjustments to tol in this case are non-optimal, in that they cannot balance error between specific entries of the solution  $x$ , only the aggregate error in the overall solution vector.

We further note that not all of the SUNDIALS-provided iterative linear solvers support the full range of the above options (e.g., separate left/right preconditioning), and that some of the SUNDIALS packages only utilize a subset of these options. Further details on these exceptions are described in the documentation for each SUNLINSOL implementation, or for each SUNDIALS package.

### 8.0.1 SUNLinearSolver core functions

The core linear solver functions consist of four required routines to get the linear solver type (`SUNLinSolGetType`), initialize the linear solver object once all solver-specific options have been set (`SUNLinSolInitialize`), set up the linear solver object to utilize an updated matrix  $A$  (`SUNLinSolSetup`), and solve the linear system  $Ax = b$  (`SUNLinSolSolve`). The remaining routine for destruction of the linear solver object (`SUNLinSolFree`) is optional.

#### `SUNLinSolGetType`

Call `type = SUNLinSolGetType(LS);`

Description The *required* function `SUNLinSolGetType` returns the type identifier for the linear solver LS. It is used to determine the solver type (direct or iterative) from the abstract `SUNLinearSolver` interface. This is used to assess compatibility with SUNDIALS-provided linear solver interfaces.

Arguments LS (`SUNLinearSolver`) a SUNLINSOL object.

Return value The return value `type` (of type `int`) will be one of the following:

`SUNNONLINEARSOLVER_DIRECT` 0, the SUNLINSOL module uses direct methods to solve the linear system.

`SUNNONLINEARSOLVER_ITERATIVE` 1, the SUNLINSOL module iteratively solves the linear system, stopping when the linear residual is within a prescribed tolerance.

Notes

#### `SUNLinSolInitialize`

Call `retval = SUNLinSolInitialize(LS);`

Description The *required* function `SUNLinSolInitialize` performs linear solver initialization (assumes that all solver-specific options have been set).

Arguments LS (`SUNLinearSolver`) a SUNLINSOL object.

Return value This should return zero for a successful call, and a negative value for a failure, ideally returning one of the generic error codes listed in Table 8.1.

Notes

**SUNLinSolSetup**

Call	<code>retval = SUNLinSolSetup(LS, A);</code>
Description	The <i>required</i> function <code>SUNLinSolSetup</code> performs any linear solver setup needed, based on an updated system SUNMATRIX <code>A</code> . This may be called frequently (e.g. with a full Newton method) or infrequently (for a modified Newton method), based on the type of integrator and/or nonlinear solver requesting the solves.
Arguments	<code>LS</code> ( <code>SUNLinearSolver</code> ) a SUNLINSOL object. <code>A</code> ( <code>SUNMatrix</code> ) a SUNMATRIX object.
Return value	This should return zero for a successful call, a positive value for a recoverable failure and a negative value for an unrecoverable failure, ideally returning one of the generic error codes listed in Table 8.1.
Notes	

**SUNLinSolSolve**

Call	<code>retval = SUNLinSolSolve(LS, A, x, b, tol);</code>
Description	The <i>required</i> function <code>SUNLinSolSolve</code> solves a linear system $Ax = b$ .
Arguments	<code>LS</code> ( <code>SUNLinearSolver</code> ) a SUNLINSOL object. <code>A</code> ( <code>SUNMatrix</code> ) a SUNMATRIX object. <code>x</code> ( <code>N_Vector</code> ) a NVECTOR object. <code>b</code> ( <code>N_Vector</code> ) a NVECTOR object. <code>tol</code> ( <code>realtype</code> ) the desired linear solver tolerance.
Return value	This should return zero for a successful call, a positive value for a recoverable failure and a negative value for an unrecoverable failure, ideally returning one of the generic error codes listed in Table 8.1.
Notes	<b>Direct solvers:</b> can ignore the <code>*tol*</code> argument.  <b>Matrix-free solvers:</b> can ignore the SUNMATRIX input <code>A</code> since a NULL argument will be passed (these should instead rely on the matrix-vector product function supplied through the routine <code>SUNLinSolSetATimes</code> .  <b>Iterative solvers:</b> These should attempt to solve to the specified tolerance <code>tol</code> in a weighted 2-norm. If the solver does not support scaling then it should just use a 2-norm.

**SUNLinSolFree**

Call	<code>retval = SUNLinSolFree(LS);</code>
Description	The <i>optional</i> function <code>SUNLinSolFree</code> frees memory allocated by the linear solver.
Arguments	<code>LS</code> ( <code>SUNLinearSolver</code> ) a SUNLINSOL object.
Return value	This should return zero for a successful call and a negative value for a failure.
Notes	

## 8.0.2 SUNLinearSolver set functions

The following set functions are used to supply linear solver modules with functions defined by the SUNDIALS packages and to modify solver parameters. Only the routine for setting the matrix-vector product routine is required, and that is only for matrix-free linear solver modules. Otherwise, all other set functions are optional. SUNLINSOL implementations that do not provide the functionality for any optional routine should leave the corresponding function pointer NULL instead of supplying a dummy routine.

**SUNLinSolSetATimes**

Call	<code>retval = SUNLinSolSetATimes(LS, A_data, ATimes);</code>
Description	<p>The function <code>SUNLinSolSetATimes</code> is required for matrix-free linear solvers; otherwise it is optional.</p> <p>This routine provides an <code>ATimesFn</code> function pointer, as well as a <code>void *</code> pointer to a data structure used by this routine, to a linear solver object. SUNDIALS packages will call this function to set the matrix-vector product function to either a solver-provided difference-quotient via vector operations or a user-supplied solver-specific routine.</p>
Arguments	<p><code>LS</code> (<code>SUNLinearSolver</code>) a SUNLINSOL object.</p> <p><code>A_data</code> (<code>void*</code>) data structure passed to <code>ATimes</code>.</p> <p><code>ATimes</code> (<code>ATimesFn</code>) function pointer implementing the matrix-vector product routine.</p>
Return value	This routine should return zero for a successful call, and a negative value for a failure, ideally returning one of the generic error codes listed in Table 8.1.
Notes	

**SUNLinSolSetPreconditioner**

Call	<code>retval = SUNLinSolSetPreconditioner(LS, Pdata, Pset, Psol);</code>
Description	<p>The <i>optional</i> function <code>SUNLinSolSetPreconditioner</code> provides <code>PSetupFn</code> and <code>PSolveFn</code> function pointers that implement the preconditioner solves <math>P_1^{-1}</math> and <math>P_2^{-1}</math> from equations (8.1)-(8.2). This routine will be called by a SUNDIALS package, which will provide translation between the generic <code>Pset</code> and <code>Psol</code> calls and the package- or user-supplied routines.</p>
Arguments	<p><code>LS</code> (<code>SUNLinearSolver</code>) a SUNLINSOL object.</p> <p><code>Pdata</code> (<code>void*</code>) data structure passed to both <code>Pset</code> and <code>Psol</code>.</p> <p><code>Pset</code> (<code>PSetupFn</code>) function pointer implementing the preconditioner setup.</p> <p><code>Psol</code> (<code>PSolveFn</code>) function pointer implementing the preconditioner solve.</p>
Return value	This routine should return zero for a successful call, and a negative value for a failure, ideally returning one of the generic error codes listed in Table 8.1.
Notes	

**SUNLinSolSetScalingVectors**

Call	<code>retval = SUNLinSolSetScalingVectors(LS, s1, s2);</code>
Description	<p>The <i>optional</i> function <code>SUNLinSolSetScalingVectors</code> provides left/right scaling vectors for the linear system solve. Here, <code>s1</code> and <code>s2</code> are <code>NVECTOR</code> of positive scale factors containing the diagonal of the matrices <math>S_1</math> and <math>S_2</math> from equations (8.1)-(8.2), respectively. Neither of these vectors need to be tested for positivity, and a <code>NULL</code> argument for either indicates that the corresponding scaling matrix is the identity.</p>
Arguments	<p><code>LS</code> (<code>SUNLinearSolver</code>) a SUNLINSOL object.</p> <p><code>s1</code> (<code>N_Vector</code>) diagonal of the matrix <math>S_1</math></p> <p><code>s2</code> (<code>N_Vector</code>) diagonal of the matrix <math>S_2</math></p>
Return value	This routine should return zero for a successful call, and a negative value for a failure, ideally returning one of the generic error codes listed in Table 8.1.
Notes	

### 8.0.3 SUNLinearSolver get functions

The following get functions allow SUNDIALS packages to retrieve results from the linear solve. All routines are optional.

**SUNLinSolNumIters**

Call `its = SUNLinSolNumIters(LS);`

Description The *optional* function `SUNLinSolNumIters` should return the number of linear iterations performed in the last ‘solve’ call.

Arguments `LS` (`SUNLinearSolver`) a `SUNLINSOL` object.

Return value `int` containing the number of iterations

Notes

**SUNLinSolResNorm**

Call `rnorm = SUNLinSolResNorm(LS);`

Description The *optional* function `SUNLinSolResNorm` should return the final residual norm from the last ‘solve’ call.

Arguments `LS` (`SUNLinearSolver`) a `SUNLINSOL` object.

Return value `realtype` containing the final residual norm

Notes

**SUNLinSolResid**

Call `rvec = SUNLinSolResid(LS);`

Description If an iterative method computes the preconditioned initial residual and returns with a successful solve without performing any iterations (i.e. either the initial guess or the preconditioner is sufficiently accurate), then this *optional* routine may be called by the SUNDIALS package. This routine should return the `NVECTOR` containing the preconditioned initial residual vector

Arguments `LS` (`SUNLinearSolver`) a `SUNLINSOL` object.

Return value `N_Vector` containing the final residual vector

Notes Since `N_Vector` is actually a pointer, and the results are not modified, this routine should *not* require additional memory allocation. If the `SUNLINSOL` object does not retain a vector for this purpose, then this function pointer should be left `NULL` in the implementation.

**SUNLinSolLastFlag**

Call `lflag = SUNLinSolLastFlag(LS);`

Description The *optional* function `SUNLinSolLastFlag` should return the last error flag encountered within the linear solver. This is not called by the SUNDIALS packages directly; it allows the user to investigate linear solver issues after a failed solve.

Arguments `LS` (`SUNLinearSolver`) a `SUNLINSOL` object.

Return value `long int` containing the most recent error flag

Notes

**SUNLinSolSpace**

Call `retval = SUNLinSolSpace(LS, &lrw, &liw);`

Description The *optional* function `SUNLinSolSpace` should return the storage requirements for the linear solver `LS`.

Arguments `LS` (`SUNLinearSolver`) a `SUNLINSOL` object.

	<code>lrw (long int*)</code> the number of realtype words stored by the linear solver.
	<code>liw (long int*)</code> the number of integer words stored by the linear solver.
Return value	This should return zero for a successful call, and a negative value for a failure, ideally returning one of the generic error codes listed in Table 8.1.
Notes	This function is advisory only, for use in determining a user's total space requirements.

#### 8.0.4 Functions provided by SUNDIALS packages

To interface with the SUNLINSOL modules, the SUNDIALS packages supply a variety of routines for evaluating the matrix-vector product, and setting up and applying the preconditioner. These package-provided routines translate between the user-supplied ODE, DAE or nonlinear systems and the generic interfaces to the linear systems of equations that result in their solution. The types for functions provided to a SUNLINSOL module are defined in the header file `sundials/sundials_iterative.h`, and are described below.

##### ATimesFn

Definition	<code>typedef int (*ATimesFn)(void *A_data, N_Vector v, N_Vector z);</code>
Purpose	These functions compute the action of a matrix on a vector, performing the operation $z = Av$ . Memory for <code>z</code> should already be allocated prior to calling this function. The vector <code>v</code> should be left unchanged.
Arguments	<code>A_data</code> is a pointer to client data, the same as that supplied to <code>SUNLinSolSetATimes</code> . <code>v</code> is the input vector to multiply. <code>z</code> is the output vector computed.
Return value	This routine should return 0 if successful and a non-zero value if unsuccessful.
Notes	

##### PSetupFn

Definition	<code>typedef int (*PSetupFn)(void *P_data)</code>
Purpose	These functions set up any requisite problem data in preparation for calls to the corresponding <code>PSolveFn</code> .
Arguments	<code>P_data</code> is a pointer to client data, the same pointer as that supplied to the routine <code>SUNLinSolSetPreconditioner</code> .
Return value	This routine should return 0 if successful and a non-zero value if unsuccessful.
Notes	

##### PSolveFn

Definition	<code>typedef int (*PSolveFn)(void *P_data, N_Vector r, N_Vector z, realtype tol, int lr)</code>
Purpose	These functions solve the preconditioner equation $Pz = r$ for the vector $z$ . Memory for <code>z</code> should already be allocated prior to calling this function. The parameter <code>P_data</code> is a pointer to any information about $P$ which the function needs in order to do its job (set up by the corresponding <code>PSetupFn</code> ). The parameter <code>lr</code> is input, and indicates whether $P$ is to be taken as the left preconditioner or the right preconditioner: <code>lr = 1</code> for left and <code>lr = 2</code> for right. If preconditioning is on one side only, <code>lr</code> can be ignored. If the preconditioner is iterative, then it should strive to solve the preconditioner equation so that

$$\|Pz - r\|_{\text{wrms}} < \text{tol}$$

where the weight vector for the WRMS norm may be accessed from the main package memory structure. The vector **r** should not be modified by the **PSolveFn**.

**Arguments** **P\_data** is a pointer to client data, the same pointer as that supplied to the routine **SUNLinSolSetPreconditioner**.

**r** is the right-hand side vector for the preconditioner system

**z** is the solution vector for the preconditioner system

**tol** is the desired tolerance for an iterative preconditioner

**lr** is flag indicating whether the routine should perform left (1) or right (2) preconditioning.

**Return value** This routine should return 0 if successful and a non-zero value if unsuccessful. On a failure, a negative return value indicates an unrecoverable condition, while a positive value indicates a recoverable one, in which the calling routine may reattempt the solution after updating preconditioner data.

**Notes**

### 8.0.5 SUNLinearSolver return codes

The functions provided to SUNLINSOL modules by each SUNDIALS package, and functions within the SUNDIALS-provided SUNLINSOL implementations utilize a common set of return codes, shown in the Table 8.1. These adhere to a common pattern: 0 indicates success, a positive value corresponds to a recoverable failure, and a negative value indicates a non-recoverable failure. Aside from this pattern, the actual values of each error code are primarily to provide additional information to the user in case of a linear solver failure.

Table 8.1: Description of the **SUNLinearSolver** error codes

Name	Value	Description
SUNLS_SUCCESS	0	successful call or converged solve
SUNLS_MEM_NULL	-1	the memory argument to the function is NULL
SUNLS_ILL_INPUT	-2	an illegal input has been provided to the function
SUNLS_MEM_FAIL	-3	failed memory access or allocation
SUNLS_ATHES_FAIL_UNREC	-4	an unrecoverable failure occurred in the <b>ATHes</b> routine
SUNLS_PSET_FAIL_UNREC	-5	an unrecoverable failure occurred in the <b>Pset</b> routine
SUNLS_PSOLVE_FAIL_UNREC	-6	an unrecoverable failure occurred in the <b>Psolve</b> routine
SUNLS_PACKAGE_FAIL_UNREC	-7	an unrecoverable failure occurred in an external linear solver package
SUNLS_GS_FAIL	-8	a failure occurred during Gram-Schmidt orthogonalization (SUNLINSOL_SPGMR/SUNLINSOL_SPGFMR)
SUNLS_QRSOL_FAIL	-9	a singular <i>R</i> matrix was encountered in a QR factorization (SUNLINSOL_SPGMR/SUNLINSOL_SPGFMR)

*continued on next page*

Name	Value	Description
SUNLS_RES_REDUCED	1	an iterative solver reduced the residual, but did not converge to the desired tolerance
SUNLS_CONV_FAIL	2	an iterative solver did not converge (and the residual was not reduced)
SUNLS_ATIMES_FAIL_REC	3	a recoverable failure occurred in the <code>ATimes</code> routine
SUNLS_PSET_FAIL_REC	4	a recoverable failure occurred in the <code>Pset</code> routine
SUNLS_PSOLVE_FAIL_REC	5	a recoverable failure occurred in the <code>Psolve</code> routine
SUNLS_PACKAGE_FAIL_REC	6	a recoverable failure occurred in an external linear solver package
SUNLS_QRFACT_FAIL	7	a singular matrix was encountered during a QR factorization (SUNLINSOL_SPGMR/SUNLINSOL_SPGMR)
SUNLS_LUFACT_FAIL	8	a singular matrix was encountered during a LU factorization (SUNLINSOL_DENSE/SUNLINSOL_BAND)

### 8.0.6 The generic SUNLinearSolver module

SUNDIALS packages interact with specific SUNLINSOL implementations through the generic SUNLINSOL module on which all other SUNLINSOL implementations are built. The `SUNLinearSolver` type is a pointer to a structure containing an implementation-dependent *content* field, and an *ops* field. The type `SUNLinearSolver` is defined as

```
typedef struct _generic_SUNLinearSolver *SUNLinearSolver;
```

```
struct _generic_SUNLinearSolver {
    void *content;
    struct _generic_SUNLinearSolver_Ops *ops;
};
```

where the `_generic_SUNLinearSolver_Ops` structure is a list of pointers to the various actual linear solver operations provided by a specific implementation. The `_generic_SUNLinearSolver_Ops` structure is defined as

```
struct _generic_SUNLinearSolver_Ops {
    SUNLinearSolver_Type (*gettype)(SUNLinearSolver);
    int (*setatimes)(SUNLinearSolver, void*, ATimesFn);
    int (*setpreconditioner)(SUNLinearSolver, void*,
                             PSetupFn, PSolveFn);
    int (*setscalingvectors)(SUNLinearSolver,
                             N_Vector, N_Vector);
    int (*initialize)(SUNLinearSolver);
    int (*setup)(SUNLinearSolver, SUNMatrix);
    int (*solve)(SUNLinearSolver, SUNMatrix, N_Vector,
                 N_Vector, realtype);
    int (*numiters)(SUNLinearSolver);
    realtype (*resnorm)(SUNLinearSolver);
    long int (*lastflag)(SUNLinearSolver);
    int (*space)(SUNLinearSolver, long int*, long int*);
    N_Vector (*resid)(SUNLinearSolver);
    int (*free)(SUNLinearSolver);
};
```



The generic SUNLINSOL module defines and implements the linear solver operations defined in Sections 8.0.1-8.0.3. These routines are in fact only wrappers to the linear solver operations defined by a particular SUNLINSOL implementation, which are accessed through the *ops* field of the **SUNLinearSolver** structure. To illustrate this point we show below the implementation of a typical linear solver operation from the generic SUNLINSOL module, namely **SUNLinSolInitialize**, which initializes a SUNLINSOL object for use after it has been created and configured, and returns a flag denoting a successful/failed operation:

```
int SUNLinSolInitialize(SUNLinearSolver S)
{
    return ((int) S->ops->initialize(S));
}
```

## 8.1 Compatibility of SUNLinearSolver modules

We note that not all SUNLINSOL types are compatible with all SUNMATRIX and NVECTOR types provided with SUNDIALS. In Table 8.2 we show the matrix-based linear solvers available as SUNLINSOL modules, and the compatible matrix implementations. Recall that Table 4.1 shows the compatibility between all SUNLINSOL modules and vector implementations.

Table 8.2: SUNDIALS matrix-based linear solvers and matrix implementations that can be used for each.

Linear Solver Interface	Dense Matrix	Banded Matrix	Sparse Matrix	User Supplied
Dense	✓			✓
Band		✓		✓
LapackDense	✓			✓
LapackBand		✓		✓
KLU			✓	✓
SUPERLUMT			✓	✓
User supplied	✓	✓	✓	✓

## 8.2 Implementing a custom SUNLinearSolver module

A particular implementation of the SUNLINSOL module must:

- Specify the *content* field of the **SUNLinearSolver** object.
- Define and implement a minimal subset of the linear solver operations. See the documentation for each SUNDIALS linear solver interface to determine which SUNLINSOL operations they require. Note that the names of these routines should be unique to that implementation in order to permit using more than one SUNLINSOL module (each with different **SUNLinearSolver** internal data representations) in the same code.
- Define and implement user-callable constructor and destructor routines to create and free a **SUNLinearSolver** with the new *content* field and with *ops* pointing to the new linear solver operations.

We note that the function pointers for all unsupported optional routines should be set to NULL in the *ops* structure. This allows the SUNDIALS package that is using the SUNLINSOL object to know that the associated functionality is not supported.

Additionally, a SUNLINSOL implementation *may* do the following:

- Define and implement additional user-callable “set” routines acting on the `SUNLinearSolver`, e.g., for setting various configuration options to tune the linear solver to a particular problem.
- Provide additional user-callable “get” routines acting on the `SUNLinearSolver` object, e.g., for returning various solve statistics.

### 8.3 The `SUNLinearSolver_Dense` implementation

The dense implementation of the `SUNLINSOL` module provided with `SUNDIALS`, `SUNLINSOL_DENSE`, is designed to be used with the corresponding `SUNMATRIX_DENSE` matrix type, and one of the serial or shared-memory `NVECTOR` implementations (`NVECTOR_SERIAL`, `NVECTOR_OPENMP` or `NVECTOR_PTHREADS`).

#### 8.3.1 `SUNLINSOL_DENSE` usage

The header file to include when using this module is `sunlinsol/sunlinsol_dense.h`. The `SUNLINSOL_DENSE` module is accessible from all `SUNDIALS` solvers *without* linking to the `libsundials_sunlinsoldense` module library.

The module `SUNLINSOL_DENSE` provides the following user-callable constructor routine:

##### `SUNLinSol_Dense`

Call `LS = SUNLinSol_Dense(y, A);`

Description The function `SUNLinSol_Dense` creates and allocates memory for a dense `SUNLinearSolver` object.

Arguments `y` (`N.Vector`) a template for cloning vectors needed within the solver  
`A` (`SUNMatrix`) a `SUNMATRIX_DENSE` matrix template for cloning matrices needed within the solver

Return value This returns a `SUNLinearSolver` object. If either `A` or `y` are incompatible then this routine will return `NULL`.

Notes This routine will perform consistency checks to ensure that it is called with consistent `NVECTOR` and `SUNMATRIX` implementations. These are currently limited to the `SUNMATRIX_DENSE` matrix type and the `NVECTOR_SERIAL`, `NVECTOR_OPENMP`, and `NVECTOR_PTHREADS` vector types. As additional compatible matrix and vector implementations are added to `SUNDIALS`, these will be included within this compatibility check.

For backwards compatibility, we also provide the wrapper function,

- `SUNDenseLinearSolver`

Wrapper function for `SUNLinSol_Dense`, with identical input and output arguments.

For solvers that include a Fortran interface module, the `SUNLINSOL_DENSE` module also includes a Fortran-callable function for creating a `SUNLinearSolver` object.

##### `FSUNDENSELINSOLINIT`

Call `FSUNDENSELINSOLINIT(code, ier)`

Description The function `FSUNDENSELINSOLINIT` can be called for Fortran programs to create a dense `SUNLinearSolver` object.

Arguments `code` (`int*`) is an integer input specifying the solver id (1 for `CVODE`, 2 for `IDA`, 3 for `KINSOL`, and 4 for `ARKODE`).

Return value `ier` is a return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes This routine must be called *after* both the NVECTOR and SUNMATRIX objects have been initialized.

Additionally, when using ARKODE with a non-identity mass matrix, the SUNLINSOL\_DENSE module includes a Fortran-callable function for creating a SUNLinearSolver mass matrix solver object.

#### FSUNMASSDENSELINSOLINIT

Call FSUNMASSDENSELINSOLINIT(*ier*)

Description The function FSUNMASSDENSELINSOLINIT can be called for Fortran programs to create a dense SUNLinearSolver object for mass matrix linear systems.

Arguments

Return value *ier* is a `int` return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes This routine must be called *after* both the NVECTOR and SUNMATRIX mass-matrix objects have been initialized.

### 8.3.2 SUNLINSOL\_DENSE description

The SUNLINSOL\_DENSE module defines the *content* field of a SUNLinearSolver to be the following structure:

```
struct _SUNLinearSolverContent_Dense {
    sunindextype N;
    sunindextype *pivots;
    long int last_flag;
};
```

These entries of the *content* field contain the following information:

**N** - size of the linear system,

**pivots** - index array for partial pivoting in LU factorization,

**last\_flag** - last error return flag from internal function evaluations.

This solver is constructed to perform the following operations:

- The “setup” call performs a *LU* factorization with partial (row) pivoting ( $\mathcal{O}(N^3)$  cost),  $PA = LU$ , where  $P$  is a permutation matrix,  $L$  is a lower triangular matrix with 1’s on the diagonal, and  $U$  is an upper triangular matrix. This factorization is stored in-place on the input SUNMATRIX\_DENSE object  $A$ , with pivoting information encoding  $P$  stored in the **pivots** array.
- The “solve” call performs pivoting and forward and backward substitution using the stored **pivots** array and the *LU* factors held in the SUNMATRIX\_DENSE object ( $\mathcal{O}(N^2)$  cost).

The SUNLINSOL\_DENSE module defines dense implementations of all “direct” linear solver operations listed in Sections 8.0.1-8.0.3:

- **SUNLinSolGetType\_Dense**
- **SUNLinSolInitialize\_Dense** – this does nothing, since all consistency checks are performed at solver creation.
- **SUNLinSolSetup\_Dense** – this performs the *LU* factorization.
- **SUNLinSolSolve\_Dense** – this uses the *LU* factors and **pivots** array to perform the solve.
- **SUNLinSolLastFlag\_Dense**
- **SUNLinSolSpace\_Dense** – this only returns information for the storage *within* the solver object, i.e. storage for **N**, **last\_flag**, and **pivots**.
- **SUNLinSolFree\_Dense**

## 8.4 The SUNLinearSolver\_Band implementation

The band implementation of the SUNLINSOL module provided with SUNDIALS, SUNLINSOL\_BAND, is designed to be used with the corresponding SUNMATRIX\_BAND matrix type, and one of the serial or shared-memory NVECTOR implementations (NVECTOR\_SERIAL, NVECTOR\_OPENMP or NVECTOR\_PTHREADS).

### 8.4.1 SUNLINSOL\_BAND usage

The header file to include when using this module is `sunlinsol/sunlinsol.band.h`. The SUNLINSOL\_BAND module is accessible from all SUNDIALS solvers *without* linking to the `libsundials_sunlinsolband` module library.

The module SUNLINSOL\_BAND provides the following user-callable constructor routine:

#### SUNLinSol\_Band

Call	<code>LS = SUNLinSol_Band(y, A);</code>
Description	The function <code>SUNLinSol_Band</code> creates and allocates memory for a band <code>SUNLinearSolver</code> object.
Arguments	<code>y</code> ( <code>N_Vector</code> ) a template for cloning vectors needed within the solver <code>A</code> ( <code>SUNMatrix</code> ) a <code>SUNMATRIX_BAND</code> matrix template for cloning matrices needed within the solver
Return value	This returns a <code>SUNLinearSolver</code> object. If either <code>A</code> or <code>y</code> are incompatible then this routine will return <code>NULL</code> .
Notes	This routine will perform consistency checks to ensure that it is called with consistent <code>NVECTOR</code> and <code>SUNMATRIX</code> implementations. These are currently limited to the <code>SUNMATRIX_DENSE</code> matrix type and the <code>NVECTOR_SERIAL</code> , <code>NVECTOR_OPENMP</code> , and <code>NVECTOR_PTHREADS</code> vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.  Additionally, this routine will verify that the input matrix <code>A</code> is allocated with appropriate upper bandwidth storage for the <i>LU</i> factorization.

For backwards compatibility, we also provide the wrapper functions:

- `SUNBandLinearSolver`

Wrapper function for `SUNLinSol_Band`, with identical input and output arguments.

For solvers that include a Fortran interface module, the SUNLINSOL\_BAND module also includes a Fortran-callable function for creating a `SUNLinearSolver` object.

#### FSUNBANDLINSOLINIT

Call	<code>FSUNBANDLINSOLINIT(code, ier)</code>
Description	The function <code>FSUNBANDLINSOLINIT</code> can be called for Fortran programs to create a band <code>SUNLinearSolver</code> object.
Arguments	<code>code</code> ( <code>int*</code> ) is an integer input specifying the solver id (1 for <code>CVODE</code> , 2 for <code>IDA</code> , 3 for <code>KINSOL</code> , and 4 for <code>ARKODE</code> ).
Return value	<code>ier</code> is a return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes	This routine must be called <i>after</i> both the <code>NVECTOR</code> and <code>SUNMATRIX</code> objects have been initialized.

Additionally, when using `ARKODE` with a non-identity mass matrix, the SUNLINSOL\_BAND module includes a Fortran-callable function for creating a `SUNLinearSolver` mass matrix solver object.

**FSUNMASSBANDLINSOLINIT**

Call `FSUNMASSBANDLINSOLINIT(ier)`

Description The function `FSUNMASSBANDLINSOLINIT` can be called for Fortran programs to create a band `SUNLinearSolver` object for mass matrix linear systems.

Arguments

Return value `ier` is a `int` return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes This routine must be called *after* both the `NVECTOR` and `SUNMATRIX` mass-matrix objects have been initialized.

**8.4.2 SUNLINSOL\_BAND description**

The `SUNLINSOL_BAND` module defines the *content* field of a `SUNLinearSolver` to be the following structure:

```
struct _SUNLinearSolverContent_Band {
    sunindextype N;
    sunindextype *pivots;
    long int last_flag;
};
```

These entries of the *content* field contain the following information:

**N** - size of the linear system,

**pivots** - index array for partial pivoting in LU factorization,

**last\_flag** - last error return flag from internal function evaluations.

This solver is constructed to perform the following operations:

- The “setup” call performs a *LU* factorization with partial (row) pivoting,  $PA = LU$ , where  $P$  is a permutation matrix,  $L$  is a lower triangular matrix with 1’s on the diagonal, and  $U$  is an upper triangular matrix. This factorization is stored in-place on the input `SUNMATRIX_BAND` object  $A$ , with pivoting information encoding  $P$  stored in the **pivots** array.
- The “solve” call performs pivoting and forward and backward substitution using the stored **pivots** array and the *LU* factors held in the `SUNMATRIX_BAND` object.
- $A$  must be allocated to accommodate the increase in upper bandwidth that occurs during factorization. More precisely, if  $A$  is a band matrix with upper bandwidth `mu` and lower bandwidth `m1`, then the upper triangular factor  $U$  can have upper bandwidth as big as  $\text{smu} = \text{MIN}(N-1, \text{mu} + \text{m1})$ . The lower triangular factor  $L$  has lower bandwidth `m1`.



The `SUNLINSOL_BAND` module defines band implementations of all “direct” linear solver operations listed in Sections 8.0.1-8.0.3:

- `SUNLinSolGetType_Band`
- `SUNLinSolInitialize_Band` – this does nothing, since all consistency checks are performed at solver creation.
- `SUNLinSolSetup_Band` – this performs the *LU* factorization.
- `SUNLinSolSolve_Band` – this uses the *LU* factors and **pivots** array to perform the solve.
- `SUNLinSolLastFlag_Band`
- `SUNLinSolSpace_Band` – this only returns information for the storage *within* the solver object, i.e. storage for `N`, `last_flag`, and **pivots**.
- `SUNLinSolFree_Band`

## 8.5 The SUNLinearSolver\_LapackDense implementation

The LAPACK dense implementation of the SUNLINSOL module provided with SUNDIALS, SUNLINSOL\_LAPACKDENSE, is designed to be used with the corresponding SUNMATRIX\_DENSE matrix type, and one of the serial or shared-memory NVECTOR implementations (NVECTOR\_SERIAL, NVECTOR\_OPENMP, or NVECTOR\_PTHREADS).

### 8.5.1 SUNLINSOL\_LAPACKDENSE usage

The header file to include when using this module is `sunlinsol/sunlinsol_lapackdense.h`. The installed module library to link to is `libsundials_sunlinsollapackdense.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

The module SUNLINSOL\_LAPACKDENSE provides the following user-callable constructor routine:

#### SUNLinSol\_LapackDense

Call	<code>LS = SUNLinSol_LapackDense(y, A);</code>
Description	The function <code>SUNLinSol_LapackDense</code> creates and allocates memory for a LAPACK-based, dense <code>SUNLinearSolver</code> object.
Arguments	<code>y</code> ( <code>N_Vector</code> ) a template for cloning vectors needed within the solver <code>A</code> ( <code>SUNMatrix</code> ) a <code>SUNMATRIX_DENSE</code> matrix template for cloning matrices needed within the solver
Return value	This returns a <code>SUNLinearSolver</code> object. If either <code>A</code> or <code>y</code> are incompatible then this routine will return <code>NULL</code> .
Notes	This routine will perform consistency checks to ensure that it is called with consistent <code>NVECTOR</code> and <code>SUNMATRIX</code> implementations. These are currently limited to the <code>SUNMATRIX_DENSE</code> matrix type and the <code>NVECTOR_SERIAL</code> , <code>NVECTOR_OPENMP</code> , and <code>NVECTOR_PTHREADS</code> vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.

For backwards compatibility, we also provide the wrapper function,

- `SUNLapackDense`

Wrapper function for `SUNLinSol_LapackDense`, with identical input and output arguments.

For solvers that include a Fortran interface module, the SUNLINSOL\_LAPACKDENSE module also includes a Fortran-callable function for creating a `SUNLinearSolver` object.

#### FSUNLAPACKDENSEINIT

Call	<code>FSUNLAPACKDENSEINIT(code, ier)</code>
Description	The function <code>FSUNLAPACKDENSEINIT</code> can be called for Fortran programs to create a LAPACK-based dense <code>SUNLinearSolver</code> object.
Arguments	<code>code</code> ( <code>int*</code> ) is an integer input specifying the solver id (1 for <code>CVODE</code> , 2 for <code>IDA</code> , 3 for <code>KINSOL</code> , and 4 for <code>ARKODE</code> ).
Return value	<code>ier</code> is a return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes	This routine must be called <i>after</i> both the <code>NVECTOR</code> and <code>SUNMATRIX</code> objects have been initialized.

Additionally, when using `ARKODE` with a non-identity mass matrix, the SUNLINSOL\_LAPACKDENSE module includes a Fortran-callable function for creating a `SUNLinearSolver` mass matrix solver object.

**FSUNMASSLAPACKDENSEINIT**

Call `FSUNMASSLAPACKDENSEINIT(ier)`

Description The function `FSUNMASSLAPACKDENSEINIT` can be called for Fortran programs to create a LAPACK-based, dense `SUNLinearSolver` object for mass matrix linear systems.

Arguments

Return value `ier` is a `int` return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes This routine must be called *after* both the `NVECTOR` and `SUNMATRIX` mass-matrix objects have been initialized.

**8.5.2 SUNLINSOL\_LAPACKDENSE description**

The `SUNLINSOL_LAPACKDENSE` module defines the *content* field of a `SUNLinearSolver` to be the following structure:

```
struct _SUNLinearSolverContent_Dense {
    sunindextype N;
    sunindextype *pivots;
    long int last_flag;
};
```

These entries of the *content* field contain the following information:

**N** - size of the linear system,

**pivots** - index array for partial pivoting in LU factorization,

**last\_flag** - last error return flag from internal function evaluations.

The `SUNLINSOL_LAPACKDENSE` module is a `SUNLINSOL` wrapper for the LAPACK dense matrix factorization and solve routines, `*GETRF` and `*GETRS`, where `*` is either `D` or `S`, depending on whether `SUNDIALS` was configured to have `realtype` set to `double` or `single`, respectively (see Section 4.2). In order to use the `SUNLINSOL_LAPACKDENSE` module it is assumed that LAPACK has been installed on the system prior to installation of `SUNDIALS`, and that `SUNDIALS` has been configured appropriately to link with LAPACK (see Appendix A for details). We note that since there do not exist 128-bit floating-point factorization and solve routines in LAPACK, this interface cannot be compiled when using `extended` precision for `realtype`. Similarly, since there do not exist 64-bit integer LAPACK routines, the `SUNLINSOL_LAPACKDENSE` module also cannot be compiled when using `int64_t` for the `sunindextype`.



This solver is constructed to perform the following operations:

- The “setup” call performs a *LU* factorization with partial (row) pivoting ( $\mathcal{O}(N^3)$  cost),  $PA = LU$ , where  $P$  is a permutation matrix,  $L$  is a lower triangular matrix with 1’s on the diagonal, and  $U$  is an upper triangular matrix. This factorization is stored in-place on the input `SUNMATRIX_DENSE` object  $A$ , with pivoting information encoding  $P$  stored in the `pivots` array.
- The “solve” call performs pivoting and forward and backward substitution using the stored `pivots` array and the *LU* factors held in the `SUNMATRIX_DENSE` object ( $\mathcal{O}(N^2)$  cost).

The `SUNLINSOL_LAPACKDENSE` module defines dense implementations of all “direct” linear solver operations listed in Sections 8.0.1-8.0.3:

- `SUNLinSolGetType_LapackDense`
- `SUNLinSolInitialize_LapackDense` – this does nothing, since all consistency checks are performed at solver creation.

- `SUNLinSolSetup_LapackDense` – this calls either `DGETRF` or `SGETRF` to perform the *LU* factorization.
- `SUNLinSolSolve_LapackDense` – this calls either `DGETRS` or `SGETRS` to use the *LU* factors and `pivots` array to perform the solve.
- `SUNLinSolLastFlag_LapackDense`
- `SUNLinSolSpace_LapackDense` – this only returns information for the storage *within* the solver object, i.e. storage for `N`, `last_flag`, and `pivots`.
- `SUNLinSolFree_LapackDense`

## 8.6 The SUNLinearSolver\_LapackBand implementation

The LAPACK band implementation of the SUNLINSOL module provided with SUNDIALS, `SUNLINSOL_LAPACKBAND`, is designed to be used with the corresponding `SUNMATRIX_BAND` matrix type, and one of the serial or shared-memory `NVECTOR` implementations (`NVECTOR_SERIAL`, `NVECTOR_OPENMP`, or `NVECTOR_PTHREADS`).

### 8.6.1 SUNLINSOL\_LAPACKBAND usage

The header file to include when using this module is `sunlinsol/sunlinsol_lapackband.h`. The installed module library to link to is `libsundials_sunlinsollapackband.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

The module `SUNLINSOL_LAPACKBAND` provides the following user-callable routine:

<b>SUNLinSol_LapackBand</b>
-----------------------------

Call	<code>LS = SUNLinSol_LapackBand(y, A);</code>
Description	The function <code>SUNLinSol_LapackBand</code> creates and allocates memory for a LAPACK-based, band <code>SUNLinearSolver</code> object.
Arguments	<code>y</code> ( <code>N_Vector</code> ) a template for cloning vectors needed within the solver <code>A</code> ( <code>SUNMatrix</code> ) a <code>SUNMATRIX_BAND</code> matrix template for cloning matrices needed within the solver
Return value	This returns a <code>SUNLinearSolver</code> object. If either <code>A</code> or <code>y</code> are incompatible then this routine will return <code>NULL</code> .
Notes	This routine will perform consistency checks to ensure that it is called with consistent <code>NVECTOR</code> and <code>SUNMATRIX</code> implementations. These are currently limited to the <code>SUNMATRIX_BAND</code> matrix type and the <code>NVECTOR_SERIAL</code> , <code>NVECTOR_OPENMP</code> , and <code>NVECTOR_PTHREADS</code> vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.  Additionally, this routine will verify that the input matrix <code>A</code> is allocated with appropriate upper bandwidth storage for the <i>LU</i> factorization.

For backwards compatibility, we also provide the wrapper functions:

- `SUNLapackBand`

Wrapper function for `SUNLinSol_LapackBand`, with identical input and output arguments.

For solvers that include a Fortran interface module, the `SUNLINSOL_LAPACKBAND` module also includes a Fortran-callable function for creating a `SUNLinearSolver` object.



**FSUNLAPACKBANDINIT**

Call	FSUNLAPACKBANDINIT( <i>code</i> , <i>ier</i> )
Description	The function FSUNLAPACKBANDINIT can be called for Fortran programs to create a LAPACK-based band SUNLinearSolver object.
Arguments	<i>code</i> ( <i>int*</i> ) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).
Return value	<i>ier</i> is a return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes	This routine must be called <i>after</i> both the NVECTOR and SUNMATRIX objects have been initialized.

Additionally, when using ARKODE with a non-identity mass matrix, the SUNLINSOL\_LAPACKBAND module includes a Fortran-callable function for creating a SUNLinearSolver mass matrix solver object.

**FSUNMASSLAPACKBANDINIT**

Call	FSUNMASSLAPACKBANDINIT( <i>ier</i> )
Description	The function FSUNMASSLAPACKBANDINIT can be called for Fortran programs to create a LAPACK-based, band SUNLinearSolver object for mass matrix linear systems.
Arguments	
Return value	<i>ier</i> is a <i>int</i> return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes	This routine must be called <i>after</i> both the NVECTOR and SUNMATRIX mass-matrix objects have been initialized.

**8.6.2 SUNLINSOL\_LAPACKBAND description**

The SUNLINSOL\_LAPACKBAND module defines the *content* field of a SUNLinearSolver to be the following structure:

```
struct _SUNLinearSolverContent_Band {
    sunindextype N;
    sunindextype *pivots;
    long int last_flag;
};
```

These entries of the *content* field contain the following information:

**N** - size of the linear system,


**pivots** - index array for partial pivoting in LU factorization,

**last\_flag** - last error return flag from internal function evaluations.

The SUNLINSOL\_LAPACKBAND module is a SUNLINSOL wrapper for the LAPACK band matrix factorization and solve routines, \*GBTRF and \*GBTRS, where \* is either D or S, depending on whether SUNDIALS was configured to have *realtype* set to *double* or *single*, respectively (see Section 4.2). In order to use the SUNLINSOL\_LAPACKBAND module it is assumed that LAPACK has been installed on the system prior to installation of SUNDIALS, and that SUNDIALS has been configured appropriately to link with LAPACK (see Appendix A for details). We note that since there do not exist 128-bit floating-point factorization and solve routines in LAPACK, this interface cannot be compiled when using *extended* precision for *realtype*. Similarly, since there do not exist 64-bit integer LAPACK routines, the SUNLINSOL\_LAPACKBAND module also cannot be compiled when using *int64.t* for the *sunindextype*.

This solver is constructed to perform the following operations:



- The “setup” call performs a  $LU$  factorization with partial (row) pivoting,  $PA = LU$ , where  $P$  is a permutation matrix,  $L$  is a lower triangular matrix with 1’s on the diagonal, and  $U$  is an upper triangular matrix. This factorization is stored in-place on the input `SUNMATRIX_BAND` object  $A$ , with pivoting information encoding  $P$  stored in the `pivots` array.
- The “solve” call performs pivoting and forward and backward substitution using the stored `pivots` array and the  $LU$  factors held in the `SUNMATRIX_BAND` object.
-   $A$  must be allocated to accommodate the increase in upper bandwidth that occurs during factorization. More precisely, if  $A$  is a band matrix with upper bandwidth `mu` and lower bandwidth `ml`, then the upper triangular factor  $U$  can have upper bandwidth as big as `smu = MIN(N-1, mu+ml)`. The lower triangular factor  $L$  has lower bandwidth `ml`.

The `SUNLINSOL_LAPACKBAND` module defines band implementations of all “direct” linear solver operations listed in Sections 8.0.1-8.0.3:

- `SUNLinSolGetType_LapackBand`
- `SUNLinSolInitialize_LapackBand` – this does nothing, since all consistency checks are performed at solver creation.
- `SUNLinSolSetup_LapackBand` – this calls either `DGBTRF` or `SGBTRF` to perform the  $LU$  factorization.
- `SUNLinSolSolve_LapackBand` – this calls either `DGBTRS` or `SGBTRS` to use the  $LU$  factors and `pivots` array to perform the solve.
- `SUNLinSolLastFlag_LapackBand`
- `SUNLinSolSpace_LapackBand` – this only returns information for the storage *within* the solver object, i.e. storage for `N`, `last_flag`, and `pivots`.
- `SUNLinSolFree_LapackBand`

## 8.7 The SUNLinearSolver\_KLU implementation

The KLU implementation of the `SUNLINSOL` module provided with SUNDIALS, `SUNLINSOL_KLU`, is designed to be used with the corresponding `SUNMATRIX_SPARSE` matrix type, and one of the serial or shared-memory `NVECTOR` implementations (`NVECTOR_SERIAL`, `NVECTOR_OPENMP`, or `NVECTOR_PTHREADS`).

### 8.7.1 SUNLINSOL\_KLU usage

The header file to include when using this module is `sunlinsol/sunlinsol_klu.h`. The installed module library to link to is `libsundials_sunlinsolklu.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

The module `SUNLINSOL_KLU` provides the following user-callable routines:

<code>SUNLinSol_KLU</code>	
Call	<code>LS = SUNLinSol_KLU(y, A);</code>
Description	The function <code>SUNLinSol_KLU</code> creates and allocates memory for a <code>SUNLINSOL_KLU</code> object.
Arguments	<code>y</code> ( <code>N_Vector</code> ) a template for cloning vectors needed within the solver <code>A</code> ( <code>SUNMatrix</code> ) a <code>SUNMATRIX_SPARSE</code> matrix template for cloning matrices needed within the solver
Return value	This returns a <code>SUNLinearSolver</code> object. If either <code>A</code> or <code>y</code> are incompatible then this routine will return <code>NULL</code> .

Notes This routine will perform consistency checks to ensure that it is called with consistent NVECTOR and SUNMATRIX implementations. These are currently limited to the SUNMATRIX\_SPARSE matrix type (using either CSR or CSC storage formats) and the NVECTOR\_SERIAL, NVECTOR\_OPENMP, and NVECTOR\_PTHREADS vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.

#### SUNLinSol\_KLUReInit

Call `retval = SUNLinSol_KLUReInit(LS, A, nnz, reinit_type);`

Description The function `SUNLinSol_KLUReInit` reinitializes memory and flags for a new factorization (symbolic and numeric) to be conducted at the next solver setup call. This routine is useful in the cases where the number of nonzeros has changed or if the structure of the linear system has changed which would require a new symbolic (and numeric factorization).

Arguments

LS	( <code>SUNLinearSolver</code> ) a template for cloning vectors needed within the solver
A	( <code>SUNMatrix</code> ) a <code>SUNMATRIX_SPARSE</code> matrix template for cloning matrices needed within the solver
nnz	( <code>sunindextype</code> ) the new number of nonzeros in the matrix
reinit_type	( <code>int</code> ) flag governing the level of reinitialization. The allowed values are: <ul style="list-style-type: none"> <li>• <code>SUNKLU_REINIT_FULL</code> – The Jacobian matrix will be destroyed and a new one will be allocated based on the <code>nnz</code> value passed to this call. New symbolic and numeric factorizations will be completed at the next solver setup.</li> <li>• <code>SUNKLU_REINIT_PARTIAL</code> – Only symbolic and numeric factorizations will be completed. It is assumed that the Jacobian size has not exceeded the size of <code>nnz</code> given in the sparse matrix provided to the original constructor routine (or the previous <code>SUNLinSol_KLUReInit</code> call).</li> </ul>

Return value The return values from this function are `SUNLS_MEM_NULL` (either `S` or `A` are `NULL`), `SUNLS_ILL_INPUT` (`A` does not have type `SUNMATRIX_SPARSE` or `reinit_type` is invalid), `SUNLS_MEM_FAIL` (reallocation of the sparse matrix failed) or `SUNLS_SUCCESS`.

Notes This routine will perform consistency checks to ensure that it is called with consistent NVECTOR and SUNMATRIX implementations. These are currently limited to the SUNMATRIX\_SPARSE matrix type (using either CSR or CSC storage formats) and the NVECTOR\_SERIAL, NVECTOR\_OPENMP, and NVECTOR\_PTHREADS vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.

This routine assumes no other changes to solver use are necessary.

#### SUNLinSol\_KLUSetOrdering

Call `retval = SUNLinSol_KLUSetOrdering(LS, ordering);`

Description This function sets the ordering used by KLU for reducing fill in the linear solve.

Arguments

LS	( <code>SUNLinearSolver</code> ) the <code>SUNLINSOL_KLU</code> object
ordering	( <code>int</code> ) flag indication the reordering algorithm to use. Options include: <ul style="list-style-type: none"> <li>0 AMD,</li> <li>1 COLAMD, and</li> <li>2 the natural ordering.</li> </ul>

The default is 1 for COLAMD.

**Return value** The return values from this function are `SUNLS_MEM_NULL` (`S` is `NULL`), `SUNLS_ILL_INPUT` (invalid `ordering`), or `SUNLS_SUCCESS`.

**Notes**

For backwards compatibility, we also provide the wrapper functions, each with identical input and output arguments to the routines that they wrap:

- `SUNKLU`  
Wrapper function for `SUNLinSol_KLU`
- `SUNKLUREInit`  
Wrapper function for `SUNLinSol_KLUREInit`
- `SUNKLUSetOrdering`  
Wrapper function for `SUNLinSol_KLUSetOrdering`

For solvers that include a Fortran interface module, the `SUNLINSOL_KLU` module also includes a Fortran-callable function for creating a `SUNLinearSolver` object.

#### FSUNKLUINIT

**Call** `FSUNKLUINIT(code, ier)`

**Description** The function `FSUNKLUINIT` can be called for Fortran programs to create a `SUNLINSOL_KLU` object.

**Arguments** `code` (`int*`) is an integer input specifying the solver id (1 for `CVODE`, 2 for `IDA`, 3 for `KINSOL`, and 4 for `ARKODE`).

**Return value** `ier` is a return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

**Notes** This routine must be called *after* both the `NVECTOR` and `SUNMATRIX` objects have been initialized.

Additionally, when using `ARKODE` with a non-identity mass matrix, the `SUNLINSOL_KLU` module includes a Fortran-callable function for creating a `SUNLinearSolver` mass matrix solver object.

#### FSUNMASSKLUIT

**Call** `FSUNMASSKLUIT(ier)`

**Description** The function `FSUNMASSKLUIT` can be called for Fortran programs to create a `SUNLINSOL_KLU` object for mass matrix linear systems.

**Arguments**

**Return value** `ier` is a `int` return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

**Notes** This routine must be called *after* both the `NVECTOR` and `SUNMATRIX` mass-matrix objects have been initialized.

The `SUNLinSol_KLUREInit` and `SUNLinSol_KLUSetOrdering` routines also support Fortran interfaces for the system and mass matrix solvers:

#### FSUNKLUREINIT

**Call** `FSUNKLUREINIT(code, nnz, reinit_type, ier)`

**Description** The function `FSUNKLUREINIT` can be called for Fortran programs to re-initialize a `SUNLINSOL_KLU` object.

Arguments	<b>code</b>	( <b>int*</b> ) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).
	<b>nnz</b>	( <b>sunindextype*</b> ) the new number of nonzeros in the matrix
	<b>reinit_type</b>	( <b>int*</b> ) flag governing the level of reinitialization. The allowed values are: <ul style="list-style-type: none"> <li>1 – The Jacobian matrix will be destroyed and a new one will be allocated based on the <b>nnz</b> value passed to this call. New symbolic and numeric factorizations will be completed at the next solver setup.</li> <li>2 – Only symbolic and numeric factorizations will be completed. It is assumed that the Jacobian size has not exceeded the size of <b>nnz</b> given in the sparse matrix provided to the original constructor routine (or the previous <b>SUNLinSol_KLUReInit</b> call).</li> </ul>
Return value	<b>ier</b>	is a <b>int</b> return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes		See <b>SUNLinSol_KLUReInit</b> for complete further documentation of this routine.

**FSUNMASSKLUREINIT**

Call	<b>FSUNMASSKLUREINIT</b> ( <b>nnz</b> , <b>reinit_type</b> , <b>ier</b> )	
Description	The function <b>FSUNMASSKLUREINIT</b> can be called for Fortran programs to re-initialize a <b>SUNLINSOL_KLU</b> object for mass matrix linear systems.	
Arguments	The arguments are identical to <b>FSUNKLUREINIT</b> above, except that <b>code</b> is not needed since mass matrix linear systems only arise in ARKODE.	
Return value	<b>ier</b> is a <b>int</b> return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.	
Notes	See <b>SUNLinSol_KLUReInit</b> for complete further documentation of this routine.	

**FSUNKLUSETORDERING**

Call	<b>FSUNKLUSETORDERING</b> ( <b>code</b> , <b>ordering</b> , <b>ier</b> )	
Description	The function <b>FSUNKLUSETORDERING</b> can be called for Fortran programs to change the reordering algorithm used by KLU.	
Arguments	<b>code</b>	( <b>int*</b> ) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).
	<b>ordering</b>	( <b>int*</b> ) flag indication the reordering algorithm to use. Options include: <ul style="list-style-type: none"> <li>0 AMD,</li> <li>1 COLAMD, and</li> <li>2 the natural ordering.</li> </ul> The default is 1 for COLAMD.
Return value	<b>ier</b> is a <b>int</b> return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.	
Notes	See <b>SUNLinSol_KLUSetOrdering</b> for complete further documentation of this routine.	

**FSUNMASSKLUSETORDERING**

Call	<b>FSUNMASSKLUSETORDERING</b> ( <b>ier</b> )	
Description	The function <b>FSUNMASSKLUSETORDERING</b> can be called for Fortran programs to change the reordering algorithm used by KLU for mass matrix linear systems.	
Arguments	The arguments are identical to <b>FSUNKLUSETORDERING</b> above, except that <b>code</b> is not needed since mass matrix linear systems only arise in ARKODE.	

Return value **ier** is a **int** return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes See `SUNLinSol_KLUSetOrdering` for complete further documentation of this routine.

## 8.7.2 SUNLINSOL\_KLU description

The `SUNLINSOL_KLU` module defines the *content* field of a `SUNLinearSolver` to be the following structure:

```
struct _SUNLinearSolverContent_KLU {
    long int      last_flag;
    int           first_factorize;
    sun_klu_symbolic *symbolic;
    sun_klu_numeric *numeric;
    sun_klu_common common;
    sunindextype   (*klu_solver)(sun_klu_symbolic*, sun_klu_numeric*,
                                sunindextype, sunindextype,
                                double*, sun_klu_common*);
};
```

These entries of the *content* field contain the following information:

**last\_flag** - last error return flag from internal function evaluations,

**first\_factorize** - flag indicating whether the factorization has ever been performed,

**symbolic** - KLU storage structure for symbolic factorization components,

**numeric** - KLU storage structure for numeric factorization components,

**common** - storage structure for common KLU solver components,

**klu\_solver** – pointer to the appropriate KLU solver function (depending on whether it is using a CSR or CSC sparse matrix).



The `SUNLINSOL_KLU` module is a `SUNLINSOL` wrapper for the KLU sparse matrix factorization and solver library written by Tim Davis [1, 15]. In order to use the `SUNLINSOL_KLU` interface to KLU, it is assumed that KLU has been installed on the system prior to installation of `SUNDIALS`, and that `SUNDIALS` has been configured appropriately to link with KLU (see Appendix A for details). Additionally, this wrapper only supports double-precision calculations, and therefore cannot be compiled if `SUNDIALS` is configured to have **realtype** set to either **extended** or **single** (see Section 4.2). Since the KLU library supports both 32-bit and 64-bit integers, this interface will be compiled for either of the available **sunindextype** options.

The KLU library has a symbolic factorization routine that computes the permutation of the linear system matrix to block triangular form and the permutations that will pre-order the diagonal blocks (the only ones that need to be factored) to reduce fill-in (using AMD, COLAMD, CHOLAMD, natural, or an ordering given by the user). Of these ordering choices, the default value in the `SUNLINSOL_KLU` module is the COLAMD ordering.

KLU breaks the factorization into two separate parts. The first is a symbolic factorization and the second is a numeric factorization that returns the factored matrix along with final pivot information. KLU also has a refactor routine that can be called instead of the numeric factorization. This routine will reuse the pivot information. This routine also returns diagnostic information that a user can examine to determine if numerical stability is being lost and a full numerical factorization should be done instead of the refactor.

Since the linear systems that arise within the context of `SUNDIALS` calculations will typically have identical sparsity patterns, the `SUNLINSOL_KLU` module is constructed to perform the following operations:

- The first time that the “setup” routine is called, it performs the symbolic factorization, followed by an initial numerical factorization.
- On subsequent calls to the “setup” routine, it calls the appropriate KLU “refactor” routine, followed by estimates of the numerical conditioning using the relevant “rcond”, and if necessary “condest”, routine(s). If these estimates of the condition number are larger than  $\varepsilon^{-2/3}$  (where  $\varepsilon$  is the double-precision unit roundoff), then a new factorization is performed.
- The module includes the routine `SUNKLUReInit`, that can be called by the user to force a full or partial refactorization at the next “setup” call.
- The “solve” call performs pivoting and forward and backward substitution using the stored KLU data structures. We note that in this solve KLU operates on the native data arrays for the right-hand side and solution vectors, without requiring costly data copies.

The `SUNLINSOL_KLU` module defines implementations of all “direct” linear solver operations listed in Sections 8.0.1-8.0.3:

- `SUNLinSolGetType_KLU`
- `SUNLinSolInitialize_KLU` – this sets the `first_factorize` flag to 1, forcing both symbolic and numerical factorizations on the subsequent “setup” call.
- `SUNLinSolSetup_KLU` – this performs either a *LU* factorization or refactorization of the input matrix.
- `SUNLinSolSolve_KLU` – this calls the appropriate KLU solve routine to utilize the *LU* factors to solve the linear system.
- `SUNLinSolLastFlag_KLU`
- `SUNLinSolSpace_KLU` – this only returns information for the storage within the solver *interface*, i.e. storage for the integers `last_flag` and `first_factorize`. For additional space requirements, see the KLU documentation.
- `SUNLinSolFree_KLU`

## 8.8 The SUNLinearSolver\_SuperLUMT implementation

The SUPERLUMT implementation of the SUNLINSOL module provided with SUNDIALS, `SUNLINSOL_SUPERLUMT`, is designed to be used with the corresponding `SUNMATRIX_SPARSE` matrix type, and one of the serial or shared-memory `NVECTOR` implementations (`NVECTOR_SERIAL`, `NVECTOR_OPENMP`, or `NVECTOR_PTHREADS`). While these are compatible, it is not recommended to use a threaded vector module with `SUNLINSOL_SUPERLUMT` unless it is the `NVECTOR_OPENMP` module and the SUPERLUMT library has also been compiled with OpenMP.

### 8.8.1 SUNLINSOL\_SUPERLUMT usage

The header file to include when using this module is `sunlinsol/sunlinsol_superluml.h`. The installed module library to link to is `libsundials_sunlinsolsuperluml.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

The module `SUNLINSOL_SUPERLUMT` provides the following user-callable routines:

**SUNLinSol\_SuperLUMT**

Call	LS = SUNLinSol_SuperLUMT(y, A, num_threads);		
Description	The function SUNLinSol_SuperLUMT creates and allocates memory for a SUNLINSOL_SUPERLUMT object.		
Arguments	y	(N_Vector) a template for cloning vectors needed within the solver	
	A	(SUNMatrix) a SUNMATRIX_SPARSE matrix template for cloning matrices needed within the solver	
	num_threads	(int) desired number of threads (OpenMP or Pthreads, depending on how SUPERLUMT was installed) to use during the factorization steps	
Return value	This returns a SUNLinearSolver object. If either A or y are incompatible then this routine will return NULL.		
Notes	This routine analyzes the input matrix and vector to determine the linear system size and to assess compatibility with the SUPERLUMT library.		
	This routine will perform consistency checks to ensure that it is called with consistent NVECTOR and SUNMATRIX implementations. These are currently limited to the SUNMATRIX_SPARSE matrix type (using either CSR or CSC storage formats) and the NVECTOR_SERIAL, NVECTOR_OPENMP, and NVECTOR_PTHREADS vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.		
	The num_threads argument is not checked and is passed directly to SUPERLUMT routines.		

**SUNLinSol\_SuperLUMTSetOrdering**

Call	<code>retval = SUNLinSol_SuperLUMTSetOrdering(LS, ordering);</code>		
Description	The function <code>SUNLinSol_SuperLUMTSetOrdering</code> sets the ordering used by <code>SUPERLUMT</code> for reducing fill in the linear solve.		
Arguments	LS	(SUNLinearSolver) the <code>SUNLINSOL_SUPERLUMT</code> object	
	ordering	(int) a flag indicating the ordering algorithm, options are:	
		0 natural ordering 1 minimal degree ordering on $A^T A$ 2 minimal degree ordering on $A^T + A$ 3 COLAMD ordering for unsymmetric matrices  The default is 3 for COLAMD.	
Return value	The return values from this function are <code>SUNLS_MEM_NULL</code> (S is <code>NULL</code> ), <code>SUNLS_ILL_INPUT</code> (invalid <code>ordering_choice</code> ), or <code>SUNLS_SUCCESS</code> .		

## Notes

For backwards compatibility, we also provide the wrapper functions, each with identical input and output arguments to the routines that they wrap:

- **SUNSuperLUMT**  
Wrapper function for `SUNLinSol_SuperLUMT`
- **SUNSuperLUMTSetOrdering**  
Wrapper function for `SUNLinSol_SuperLUMTSetOrdering`

For solvers that include a Fortran interface module, the `SUNLINSOL_SUPERLUMT` module also includes a Fortran-callable function for creating a `SUNLinearSolver` object.



**FSUNSUPERLUMTINIT**

Call	FSUNSUPERLUMTINIT( <i>code</i> , <i>num_threads</i> , <i>ier</i> )
Description	The function FSUNSUPERLUMTINIT can be called for Fortran programs to create a SUNLINSOL_KLU object.
Arguments	<p><i>code</i> (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).</p> <p><i>num_threads</i> (int*) desired number of threads (OpenMP or Pthreads, depending on how SUPERLUMT was installed) to use during the factorization steps</p>
Return value	<i>ier</i> is a return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes	This routine must be called <i>after</i> both the NVECTOR and SUNMATRIX objects have been initialized.

Additionally, when using ARKODE with a non-identity mass matrix, the SUNLINSOL\_SUPERLUMT module includes a Fortran-callable function for creating a **SUNLinearSolver** mass matrix solver object.

**FSUNMASSSUPERLUMTINIT**

Call	FSUNMASSSUPERLUMTINIT( <i>num_threads</i> , <i>ier</i> )
Description	The function FSUNMASSSUPERLUMTINIT can be called for Fortran programs to create a SUNLINSOL_SUPERLUMT object for mass matrix linear systems.
Arguments	<i>num_threads</i> (int*) desired number of threads (OpenMP or Pthreads, depending on how SUPERLUMT was installed) to use during the factorization steps.
Return value	<i>ier</i> is a int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes	This routine must be called <i>after</i> both the NVECTOR and SUNMATRIX mass-matrix objects have been initialized.

The **SUNLinSol\_SuperLUMTSetOrdering** routine also supports Fortran interfaces for the system and mass matrix solvers:

**FSUNSUPERLUMTSETORDERING**

Call	FSUNSUPERLUMTSETORDERING( <i>code</i> , <i>ordering</i> , <i>ier</i> )
Description	The function FSUNSUPERLUMTSETORDERING can be called for Fortran programs to update the ordering algorithm in a SUNLINSOL_SUPERLUMT object.
Arguments	<p><i>code</i> (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).</p> <p><i>ordering</i> (int*) a flag indicating the ordering algorithm, options are:</p> <ul style="list-style-type: none"> <li>0 natural ordering</li> <li>1 minimal degree ordering on <math>A^T A</math></li> <li>2 minimal degree ordering on <math>A^T + A</math></li> <li>3 COLAMD ordering for unsymmetric matrices</li> </ul> <p>The default is 3 for COLAMD.</p>
Return value	<i>ier</i> is a int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes	See <b>SUNLinSol_SuperLUMTSetOrdering</b> for complete further documentation of this routine.

**FSUNMASSUPERLUMTSETORDERING**

Call	FSUNMASSUPERLUMTSETORDERING(ordering, ier)
Description	The function FSUNMASSUPERLUMTSETORDERING can be called for Fortran programs to update the ordering algorithm in a SUNLINSOL_SUPERLUMT object for mass matrix linear systems.
Arguments	<p>ordering (int*) a flag indicating the ordering algorithm, options are:</p> <ul style="list-style-type: none"> <li>0 natural ordering</li> <li>1 minimal degree ordering on <math>A^T A</math></li> <li>2 minimal degree ordering on <math>A^T + A</math></li> <li>3 COLAMD ordering for unsymmetric matrices</li> </ul> <p>The default is 3 for COLAMD.</p>
Return value	ier is a int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes	See SUNLinSol_SuperLUMTSetOrdering for complete further documentation of this routine.

**8.8.2 SUNLINSOL\_SUPERLUMT description**

The SUNLINSOL\_SUPERLUMT module defines the *content* field of a SUNLinearSolver to be the following structure:

```

struct _SUNLinearSolverContent_SuperLUMT {
    long int    last_flag;
    int         first_factorize;
    SuperMatrix *A, *AC, *L, *U, *B;
    Gstat_t     *Gstat;
    sunindextype *perm_r, *perm_c;
    sunindextype N;
    int         num_threads;
    realtype     diag_pivot_thresh;
    int         ordering;
    superlumt_options_t *options;
};

```

These entries of the *content* field contain the following information:

**last\_flag** - last error return flag from internal function evaluations,

**first\_factorize** - flag indicating whether the factorization has ever been performed,

**A, AC, L, U, B** - SuperMatrix pointers used in solve,

**Gstat** - GStat\_t object used in solve,

**perm\_r, perm\_c** - permutation arrays used in solve,

**N** - size of the linear system,

**num\_threads** - number of OpenMP/Pthreads threads to use,

**diag\_pivot\_thresh** - threshold on diagonal pivoting,

**ordering** - flag for which reordering algorithm to use,

**options** - pointer to SUPERLUMT options structure.



The SUNLINSOL\_SUPERLUMT module is a SUNLINSOL wrapper for the SUPERLUMT sparse matrix factorization and solver library written by X. Sherry Li [2, 29, 17]. The package performs matrix factorization using threads to enhance efficiency in shared memory parallel environments. It should be noted that threads are only used in the factorization step. In order to use the SUNLINSOL\_SUPERLUMT interface to SUPERLUMT, it is assumed that SUPERLUMT has been installed on the system prior to installation of SUNDIALS, and that SUNDIALS has been configured appropriately to link with SUPERLUMT (see Appendix A for details). Additionally, this wrapper only supports single- and double-precision calculations, and therefore cannot be compiled if SUNDIALS is configured to have `realtype` set to `extended` (see Section 4.2). Moreover, since the SUPERLUMT library may be installed to support either 32-bit or 64-bit integers, it is assumed that the SUPERLUMT library is installed using the same integer precision as the SUNDIALS `sunindextype` option.

The SUPERLUMT library has a symbolic factorization routine that computes the permutation of the linear system matrix to reduce fill-in on subsequent  $LU$  factorizations (using COLAMD, minimal degree ordering on  $A^T * A$ , minimal degree ordering on  $A^T + A$ , or natural ordering). Of these ordering choices, the default value in the SUNLINSOL\_SUPERLUMT module is the COLAMD ordering.

Since the linear systems that arise within the context of SUNDIALS calculations will typically have identical sparsity patterns, the SUNLINSOL\_SUPERLUMT module is constructed to perform the following operations:

- The first time that the “setup” routine is called, it performs the symbolic factorization, followed by an initial numerical factorization.
- On subsequent calls to the “setup” routine, it skips the symbolic factorization, and only refactors the input matrix.
- The “solve” call performs pivoting and forward and backward substitution using the stored SUPERLUMT data structures. We note that in this solve SUPERLUMT operates on the native data arrays for the right-hand side and solution vectors, without requiring costly data copies.

The SUNLINSOL\_SUPERLUMT module defines implementations of all “direct” linear solver operations listed in Sections 8.0.1-8.0.3:

- `SUNLinSolGetType_SuperLUMT`
- `SUNLinSolInitialize_SuperLUMT` – this sets the `first_factorize` flag to 1 and resets the internal SUPERLUMT statistics variables.
- `SUNLinSolSetup_SuperLUMT` – this performs either a  $LU$  factorization or refactorization of the input matrix.
- `SUNLinSolSolve_SuperLUMT` – this calls the appropriate SUPERLUMT solve routine to utilize the  $LU$  factors to solve the linear system.
- `SUNLinSolLastFlag_SuperLUMT`
- `SUNLinSolSpace_SuperLUMT` – this only returns information for the storage within the solver *interface*, i.e. storage for the integers `last_flag` and `first_factorize`. For additional space requirements, see the SUPERLUMT documentation.
- `SUNLinSolFree_SuperLUMT`

## 8.9 The SUNLinearSolver\_SPGMR implementation

The SPGMR (Scaled, Preconditioned, Generalized Minimum Residual [33]) implementation of the SUNLINSOL module provided with SUNDIALS, SUNLINSOL\_SPGMR, is an iterative linear solver that is designed to be compatible with any NVECTOR implementation (serial, threaded, parallel, and user-supplied) that supports a minimal subset of operations (`N_VClone`, `N_VDotProd`, `N_VScale`, `N_VLinearSum`, `N_VProd`, `N_VConst`, `N_VDiv`, and `N_VDestroy`). When using Classical Gram-Schmidt, the optional function `N_VDotProdMulti` may be supplied for increased efficiency.

### 8.9.1 SUNLINSOL\_SPGMR usage

The header file to include when using this module is `sunlinsol/sunlinsol_spgmr.h`. The SUNLINSOL\_SPGMR module is accessible from all SUNDIALS solvers *without* linking to the `libsundials_sunlinsolspgmr` module library.

The module SUNLINSOL\_SPGMR provides the following user-callable routines:

#### SUNLinSol\_SPGMR

Call	<code>LS = SUNLinSol_SPGMR(y, pretype, maxl);</code>
Description	The function <code>SUNLinSol_SPGMR</code> creates and allocates memory for a SPGMR <code>SUNLinearSolver</code> .
Arguments	<p><code>y</code> (N_Vector) a template for cloning vectors needed within the solver</p> <p><code>pretype</code> (int) flag indicating the desired type of preconditioning, allowed values are:</p> <ul style="list-style-type: none"> <li>• <code>PREC_NONE</code> (0)</li> <li>• <code>PREC_LEFT</code> (1)</li> <li>• <code>PREC_RIGHT</code> (2)</li> <li>• <code>PREC_BOTH</code> (3)</li> </ul> <p>Any other integer input will result in the default (no preconditioning).</p> <p><code>maxl</code> (int) the number of Krylov basis vectors to use. values <math>\leq 0</math> will result in the default value (5).</p>
Return value	This returns a <code>SUNLinearSolver</code> object. If either <code>y</code> is incompatible then this routine will return <code>NULL</code> .
Notes	<p>This routine will perform consistency checks to ensure that it is called with a consistent NVECTOR implementation (i.e. that it supplies the requisite vector operations). If <code>y</code> is incompatible, then this routine will return <code>NULL</code>.</p> <p>We note that some SUNDIALS solvers are designed to only work with left preconditioning (IDA and IDAS) and others with only right preconditioning (KINSOL). While it is possible to configure a SUNLINSOL_SPGMR object to use any of the preconditioning options with these solvers, this use mode is not supported and may result in inferior performance.</p>

#### SUNLinSol\_SPGMRSetPrecType

Call	<code>retval = SUNLinSol_SPGMRSetPrecType(LS, pretype);</code>
Description	The function <code>SUNLinSol_SPGMRSetPrecType</code> updates the type of preconditioning to use in the SUNLINSOL_SPGMR object.
Arguments	<p><code>LS</code> (SUNLinearSolver) the SUNLINSOL_SPGMR object to update</p> <p><code>pretype</code> (int) flag indicating the desired type of preconditioning, allowed values match those discussed in <code>SUNLinSol_SPGMR</code>.</p>
Return value	This routine will return with one of the error codes <code>SUNLS_ILL_INPUT</code> (illegal <code>pretype</code> ), <code>SUNLS_MEM_NULL</code> (S is NULL) or <code>SUNLS_SUCCESS</code> .
Notes	

#### SUNLinSol\_SPGMRSetGSType

Call	<code>retval = SUNLinSol_SPGMRSetGSType(LS, gstype);</code>
Description	The function <code>SUNLinSol_SPGMRSetGSType</code> sets the type of Gram-Schmidt orthogonalization to use in the SUNLINSOL_SPGMR object.
Arguments	<p><code>LS</code> (SUNLinearSolver) the SUNLINSOL_SPGMR object to update</p> <p><code>gstype</code> (int) flag indicating the desired orthogonalization algorithm; allowed values are:</p>

- MODIFIED\_GS (1)
- CLASSICAL\_GS (2)

Any other integer input will result in a failure, returning error code `SUNLS_ILL_INPUT`.

Return value This routine will return with one of the error codes `SUNLS_ILL_INPUT` (illegal `pretype`), `SUNLS_MEM_NULL` (`S` is `NULL`) or `SUNLS_SUCCESS`.

Notes

#### `SUNLinSol_SPGMRSetMaxRestarts`

Call `retval = SUNLinSol_SPGMRSetMaxRestarts(LS, maxrs);`

Description The function `SUNLinSol_SPGMRSetMaxRestarts` sets the number of GMRES restarts to allow in the `SUNLINSOL_SPGMR` object.

Arguments `LS` (`SUNLinearSolver`) the `SUNLINSOL_SPGMR` object to update  
`maxrs` (`int`) integer indicating number of restarts to allow. A negative input will result in the default of 0.

Return value This routine will return with one of the error codes `SUNLS_MEM_NULL` (`S` is `NULL`) or `SUNLS_SUCCESS`.

Notes

For backwards compatibility, we also provide the wrapper functions, each with identical input and output arguments to the routines that they wrap:

- `SUNSPGMR`  
 Wrapper function for `SUNLinSol_SPGMR`
- `SUNSPGMRSetPrecType`  
 Wrapper function for `SUNLinSol_SPGMRSetPrecType`
- `SUNSPGMRSetGSType`  
 Wrapper function for `SUNLinSol_SPGMRSetGSType`
- `SUNSPGMRSetMaxRestarts`  
 Wrapper function for `SUNLinSol_SPGMRSetMaxRestarts`

For solvers that include a Fortran interface module, the `SUNLINSOL_SPGMR` module also includes a Fortran-callable function for creating a `SUNLinearSolver` object.

#### `FSUNSPGMRINIT`

Call `FSUNSPGMRINIT(code, pretype, maxl, ier)`

Description The function `FSUNSPGMRINIT` can be called for Fortran programs to create a `SUNLINSOL_SPGMR` object.

Arguments `code` (`int*`) is an integer input specifying the solver id (1 for `CVODE`, 2 for `IDA`, 3 for `KINSOL`, and 4 for `ARKODE`).  
`pretype` (`int*`) flag indicating desired preconditioning type  
`maxl` (`int*`) flag indicating Krylov subspace size

Return value `ier` is a return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes This routine must be called *after* the `NVECTOR` object has been initialized.

Allowable values for `pretype` and `maxl` are the same as for the C function `SUNLinSol_SPGMR`.

Additionally, when using `ARKODE` with a non-identity mass matrix, the `SUNLINSOL_SPGMR` module includes a Fortran-callable function for creating a `SUNLinearSolver` mass matrix solver object.

**FSUNMASSSPGMRINIT**

**Call** FSUNMASSSPGMRINIT(*pretype*, *maxl*, *ier*)

**Description** The function FSUNMASSSPGMRINIT can be called for Fortran programs to create a SUNLINSOL\_SPGMR object for mass matrix linear systems.

**Arguments** *pretype* (*int\**) flag indicating desired preconditioning type  
*maxl* (*int\**) flag indicating Krylov subspace size

**Return value** *ier* is a *int* return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

**Notes** This routine must be called *after* the NVECTOR object has been initialized.  
 Allowable values for *pretype* and *maxl* are the same as for the C function SUNLinSol\_SPGMR. The SUNLinSol\_SPGMRSetPrecType, SUNLinSol\_SPGMRSetGSType and SUNLinSol\_SPGMRSetMaxRestarts routines also support Fortran interfaces for the system and mass matrix solvers

**FSUNSPGMRSETGSTYPE**

**Call** FSUNSPGMRSETGSTYPE(*code*, *gstype*, *ier*)

**Description** The function FSUNSPGMRSETGSTYPE can be called for Fortran programs to change the Gram-Schmidt orthogonalization algorithm.

**Arguments** *code* (*int\**) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).  
*gstype* (*int\**) flag indicating the desired orthogonalization algorithm.

**Return value** *ier* is a *int* return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

**Notes** See SUNLinSol\_SPGMRSetGSType for complete further documentation of this routine.

**FSUNMASSSPGMRSETGSTYPE**

**Call** FSUNMASSSPGMRSETGSTYPE(*gstype*, *ier*)

**Description** The function FSUNMASSSPGMRSETGSTYPE can be called for Fortran programs to change the Gram-Schmidt orthogonalization algorithm for mass matrix linear systems.

**Arguments** The arguments are identical to FSUNSPGMRSETGSTYPE above, except that *code* is not needed since mass matrix linear systems only arise in ARKODE.

**Return value** *ier* is a *int* return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

**Notes** See SUNLinSol\_SPGMRSetGSType for complete further documentation of this routine.

**FSUNSPGMRSETPRECTYPE**

**Call** FSUNSPGMRSETPRECTYPE(*code*, *pretype*, *ier*)

**Description** The function FSUNSPGMRSETPRECTYPE can be called for Fortran programs to change the type of preconditioning to use.

**Arguments** *code* (*int\**) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).  
*pretype* (*int\**) flag indicating the type of preconditioning to use.

**Return value** *ier* is a *int* return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

**Notes** See SUNLinSol\_SPGMRSetPrecType for complete further documentation of this routine.

**FSUNMASSSPGMRSETPRECTYPE**

Call	FSUNMASSSPGMRSETPRECTYPE( <i>pretype</i> , <i>ier</i> )
Description	The function FSUNMASSSPGMRSETPRECTYPE can be called for Fortran programs to change the type of preconditioning for mass matrix linear systems.
Arguments	The arguments are identical to FSUNSPGMRSETPRECTYPE above, except that <i>code</i> is not needed since mass matrix linear systems only arise in ARKODE.
Return value	<i>ier</i> is a <i>int</i> return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes	See SUNLinSol_SPGMRSetPrecType for complete further documentation of this routine.

**FSUNSPGMRSETMAXRS**

Call	FSUNSPGMRSETMAXRS( <i>code</i> , <i>maxrs</i> , <i>ier</i> )
Description	The function FSUNSPGMRSETMAXRS can be called for Fortran programs to change the maximum number of restarts allowed for SPGMR.
Arguments	<i>code</i> ( <i>int*</i> ) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE). <i>maxrs</i> ( <i>int*</i> ) maximum allowed number of restarts.
Return value	<i>ier</i> is a <i>int</i> return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes	See SUNLinSol_SPGMRSetMaxRestarts for complete further documentation of this routine.

**FSUNMASSSPGMRSETMAXRS**

Call	FSUNMASSSPGMRSETMAXRS( <i>maxrs</i> , <i>ier</i> )
Description	The function FSUNMASSSPGMRSETMAXRS can be called for Fortran programs to change the maximum number of restarts allowed for SPGMR for mass matrix linear systems.
Arguments	The arguments are identical to FSUNSPGMRSETMAXRS above, except that <i>code</i> is not needed since mass matrix linear systems only arise in ARKODE.
Return value	<i>ier</i> is a <i>int</i> return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes	See SUNLinSol_SPGMRSetMaxRestarts for complete further documentation of this routine.

**8.9.2 SUNLINSOL\_SPGMR description**

The SUNLINSOL\_SPGMR module defines the *content* field of a SUNLinearSolver to be the following structure:

```
struct _SUNLinearSolverContent_SPGMR {
    int maxl;
    int pretype;
    int gstype;
    int max_restarts;
    int numiters;
    realtype resnorm;
    long int last_flag;
    ATimesFn ATimes;
    void* ATData;
    PSetupFn Psetup;
```

```

PSolveFn Psolve;
void* PData;
N_Vector s1;
N_Vector s2;
N_Vector *V;
realtype **Hes;
realtype *givens;
N_Vector xcor;
realtype *yg;
N_Vector vtemp;
};

```

These entries of the *content* field contain the following information:

**maxl** - number of GMRES basis vectors to use (default is 5),

**pretype** - flag for type of preconditioning to employ (default is none),

**gstype** - flag for type of Gram-Schmidt orthogonalization (default is modified Gram-Schmidt),

**max\_restarts** - number of GMRES restarts to allow (default is 0),

**numiters** - number of iterations from the most-recent solve,

**resnorm** - final linear residual norm from the most-recent solve,

**last\_flag** - last error return flag from an internal function,

**ATimes** - function pointer to perform  $Av$  product,

**ATData** - pointer to structure for **ATimes**,

**Psetup** - function pointer to preconditioner setup routine,

**Psolve** - function pointer to preconditioner solve routine,

**PData** - pointer to structure for **Psetup** and **Psolve**,

**s1, s2** - vector pointers for supplied scaling matrices (default is NULL),

**V** - the array of Krylov basis vectors  $v_1, \dots, v_{\max l+1}$ , stored in  $V[0], \dots, V[\max l]$ . Each  $v_i$  is a vector of type NVECTOR.,

**Hes** - the  $(\max l + 1) \times \max l$  Hessenberg matrix. It is stored row-wise so that the  $(i,j)$ th element is given by  $Hes[i][j]$ .,

**givens** - a length  $2*\max l$  array which represents the Givens rotation matrices that arise in the GMRES

algorithm. These matrices are  $F_0, F_1, \dots, F_j$ , where  $F_i =$

$$\begin{bmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & & c_i & -s_i & \\ & & & s_i & c_i & \\ & & & & & 1 \\ & & & & & & \ddots & \\ & & & & & & & 1 \end{bmatrix},$$

are represented in the **givens** vector as  $givens[0] = c_0$ ,  $givens[1] = s_0$ ,  $givens[2] = c_1$ ,  $givens[3] = s_1, \dots, givens[2j] = c_j$ ,  $givens[2j+1] = s_j$ .,

**xcor** - a vector which holds the scaled, preconditioned correction to the initial guess,



**yg** - a length (maxl+1) array of `realtype` values used to hold “short” vectors (e.g.  $y$  and  $g$ ),

**vtemp** - temporary vector storage.

This solver is constructed to perform the following operations:

- During construction, the `xcor` and `vtemp` arrays are cloned from a template `NVECTOR` that is input, and default solver parameters are set.
- User-facing “set” routines may be called to modify default solver parameters.
- Additional “set” routines are called by the SUNDIALS solver that interfaces with `SUNLINSOL_SPGMR` to supply the `ATimes`, `PSetup`, and `Psolve` function pointers and `s1` and `s2` scaling vectors.
- In the “initialize” call, the remaining solver data is allocated (`V`, `Hes`, `givens`, and `yg` )
- In the “setup” call, any non-NULL `PSetup` function is called. Typically, this is provided by the SUNDIALS solver itself, that translates between the generic `PSetup` function and the solver-specific routine (solver-supplied or user-supplied).
- In the “solve” call, the GMRES iteration is performed. This will include scaling, preconditioning, and restarts if those options have been supplied.

The `SUNLINSOL_SPGMR` module defines implementations of all “iterative” linear solver operations listed in Sections 8.0.1-8.0.3:

- `SUNLinSolGetType_SPGMR`
- `SUNLinSolInitialize_SPGMR`
- `SUNLinSolSetATimes_SPGMR`
- `SUNLinSolSetPreconditioner_SPGMR`
- `SUNLinSolSetScalingVectors_SPGMR`
- `SUNLinSolSetup_SPGMR`
- `SUNLinSolSolve_SPGMR`
- `SUNLinSolNumIters_SPGMR`
- `SUNLinSolResNorm_SPGMR`
- `SUNLinSolResid_SPGMR`
- `SUNLinSolLastFlag_SPGMR`
- `SUNLinSolSpace_SPGMR`
- `SUNLinSolFree_SPGMR`

## 8.10 The SUNLinearSolver\_SPFGMR implementation

The SPFGMR (Scaled, Preconditioned, Flexible, Generalized Minimum Residual [32]) implementation of the `SUNLINSOL` module provided with SUNDIALS, `SUNLINSOL_SPFGMR`, is an iterative linear solver that is designed to be compatible with any `NVECTOR` implementation (serial, threaded, parallel, and user-supplied) that supports a minimal subset of operations (`N_VClone`, `N_VDotProd`, `N_VScale`, `N_VLinearSum`, `N_VProd`, `N_VConst`, `N_VDiv`, and `N_VDestroy`). When using Classical Gram-Schmidt, the optional function `N_VDotProdMulti` may be supplied for increased efficiency. Unlike the other Krylov iterative linear solvers supplied with SUNDIALS, FGMRES is specifically designed to work with a changing preconditioner (e.g. from an iterative method).

### 8.10.1 SUNLINSOL\_SPFGMR usage

The header file to include when using this module is `sunlinsol/sunlinsol.spfgmr.h`. The SUNLINSOL\_SPFGMR module is accessible from all SUNDIALS solvers *without* linking to the `libsundials_sunlinsolspfgmr` module library.

The module SUNLINSOL\_SPFGMR provides the following user-callable routines:

#### SUNLinSol\_SPFGMR

Call	<code>LS = SUNLinSol_SPFGMR(y, pretype, maxl);</code>
Description	The function <code>SUNLinSol_SPFGMR</code> creates and allocates memory for a SPFGMR <code>SUNLinearSolver</code> .
Arguments	<p><code>y</code> (<code>N_Vector</code>) a template for cloning vectors needed within the solver</p> <p><code>pretype</code> (<code>int</code>) flag indicating the desired type of preconditioning, allowed values are:</p> <ul style="list-style-type: none"> <li>• <code>PREC_NONE</code> (0)</li> <li>• <code>PREC_LEFT</code> (1)</li> <li>• <code>PREC_RIGHT</code> (2)</li> <li>• <code>PREC_BOTH</code> (3)</li> </ul> <p>Any other integer input will result in the default (no preconditioning).</p> <p><code>maxl</code> (<code>int</code>) the number of Krylov basis vectors to use. values <math>\leq 0</math> will result in the default value (5).</p>
Return value	This returns a <code>SUNLinearSolver</code> object. If either <code>y</code> is incompatible then this routine will return <code>NULL</code> .
Notes	<p>This routine will perform consistency checks to ensure that it is called with a consistent <code>NVECTOR</code> implementation (i.e. that it supplies the requisite vector operations). If <code>y</code> is incompatible, then this routine will return <code>NULL</code>.</p> <p>We note that some SUNDIALS solvers are designed to only work with left preconditioning (IDA and IDAS). While it is possible to use a right-preconditioned <code>SUNLINSOL_SPFGMR</code> object for these packages, this use mode is not supported and may result in inferior performance.</p>

#### SUNLinSol\_SPFGMRSetPrecType

Call	<code>retval = SUNLinSol_SPFGMRSetPrecType(LS, pretype);</code>
Description	The function <code>SUNLinSol_SPFGMRSetPrecType</code> updates the type of preconditioning to use in the <code>SUNLINSOL_SPFGMR</code> object.
Arguments	<p><code>LS</code> (<code>SUNLinearSolver</code>) the <code>SUNLINSOL_SPFGMR</code> object to update</p> <p><code>pretype</code> (<code>int</code>) flag indicating the desired type of preconditioning, allowed values match those discussed in <code>SUNLinSol_SPFGMR</code>.</p>
Return value	This routine will return with one of the error codes <code>SUNLS_ILL_INPUT</code> (illegal <code>pretype</code> ), <code>SUNLS_MEM_NULL</code> ( <code>S</code> is <code>NULL</code> ) or <code>SUNLS_SUCCESS</code> .
Notes	

#### SUNLinSol\_SPFGMRSetGSType

Call	<code>retval = SUNLinSol_SPFGMRSetGSType(LS, gstype);</code>
Description	The function <code>SUNLinSol_SPFGMRSetPrecType</code> sets the type of Gram-Schmidt orthogonalization to use in the <code>SUNLINSOL_SPFGMR</code> object.
Arguments	<p><code>LS</code> (<code>SUNLinearSolver</code>) the <code>SUNLINSOL_SPFGMR</code> object to update</p> <p><code>gstype</code> (<code>int</code>) flag indicating the desired orthogonalization algorithm; allowed values are:</p>

- MODIFIED\_GS (1)
- CLASSICAL\_GS (2)

Any other integer input will result in a failure, returning error code `SUNLS_ILL_INPUT`.

Return value This routine will return with one of the error codes `SUNLS_ILL_INPUT` (illegal `pretype`), `SUNLS_MEM_NULL` (`S` is `NULL`) or `SUNLS_SUCCESS`.

Notes

#### `SUNLinSol_SPFGMRSetMaxRestarts`

Call `retval = SUNLinSol_SPFGMRSetMaxRestarts(LS, maxrs);`

Description The function `SUNLinSol_SPFGMRSetMaxRestarts` sets the number of GMRES restarts to allow in the `SUNLINSOL_SPFGMR` object.

Arguments `LS` (`SUNLinearSolver`) the `SUNLINSOL_SPFGMR` object to update  
`maxrs` (`int`) integer indicating number of restarts to allow. A negative input will result in the default of 0.

Return value This routine will return with one of the error codes `SUNLS_MEM_NULL` (`S` is `NULL`) or `SUNLS_SUCCESS`.

Notes

For backwards compatibility, we also provide the wrapper functions, each with identical input and output arguments to the routines that they wrap:

- `SUNSPFGMR`  
 Wrapper function for `SUNLinSol_SPFGMR`
- `SUNSPFGMRSetPrecType`  
 Wrapper function for `SUNLinSol_SPFGMRSetPrecType`
- `SUNSPFGMRSetGSType`  
 Wrapper function for `SUNLinSol_SPFGMRSetGSType`
- `SUNSPFGMRSetMaxRestarts`  
 Wrapper function for `SUNLinSol_SPFGMRSetMaxRestarts`

For solvers that include a Fortran interface module, the `SUNLINSOL_SPFGMR` module also includes a Fortran-callable function for creating a `SUNLinearSolver` object.

#### `FSUNSPFGMRINIT`

Call `FSUNSPFGMRINIT(code, pretype, maxl, ier)`

Description The function `FSUNSPFGMRINIT` can be called for Fortran programs to create a `SUNLINSOL_SPFGMR` object.

Arguments `code` (`int*`) is an integer input specifying the solver id (1 for `CVODE`, 2 for `IDA`, 3 for `KINSOL`, and 4 for `ARKODE`).  
`pretype` (`int*`) flag indicating desired preconditioning type  
`maxl` (`int*`) flag indicating Krylov subspace size

Return value `ier` is a return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes This routine must be called *after* the `NVECTOR` object has been initialized.

Allowable values for `pretype` and `maxl` are the same as for the C function `SUNLinSol_SPFGMR`.

Additionally, when using `ARKODE` with a non-identity mass matrix, the `SUNLINSOL_SPFGMR` module includes a Fortran-callable function for creating a `SUNLinearSolver` mass matrix solver object.

**FSUNMASSSPFGMRINIT**

Call FSUNMASSSPFGMRINIT(*pretype*, *maxl*, *ier*)

Description The function FSUNMASSSPFGMRINIT can be called for Fortran programs to create a SUNLINSOL\_SPFGMR object for mass matrix linear systems.

Arguments *pretype* (*int\**) flag indicating desired preconditioning type  
*maxl* (*int\**) flag indicating Krylov subspace size

Return value *ier* is a *int* return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes This routine must be called *after* the NVECTOR object has been initialized.

Allowable values for *pretype* and *maxl* are the same as for the C function SUNLinSol\_SPFGMR.

The SUNLinSol\_SPFGMRSetPrecType, SUNLinSol\_SPFGMRSetGStype and SUNLinSol\_SPFGMRSetMaxRestarts routines also support Fortran interfaces for the system and mass matrix solvers

**FSUNSPFGMRSETGSTYPE**

Call FSUNSPFGMRSETGSTYPE(*code*, *gstype*, *ier*)

Description The function FSUNSPFGMRSETGSTYPE can be called for Fortran programs to change the Gram-Schmidt orthogonalization algorithm.

Arguments *code* (*int\**) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).  
*gstype* (*int\**) flag indicating the desired orthogonalization algorithm.

Return value *ier* is a *int* return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes See SUNLinSol\_SPFGMRSetGStype for complete further documentation of this routine.

**FSUNMASSSPFGMRSETGSTYPE**

Call FSUNMASSSPFGMRSETGSTYPE(*gstype*, *ier*)

Description The function FSUNMASSSPFGMRSETGSTYPE can be called for Fortran programs to change the Gram-Schmidt orthogonalization algorithm for mass matrix linear systems.

Arguments The arguments are identical to FSUNSPFGMRSETGSTYPE above, except that *code* is not needed since mass matrix linear systems only arise in ARKODE.

Return value *ier* is a *int* return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes See SUNLinSol\_SPFGMRSetGStype for complete further documentation of this routine.

**FSUNSPFGMRSETPRECTYPE**

Call FSUNSPFGMRSETPRECTYPE(*code*, *pretype*, *ier*)

Description The function FSUNSPFGMRSETPRECTYPE can be called for Fortran programs to change the type of preconditioning to use.

Arguments *code* (*int\**) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).  
*pretype* (*int\**) flag indication the type of preconditioning to use.

Return value *ier* is a *int* return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes See SUNLinSol\_SPFGMRSetPrecType for complete further documentation of this routine.

**FSUNMASSSPFGMRSETPRECTYPE**

Call	FSUNMASSSPFGMRSETPRECTYPE( <i>pretype</i> , <i>ier</i> )
Description	The function FSUNMASSSPFGMRSETPRECTYPE can be called for Fortran programs to change the type of preconditioning for mass matrix linear systems.
Arguments	The arguments are identical to FSUNSPFGMRSETPRECTYPE above, except that <i>code</i> is not needed since mass matrix linear systems only arise in ARKODE.
Return value	<i>ier</i> is a <i>int</i> return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes	See SUNLinSol_SPFGMRSetPrecType for complete further documentation of this routine.

**FSUNSPFGMRSETMAXRS**

Call	FSUNSPFGMRSETMAXRS( <i>code</i> , <i>maxrs</i> , <i>ier</i> )
Description	The function FSUNSPFGMRSETMAXRS can be called for Fortran programs to change the maximum number of restarts allowed for SPFGMR.
Arguments	<i>code</i> ( <i>int*</i> ) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE). <i>maxrs</i> ( <i>int*</i> ) maximum allowed number of restarts.
Return value	<i>ier</i> is a <i>int</i> return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes	See SUNLinSol_SPFGMRSetMaxRestarts for complete further documentation of this routine.

**FSUNMASSSPFGMRSETMAXRS**

Call	FSUNMASSSPFGMRSETMAXRS( <i>maxrs</i> , <i>ier</i> )
Description	The function FSUNMASSSPFGMRSETMAXRS can be called for Fortran programs to change the maximum number of restarts allowed for SPFGMR for mass matrix linear systems.
Arguments	The arguments are identical to FSUNSPFGMRSETMAXRS above, except that <i>code</i> is not needed since mass matrix linear systems only arise in ARKODE.
Return value	<i>ier</i> is a <i>int</i> return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes	See SUNLinSol_SPFGMRSetMaxRestarts for complete further documentation of this routine.

**8.10.2 SUNLINSOL\_SPFGMR description**

The SUNLINSOL\_SPFGMR module defines the *content* field of a SUNLinearSolver to be the following structure:

```
struct _SUNLinearSolverContent_SPFGMR {
    int maxl;
    int pretype;
    int gstype;
    int max_restarts;
    int numiters;
    realtype resnorm;
    long int last_flag;
    ATimesFn ATimes;
    void* ATData;
    PSetupFn Psetup;
```

```

PSolveFn Psolve;
void* PData;
N_Vector s1;
N_Vector s2;
N_Vector *V;
N_Vector *Z;
realtype **Hes;
realtype *givens;
N_Vector xcor;
realtype *yg;
N_Vector vtemp;
};

```

These entries of the *content* field contain the following information:

**maxl** - number of FGMRES basis vectors to use (default is 5),

**pretype** - flag for use of preconditioning (default is none),

**gstyle** - flag for type of Gram-Schmidt orthogonalization (default is modified Gram-Schmidt),

**max\_restarts** - number of FGMRES restarts to allow (default is 0),

**numiters** - number of iterations from the most-recent solve,

**resnorm** - final linear residual norm from the most-recent solve,

**last\_flag** - last error return flag from an internal function,

**ATimes** - function pointer to perform  $Av$  product,

**ATData** - pointer to structure for **ATimes**,

**Psetup** - function pointer to preconditioner setup routine,

**Psolve** - function pointer to preconditioner solve routine,

**PData** - pointer to structure for **Psetup** and **Psolve**,

**s1, s2** - vector pointers for supplied scaling matrices (default is NULL),

**V** - the array of Krylov basis vectors  $v_1, \dots, v_{\text{maxl}+1}$ , stored in  $V[0], \dots, V[\text{maxl}]$ . Each  $v_i$  is a vector of type NVECTOR.,

**Z** - the array of preconditioned Krylov basis vectors  $z_1, \dots, z_{\text{maxl}+1}$ , stored in  $Z[0], \dots, Z[\text{maxl}]$ . Each  $z_i$  is a vector of type NVECTOR.,

**Hes** - the  $(\text{maxl} + 1) \times \text{maxl}$  Hessenberg matrix. It is stored row-wise so that the (i,j)th element is given by  $\text{Hes}[i][j]$ .,

**givens** - a length  $2*\text{maxl}$  array which represents the Givens rotation matrices that arise in the FGM-

RES algorithm. These matrices are  $F_0, F_1, \dots, F_j$ , where  $F_i =$

$$\begin{bmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & & c_i & -s_i & \\ & & & s_i & c_i & \\ & & & & & 1 \\ & & & & & & \ddots \\ & & & & & & & 1 \end{bmatrix},$$

are represented in the **givens** vector as  $\text{givens}[0] = c_0, \text{givens}[1] = s_0, \text{givens}[2] = c_1, \text{givens}[3] = s_1, \dots, \text{givens}[2j] = c_j, \text{givens}[2j+1] = s_j$ .,

**xcor** - a vector which holds the scaled, preconditioned correction to the initial guess,

**yg** - a length (maxl+1) array of **realtype** values used to hold “short” vectors (e.g.  $y$  and  $g$ ),

**vtemp** - temporary vector storage.

This solver is constructed to perform the following operations:

- During construction, the **xcor** and **vtemp** arrays are cloned from a template **NVECTOR** that is input, and default solver parameters are set.
- User-facing “set” routines may be called to modify default solver parameters.
- Additional “set” routines are called by the SUNDIALS solver that interfaces with **SUNLINSOL\_SPFGMR** to supply the **ATimes**, **PSetup**, and **Psolve** function pointers and **s1** and **s2** scaling vectors.
- In the “initialize” call, the remaining solver data is allocated (**V**, **Hes**, **givens**, and **yg** )
- In the “setup” call, any non-NULL **PSetup** function is called. Typically, this is provided by the SUNDIALS solver itself, that translates between the generic **PSetup** function and the solver-specific routine (solver-supplied or user-supplied).
- In the “solve” call, the FGMRES iteration is performed. This will include scaling, preconditioning, and restarts if those options have been supplied.

The **SUNLINSOL\_SPFGMR** module defines implementations of all “iterative” linear solver operations listed in Sections 8.0.1-8.0.3:

- **SUNLinSolGetType\_SPFGMR**
- **SUNLinSolInitialize\_SPFGMR**
- **SUNLinSolSetATimes\_SPFGMR**
- **SUNLinSolSetPreconditioner\_SPFGMR**
- **SUNLinSolSetScalingVectors\_SPFGMR**
- **SUNLinSolSetup\_SPFGMR**
- **SUNLinSolSolve\_SPFGMR**
- **SUNLinSolNumIters\_SPFGMR**
- **SUNLinSolResNorm\_SPFGMR**
- **SUNLinSolResid\_SPFGMR**
- **SUNLinSolLastFlag\_SPFGMR**
- **SUNLinSolSpace\_SPFGMR**
- **SUNLinSolFree\_SPFGMR**

## 8.11 The SUNLinearSolver\_SPBCGS implementation

The SPBCGS (Scaled, Preconditioned, Bi-Conjugate Gradient, Stabilized [34]) implementation of the **SUNLINSOL** module provided with SUNDIALS, **SUNLINSOL\_SPBCGS**, is an iterative linear solver that is designed to be compatible with any **NVECTOR** implementation (serial, threaded, parallel, and user-supplied) that supports a minimal subset of operations (**N\_VClone**, **N\_VDotProd**, **N\_VScale**, **N\_VLinearSum**, **N\_VProd**, **N\_VDiv**, and **N\_VDestroy**). Unlike the SPGMR and SPFGMR algorithms, SPBCGS requires a fixed amount of memory that does not increase with the number of allowed iterations.

### 8.11.1 SUNLINSOL\_SPBCGS usage

The header file to include when using this module is `sunlinsol/sunlinsol.spbcgs.h`. The SUNLINSOL\_SPBCGS module is accessible from all SUNDIALS solvers *without* linking to the `libsundials_sunlinsolspbcgs` module library.

The module SUNLINSOL\_SPBCGS provides the following user-callable routines:

#### SUNLinSol\_SPBCGS

**Call** `LS = SUNLinSol_SPBCGS(y, pretype, maxl);`

**Description** The function `SUNLinSol_SPBCGS` creates and allocates memory for a SPBCGS `SUNLinearSolver`.

**Arguments** `y` (`N_Vector`) a template for cloning vectors needed within the solver  
**pretype** (`int`) flag indicating the desired type of preconditioning, allowed values are:

- `PREC_NONE` (0)
- `PREC_LEFT` (1)
- `PREC_RIGHT` (2)
- `PREC_BOTH` (3)

Any other integer input will result in the default (no preconditioning).

**maxl** (`int`) the number of linear iterations to allow; values  $\leq 0$  will result in the default value (5).

**Return value** This returns a `SUNLinearSolver` object. If either `y` is incompatible then this routine will return `NULL`.

**Notes** This routine will perform consistency checks to ensure that it is called with a consistent `NVECTOR` implementation (i.e. that it supplies the requisite vector operations). If `y` is incompatible, then this routine will return `NULL`.

We note that some SUNDIALS solvers are designed to only work with left preconditioning (IDA and IDAS) and others with only right preconditioning (KINSOL). While it is possible to configure a SUNLINSOL\_SPBCGS object to use any of the preconditioning options with these solvers, this use mode is not supported and may result in inferior performance.

#### SUNLinSol\_SPBCGSSetPrecType

**Call** `retval = SUNLinSol_SPBCGSSetPrecType(LS, pretype);`

**Description** The function `SUNLinSol_SPBCGSSetPrecType` updates the type of preconditioning to use in the SUNLINSOL\_SPBCGS object.

**Arguments** `LS` (`SUNLinearSolver`) the SUNLINSOL\_SPBCGS object to update  
**pretype** (`int`) flag indicating the desired type of preconditioning, allowed values match those discussed in `SUNLinSol_SPBCGS`.

**Return value** This routine will return with one of the error codes `SUNLS_ILL_INPUT` (illegal `pretype`), `SUNLS_MEM_NULL` (S is `NULL`) or `SUNLS_SUCCESS`.

**Notes**

#### SUNLinSol\_SPBCGSsetMaxl

**Call** `retval = SUNLinSol_SPBCGSsetMaxl(LS, maxl);`

**Description** The function `SUNLinSol_SPBCGSsetMaxl` updates the number of linear solver iterations to allow.

**Arguments** `LS` (`SUNLinearSolver`) the SUNLINSOL\_SPBCGS object to update  
**maxl** (`int`) flag indicating the number of iterations to allow; values  $\leq 0$  will result in the default value (5)



Return value This routine will return with one of the error codes `SUNLS_MEM_NULL` (`S` is `NULL`) or `SUNLS_SUCCESS`.

Notes

For backwards compatibility, we also provide the wrapper functions, each with identical input and output arguments to the routines that they wrap:

- `SUNSPBCGS`  
Wrapper function for `SUNLinSol_SPBCGS`
- `SUNSPBCGSSetPrecType`  
Wrapper function for `SUNLinSol_SPBCGSSetPrecType`
- `SUNSPBCGSsetMaxl`  
Wrapper function for `SUNLinSol_SPBCGSsetMaxl`

For solvers that include a Fortran interface module, the `SUNLINSOL_SPBCGS` module also includes a Fortran-callable function for creating a `SUNLinearSolver` object.

#### FSUNSPBCGSINIT

Call `FSUNSPBCGSINIT(code, pretype, maxl, ier)`

Description The function `FSUNSPBCGSINIT` can be called for Fortran programs to create a `SUNLINSOL_SPBCGS` object.

Arguments `code` (`int*`) is an integer input specifying the solver id (1 for `CVODE`, 2 for `IDA`, 3 for `KINSOL`, and 4 for `ARKODE`).  
`pretype` (`int*`) flag indicating desired preconditioning type  
`maxl` (`int*`) flag indicating number of iterations to allow

Return value `ier` is a return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes This routine must be called *after* the `NVECTOR` object has been initialized.  
Allowable values for `pretype` and `maxl` are the same as for the C function `SUNLinSol_SPBCGS`.

Additionally, when using `ARKODE` with a non-identity mass matrix, the `SUNLINSOL_SPBCGS` module includes a Fortran-callable function for creating a `SUNLinearSolver` mass matrix solver object.

#### FSUNMASSSPBCGSINIT

Call `FSUNMASSSPBCGSINIT(pretype, maxl, ier)`

Description The function `FSUNMASSSPBCGSINIT` can be called for Fortran programs to create a `SUNLINSOL_SPBCGS` object for mass matrix linear systems.

Arguments `pretype` (`int*`) flag indicating desired preconditioning type  
`maxl` (`int*`) flag indicating number of iterations to allow

Return value `ier` is a `int` return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes This routine must be called *after* the `NVECTOR` object has been initialized.  
Allowable values for `pretype` and `maxl` are the same as for the C function `SUNLinSol_SPBCGS`.

The `SUNLinSol_SPBCGSSetPrecType` and `SUNLinSol_SPBCGSsetMaxl` routines also support Fortran interfaces for the system and mass matrix solvers.

**FSUNSPBCGSSETPRECTYPE**

Call	FSUNSPBCGSSETPRECTYPE( <i>code</i> , <i>pretype</i> , <i>ier</i> )
Description	The function FSUNSPBCGSSETPRECTYPE can be called for Fortran programs to change the type of preconditioning to use.
Arguments	<i>code</i> ( <b>int*</b> ) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE). <i>pretype</i> ( <b>int*</b> ) flag indication the type of preconditioning to use.
Return value	<i>ier</i> is a <b>int</b> return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes	See SUNLinSol_SPBCGSSetPrecType for complete further documentation of this routine.

**FSUNMASSSPBCGSSETPRECTYPE**

Call	FSUNMASSSPBCGSSETPRECTYPE( <i>pretype</i> , <i>ier</i> )
Description	The function FSUNMASSSPBCGSSETPRECTYPE can be called for Fortran programs to change the type of preconditioning for mass matrix linear systems.
Arguments	The arguments are identical to FSUNSPBCGSSETPRECTYPE above, except that <i>code</i> is not needed since mass matrix linear systems only arise in ARKODE.
Return value	<i>ier</i> is a <b>int</b> return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes	See SUNLinSol_SPBCGSSetPrecType for complete further documentation of this routine.

**FSUNSPBCGSSETMAXL**

Call	FSUNSPBCGSSETMAXL( <i>code</i> , <i>maxl</i> , <i>ier</i> )
Description	The function FSUNSPBCGSSETMAXL can be called for Fortran programs to change the maximum number of iterations to allow.
Arguments	<i>code</i> ( <b>int*</b> ) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE). <i>maxl</i> ( <b>int*</b> ) the number of iterations to allow
Return value	<i>ier</i> is a <b>int</b> return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes	See SUNLinSol_SPBCGSsetMaxl for complete further documentation of this routine.

**FSUNMASSSPBCGSSETMAXL**

Call	FSUNMASSSPBCGSSETMAXL( <i>maxl</i> , <i>ier</i> )
Description	The function FSUNMASSSPBCGSSETMAXL can be called for Fortran programs to change the type of preconditioning for mass matrix linear systems.
Arguments	The arguments are identical to FSUNSPBCGSSETMAXL above, except that <i>code</i> is not needed since mass matrix linear systems only arise in ARKODE.
Return value	<i>ier</i> is a <b>int</b> return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes	See SUNLinSol_SPBCGSsetMaxl for complete further documentation of this routine.

### 8.11.2 SUNLINSOL\_SPBCGS description

The SUNLINSOL\_SPBCGS module defines the *content* field of a `SUNLinearSolver` to be the following structure:

```
struct _SUNLinearSolverContent_SPBCGS {
    int maxl;
    int pretype;
    int numiters;
    realtype resnorm;
    long int last_flag;
    ATimesFn ATimes;
    void* ATData;
    PSetupFn Psetup;
    PSolveFn Psolve;
    void* PData;
    N_Vector s1;
    N_Vector s2;
    N_Vector r;
    N_Vector r_star;
    N_Vector p;
    N_Vector q;
    N_Vector u;
    N_Vector Ap;
    N_Vector vtemp;
};
```

These entries of the *content* field contain the following information:

- maxl** - number of SPBCGS iterations to allow (default is 5),
- pretype** - flag for type of preconditioning to employ (default is none),
- numiters** - number of iterations from the most-recent solve,
- resnorm** - final linear residual norm from the most-recent solve,
- last\_flag** - last error return flag from an internal function,
- ATimes** - function pointer to perform  $Av$  product,
- ATData** - pointer to structure for **ATimes**,
- Psetup** - function pointer to preconditioner setup routine,
- Psolve** - function pointer to preconditioner solve routine,
- PData** - pointer to structure for **Psetup** and **Psolve**,
- s1, s2** - vector pointers for supplied scaling matrices (default is NULL),
- r** - a NVECTOR which holds the current scaled, preconditioned linear system residual,
- r\_star** - a NVECTOR which holds the initial scaled, preconditioned linear system residual,
- p, q, u, Ap, vtemp** - NVECTORS used for workspace by the SPBCGS algorithm.

This solver is constructed to perform the following operations:

- During construction all NVECTOR solver data is allocated, with vectors cloned from a template NVECTOR that is input, and default solver parameters are set.

- User-facing “set” routines may be called to modify default solver parameters.
- Additional “set” routines are called by the SUNDIALS solver that interfaces with SUNLINSOL\_SPBCGS to supply the `ATimes`, `PSetup`, and `Psolve` function pointers and `s1` and `s2` scaling vectors.
- In the “initialize” call, the solver parameters are checked for validity.
- In the “setup” call, any non-NULL `PSetup` function is called. Typically, this is provided by the SUNDIALS solver itself, that translates between the generic `PSetup` function and the solver-specific routine (solver-supplied or user-supplied).
- In the “solve” call the SPBCGS iteration is performed. This will include scaling and preconditioning if those options have been supplied.

The SUNLINSOL\_SPBCGS module defines implementations of all “iterative” linear solver operations listed in Sections 8.0.1-8.0.3:

- `SUNLinSolGetType_SPBCGS`
- `SUNLinSolInitialize_SPBCGS`
- `SUNLinSolSetATimes_SPBCGS`
- `SUNLinSolSetPreconditioner_SPBCGS`
- `SUNLinSolSetScalingVectors_SPBCGS`
- `SUNLinSolSetup_SPBCGS`
- `SUNLinSolSolve_SPBCGS`
- `SUNLinSolNumIters_SPBCGS`
- `SUNLinSolResNorm_SPBCGS`
- `SUNLinSolResid_SPBCGS`
- `SUNLinSolLastFlag_SPBCGS`
- `SUNLinSolSpace_SPBCGS`
- `SUNLinSolFree_SPBCGS`

## 8.12 The SUNLinearSolver\_SPTFQMR implementation

The SPTFQMR (Scaled, Preconditioned, Transpose-Free Quasi-Minimum Residual [20]) implementation of the SUNLINSOL module provided with SUNDIALS, `SUNLINSOL_SPTFQMR`, is an iterative linear solver that is designed to be compatible with any `NVECTOR` implementation (serial, threaded, parallel, and user-supplied) that supports a minimal subset of operations (`N_VClone`, `N_VDotProd`, `N_VScale`, `N_VLinearSum`, `N_VProd`, `N_VConst`, `N_VDiv`, and `N_VDestroy`). Unlike the SPGMR and SPFGMR algorithms, SPTFQMR requires a fixed amount of memory that does not increase with the number of allowed iterations.

### 8.12.1 SUNLINSOL\_SPTFQMR usage

The header file to include when using this module is `sunlinsol/sunlinsol_sptfqmr.h`. The `SUNLINSOL_SPTFQMR` module is accessible from all SUNDIALS solvers *without* linking to the `libsundials_sunlinsolsptfqmr` module library.

The module `SUNLINSOL_SPTFQMR` provides the following user-callable routines:

**SUNLinSol\_SPTFQMR**

Call	<code>LS = SUNLinSol_SPTFQMR(y, pretype, maxl);</code>
Description	The function <code>SUNLinSol_SPTFQMR</code> creates and allocates memory for a SPTFQMR <code>SUNLinearSolver</code> .
Arguments	<p><code>y</code> (<code>N_Vector</code>) a template for cloning vectors needed within the solver</p> <p><code>pretype</code> (<code>int</code>) flag indicating the desired type of preconditioning, allowed values are:</p> <ul style="list-style-type: none"> <li>• <code>PREC_NONE</code> (0)</li> <li>• <code>PREC_LEFT</code> (1)</li> <li>• <code>PREC_RIGHT</code> (2)</li> <li>• <code>PREC_BOTH</code> (3)</li> </ul> <p>Any other integer input will result in the default (no preconditioning).</p> <p><code>maxl</code> (<code>int</code>) the number of linear iterations to allow; values <math>\leq 0</math> will result in the default value (5).</p>
Return value	This returns a <code>SUNLinearSolver</code> object. If either <code>y</code> is incompatible then this routine will return <code>NULL</code> .
Notes	<p>This routine will perform consistency checks to ensure that it is called with a consistent <code>NVECTOR</code> implementation (i.e. that it supplies the requisite vector operations). If <code>y</code> is incompatible, then this routine will return <code>NULL</code>.</p> <p>We note that some SUNDIALS solvers are designed to only work with left preconditioning (<code>IDA</code> and <code>IDAS</code>) and others with only right preconditioning (<code>KINSOL</code>). While it is possible to configure a <code>SUNLINSOL_SPTFQMR</code> object to use any of the preconditioning options with these solvers, this use mode is not supported and may result in inferior performance.</p>

**SUNLinSol\_SPTFQMRSetPrecType**

Call	<code>retval = SUNLinSol_SPTFQMRSetPrecType(LS, pretype);</code>
Description	The function <code>SUNLinSol_SPTFQMRSetPrecType</code> updates the type of preconditioning to use in the <code>SUNLINSOL_SPTFQMR</code> object.
Arguments	<p><code>LS</code> (<code>SUNLinearSolver</code>) the <code>SUNLINSOL_SPTFQMR</code> object to update</p> <p><code>pretype</code> (<code>int</code>) flag indicating the desired type of preconditioning, allowed values match those discussed in <code>SUNLinSol_SPTFQMR</code>.</p>
Return value	This routine will return with one of the error codes <code>SUNLS_ILL_INPUT</code> (illegal <code>pretype</code> ), <code>SUNLS_MEM_NULL</code> ( <code>S</code> is <code>NULL</code> ) or <code>SUNLS_SUCCESS</code> .
Notes	

**SUNLinSol\_SPTFQMRSetMaxl**

Call	<code>retval = SUNLinSol_SPTFQMRSetMaxl(LS, maxl);</code>
Description	The function <code>SUNLinSol_SPTFQMRSetMaxl</code> updates the number of linear solver iterations to allow.
Arguments	<p><code>LS</code> (<code>SUNLinearSolver</code>) the <code>SUNLINSOL_SPTFQMR</code> object to update</p> <p><code>maxl</code> (<code>int</code>) flag indicating the number of iterations to allow; values <math>\leq 0</math> will result in the default value (5)</p>
Return value	This routine will return with one of the error codes <code>SUNLS_MEM_NULL</code> ( <code>S</code> is <code>NULL</code> ) or <code>SUNLS_SUCCESS</code> .
Notes	

For backwards compatibility, we also provide the wrapper functions, each with identical input and output arguments to the routines that they wrap:

- **SUNSPTFQMR**  
Wrapper function for `SUNLinSol_SPTFQMR`
- **SUNSPTFQMRSetPrecType**  
Wrapper function for `SUNLinSol_SPTFQMRSetPrecType`
- **SUNSPTFQMRSetMaxl**  
Wrapper function for `SUNLinSol_SPTFQMRSetMaxl`

For solvers that include a Fortran interface module, the `SUNLINSOL_SPTFQMR` module also includes a Fortran-callable function for creating a `SUNLinearSolver` object.

#### FSUNSPTFQMRINIT

Call `FSUNSPTFQMRINIT(code, pretype, maxl, ier)`

Description The function `FSUNSPTFQMRINIT` can be called for Fortran programs to create a `SUNLINSOL_SPTFQMR` object.

Arguments `code` (`int*`) is an integer input specifying the solver id (1 for `CVODE`, 2 for `IDA`, 3 for `KINSOL`, and 4 for `ARKODE`).  
`pretype` (`int*`) flag indicating desired preconditioning type  
`maxl` (`int*`) flag indicating number of iterations to allow

Return value `ier` is a return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes This routine must be called *after* the `NVECTOR` object has been initialized.

Allowable values for `pretype` and `maxl` are the same as for the C function `SUNLinSol_SPTFQMR`.

Additionally, when using `ARKODE` with a non-identity mass matrix, the `SUNLINSOL_SPTFQMR` module includes a Fortran-callable function for creating a `SUNLinearSolver` mass matrix solver object.

#### FSUNMASSSPTFQMRINIT

Call `FSUNMASSSPTFQMRINIT(pretype, maxl, ier)`

Description The function `FSUNMASSSPTFQMRINIT` can be called for Fortran programs to create a `SUNLINSOL_SPTFQMR` object for mass matrix linear systems.

Arguments `pretype` (`int*`) flag indicating desired preconditioning type  
`maxl` (`int*`) flag indicating number of iterations to allow

Return value `ier` is a `int` return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes This routine must be called *after* the `NVECTOR` object has been initialized.

Allowable values for `pretype` and `maxl` are the same as for the C function `SUNLinSol_SPTFQMR`.

The `SUNLinSol_SPTFQMRSetPrecType` and `SUNLinSol_SPTFQMRSetMaxl` routines also support Fortran interfaces for the system and mass matrix solvers.

#### FSUNSPTFQMRSETPRECTYPE

Call `FSUNSPTFQMRSETPRECTYPE(code, pretype, ier)`

Description The function `FSUNSPTFQMRSETPRECTYPE` can be called for Fortran programs to change the type of preconditioning to use.

Arguments `code` (`int*`) is an integer input specifying the solver id (1 for `CVODE`, 2 for `IDA`, 3 for `KINSOL`, and 4 for `ARKODE`).  
`pretype` (`int*`) flag indication the type of preconditioning to use.

Return value `ier` is a `int` return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes See `SUNLinSol_SPTFQMRSetPrecType` for complete further documentation of this routine.

#### FSUNMASSSPTFQMRSETPRECTYPE

Call `FSUNMASSSPTFQMRSETPRECTYPE(prectype, ier)`

Description The function `FSUNMASSSPTFQMRSETPRECTYPE` can be called for Fortran programs to change the type of preconditioning for mass matrix linear systems.

Arguments The arguments are identical to `FSUNSPTFQMRSETPRECTYPE` above, except that `code` is not needed since mass matrix linear systems only arise in ARKODE.

Return value `ier` is a `int` return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes See `SUNLinSol_SPTFQMRSetPrecType` for complete further documentation of this routine.

#### FSUNSPTFQMRSETMAXL

Call `FSUNSPTFQMRSETMAXL(code, maxl, ier)`

Description The function `FSUNSPTFQMRSETMAXL` can be called for Fortran programs to change the maximum number of iterations to allow.

Arguments `code` (`int*`) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).  
`maxl` (`int*`) the number of iterations to allow

Return value `ier` is a `int` return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes See `SUNLinSol_SPTFQMRSetMaxl` for complete further documentation of this routine.

#### FSUNMASSSPTFQMRSETMAXL

Call `FSUNMASSSPTFQMRSETMAXL(maxl, ier)`

Description The function `FSUNMASSSPTFQMRSETMAXL` can be called for Fortran programs to change the type of preconditioning for mass matrix linear systems.

Arguments The arguments are identical to `FSUNSPTFQMRSETMAXL` above, except that `code` is not needed since mass matrix linear systems only arise in ARKODE.

Return value `ier` is a `int` return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes See `SUNLinSol_SPTFQMRSetMaxl` for complete further documentation of this routine.

### 8.12.2 SUNLINSOL\_SPTFQMR description

The `SUNLINSOL_SPTFQMR` module defines the *content* field of a `SUNLinearSolver` to be the following structure:

```
struct _SUNLinearSolverContent_SPTFQMR {
    int maxl;
    int pretype;
    int numiters;
    realtype resnorm;
    long int last_flag;
```

```

    ATimesFn ATimes;
    void* ATData;
    PSetupFn Psetup;
    PSolveFn Psolve;
    void* PData;
    N_Vector s1;
    N_Vector s2;
    N_Vector r_star;
    N_Vector q;
    N_Vector d;
    N_Vector v;
    N_Vector p;
    N_Vector *r;
    N_Vector u;
    N_Vector vtemp1;
    N_Vector vtemp2;
    N_Vector vtemp3;
};

```

These entries of the *content* field contain the following information:

**maxl** - number of TFQMR iterations to allow (default is 5),

**pretype** - flag for type of preconditioning to employ (default is none),

**numiters** - number of iterations from the most-recent solve,

**resnorm** - final linear residual norm from the most-recent solve,

**last\_flag** - last error return flag from an internal function,

**ATimes** - function pointer to perform  $Av$  product,

**ATData** - pointer to structure for **ATimes**,

**Psetup** - function pointer to preconditioner setup routine,

**Psolve** - function pointer to preconditioner solve routine,

**PData** - pointer to structure for **Psetup** and **Psolve**,

**s1**, **s2** - vector pointers for supplied scaling matrices (default is NULL),

**r\_star** - a NVECTOR which holds the initial scaled, preconditioned linear system residual,

**q**, **d**, **v**, **p**, **u** - NVECTORS used for workspace by the SPTFQMR algorithm,

**r** - array of two NVECTORS used for workspace within the SPTFQMR algorithm,

**vtemp1**, **vtemp2**, **vtemp3** - temporary vector storage.

This solver is constructed to perform the following operations:

- During construction all NVECTOR solver data is allocated, with vectors cloned from a template NVECTOR that is input, and default solver parameters are set.
- User-facing “set” routines may be called to modify default solver parameters.
- Additional “set” routines are called by the SUNDIALS solver that interfaces with SUNLINSOL\_SPTFQMR to supply the **ATimes**, **Psetup**, and **Psolve** function pointers and **s1** and **s2** scaling vectors.
- In the “initialize” call, the solver parameters are checked for validity.



- In the “setup” call, any non-NULL `PSetup` function is called. Typically, this is provided by the SUNDIALS solver itself, that translates between the generic `PSetup` function and the solver-specific routine (solver-supplied or user-supplied).
- In the “solve” call the TFQMR iteration is performed. This will include scaling and preconditioning if those options have been supplied.

The `SUNLINSOL_SPTFQMR` module defines implementations of all “iterative” linear solver operations listed in Sections 8.0.1-8.0.3:

- `SUNLinSolGetType_SPTFQMR`
- `SUNLinSolInitialize_SPTFQMR`
- `SUNLinSolSetATimes_SPTFQMR`
- `SUNLinSolSetPreconditioner_SPTFQMR`
- `SUNLinSolSetScalingVectors_SPTFQMR`
- `SUNLinSolSetup_SPTFQMR`
- `SUNLinSolSolve_SPTFQMR`
- `SUNLinSolNumIters_SPTFQMR`
- `SUNLinSolResNorm_SPTFQMR`
- `SUNLinSolResid_SPTFQMR`
- `SUNLinSolLastFlag_SPTFQMR`
- `SUNLinSolSpace_SPTFQMR`
- `SUNLinSolFree_SPTFQMR`

## 8.13 The SUNLinearSolver\_PCG implementation

The PCG (Preconditioned Conjugate Gradient [21]) implementation of the `SUNLINSOL` module provided with SUNDIALS, `SUNLINSOL_PCG`, is an iterative linear solver that is designed to be compatible with any `NVECTOR` implementation (serial, threaded, parallel, and user-supplied) that supports a minimal subset of operations (`N_VClone`, `N_VDotProd`, `N_VScale`, `N_VLinearSum`, `N_VProd`, and `N_VDestroy`). Unlike the SPGMR and SPFGMR algorithms, PCG requires a fixed amount of memory that does not increase with the number of allowed iterations.

Unlike all of the other iterative linear solvers supplied with SUNDIALS, PCG should only be used on *symmetric* linear systems (e.g. mass matrix linear systems encountered in ARKODE). As a result, the explanation of the role of scaling and preconditioning matrices given in general must be modified in this scenario. The PCG algorithm solves a linear system  $Ax = b$  where  $A$  is a symmetric ( $A^T = A$ ), real-valued matrix. Preconditioning is allowed, and is applied in a symmetric fashion on both the right and left. Scaling is also allowed and is applied symmetrically. We denote the preconditioner and scaling matrices as follows:

- $P$  is the preconditioner (assumed symmetric),
- $S$  is a diagonal matrix of scale factors.

The matrices  $A$  and  $P$  are not required explicitly; only routines that provide  $A$  and  $P^{-1}$  as operators are required. The diagonal of the matrix  $S$  is held in a single NVECTOR, supplied by the user.

In this notation, PCG applies the underlying CG algorithm to the equivalent transformed system

$$\tilde{A}\tilde{x} = \tilde{b} \quad (8.3)$$

where

$$\begin{aligned} \tilde{A} &= SP^{-1}AP^{-1}S, \\ \tilde{b} &= SP^{-1}b, \\ \tilde{x} &= S^{-1}Px. \end{aligned} \quad (8.4)$$

The scaling matrix must be chosen so that the vectors  $SP^{-1}b$  and  $S^{-1}Px$  have dimensionless components.

The stopping test for the PCG iterations is on the L2 norm of the scaled preconditioned residual:

$$\begin{aligned} &\|\tilde{b} - \tilde{A}\tilde{x}\|_2 < \delta \\ \Leftrightarrow & \\ &\|SP^{-1}b - SP^{-1}Ax\|_2 < \delta \\ \Leftrightarrow & \\ &\|P^{-1}b - P^{-1}Ax\|_S < \delta \end{aligned}$$

where  $\|v\|_S = \sqrt{v^T S^T S v}$ , with an input tolerance  $\delta$ .

### 8.13.1 SUNLINSOL\_PCG usage

The header file to include when using this module is `sunlinsol/sunlinsol_pcg.h`. The SUNLINSOL\_PCG module is accessible from all SUNDIALS solvers *without* linking to the `libsundials_sunlinsolpcg` module library.

The module SUNLINSOL\_PCG provides the following user-callable routines:

SUNLinSol_PCG	
Call	<code>LS = SUNLinSol_PCG(y, pretype, maxl);</code>
Description	The function <code>SUNLinSol_PCG</code> creates and allocates memory for a PCG <code>SUNLinearSolver</code> .
Arguments	<p><code>y</code> (N_Vector) a template for cloning vectors needed within the solver</p> <p><code>pretype</code> (int) flag indicating whether to use preconditioning. Since the PCG algorithm is designed to only support symmetric preconditioning, then any of the <code>pretype</code> inputs <code>PREC_LEFT</code> (1), <code>PREC_RIGHT</code> (2), or <code>PREC_BOTH</code> (3) will result in use of the symmetric preconditioner; any other integer input will result in the default (no preconditioning).</p> <p><code>maxl</code> (int) the number of linear iterations to allow; values <math>\leq 0</math> will result in the default value (5).</p>
Return value	This returns a <code>SUNLinearSolver</code> object. If either <code>y</code> is incompatible then this routine will return <code>NULL</code> .
Notes	<p>This routine will perform consistency checks to ensure that it is called with a consistent NVECTOR implementation (i.e. that it supplies the requisite vector operations). If <code>y</code> is incompatible, then this routine will return <code>NULL</code>.</p> <p>Although some SUNDIALS solvers are designed to only work with left preconditioning (IDA and IDAS) and others with only right preconditioning (KINSOL), PCG should <i>only</i> be used with these packages when the linear systems are known to be <i>symmetric</i>. Since the scaling of matrix rows and columns must be identical in a symmetric matrix, symmetric preconditioning should work appropriately even for packages designed with one-sided preconditioning in mind.</p>

**SUNLinSol\_PCGSetPrecType**

Call	<code>retval = SUNLinSol_PCGSetPrecType(LS, pretype);</code>
Description	The function <code>SUNLinSol_PCGSetPrecType</code> updates the flag indicating use of preconditioning in the <code>SUNLINSOL_PCG</code> object.
Arguments	<code>LS</code> ( <code>SUNLinearSolver</code> ) the <code>SUNLINSOL_PCG</code> object to update <code>pretype</code> ( <code>int</code> ) flag indicating use of preconditioning. allowable values match those discussed in <code>SUNLinSol_PCG</code> .
Return value	This routine will return with one of the error codes <code>SUNLS_ILL_INPUT</code> (illegal <code>pretype</code> ), <code>SUNLS_MEM_NULL</code> ( <code>S</code> is <code>NULL</code> ) or <code>SUNLS_SUCCESS</code> .
Notes	

**SUNLinSol\_PCGSetMaxl**

Call	<code>retval = SUNLinSol_PCGSetMaxl(LS, maxl);</code>
Description	The function <code>SUNLinSol_PCGSetMaxl</code> updates the number of linear solver iterations to allow.
Arguments	<code>LS</code> ( <code>SUNLinearSolver</code> ) the <code>SUNLINSOL_PCG</code> object to update <code>maxl</code> ( <code>int</code> ) flag indicating the number of iterations to allow; values $\leq 0$ will result in the default value (5)
Return value	This routine will return with one of the error codes <code>SUNLS_MEM_NULL</code> ( <code>S</code> is <code>NULL</code> ) or <code>SUNLS_SUCCESS</code> .
Notes	

For backwards compatibility, we also provide the wrapper functions, each with identical input and output arguments to the routines that they wrap:

- `SUNPCG`  
Wrapper function for `SUNLinSol_PCG`
- `SUNPCGSetPrecType`  
Wrapper function for `SUNLinSol_PCGSetPrecType`
- `SUNPCGSetMaxl`  
Wrapper function for `SUNLinSol_PCGSetMaxl`

For solvers that include a Fortran interface module, the `SUNLINSOL_PCG` module also includes a Fortran-callable function for creating a `SUNLinearSolver` object.

**FSUNPCGINIT**

Call	<code>FSUNPCGINIT(code, pretype, maxl, ier)</code>
Description	The function <code>FSUNPCGINIT</code> can be called for Fortran programs to create a <code>SUNLINSOL_PCG</code> object.
Arguments	<code>code</code> ( <code>int*</code> ) is an integer input specifying the solver id (1 for <code>CVODE</code> , 2 for <code>IDA</code> , 3 for <code>KINSOL</code> , and 4 for <code>ARKODE</code> ). <code>pretype</code> ( <code>int*</code> ) flag indicating desired preconditioning type <code>maxl</code> ( <code>int*</code> ) flag indicating number of iterations to allow
Return value	<code>ier</code> is a return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes	This routine must be called <i>after</i> the <code>NVECTOR</code> object has been initialized. Allowable values for <code>pretype</code> and <code>maxl</code> are the same as for the C function <code>SUNLinSol_PCG</code> .

Additionally, when using `ARKODE` with a non-identity mass matrix, the `SUNLINSOL_PCG` module includes a Fortran-callable function for creating a `SUNLinearSolver` mass matrix solver object.

**FSUNMASSPCGINIT**

Call FSUNMASSPCGINIT(*pretype*, *maxl*, *ier*)

Description The function FSUNMASSPCGINIT can be called for Fortran programs to create a SUNLIN-SOL\_PCG object for mass matrix linear systems.

Arguments *pretype* (*int\**) flag indicating desired preconditioning type  
*maxl* (*int\**) flag indicating number of iterations to allow

Return value *ier* is a *int* return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes This routine must be called *after* the NVECTOR object has been initialized.  
 Allowable values for *pretype* and *maxl* are the same as for the C function SUNLinSol\_PCG. The SUNLinSol\_PCGSetPrecType and SUNLinSol\_PCGSetMaxl routines also support Fortran interfaces for the system and mass matrix solvers.

**FSUNPCGSETPRECTYPE**

Call FSUNPCGSETPRECTYPE(*code*, *pretype*, *ier*)

Description The function FSUNPCGSETPRECTYPE can be called for Fortran programs to change the type of preconditioning to use.

Arguments *code* (*int\**) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).  
*pretype* (*int\**) flag indication the type of preconditioning to use.

Return value *ier* is a *int* return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes See SUNLinSol\_PCGSetPrecType for complete further documentation of this routine.

**FSUNMASSPCGSETPRECTYPE**

Call FSUNMASSPCGSETPRECTYPE(*pretype*, *ier*)

Description The function FSUNMASSPCGSETPRECTYPE can be called for Fortran programs to change the type of preconditioning for mass matrix linear systems.

Arguments The arguments are identical to FSUNPCGSETPRECTYPE above, except that *code* is not needed since mass matrix linear systems only arise in ARKODE.

Return value *ier* is a *int* return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes See SUNLinSol\_PCGSetPrecType for complete further documentation of this routine.

**FSUNPCGSETMAXL**

Call FSUNPCGSETMAXL(*code*, *maxl*, *ier*)

Description The function FSUNPCGSETMAXL can be called for Fortran programs to change the maximum number of iterations to allow.

Arguments *code* (*int\**) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).  
*maxl* (*int\**) the number of iterations to allow

Return value *ier* is a *int* return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes See SUNLinSol\_PCGSetMaxl for complete further documentation of this routine.

**FSUNMASSPCGSETMAXL**

Call	FSUNMASSPCGSETMAXL(maxl, ier)
Description	The function FSUNMASSPCGSETMAXL can be called for Fortran programs to change the type of preconditioning for mass matrix linear systems.
Arguments	The arguments are identical to FSUNPCGSETMAXL above, except that <b>code</b> is not needed since mass matrix linear systems only arise in ARKODE.
Return value	<b>ier</b> is a <b>int</b> return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes	See SUNLinSol_PCGSetMaxl for complete further documentation of this routine.

**8.13.2 SUNLINSOL\_PCG description**

The SUNLINSOL\_PCG module defines the *content* field of a **SUNLinearSolver** to be the following structure:

```
struct _SUNLinearSolverContent_PCG {
    int maxl;
    int pretype;
    int numiters;
    realtype resnorm;
    long int last_flag;
    ATimesFn ATimes;
    void* ATData;
    PSetupFn Psetup;
    PSolveFn Psolve;
    void* PData;
    N_Vector s;
    N_Vector r;
    N_Vector p;
    N_Vector z;
    N_Vector Ap;
};
```

These entries of the *content* field contain the following information:

**maxl** - number of PCG iterations to allow (default is 5),

**pretype** - flag for use of preconditioning (default is none),

**numiters** - number of iterations from the most-recent solve,

**resnorm** - final linear residual norm from the most-recent solve,

**last\_flag** - last error return flag from an internal function,

**ATimes** - function pointer to perform  $Av$  product,

**ATData** - pointer to structure for **ATimes**,

**Psetup** - function pointer to preconditioner setup routine,

**Psolve** - function pointer to preconditioner solve routine,

**PData** - pointer to structure for **Psetup** and **Psolve**,

**s** - vector pointer for supplied scaling matrix (default is **NULL**),

**r** - a **NVECTOR** which holds the preconditioned linear system residual,

**p, z, Ap** - NVECTORS used for workspace by the PCG algorithm.

This solver is constructed to perform the following operations:

- During construction all NVECTOR solver data is allocated, with vectors cloned from a template NVECTOR that is input, and default solver parameters are set.
- User-facing “set” routines may be called to modify default solver parameters.
- Additional “set” routines are called by the SUNDIALS solver that interfaces with SUNLINSOL\_PCG to supply the **ATimes**, **PSetup**, and **Psolve** function pointers and **s** scaling vector.
- In the “initialize” call, the solver parameters are checked for validity.
- In the “setup” call, any non-NULL **PSetup** function is called. Typically, this is provided by the SUNDIALS solver itself, that translates between the generic **PSetup** function and the solver-specific routine (solver-supplied or user-supplied).
- In the “solve” call the PCG iteration is performed. This will include scaling and preconditioning if those options have been supplied.

The SUNLINSOL\_PCG module defines implementations of all “iterative” linear solver operations listed in Sections [8.0.1-8.0.3](#):

- **SUNLinSolGetType\_PCG**
- **SUNLinSolInitialize\_PCG**
- **SUNLinSolSetATimes\_PCG**
- **SUNLinSolSetPreconditioner\_PCG**
- **SUNLinSolSetScalingVectors\_PCG** – since PCG only supports symmetric scaling, the second NVECTOR argument to this function is ignored
- **SUNLinSolSetup\_PCG**
- **SUNLinSolSolve\_PCG**
- **SUNLinSolNumIters\_PCG**
- **SUNLinSolResNorm\_PCG**
- **SUNLinSolResid\_PCG**
- **SUNLinSolLastFlag\_PCG**
- **SUNLinSolSpace\_PCG**
- **SUNLinSolFree\_PCG**

## 8.14 SUNLinearSolver Examples

There are **SUNLinearSolver** examples that may be installed for each implementation; these make use of the functions in **test\_sunlinsol.c**. These example functions show simple usage of the **SUNLinearSolver** family of functions. The inputs to the examples depend on the linear solver type, and are output to **stdout** if the example is run without the appropriate number of command-line arguments.

The following is a list of the example functions in **test\_sunlinsol.c**:

- **Test\_SUNLinSolGetType**: Verifies the returned solver type against the value that should be returned.

- `Test_SUNLinSolInitialize`: Verifies that `SUNLinSolInitialize` can be called and returns successfully.
- `Test_SUNLinSolSetup`: Verifies that `SUNLinSolSetup` can be called and returns successfully.
- `Test_SUNLinSolSolve`: Given a `SUNMATRIX` object  $A$ , `NVECTOR` objects  $x$  and  $b$  (where  $Ax = b$ ) and a desired solution tolerance `tol`, this routine clones  $x$  into a new vector  $y$ , calls `SUNLinSolSolve` to fill  $y$  as the solution to  $Ay = b$  (to the input tolerance), verifies that each entry in  $x$  and  $y$  match to within  $10 \cdot \text{tol}$ , and overwrites  $x$  with  $y$  prior to returning (in case the calling routine would like to investigate further).
- `Test_SUNLinSolSetATimes` (iterative solvers only): Verifies that `SUNLinSolSetATimes` can be called and returns successfully.
- `Test_SUNLinSolSetPreconditioner` (iterative solvers only): Verifies that `SUNLinSolSetPreconditioner` can be called and returns successfully.
- `Test_SUNLinSolSetScalingVectors` (iterative solvers only): Verifies that `SUNLinSolSetScalingVectors` can be called and returns successfully.
- `Test_SUNLinSolLastFlag`: Verifies that `SUNLinSolLastFlag` can be called, and outputs the result to `stdout`.
- `Test_SUNLinSolNumIters` (iterative solvers only): Verifies that `SUNLinSolNumIters` can be called, and outputs the result to `stdout`.
- `Test_SUNLinSolResNorm` (iterative solvers only): Verifies that `SUNLinSolResNorm` can be called, and that the result is non-negative.
- `Test_SUNLinSolResid` (iterative solvers only): Verifies that `SUNLinSolResid` can be called.
- `Test_SUNLinSolSpace` verifies that `SUNLinSolSpace` can be called, and outputs the results to `stdout`.

We'll note that these tests should be performed in a particular order. For either direct or iterative linear solvers, `Test_SUNLinSolInitialize` must be called before `Test_SUNLinSolSetup`, which must be called before `Test_SUNLinSolSolve`. Additionally, for iterative linear solvers `Test_SUNLinSolSetATimes`, `Test_SUNLinSolSetPreconditioner` and `Test_SUNLinSolSetScalingVectors` should be called before `Test_SUNLinSolInitialize`; similarly `Test_SUNLinSolNumIters`, `Test_SUNLinSolResNorm` and `Test_SUNLinSolResid` should be called after `Test_SUNLinSolSolve`. These are called in the appropriate order in all of the example problems.

## 8.15 SUNLinearSolver functions used by IDA

In Table 8.3 below, we list the linear solver functions in the `SUNLINSOL` module used within the `IDALS` interface in the `IDA` package. In general, `IDALS` considers two non-overlapping categories of linear solvers: *matrix-based* and *matrix-free*, determined based on whether the `SUNMATRIX` object `J` passed to `IDASetLinearSolver` was not `NULL`.

Additionally, `IDALS` will consider a linear solver of either type as *iterative* if it self-identifies as `SUNLINEARSOLVER_ITERATIVE` (via the `SUNLinSolGetType` routine). Since both matrix-based and matrix-free linear solvers may be iterative, we only list `SUNLINSOL` routines that are specifically called based on this type; these routines are *in addition to* those listed for the other two categories.

As with the `SUNMATRIX` module, we emphasize that the `IDA` user does not need to know detailed usage of linear solver functions by the `IDA` code modules in order to use `IDA`. The information is presented as an implementation detail for the interested reader.

The linear solver functions listed below are marked with ✓ to indicate that they are required, or with † to indicate that they are only called if they are non-`NULL` in the `SUNLINSOL` implementation that is being used.

Table 8.3: List of linear solver function usage by IDA code modules

	Matrix-Based	Matrix-Free	Iterative
SUNLinSolGetType	✓	✓	
SUNLinSolSetATimes		✓	
SUNLinSolSetPreconditioner			†
SUNLinSolSetScalingVectors	†	†	
SUNLinSolInitialize	✓	✓	
SUNLinSolSetup	✓	✓	
SUNLinSolSolve	✓	✓	
SUNLinSolNumIters			✓
SUNLinSolResid			✓
<sup>1</sup> SUNLinSolLastFlag			
SUNLinSolFree	✓	✓	
SUNLinSolSpace	†	†	

1. Although IDALS does not call SUNLinSolLastFlag directly, this routine is available for users to query linear solver issues directly.



## Chapter 9

# Description of the SUNNonlinearSolver module

SUNDIALS time integration packages are written in terms of generic nonlinear solver operations defined by the SUNNONLINSOL API and implemented by a particular SUNNONLINSOL module of type `SUNNonlinearSolver`. Users can supply their own SUNNONLINSOL module, or use one of the modules provided with SUNDIALS.

The time integrators in SUNDIALS specify a default nonlinear solver module and as such this chapter is intended for users that wish to use a non-default nonlinear solver module or would like to provide their own nonlinear solver implementation. Users interested in using a non-default solver module may skip the description of the SUNNONLINSOL API in section 9.1 and proceed to the subsequent sections in this chapter that describe the SUNNONLINSOL modules provided with SUNDIALS.

For users interested in providing their own SUNNONLINSOL module, the following section presents the SUNNONLINSOL API and its implementation beginning with the definition of SUNNONLINSOL functions in sections 9.1.1 – 9.1.3. This is followed by the definition of functions supplied to a nonlinear solver implementation in section 9.1.4. A table of nonlinear solver return codes is given in section 9.1.5. The `SUNNonlinearSolver` type and the generic SUNNONLINSOL module are defined in section 9.1.6. Section 9.1.7 describes how SUNNONLINSOL models interface with SUNDIALS integrators providing sensitivity analysis capabilities (CVODES and IDAS). Finally, section 9.1.8 lists the requirements for supplying a custom SUNNONLINSOL module. Users wishing to supply their own SUNNONLINSOL module are encouraged to use the SUNNONLINSOL implementations provided with SUNDIALS as a template for supplying custom nonlinear solver modules.

### 9.1 The SUNNonlinearSolver API

The SUNNONLINSOL API defines several nonlinear solver operations that enable SUNDIALS integrators to utilize any SUNNONLINSOL implementation that provides the required functions. These functions can be divided into three categories. The first are the core nonlinear solver functions. The second group of functions consists of set routines to supply the nonlinear solver with functions provided by the SUNDIALS time integrators and to modify solver parameters. The final group consists of get routines for retrieving nonlinear solver statistics. All of these functions are defined in the header file `sundials/sundials_nonlinearsolver.h`.

#### 9.1.1 SUNNonlinearSolver core functions

The core nonlinear solver functions consist of two required functions to get the nonlinear solver type (`SUNNonlinSolGetType`) and solve the nonlinear system (`SUNNonlinSolSolve`). The remaining three functions for nonlinear solver initialization (`SUNNonlinSolInitialization`), setup (`SUNNonlinSolSetup`), and destruction (`SUNNonlinSolFree`) are optional.

**SUNNonlinSolGetType**

Call            `type = SUNNonlinSolGetType(NLS);`

Description   The *required* function `SUNNonlinSolGetType` returns nonlinear solver type.

Arguments     `NLS` (`SUNNonlinearSolver`) a `SUNNONLINSOL` object.

Return value   The return value `type` (of type `int`) will be one of the following:

`SUNNONLINEARSOLVER_ROOTFIND`    0, the `SUNNONLINSOL` module solves  $F(y) = 0$ .

`SUNNONLINEARSOLVER_FIXEDPOINT` 1, the `SUNNONLINSOL` module solves  $G(y) = y$ .

**SUNNonlinSolInitialize**

Call            `retval = SUNNonlinSolInitialize(NLS);`

Description   The *optional* function `SUNNonlinSolInitialize` performs nonlinear solver initialization and may perform any necessary memory allocations.

Arguments     `NLS` (`SUNNonlinearSolver`) a `SUNNONLINSOL` object.

Return value   The return value `retval` (of type `int`) is zero for a successful call and a negative value for a failure.

Notes          It is assumed all solver-specific options have been set prior to calling `SUNNonlinSolInitialize`. `SUNNONLINSOL` implementations that do not require initialization may set this operation to `NULL`.

**SUNNonlinSolSetup**

Call            `retval = SUNNonlinSolSetup(NLS, y, mem);`

Description   The *optional* function `SUNNonlinSolSetup` performs any solver setup needed for a nonlinear solve.

Arguments     `NLS` (`SUNNonlinearSolver`) a `SUNNONLINSOL` object.

`y`    (`N.Vector`) the initial iteration passed to the nonlinear solver.

`mem` (`void *`) the SUNDIALS integrator memory structure.

Return value   The return value `retval` (of type `int`) is zero for a successful call and a negative value for a failure.

Notes          SUNDIALS integrators call `SUNNonlinSolSetup` before each step attempt. `SUNNONLINSOL` implementations that do not require setup may set this operation to `NULL`.

**SUNNonlinSolSolve**

Call            `retval = SUNNonlinSolSolve(NLS, y0, y, w, tol, callLSetup, mem);`

Description   The *required* function `SUNNonlinSolSolve` solves the nonlinear system  $F(y) = 0$  or  $G(y) = y$ .

Arguments     `NLS`            (`SUNNonlinearSolver`) a `SUNNONLINSOL` object.

`y0`            (`N.Vector`) the initial iterate for the nonlinear solve. This *must* remain unchanged throughout the solution process.

`y`             (`N.Vector`) the solution to the nonlinear system.

`w`             (`N.Vector`) the solution error weight vector used for computing weighted error norms.

`tol`           (`realtype`) the requested solution tolerance in the weighted root-mean-squared norm.

`callLSetup` (`booleantype`) a flag indicating that the integrator recommends for the linear solver setup function to be called.

**mem** (void \*) the SUNDIALS integrator memory structure.

**Return value** The return value **retval** (of type **int**) is zero for a successful solve, a positive value for a recoverable error, and a negative value for an unrecoverable error.

#### SUNNonlinSolFree

**Call** **retval** = SUNNonlinSolFree(NLS);

**Description** The *optional* function **SUNNonlinSolFree** frees any memory allocated by the nonlinear solver.

**Arguments** **NLS** (**SUNNonlinearSolver**) a SUNNONLINSOL object.

**Return value** The return value **retval** (of type **int**) should be zero for a successful call, and a negative value for a failure. SUNNONLINSOL implementations that do not allocate data may set this operation to **NULL**.

### 9.1.2 SUNNonlinearSolver set functions

The following set functions are used to supply nonlinear solver modules with functions defined by the SUNDIALS integrators and to modify solver parameters. Only the routine for setting the nonlinear system defining function (**SUNNonlinSolSetSysFn**) is required. All other set functions are optional.

#### SUNNonlinSolSetSysFn

**Call** **retval** = SUNNonlinSolSetSysFn(NLS, SysFn);

**Description** The *required* function **SUNNonlinSolSetSysFn** is used to provide the nonlinear solver with the function defining the nonlinear system. This is the function  $F(y)$  in  $F(y) = 0$  for **SUNNONLINEARSOLVER\_ROOTFIND** modules or  $G(y)$  in  $G(y) = y$  for **SUNNONLINEARSOLVER\_FIXEDPOINT** modules.

**Arguments** **NLS** (**SUNNonlinearSolver**) a SUNNONLINSOL object.

**SysFn** (**SUNNonlinSolSysFn**) the function defining the nonlinear system. See section 9.1.4 for the definition of **SUNNonlinSolSysFn**.

**Return value** The return value **retval** (of type **int**) should be zero for a successful call, and a negative value for a failure.

#### SUNNonlinSolSetLSetupFn

**Call** **retval** = SUNNonlinSolSetLSetupFn(NLS, LSetupFn);

**Description** The *optional* function **SUNNonlinSolLSetupFn** is called by SUNDIALS integrators to provide the nonlinear solver with access to its linear solver setup function.

**Arguments** **NLS** (**SUNNonlinearSolver**) a SUNNONLINSOL object.

**LSetupFn** (**SUNNonlinSolLSetupFn**) a wrapper function to the SUNDIALS integrator's linear solver setup function. See section 9.1.4 for the definition of **SUNNonlinLSetupFn**.

**Return value** The return value **retval** (of type **int**) should be zero for a successful call, and a negative value for a failure.

**Notes** The **SUNNonlinLSetupFn** function sets up the linear system  $Ax = b$  where  $A = \frac{\partial F}{\partial y}$  is the linearization of the nonlinear residual function  $F(y) = 0$  (when using **SUNLINSOL** direct linear solvers) or calls the user-defined preconditioner setup function (when using **SUNLINSOL** iterative linear solvers). SUNNONLINSOL implementations that do not require solving this system, do not utilize **SUNLINSOL** linear solvers, or use **SUNLINSOL** linear solvers that do not require setup may set this operation to **NULL**.

**SUNNonlinSolSetLSolveFn**

Call	<code>retval = SUNNonlinSolSetLSolveFn(NLS, LSolveFn);</code>
Description	The <i>optional</i> function <code>SUNNonlinSolSetLSolveFn</code> is called by SUNDIALS integrators to provide the nonlinear solver with access to its linear solver solve function.
Arguments	<code>NLS</code> ( <code>SUNNonlinearSolver</code> ) a <code>SUNNONLINSOL</code> object <code>LSolveFn</code> ( <code>SUNNonlinSolLSolveFn</code> ) a wrapper function to the SUNDIALS integrator's linear solver solve function. See section 9.1.4 for the definition of <code>SUNNonlinSolLSolveFn</code> .
Return value	The return value <code>retval</code> (of type <code>int</code> ) should be zero for a successful call, and a negative value for a failure.
Notes	The <code>SUNNonlinLSolveFn</code> function solves the linear system $Ax = b$ where $A = \frac{\partial F}{\partial y}$ is the linearization of the nonlinear residual function $F(y) = 0$ . <code>SUNNONLINSOL</code> implementations that do not require solving this system or do not use <code>SUNLINSOL</code> linear solvers may set this operation to <code>NULL</code> .

**SUNNonlinSolSetConvTestFn**

Call	<code>retval = SUNNonlinSolSetConvTestFn(NLS, CTestFn);</code>
Description	The <i>optional</i> function <code>SUNNonlinSolSetConvTestFn</code> is used to provide the nonlinear solver with a function for determining if the nonlinear solver iteration has converged. This is typically called by SUNDIALS integrators to define their nonlinear convergence criteria, but may be replaced by the user.
Arguments	<code>NLS</code> ( <code>SUNNonlinearSolver</code> ) a <code>SUNNONLINSOL</code> object. <code>CTestFn</code> ( <code>SUNNonlineSolConvTestFn</code> ) a SUNDIALS integrator's nonlinear solver convergence test function. See section 9.1.4 for the definition of <code>SUNNonlinSolConvTestFn</code> .
Return value	The return value <code>retval</code> (of type <code>int</code> ) should be zero for a successful call, and a negative value for a failure.
Notes	<code>SUNNONLINSOL</code> implementations utilizing their own convergence test criteria may set this function to <code>NULL</code> .

**SUNNonlinSolSetMaxIters**

Call	<code>retval = SUNNonlinSolSetMaxIters(NLS, maxiters);</code>
Description	The <i>optional</i> function <code>SUNNonlinSolSetMaxIters</code> sets the maximum number of nonlinear solver iterations. This is typically called by SUNDIALS integrators to define their default iteration limit, but may be adjusted by the user.
Arguments	<code>NLS</code> ( <code>SUNNonlinearSolver</code> ) a <code>SUNNONLINSOL</code> object. <code>maxiters</code> ( <code>int</code> ) the maximum number of nonlinear iterations.
Return value	The return value <code>retval</code> (of type <code>int</code> ) should be zero for a successful call, and a negative value for a failure (e.g., <code>maxiters &lt; 1</code> ).

### 9.1.3 SUNNonlinearSolver get functions

The following get functions allow SUNDIALS integrators to retrieve nonlinear solver statistics. The routine for getting the current total number of iterations (`SUNNonlinSolGetNumIters` is optional. The routine for getting the current nonlinear solver iteration (`SUNNonlinSolGetCurIter`) is required when using the convergence test provided by the SUNDIALS integrator or by the ARKODE and CVODE linear solver interfaces. Otherwise, `SUNNonlinSolGetCurIter` is optional.

**SUNNonlinSolGetNumIters**

Call	<code>retval = SUNNonlinSolGetNumIters(NLS, numiters);</code>
Description	The <i>optional</i> function <code>SUNNonlinSolGetNumIters</code> returns the total number of nonlinear solver iterations. This is typically called by the SUNDIALS integrator to store the nonlinear solver statistics, but may also be called by the user.
Arguments	<code>NLS</code> ( <code>SUNNonlinearSolver</code> ) a <code>SUNNONLINSOL</code> object <code>numiters</code> ( <code>long int*</code> ) the total number of nonlinear solver iterations.
Return value	The return value <code>retval</code> (of type <code>int</code> ) should be zero for a successful call, and a negative value for a failure.

**SUNNonlinSolGetCurIter**

Call	<code>retval = SUNNonlinSolGetCurIter(NLS, iter);</code>
Description	The function <code>SUNNonlinSolGetCurIter</code> returns the iteration index of the current nonlinear solve. This function is <i>required</i> when using SUNDIALS integrator-provided convergence tests or when using a <code>SUNLINSOL</code> spils linear solver; otherwise it is <i>optional</i> .
Arguments	<code>NLS</code> ( <code>SUNNonlinearSolver</code> ) a <code>SUNNONLINSOL</code> object <code>iter</code> ( <code>int*</code> ) the nonlinear solver iteration in the current solve starting from zero.
Return value	The return value <code>retval</code> (of type <code>int</code> ) should be zero for a successful call, and a negative value for a failure.

**9.1.4 Functions provided by SUNDIALS integrators**

To interface with `SUNNONLINSOL` modules, the SUNDIALS integrators supply a variety of routines for evaluating the nonlinear system, calling the `SUNLINSOL` setup and solve functions, and testing the nonlinear iteration for convergence. These integrator-provided routines translate between the user-supplied ODE or DAE systems and the generic interfaces to the nonlinear or linear systems of equations that result in their solution. The types for functions provided to a `SUNNONLINSOL` module are defined in the header file `sundials/sundials_nonlinearsolver.h`, and are described below.

**SUNNonlinSolSysFn**

Definition	<code>typedef int (*SUNNonlinSolSysFn)(N_Vector y, N_Vector F, void* mem);</code>
Purpose	These functions evaluate the nonlinear system $F(y)$ for <code>SUNNONLINEARSOLVER_ROOTFIND</code> type modules or $G(y)$ for <code>SUNNONLINEARSOLVER_FIXEDPOINT</code> type modules. Memory for <code>F</code> must be allocated prior to calling this function. The vector <code>y</code> <i>must</i> be left unchanged.
Arguments	<code>y</code> is the state vector at which the nonlinear system should be evaluated. <code>F</code> is the output vector containing $F(y)$ or $G(y)$ , depending on the solver type. <code>mem</code> is the SUNDIALS integrator memory structure.
Return value	The return value <code>retval</code> (of type <code>int</code> ) is zero for a successful solve, a positive value for a recoverable error, and a negative value for an unrecoverable error.

**SUNNonlinSolLSetupFn**

Definition	<code>typedef int (*SUNNonlinSolLSetupFn)(N_Vector y, N_Vector F,   booleantype jbad,   booleantype* jcur, void* mem);</code>
Purpose	These functions are wrappers to the SUNDIALS integrator's function for setting up linear solves with <code>SUNLINSOL</code> modules.

Arguments	<p><b>y</b> is the state vector at which the linear system should be setup.</p> <p><b>F</b> is the value of the nonlinear system function at <b>y</b>.</p> <p><b>jbad</b> is an input indicating whether the nonlinear solver believes that <math>A</math> has gone stale (SUNTRUE) or not (SUNFALSE).</p> <p><b>jcur</b> is an output indicating whether the routine has updated the Jacobian <math>A</math> (SUNTRUE) or not (SUNFALSE).</p> <p><b>mem</b> is the SUNDIALS integrator memory structure.</p>
Return value	The return value <b>retval</b> (of type <b>int</b> ) is zero for a successful solve, a positive value for a recoverable error, and a negative value for an unrecoverable error.
Notes	The <b>SUNNonlinLSetupFn</b> function sets up the linear system $Ax = b$ where $A = \frac{\partial F}{\partial y}$ is the linearization of the nonlinear residual function $F(y) = 0$ (when using SUNLINSOL direct linear solvers) or calls the user-defined preconditioner setup function (when using SUNLINSOL iterative linear solvers). SUNNONLINSOL implementations that do not require solving this system, do not utilize SUNLINSOL linear solvers, or use SUNLINSOL linear solvers that do not require setup may ignore these functions.

#### **SUNNonlinSolLSolveFn**

Definition	<code>typedef int (*SUNNonlinSolLSolveFn)(N_Vector y, N_Vector b, void* mem);</code>
Purpose	These functions are wrappers to the SUNDIALS integrator's function for solving linear systems with SUNLINSOL modules.
Arguments	<p><b>y</b> is the input vector containing the current nonlinear iteration.</p> <p><b>b</b> contains the right-hand side vector for the linear solve on input and the solution to the linear system on output.</p> <p><b>mem</b> is the SUNDIALS integrator memory structure.</p>
Return value	The return value <b>retval</b> (of type <b>int</b> ) is zero for a successful solve, a positive value for a recoverable error, and a negative value for an unrecoverable error.
Notes	The <b>SUNNonlinLSolveFn</b> function solves the linear system $Ax = b$ where $A = \frac{\partial F}{\partial y}$ is the linearization of the nonlinear residual function $F(y) = 0$ . SUNNONLINSOL implementations that do not require solving this system or do not use SUNLINSOL linear solvers may ignore these functions.

#### **SUNNonlinSolConvTestFn**

Definition	<code>typedef int (*SUNNonlinSolConvTestFn)(SUNNonlinearSolver NLS, N_Vector y, N_Vector del, realtype tol, N_Vector ewt, void* mem);</code>
Purpose	These functions are SUNDIALS integrator-specific convergence tests for nonlinear solvers and are typically supplied by each SUNDIALS integrator, but users may supply custom problem-specific versions as desired.
Arguments	<p><b>NLS</b> is the SUNNONLINSOL object.</p> <p><b>y</b> is the current nonlinear iterate.</p> <p><b>del</b> is the difference between the current and prior nonlinear iterates.</p> <p><b>tol</b> is the nonlinear solver tolerance.</p> <p><b>ewt</b> is the weight vector used in computing weighted norms.</p> <p><b>mem</b> is the SUNDIALS integrator memory structure.</p>
Return value	The return value of this routine will be a negative value if an unrecoverable error occurred or one of the following:
	<b>SUN-NLS-SUCCESS</b> the iteration is converged.

	SUN-NLS_CONTINUE	the iteration has not converged, keep iterating.
	SUN-NLS_CONV_RECVR	the iteration appears to be diverging, try to recover.
Notes	The tolerance passed to this routine by SUNDIALS integrators is the tolerance in a weighted root-mean-squared norm with error weight vector <b>ewt</b> . SUNNONLINSOL modules utilizing their own convergence criteria may ignore these functions.	

### 9.1.5 SUNNonlinearSolver return codes

The functions provided to SUNNONLINSOL modules by each SUNDIALS integrator, and functions within the SUNDIALS-provided SUNNONLINSOL implementations utilize a common set of return codes, shown below in Table 9.1. Here, negative values correspond to non-recoverable failures, positive values to recoverable failures, and zero to a successful call.

Table 9.1: Description of the SUNNonlinearSolver return codes

Name	Value	Description
SUN-NLS_SUCCESS	0	successful call or converged solve
SUN-NLS_CONTINUE	1	the nonlinear solver is not converged, keep iterating
SUN-NLS_CONV_RECVR	2	the nonlinear solver appears to be diverging, try to recover
SUN-NLS_MEM_NULL	-1	a memory argument is NULL
SUN-NLS_MEM_FAIL	-2	a memory access or allocation failed
SUN-NLS_ILL_INPUT	-3	an illegal input option was provided

### 9.1.6 The generic SUNNonlinearSolver module

SUNDIALS integrators interact with specific SUNNONLINSOL implementations through the generic SUNNONLINSOL module on which all other SUNNONLINSOL implementations are built. The `SUNNonlinearSolver` type is a pointer to a structure containing an implementation-dependent *content* field and an *ops* field. The type `SUNNonlinearSolver` is defined as follows:

```
typedef struct _generic_SUNNonlinearSolver *SUNNonlinearSolver;
```

```
struct _generic_SUNNonlinearSolver {
    void *content;
    struct _generic_SUNNonlinearSolver_Ops *ops;
};
```

where the `_generic_SUNNonlinearSolver_Ops` structure is a list of pointers to the various actual nonlinear solver operations provided by a specific implementation. The `_generic_SUNNonlinearSolver_Ops` structure is defined as

```
struct _generic_SUNNonlinearSolver_Ops {
    SUNNonlinearSolver_Type (*gettype)(SUNNonlinearSolver);
    int (*initialize)(SUNNonlinearSolver);
    int (*setup)(SUNNonlinearSolver, N_Vector, void*);
    int (*solve)(SUNNonlinearSolver, N_Vector, N_Vector,
                 N_Vector, realtype, booleantype, void*);
    int (*free)(SUNNonlinearSolver);
    int (*setsysfn)(SUNNonlinearSolver, SUNNonlinSolSysFn);
    int (*setlssetupfn)(SUNNonlinearSolver, SUNNonlinSolLSSetupFn);
    int (*setlsolvefn)(SUNNonlinearSolver, SUNNonlinSolLSolveFn);
    int (*setctestfn)(SUNNonlinearSolver, SUNNonlinSolConvTestFn);
    int (*setmaxiters)(SUNNonlinearSolver, int);
```

```

    int                (*getnumiters)(SUNNonlinearSolver, long int*);
    int                (*getcuriter)(SUNNonlinearSolver, int*);
};

```

The generic SUNNONLINSOL module defines and implements the nonlinear solver operations defined in Sections 9.1.1 – 9.1.3. These routines are in fact only wrappers to the nonlinear solver operations provided by a particular SUNNONLINSOL implementation, which are accessed through the ops field of the SUNNonlinearSolver structure. To illustrate this point we show below the implementation of a typical nonlinear solver operation from the generic SUNNONLINSOL module, namely SUNNonlinSolSolve, which solves the nonlinear system and returns a flag denoting a successful or failed solve:

```

int SUNNonlinSolSolve(SUNNonlinearSolver NLS,
                     N_Vector y0, N_Vector y,
                     N_Vector w, realtype tol,
                     booleantype callLSetup, void* mem)
{
    return((int) NLS->ops->solve(NLS, y0, y, w, tol, callLSetup, mem));
}

```

### 9.1.7 Usage with sensitivity enabled integrators

When used with SUNDIALS packages that support sensitivity analysis capabilities (e.g., CVODES and IDAS) a special NVECTOR module is used to interface with SUNNONLINSOL modules for solves involving sensitivity vectors stored in an NVECTOR array. As described below, the NVECTOR\_SENSWRAPPER module is an NVECTOR implementation where the vector content is an NVECTOR array. This wrapper vector allows SUNNONLINSOL modules to operate on data stored as a collection of vectors.

For all SUNDIALS-provided SUNNONLINSOL modules a special constructor wrapper is provided so users do not need to interact directly with the NVECTOR\_SENSWRAPPER module. These constructors follow the naming convention SUNNonlinSol\*\*\*Sens(count,...) where \*\*\* is the name of the SUNNONLINSOL module, count is the size of the vector wrapper, and ... are the module-specific constructor arguments.

#### The NVECTOR\_SENSWRAPPER module

This section describes the NVECTOR\_SENSWRAPPER implementation of an NVECTOR. To access the NVECTOR\_SENSWRAPPER module, include the header file `sundials/sundials_nvector_senswrapper.h`.

The NVECTOR\_SENSWRAPPER module defines an N\_Vector implementing all of the standard vectors operations defined in Table 6.2 but with some changes to how operations are computed in order to accommodate operating on a collection of vectors.

1. Element-wise vector operations are computed on a vector-by-vector basis. For example, the linear sum of two wrappers containing  $n_v$  vectors of length  $n$ , `N_VLinearSum(a,x,b,y,z)`, is computed as

$$z_{j,i} = ax_{j,i} + by_{j,i}, \quad i = 0, \dots, n-1, \quad j = 0, \dots, n_v-1.$$

2. The dot product of two wrappers containing  $n_v$  vectors of length  $n$  is computed as if it were the dot product of two vectors of length  $nn_v$ . Thus `d = N_VDotProd(x,y)` is

$$d = \sum_{j=0}^{n_v-1} \sum_{i=0}^{n-1} x_{j,i} y_{j,i}.$$

3. All norms are computed as the maximum of the individual norms of the  $n_v$  vectors in the wrapper. For example, the weighted root mean square norm `m = N_VWrmsNorm(x, w)` is

$$m = \max_j \sqrt{\left( \frac{1}{n} \sum_{i=0}^{n-1} (x_{j,i} w_{j,i})^2 \right)}$$



To enable usage alongside other NVECTOR modules the NVECTOR\_SENSWRAPPER functions implementing vector operations have `_SensWrapper` appended to the generic vector operation name.

The NVECTOR\_SENSWRAPPER module provides the following constructors for creating an NVECTOR\_SENSWRAPPER:

#### `N_VNewEmpty_SensWrapper`

Call	<code>w = N_VNewEmpty_SensWrapper(count);</code>
Description	The function <code>N_VNewEmpty_SensWrapper</code> creates an empty NVECTOR_SENSWRAPPER wrapper with space for <code>count</code> vectors.
Arguments	<code>count</code> ( <code>int</code> ) the number of vectors the wrapper will contain.
Return value	The return value <code>w</code> (of type <code>N_Vector</code> ) will be a NVECTOR object if the constructor exits successfully, otherwise <code>w</code> will be <code>NULL</code> .

#### `N_VNew_SensWrapper`

Call	<code>w = N_VNew_SensWrapper(count, y);</code>
Description	The function <code>N_VNew_SensWrapper</code> creates an NVECTOR_SENSWRAPPER wrapper containing <code>count</code> vectors cloned from <code>y</code> .
Arguments	<code>count</code> ( <code>int</code> ) the number of vectors the wrapper will contain. <code>y</code> ( <code>N_Vector</code> ) the template vectors to use in creating the vector wrapper.
Return value	The return value <code>w</code> (of type <code>N_Vector</code> ) will be a NVECTOR object if the constructor exits successfully, otherwise <code>w</code> will be <code>NULL</code> .

The NVECTOR\_SENSWRAPPER implementation of the NVECTOR module defines the *content* field of the `N_Vector` to be a structure containing an `N_Vector` array, the number of vectors in the vector array, and a boolean flag indicating ownership of the vectors in the vector array.

```
struct _N_VectorContent_SensWrapper {
    N_Vector* vecs;
    int nvecs;
    boolean_t own_vecs;
};
```

The following macros are provided to access the content of an NVECTOR\_SENSWRAPPER vector.

- `NV_CONTENT_SW(v)` - provides access to the content structure
- `NV_VECS_SW(v)` - provides access to the vector array
- `NV_NVECS_SW(v)` - provides access to the number of vectors
- `NV_OWN_VECS_SW(v)` - provides access to the ownership flag
- `NV_VEC_SW(v,i)` - provides access to the *i*-th vector in the vector array

### 9.1.8 Implementing a Custom SUNNonlinearSolver Module

A SUNNONLINSOL implementation *must* do the following:

1. Specify the content of the SUNNONLINSOL module.
2. Define and implement the required nonlinear solver operations defined in Sections 9.1.1 – 9.1.3. Note that the names of the module routines should be unique to that implementation in order to permit using more than one SUNNONLINSOL module (each with different SUNNonlinearSolver internal data representations) in the same code.

3. Define and implement a user-callable constructor to create a `SUNNonlinearSolver` object.

Additionally, a `SUNNonlinearSolver` implementation *may* do the following:

1. Define and implement additional user-callable “set” routines acting on the `SUNNonlinearSolver` object, e.g., for setting various configuration options to tune the performance of the nonlinear solve algorithm.
2. Provide additional user-callable “get” routines acting on the `SUNNonlinearSolver` object, e.g., for returning various solve statistics.

## 9.2 The `SUNNonlinearSolver_Newton` implementation

This section describes the `SUNNONLINSOL` implementation of Newton’s method. To access the `SUNNONLINSOL_NEWTON` module, include the header file `sunnonlinsol/sunnonlinsol_newton.h`. We note that the `SUNNONLINSOL_NEWTON` module is accessible from `SUNDIALS` integrators *without* separately linking to the `libsundials_sunnonlinsolnewton` module library.

### 9.2.1 `SUNNonlinearSolver_Newton` description

To find the solution to

$$F(y) = 0 \tag{9.1}$$

given an initial guess  $y^{(0)}$ , Newton’s method computes a series of approximate solutions

$$y^{(m+1)} = y^{(m)} + \delta^{(m+1)} \tag{9.2}$$

where  $m$  is the Newton iteration index, and the Newton update  $\delta^{(m+1)}$  is the solution of the linear system

$$A(y^{(m)})\delta^{(m+1)} = -F(y^{(m)}), \tag{9.3}$$

in which  $A$  is the Jacobian matrix

$$A \equiv \partial F / \partial y. \tag{9.4}$$

Depending on the linear solver used, the `SUNNONLINSOL_NEWTON` module will employ either a Modified Newton method, or an Inexact Newton method [5, 9, 16, 18, 28]. When used with a direct linear solver, the Jacobian matrix  $A$  is held constant during the Newton iteration, resulting in a Modified Newton method. With a matrix-free iterative linear solver, the iteration is an Inexact Newton method.

In both cases, calls to the integrator-supplied `SUNNonlinSolSetupFn` function are made infrequently to amortize the increased cost of matrix operations (updating  $A$  and its factorization within direct linear solvers, or updating the preconditioner within iterative linear solvers). Specifically, `SUNNONLINSOL_NEWTON` will call the `SUNNonlinSolSetupFn` function in two instances:

- (a) when requested by the integrator (the input `callSetSetup` is `SUNTRUE`) before attempting the Newton iteration, or
- (b) when reattempting the nonlinear solve after a recoverable failure occurs in the Newton iteration with stale Jacobian information (`jcur` is `SUNFALSE`). In this case, `SUNNONLINSOL_NEWTON` will set `jbad` to `SUNTRUE` before calling the `SUNNonlinSolSetupFn` function.

Whether the Jacobian matrix  $A$  is fully or partially updated depends on logic unique to each integrator-supplied `SUNNonlinSolSetupFn` routine. We refer to the discussion of nonlinear solver strategies provided in Chapter 2 for details on this decision.

The default maximum number of iterations and the stopping criteria for the Newton iteration are supplied by the `SUNDIALS` integrator when `SUNNONLINSOL_NEWTON` is attached to it. Both the maximum number of iterations and the convergence test function may be modified by the user by calling the `SUNNonlinSolSetMaxIters` and/or `SUNNonlinSolSetConvTestFn` functions after attaching the `SUNNONLINSOL_NEWTON` object to the integrator.

### 9.2.2 SUNNonlinearSolver\_Newton functions

The `SUNNONLINSOL_NEWTON` module provides the following constructors for creating a `SUNNonlinearSolver` object.

#### `SUNNonlinSol_Newton`

Call	<code>NLS = SUNNonlinSol_Newton(y);</code>
Description	The function <code>SUNNonlinSol_Newton</code> creates a <code>SUNNonlinearSolver</code> object for use with SUNDIALS integrators to solve nonlinear systems of the form $F(y) = 0$ using Newton's method.
Arguments	<code>y</code> ( <code>N_Vector</code> ) a template for cloning vectors needed within the solver.
Return value	The return value <code>NLS</code> (of type <code>SUNNonlinearSolver</code> ) will be a <code>SUNNONLINSOL</code> object if the constructor exits successfully, otherwise <code>NLS</code> will be <code>NULL</code> .

#### `SUNNonlinSol_NewtonSens`

Call	<code>NLS = SUNNonlinSol_NewtonSens(count, y);</code>
Description	The function <code>SUNNonlinSol_NewtonSens</code> creates a <code>SUNNonlinearSolver</code> object for use with SUNDIALS sensitivity enabled integrators (CVODES and IDAS) to solve nonlinear systems of the form $F(y) = 0$ using Newton's method.
Arguments	<p><code>count</code> (<code>int</code>) the number of vectors in the nonlinear solve. When integrating a system containing <code>Ns</code> sensitivities the value of <code>count</code> is:</p> <ul style="list-style-type: none"> <li>• <code>Ns+1</code> if using a <i>simultaneous</i> corrector approach.</li> <li>• <code>Ns</code> if using a <i>staggered</i> corrector approach.</li> </ul> <p><code>y</code> (<code>N_Vector</code>) a template for cloning vectors needed within the solver.</p>
Return value	The return value <code>NLS</code> (of type <code>SUNNonlinearSolver</code> ) will be a <code>SUNNONLINSOL</code> object if the constructor exits successfully, otherwise <code>NLS</code> will be <code>NULL</code> .

The `SUNNONLINSOL_NEWTON` module implements all of the functions defined in sections 9.1.1 – 9.1.3 except for the `SUNNonlinSolSetup` function. The `SUNNONLINSOL_NEWTON` functions have the same names as those defined by the generic `SUNNONLINSOL` API with `_Newton` appended to the function name. Unless using the `SUNNONLINSOL_NEWTON` module as a standalone nonlinear solver the generic functions defined in sections 9.1.1 – 9.1.3 should be called in favor of the `SUNNONLINSOL_NEWTON`-specific implementations.

The `SUNNONLINSOL_NEWTON` module also defines the following additional user-callable function.

#### `SUNNonlinSolGetSysFn_Newton`

Call	<code>retval = SUNNonlinSolGetSysFn_Newton(NLS, SysFn);</code>
Description	The function <code>SUNNonlinSolGetSysFn_Newton</code> returns the residual function that defines the nonlinear system.
Arguments	<p><code>NLS</code> (<code>SUNNonlinearSolver</code>) a <code>SUNNONLINSOL</code> object</p> <p><code>SysFn</code> (<code>SUNNonlinSolSysFn*</code>) the function defining the nonlinear system.</p>
Return value	The return value <code>retval</code> (of type <code>int</code> ) should be zero for a successful call, and a negative value for a failure.
Notes	This function is intended for users that wish to evaluate the nonlinear residual in a custom convergence test function for the <code>SUNNONLINSOL_NEWTON</code> module. We note that <code>SUNNONLINSOL_NEWTON</code> will not leverage the results from any user calls to <code>SysFn</code> .

### 9.2.3 SUNNonlinearSolver\_Newton content

The *content* field of the SUNNONLINSOL\_NEWTON module is the following structure.

```
struct _SUNNonlinearSolverContent_Newton {

    SUNNonlinSolSysFn    Sys;
    SUNNonlinSolLSetupFn LSetup;
    SUNNonlinSolLSolveFn LSolve;
    SUNNonlinSolConvTestFn CTest;

    N_Vector    delta;
    booleantype jcur;
    int         curiter;
    int         maxiters;
    long int    niters;
};
```

These entries of the *content* field contain the following information:

**Sys** - the function for evaluating the nonlinear system,  
**LSetup** - the package-supplied function for setting up the linear solver,  
**LSolve** - the package-supplied function for performing a linear solve,  
**CTest** - the function for checking convergence of the Newton iteration,  
**delta** - the Newton iteration update vector,  
**jcur** - the Jacobian status (SUNTRUE = current, SUNFALSE = stale),  
**curiter** - the current number of iterations in the solve attempt,  
**maxiters** - the maximum number of Newton iterations allowed in a solve, and  
**niters** - the total number of nonlinear iterations across all solves.

### 9.2.4 SUNNonlinearSolver\_Newton Fortran interface

For SUNDIALS integrators that include a Fortran interface, the SUNNONLINSOL\_NEWTON module also includes a Fortran-callable function for creating a **SUNNonlinearSolver** object.

#### FSUNNEWTONINIT

Call **FSUNNEWTONINIT**(code, ier);

Description The function **FSUNNEWTONINIT** can be called for Fortran programs to create a **SUNNonlinearSolver** object for use with SUNDIALS integrators to solve nonlinear systems of the form  $F(y) = 0$  with Newton's method.

Arguments **code** (**int\***) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).

Return value **ier** is a return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

## 9.3 The SUNNonlinearSolver\_FixedPoint implementation

This section describes the SUNNONLINSOL implementation of a fixed point (functional) iteration with optional Anderson acceleration. To access the SUNNONLINSOL\_FIXEDPOINT module, include the header file `sunnonlinsol/sunnonlinsol.fixedpoint.h`. We note that the SUNNONLINSOL\_FIXEDPOINT module is accessible from SUNDIALS integrators *without* separately linking to the `libsundials_sunnonlinsolfixedpoint` module library.

### 9.3.1 SUNNonlinearSolver\_FixedPoint description

To find the solution to

$$G(y) = y \quad (9.5)$$

given an initial guess  $y^{(0)}$ , the fixed point iteration computes a series of approximate solutions

$$y^{(n+1)} = G(y^{(n)}) \quad (9.6)$$

where  $n$  is the iteration index. The convergence of this iteration may be accelerated using Anderson's method [3, 35, 19, 30]. With Anderson acceleration using subspace size  $m$ , the series of approximate solutions can be formulated as the linear combination

$$y^{(n+1)} = \sum_{i=0}^{m_n} \alpha_i^{(n)} G(y^{(n-m_n+i)}) \quad (9.7)$$

where  $m_n = \min\{m, n\}$  and the factors

$$\alpha^{(n)} = (\alpha_0^{(n)}, \dots, \alpha_{m_n}^{(n)}) \quad (9.8)$$

solve the minimization problem  $\min_{\alpha} \|F_n \alpha^T\|_2$  under the constraint that  $\sum_{i=0}^{m_n} \alpha_i = 1$  where

$$F_n = (f_{n-m_n}, \dots, f_n) \quad (9.9)$$

with  $f_i = G(y^{(i)}) - y^{(i)}$ . Due to this constraint, in the limit of  $m = 0$  the accelerated fixed point iteration formula (9.7) simplifies to the standard fixed point iteration (9.6).

Following the recommendations made in [35], the SUNNONLINSOL\_FIXEDPOINT implementation computes the series of approximate solutions as

$$y^{(n+1)} = G(y^{(n)}) - \sum_{i=0}^{m_n-1} \gamma_i^{(n)} \Delta g_{n-m_n+i} \quad (9.10)$$

with  $\Delta g_i = G(y^{(i+1)}) - G(y^{(i)})$  and where the factors

$$\gamma^{(n)} = (\gamma_0^{(n)}, \dots, \gamma_{m_n-1}^{(n)}) \quad (9.11)$$

solve the unconstrained minimization problem  $\min_{\gamma} \|f_n - \Delta F_n \gamma^T\|_2$  where

$$\Delta F_n = (\Delta f_{n-m_n}, \dots, \Delta f_{n-1}), \quad (9.12)$$

with  $\Delta f_i = f_{i+1} - f_i$ . The least-squares problem is solved by applying a QR factorization to  $\Delta F_n = Q_n R_n$  and solving  $R_n \gamma = Q_n^T f_n$ .

The acceleration subspace size  $m$  is required when constructing the SUNNONLINSOL\_FIXEDPOINT object. The default maximum number of iterations and the stopping criteria for the fixed point iteration are supplied by the SUNDIALS integrator when SUNNONLINSOL\_FIXEDPOINT is attached to it. Both the maximum number of iterations and the convergence test function may be modified by the user by calling `SUNNonlinSolSetMaxIters` and `SUNNonlinSolSetConvTestFn` functions after attaching the SUNNONLINSOL\_FIXEDPOINT object to the integrator.

### 9.3.2 SUNNonlinearSolver\_FixedPoint functions

The SUNNONLINSOL\_FIXEDPOINT module provides the following constructors for creating a `SUNNonlinearSolver` object.

**SUNNonlinSol\_FixedPoint**

Call	<code>NLS = SUNNonlinSol_FixedPoint(y, m);</code>
Description	The function <code>SUNNonlinSol_FixedPoint</code> creates a <code>SUNNonlinearSolver</code> object for use with SUNDIALS integrators to solve nonlinear systems of the form $G(y) = y$ .
Arguments	<code>y</code> ( <code>N_Vector</code> ) a template for cloning vectors needed within the solver <code>m</code> ( <code>int</code> ) the number of acceleration vectors to use
Return value	The return value <code>NLS</code> (of type <code>SUNNonlinearSolver</code> ) will be a <code>SUNNONLINSOL</code> object if the constructor exits successfully, otherwise <code>NLS</code> will be <code>NULL</code> .

**SUNNonlinSol\_FixedPointSens**

Call	<code>NLS = SUNNonlinSol_FixedPointSens(count, y, m);</code>
Description	The function <code>SUNNonlinSol_FixedPointSens</code> creates a <code>SUNNonlinearSolver</code> object for use with SUNDIALS sensitivity enabled integrators (CVODES and IDAS) to solve nonlinear systems of the form $G(y) = y$ .
Arguments	<code>count</code> ( <code>int</code> ) the number of vectors in the nonlinear solve. When integrating a system containing <code>Ns</code> sensitivities the value of <code>count</code> is: <ul style="list-style-type: none"> <li>• <code>Ns+1</code> if using a <i>simultaneous</i> corrector approach.</li> <li>• <code>Ns</code> if using a <i>staggered</i> corrector approach.</li> </ul> <code>y</code> ( <code>N_Vector</code> ) a template for cloning vectors needed within the solver. <code>m</code> ( <code>int</code> ) the number of acceleration vectors to use.
Return value	The return value <code>NLS</code> (of type <code>SUNNonlinearSolver</code> ) will be a <code>SUNNONLINSOL</code> object if the constructor exits successfully, otherwise <code>NLS</code> will be <code>NULL</code> .

Since the accelerated fixed point iteration (9.6) does not require the setup or solution of any linear systems, the `SUNNONLINSOL_FIXEDPOINT` module implements all of the functions defined in sections 9.1.1 – 9.1.3 except for the `SUNNonlinSolSetup`, `SUNNonlinSolSetLSetupFn`, and `SUNNonlinSolSetLSolveFn` functions, that are set to `NULL`. The `SUNNONLINSOL_FIXEDPOINT` functions have the same names as those defined by the generic `SUNNONLINSOL` API with `_FixedPoint` appended to the function name. Unless using the `SUNNONLINSOL_FIXEDPOINT` module as a standalone nonlinear solver the generic functions defined in sections 9.1.1 – 9.1.3 should be called in favor of the `SUNNONLINSOL_FIXEDPOINT`-specific implementations.

The `SUNNONLINSOL_FIXEDPOINT` module also defines the following additional user-callable function.

**SUNNonlinSolGetSysFn\_FixedPoint**

Call	<code>retval = SUNNonlinSolGetSysFn_FixedPoint(NLS, SysFn);</code>
Description	The function <code>SUNNonlinSolGetSysFn_FixedPoint</code> returns the fixed-point function that defines the nonlinear system.
Arguments	<code>NLS</code> ( <code>SUNNonlinearSolver</code> ) a <code>SUNNONLINSOL</code> object <code>SysFn</code> ( <code>SUNNonlinSolSysFn*</code> ) the function defining the nonlinear system.
Return value	The return value <code>retval</code> (of type <code>int</code> ) should be zero for a successful call, and a negative value for a failure.
Notes	This function is intended for users that wish to evaluate the fixed-point function in a custom convergence test function for the <code>SUNNONLINSOL_FIXEDPOINT</code> module. We note that <code>SUNNONLINSOL_FIXEDPOINT</code> will not leverage the results from any user calls to <code>SysFn</code> .

### 9.3.3 SUNNonlinearSolver\_FixedPoint content

The *content* field of the SUNNONLINSOL\_FIXEDPOINT module is the following structure.

```
struct _SUNNonlinearSolverContent_FixedPoint {

    SUNNonlinSolSysFn    Sys;
    SUNNonlinSolConvTestFn CTest;

    int      m;
    int      *imap;
    realtype *R;
    realtype *gamma;
    realtype *cvals;
    N_Vector *df;
    N_Vector *dg;
    N_Vector *q;
    N_Vector *Xvecs;
    N_Vector yprev;
    N_Vector gy;
    N_Vector fold;
    N_Vector gold;
    N_Vector delta;
    int      curiter;
    int      maxiters;
    long int niters;
};
```

The following entries of the *content* field are always allocated:

**Sys** - function for evaluating the nonlinear system,  
**CTest** - function for checking convergence of the fixed point iteration,  
**yprev** - **N\_Vector** used to store previous fixed-point iterate,  
**gy** - **N\_Vector** used to store  $G(y)$  in fixed-point algorithm,  
**delta** - **N\_Vector** used to store difference between successive fixed-point iterates,  
**curiter** - the current number of iterations in the solve attempt,  
**maxiters** - the maximum number of fixed-point iterations allowed in a solve, and  
**niters** - the total number of nonlinear iterations across all solves.

**m** - number of acceleration vectors,

If Anderson acceleration is requested (i.e.,  $m > 0$  in the call to `SUNNonlinSol_FixedPoint`), then the following items are also allocated within the *content* field:

**imap** - index array used in acceleration algorithm (length **m**)  
**R** - small matrix used in acceleration algorithm (length **m\*m**)  
**gamma** - small vector used in acceleration algorithm (length **m**)  
**cvals** - small vector used in acceleration algorithm (length **m+1**)  
**df** - array of **N\_Vectors** used in acceleration algorithm (length **m**)  
**dg** - array of **N\_Vectors** used in acceleration algorithm (length **m**)  
**q** - array of **N\_Vectors** used in acceleration algorithm (length **m**)  
**Xvecs** - **N\_Vector** pointer array used in acceleration algorithm (length **m+1**)  
**fold** - **N\_Vector** used in acceleration algorithm  
**gold** - **N\_Vector** used in acceleration algorithm

### 9.3.4 SUNNonlinearSolver\_FixedPoint Fortran interface

For SUNDIALS integrators that include a Fortran interface, the SUNNONLINSOL\_FIXEDPOINT module also includes a Fortran-callable function for creating a `SUNNonlinearSolver` object.

<b>FSUNFIXEDPOINTINIT</b>
---------------------------

Call	<code>FSUNFIXEDPOINTINIT(code, m, ier);</code>
Description	The function <code>FSUNFIXEDPOINTINIT</code> can be called for Fortran programs to create a <code>SUNNonlinearSolver</code> object for use with SUNDIALS integrators to solve nonlinear systems of the form $G(y) = y$ .
Arguments	<p><code>code</code> (<code>int*</code>) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).</p> <p><code>m</code> (<code>int*</code>) is an integer input specifying the number of acceleration vectors.</p>
Return value	<code>ier</code> is a return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.



## Appendix A

# SUNDIALS Package Installation Procedure

The installation of any SUNDIALS package is accomplished by installing the SUNDIALS suite as a whole, according to the instructions that follow. The same procedure applies whether or not the downloaded file contains one or all solvers in SUNDIALS.

The SUNDIALS suite (or individual solvers) are distributed as compressed archives (`.tar.gz`). The name of the distribution archive is of the form `solver-x.y.z.tar.gz`, where *solver* is one of: `sundials`, `cvode`, `cvodes`, `arkode`, `ida`, `idas`, or `kinsol`, and `x.y.z` represents the version number (of the SUNDIALS suite or of the individual solver). To begin the installation, first uncompress and expand the sources, by issuing

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

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

Starting with version 2.6.0 of SUNDIALS, CMake is the only supported method of installation. The explanations of the installation procedure begins with a few common observations:

- The remainder of this chapter will follow these conventions:

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

***builddir*** is the (temporary) directory under which SUNDIALS is built.

***instdir*** is the directory under which the SUNDIALS exported header files and libraries will be installed. Typically, header files are exported under a directory `instdir/include` while libraries are installed under `instdir/lib`, with *instdir* specified at configuration time.

- For SUNDIALS CMake-based installation, in-source builds are prohibited; in other words, the build directory *builddir* can **not** be the same as *solverdir* and such an attempt will lead to an error. This prevents “polluting” the source tree and allows efficient builds for different configurations and/or options.
- The installation directory *instdir* can **not** be the same as the source directory *solverdir*.
- By default, only the libraries and header files are exported to the installation directory *instdir*. If enabled by the user (with the appropriate toggle for CMake), the examples distributed with SUNDIALS will be built together with the solver libraries but the installation step will result in exporting (by default in a subdirectory of the installation directory) the example sources and sample outputs together with automatically generated configuration files that reference the *installed* SUNDIALS headers and libraries. As such, these configuration files for the SUNDIALS examples can be used as “templates” for your own problems. CMake installs `CMakeLists.txt` files and also (as an option available only under Unix/Linux) `Makefile` files. Note this installation



approach also allows the option of building the SUNDIALS examples without having to install them. (This can be used as a sanity check for the freshly built libraries.)

- Even if generation of shared libraries is enabled, only static libraries are created for the FCMIX modules. (Because of the use of fixed names for the Fortran user-provided subroutines, FCMIX shared libraries would result in “undefined symbol” errors at link time.)

## A.1 CMake-based installation

CMake-based installation provides a platform-independent build system. CMake can generate Unix and Linux Makefiles, as well as KDevelop, Visual Studio, and (Apple) XCode project files from the same configuration file. In addition, CMake also provides a GUI front end and which allows an interactive build and installation process.

The SUNDIALS build process requires CMake version 3.1.3 or higher and a working C compiler. On Unix-like operating systems, it also requires Make (and **curses**, including its development libraries, for the GUI front end to CMake, **ccmake**), while on Windows it requires Visual Studio. CMake is continually adding new features, and the latest version can be downloaded from <http://www.cmake.org>. Build instructions for CMake (only necessary for Unix-like systems) can be found on the CMake website. Once CMake is installed, Linux/Unix users will be able to use **ccmake**, while Windows users will be able to use **CMakeSetup**.

As previously noted, when using CMake to configure, build and install SUNDIALS, it is always required to use a separate build directory. While in-source builds are possible, they are explicitly prohibited by the SUNDIALS CMake scripts (one of the reasons being that, unlike autotools, CMake does not provide a **make distclean** procedure and it is therefore difficult to clean-up the source tree after an in-source build). By ensuring a separate build directory, it is an easy task for the user to clean-up all traces of the build by simply removing the build directory. CMake does generate a **make clean** which will remove files generated by the compiler and linker.

### A.1.1 Configuring, building, and installing on Unix-like systems

The default CMake configuration will build all included solvers and associated examples and will build static and shared libraries. The *instdir* defaults to */usr/local* and can be changed by setting the **CMAKE\_INSTALL\_PREFIX** variable. Support for FORTRAN and all other options are disabled.

CMake can be used from the command line with the **cmake** command, or from a **curses**-based GUI by using the **ccmake** command. Examples for using both methods will be presented. For the examples shown it is assumed that there is a top level SUNDIALS directory with appropriate source, build and install directories:

```
% mkdir (...)sundials/instdir
% mkdir (...)sundials/builddir
% cd (...)sundials/builddir
```

#### Building with the GUI

Using CMake with the GUI follows this general process:

- Select and modify values, run configure (c key)
- New values are denoted with an asterisk
- To set a variable, move the cursor to the variable and press enter
  - If it is a boolean (ON/OFF) it will toggle the value
  - If it is string or file, it will allow editing of the string
  - For file and directories, the <tab> key can be used to complete

- Repeat until all values are set as desired and the generate option is available (g key)
- Some variables (advanced variables) are not visible right away
- To see advanced variables, toggle to advanced mode (t key)
- To search for a variable press / key, and to repeat the search, press the n key

To build the default configuration using the GUI, from the *builddir* enter the `ccmake` command and point to the *solverdir*:

```
% ccmake ../solverdir
```

The default configuration screen is shown in Figure A.1.

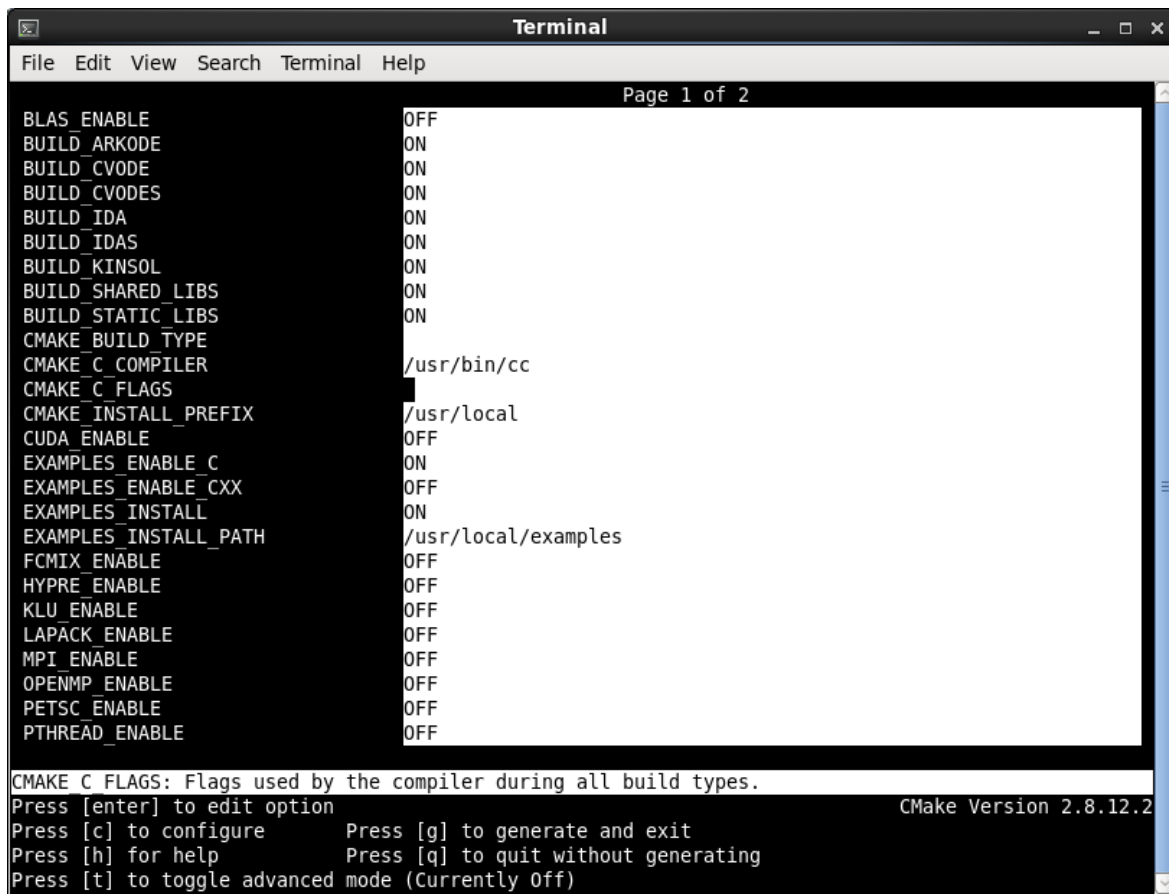


Figure A.1: Default configuration screen. Note: Initial screen is empty. To get this default configuration, press 'c' repeatedly (accepting default values denoted with asterisk) until the 'g' option is available.

The default *instldir* for both SUNDIALS and corresponding examples can be changed by setting the `CMAKE_INSTALL_PREFIX` and the `EXAMPLES_INSTALL_PATH` as shown in figure A.2.

Pressing the (g key) will generate makefiles including all dependencies and all rules to build SUNDIALS on this system. Back at the command prompt, you can now run:

```
% make
```

To install SUNDIALS in the installation directory specified in the configuration, simply run:

```
% make install
```

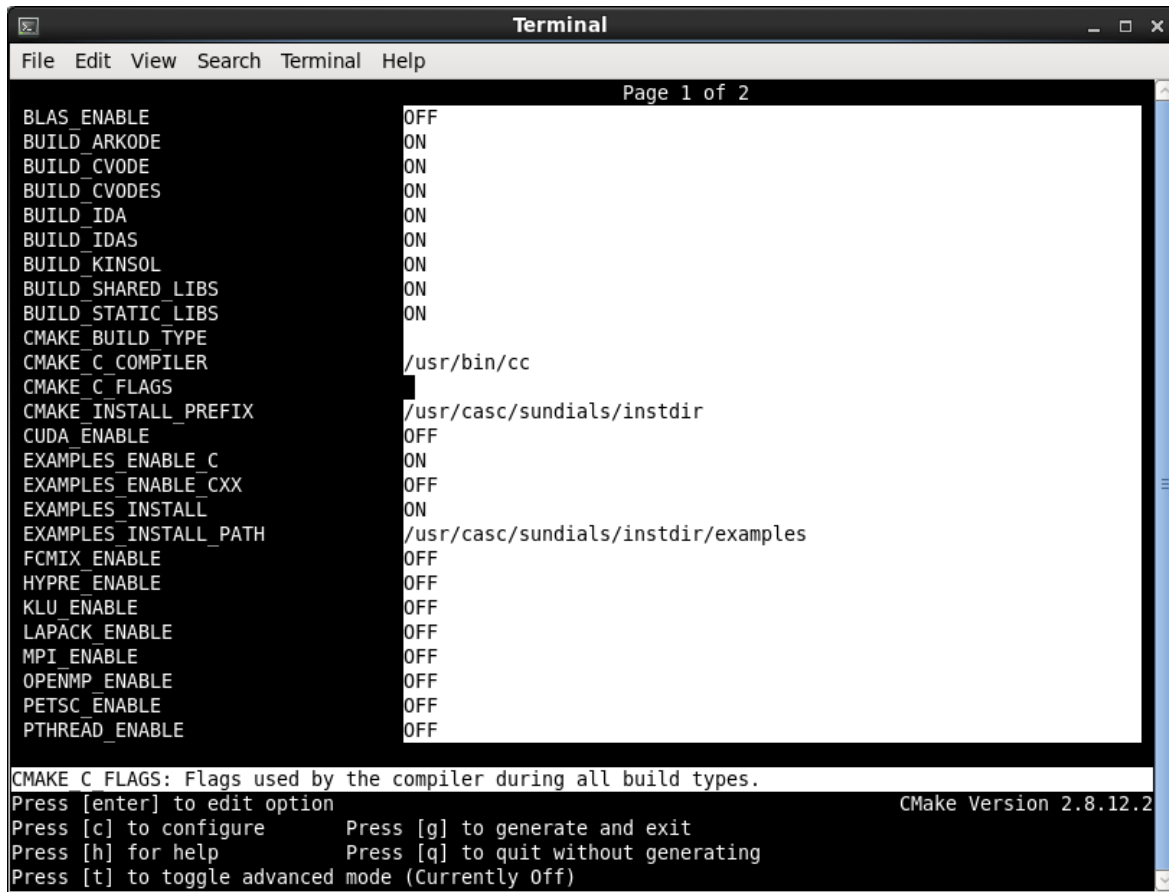


Figure A.2: Changing the *instdir* for SUNDIALS and corresponding examples

### Building from the command line

Using CMake from the command line is simply a matter of specifying CMake variable settings with the `cmake` command. The following will build the default configuration:

```

% cmake -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
> ../solverdir
% make
% make install

```

### A.1.2 Configuration options (Unix/Linux)

A complete list of all available options for a CMake-based SUNDIALS configuration is provide below. Note that the default values shown are for a typical configuration on a Linux system and are provided as illustration only.

**BLAS\_ENABLE** - Enable BLAS support

Default: OFF

Note: Setting this option to ON will trigger additional CMake options. See additional information on building with BLAS enabled in [A.1.4](#).

**BLAS\_LIBRARIES** - BLAS library

Default: /usr/lib/libblas.so

Note: CMake will search for libraries in your `LD_LIBRARY_PATH` prior to searching default system paths.

`BUILD_ARKODE` - Build the ARKODE library  
Default: ON

`BUILD_CVODE` - Build the CVODE library  
Default: ON

`BUILD_CVODES` - Build the CVODES library  
Default: ON

`BUILD_IDA` - Build the IDA library  
Default: ON

`BUILD_IDAS` - Build the IDAS library  
Default: ON

`BUILD_KINSOL` - Build the KINSOL library  
Default: ON

`BUILD_SHARED_LIBS` - Build shared libraries  
Default: ON

`BUILD_STATIC_LIBS` - Build static libraries  
Default: ON

`CMAKE_BUILD_TYPE` - Choose the type of build, options are: `None` (`CMAKE_C_FLAGS` used), `Debug`, `Release`, `RelWithDebInfo`, and `MinSizeRel`  
Default:  
Note: Specifying a build type will trigger the corresponding build type specific compiler flag options below which will be appended to the flags set by `CMAKE_<language>_FLAGS`.

`CMAKE_C_COMPILER` - C compiler  
Default: `/usr/bin/cc`

`CMAKE_C_FLAGS` - Flags for C compiler  
Default:

`CMAKE_C_FLAGS_DEBUG` - Flags used by the C compiler during debug builds  
Default: `-g`

`CMAKE_C_FLAGS_MINSIZEREL` - Flags used by the C compiler during release minsize builds  
Default: `-Os -DNDEBUG`

`CMAKE_C_FLAGS_RELEASE` - Flags used by the C compiler during release builds  
Default: `-O3 -DNDEBUG`

`CMAKE_CXX_COMPILER` - C++ compiler  
Default: `/usr/bin/c++`  
Note: A C++ compiler (and all related options) are only triggered if C++ examples are enabled (`EXAMPLES_ENABLE_CXX` is ON). All SUNDIALS solvers can be used from C++ applications by default without setting any additional configuration options.

`CMAKE_CXX_FLAGS` - Flags for C++ compiler  
Default:

`CMAKE_CXX_FLAGS_DEBUG` - Flags used by the C++ compiler during debug builds  
Default: `-g`

**CMAKE\_CXX\_FLAGS\_MINSIZEREL** - Flags used by the C++ compiler during release minsize builds

Default: -Os -DNDEBUG

**CMAKE\_CXX\_FLAGS\_RELEASE** - Flags used by the C++ compiler during release builds

Default: -O3 -DNDEBUG

**CMAKE\_Fortran\_COMPILER** - Fortran compiler

Default: /usr/bin/gfortran

Note: Fortran support (and all related options) are triggered only if either Fortran-C support is enabled (**FCMIX\_ENABLE** is ON) or BLAS/LAPACK support is enabled (**BLAS\_ENABLE** or **LAPACK\_ENABLE** is ON).

**CMAKE\_Fortran\_FLAGS** - Flags for Fortran compiler

Default:

**CMAKE\_Fortran\_FLAGS\_DEBUG** - Flags used by the Fortran compiler during debug builds

Default: -g

**CMAKE\_Fortran\_FLAGS\_MINSIZEREL** - Flags used by the Fortran compiler during release minsize builds

Default: -Os

**CMAKE\_Fortran\_FLAGS\_RELEASE** - Flags used by the Fortran compiler during release builds

Default: -O3

**CMAKE\_INSTALL\_PREFIX** - Install path prefix, prepended onto install directories

Default: /usr/local

Note: The user must have write access to the location specified through this option. Exported SUNDIALS header files and libraries will be installed under subdirectories **include** and **lib** of **CMAKE\_INSTALL\_PREFIX**, respectively.

**CUDA\_ENABLE** - Build the SUNDIALS CUDA vector module.

Default: OFF

**EXAMPLES\_ENABLE\_C** - Build the SUNDIALS C examples

Default: ON

**EXAMPLES\_ENABLE\_CUDA** - Build the SUNDIALS CUDA examples

Default: OFF

Note: You need to enable CUDA support to build these examples.

**EXAMPLES\_ENABLE\_CXX** - Build the SUNDIALS C++ examples

Default: OFF

**EXAMPLES\_ENABLE\_RAJA** - Build the SUNDIALS RAJA examples

Default: OFF

Note: You need to enable CUDA and RAJA support to build these examples.

**EXAMPLES\_ENABLE\_F77** - Build the SUNDIALS Fortran77 examples

Default: ON (if **FCMIX\_ENABLE** is ON)

**EXAMPLES\_ENABLE\_F90** - Build the SUNDIALS Fortran90 examples

Default: OFF

**EXAMPLES\_INSTALL** - Install example files

Default: ON

Note: This option is triggered when any of the SUNDIALS example programs are enabled (**EXAMPLES\_ENABLE\_<language>** is ON). If the user requires installation of example programs then the sources and sample output files for all SUNDIALS modules that are currently enabled will be exported to the directory specified by **EXAMPLES\_INSTALL\_PATH**. A CMake configuration

script will also be automatically generated and exported to the same directory. Additionally, if the configuration is done under a Unix-like system, makefiles for the compilation of the example programs (using the installed SUNDIALS libraries) will be automatically generated and exported to the directory specified by `EXAMPLES_INSTALL_PATH`.

`EXAMPLES_INSTALL_PATH` - Output directory for installing example files

Default: `/usr/local/examples`

Note: The actual default value for this option will be an `examples` subdirectory created under `CMAKE_INSTALL_PREFIX`.

`FCMIX_ENABLE` - Enable Fortran-C support

Default: `OFF`

`HYPRE_ENABLE` - Enable *hypre* support

Default: `OFF`

Note: See additional information on building with *hypre* enabled in [A.1.4](#).

`HYPRE_INCLUDE_DIR` - Path to *hypre* header files

`HYPRE_LIBRARY_DIR` - Path to *hypre* installed library files

`KLU_ENABLE` - Enable KLU support

Default: `OFF`

Note: See additional information on building with KLU enabled in [A.1.4](#).

`KLU_INCLUDE_DIR` - Path to SuiteSparse header files

`KLU_LIBRARY_DIR` - Path to SuiteSparse installed library files

`LAPACK_ENABLE` - Enable LAPACK support

Default: `OFF`

Note: Setting this option to `ON` will trigger additional CMake options. See additional information on building with LAPACK enabled in [A.1.4](#).

`LAPACK_LIBRARIES` - LAPACK (and BLAS) libraries

Default: `/usr/lib/liblapack.so;/usr/lib/libblas.so`

Note: CMake will search for libraries in your `LD_LIBRARY_PATH` prior to searching default system paths.

`MPI_ENABLE` - Enable MPI support (build the parallel nvector).

Default: `OFF`

Note: Setting this option to `ON` will trigger several additional options related to MPI.

`MPI_C_COMPILER` - `mpicc` program

Default:

`MPI_CXX_COMPILER` - `mpicxx` program

Default:

Note: This option is triggered only if MPI is enabled (`MPI_ENABLE` is `ON`) and C++ examples are enabled (`EXAMPLES_ENABLE_CXX` is `ON`). All SUNDIALS solvers can be used from C++ MPI applications by default without setting any additional configuration options other than `MPI_ENABLE`.

`MPI_Fortran_COMPILER` - `mpif77` or `mpif90` program

Default:

Note: This option is triggered only if MPI is enabled (`MPI_ENABLE` is `ON`), Fortran-C support is enabled (`FCMIX_ENABLE` is `ON`), and Fortran77 or Fortran90 examples are enabled (`EXAMPLES_ENABLE_F77` or `EXAMPLES_ENABLE_F90` are `ON`).

**MPIEXEC\_EXECUTABLE** - Specify the executable for running MPI programs

Default: `mpirun`

Note: This option is triggered only if MPI is enabled (`MPI_ENABLE` is ON).

**OPENMP\_ENABLE** - Enable OpenMP support (build the OpenMP nvector).

Default: OFF

**PETSC\_ENABLE** - Enable PETSc support

Default: OFF

Note: See additional information on building with PETSc enabled in [A.1.4](#).

**PETSC\_INCLUDE\_DIR** - Path to PETSc header files

**PETSC\_LIBRARY\_DIR** - Path to PETSc installed library files

**PTHREAD\_ENABLE** - Enable Pthreads support (build the Pthreads nvector).

Default: OFF

**RAJA\_ENABLE** - Enable RAJA support (build the RAJA nvector).

Default: OFF

Note: You need to enable CUDA in order to build the RAJA vector module.

**SUNDIALS\_F77\_FUNC\_CASE** - **advanced option** - Specify the case to use in the Fortran name-mangling scheme, options are: `lower` or `upper`

Default:

Note: The build system will attempt to infer the Fortran name-mangling scheme using the Fortran compiler. This option should only be used if a Fortran compiler is not available or to override the inferred or default (`lower`) scheme if one can not be determined. If used, `SUNDIALS_F77_FUNC_UNDERSCORES` must also be set.

**SUNDIALS\_F77\_FUNC\_UNDERSCORES** - **advanced option** - Specify the number of underscores to append in the Fortran name-mangling scheme, options are: `none`, `one`, or `two`

Default:

Note: The build system will attempt to infer the Fortran name-mangling scheme using the Fortran compiler. This option should only be used if a Fortran compiler is not available or to override the inferred or default (`one`) scheme if one can not be determined. If used, `SUNDIALS_F77_FUNC_CASE` must also be set.

**SUNDIALS\_INDEX\_TYPE** - **advanced**

Integer type used for SUNDIALS indices. The size must match the size provided for the `SUNDIALS_INDEX_SIZE` option.

Default:

Note: In past SUNDIALS versions, a user could set this option to `INT64_T` to use 64-bit integers, or `INT32_T` to use 32-bit integers. Starting in SUNDIALS 3.2.0, these special values are deprecated. For SUNDIALS 3.2.0 and up, a user will only need to use the `SUNDIALS_INDEX_SIZE` option in most cases.

**SUNDIALS\_INDEX\_SIZE** - Integer size (in bits) used for indices in SUNDIALS, options are: `32` or `64`

Default: `64`

Note: The build system tries to find an integer type of appropriate size. Candidate 64-bit integer types are (in order of preference): `int64_t`, `__int64`, `long long`, and `long`. Candidate 32-bit integers are (in order of preference): `int32_t`, `int`, and `long`. The advanced option, `SUNDIALS_INDEX_TYPE` can be used to provide a type not listed here.

**SUNDIALS\_PRECISION** - Precision used in SUNDIALS, options are: `double`, `single`, or `extended`

Default: `double`



**SUPERLUMT\_ENABLE** - Enable SuperLU\_MT support

Default: OFF

Note: See additional information on building with SuperLU\_MT enabled in [A.1.4](#).

**SUPERLUMT\_INCLUDE\_DIR** - Path to SuperLU\_MT header files (typically SRC directory)

**SUPERLUMT\_LIBRARY\_DIR** - Path to SuperLU\_MT installed library files

**SUPERLUMT\_THREAD\_TYPE** - Must be set to Pthread or OpenMP

Default: Pthread

**USE\_GENERIC\_MATH** - Use generic (stdc) math libraries

Default: ON

### xSDK Configuration Options

SUNDIALS supports CMake configuration options defined by the Extreme-scale Scientific Software Development Kit (xSDK) community policies (see <https://xsdk.info> for more information). xSDK CMake options are unused by default but may be activated by setting **USE\_XSDK\_DEFAULTS** to ON.

When xSDK options are active, they will overwrite the corresponding SUNDIALS option and may have different default values (see details below). As such the equivalent SUNDIALS options should not be used when configuring with xSDK options. In the GUI front end to CMake (**ccmake**), setting **USE\_XSDK\_DEFAULTS** to ON will hide the corresponding SUNDIALS options as advanced CMake variables. During configuration, messages are output detailing which xSDK flags are active and the equivalent SUNDIALS options that are replaced. Below is a complete list xSDK options and the corresponding SUNDIALS options if applicable.



**TPL\_BLAS\_LIBRARIES** - BLAS library

Default: /usr/lib/libblas.so

SUNDIALS equivalent: **BLAS\_LIBRARIES**

Note: CMake will search for libraries in your **LD\_LIBRARY\_PATH** prior to searching default system paths.

**TPL\_ENABLE\_BLAS** - Enable BLAS support

Default: OFF

SUNDIALS equivalent: **BLAS\_ENABLE**

**TPL\_ENABLE\_HYPRE** - Enable *hypre* support

Default: OFF

SUNDIALS equivalent: **HYPRE\_ENABLE**

**TPL\_ENABLE\_KLU** - Enable KLU support

Default: OFF

SUNDIALS equivalent: **KLU\_ENABLE**

**TPL\_ENABLE\_PETSC** - Enable PETSc support

Default: OFF

SUNDIALS equivalent: **PETSC\_ENABLE**

**TPL\_ENABLE\_LAPACK** - Enable LAPACK support

Default: OFF

SUNDIALS equivalent: **LAPACK\_ENABLE**

**TPL\_ENABLE\_SUPERLUMT** - Enable SuperLU\_MT support

Default: OFF

SUNDIALS equivalent: **SUPERLUMT\_ENABLE**

**TPL\_HYPRE\_INCLUDE\_DIRS** - Path to *hypre* header files

SUNDIALS equivalent: **HYPRE\_INCLUDE\_DIR**

**TPL\_HYPRE\_LIBRARIES** - *hypre* library  
SUNDIALS equivalent: N/A

**TPL\_KLU\_INCLUDE\_DIRS** - Path to KLU header files  
SUNDIALS equivalent: **KLU\_INCLUDE\_DIR**

**TPL\_KLU\_LIBRARIES** - KLU library  
SUNDIALS equivalent: N/A

**TPL\_LAPACK\_LIBRARIES** - LAPACK (and BLAS) libraries  
Default: `/usr/lib/liblapack.so;/usr/lib/libblas.so`  
SUNDIALS equivalent: **LAPACK\_LIBRARIES**  
Note: CMake will search for libraries in your **LD\_LIBRARY\_PATH** prior to searching default system paths.

**TPL\_PETSC\_INCLUDE\_DIRS** - Path to PETSc header files  
SUNDIALS equivalent: **PETSC\_INCLUDE\_DIR**

**TPL\_PETSC\_LIBRARIES** - PETSc library  
SUNDIALS equivalent: N/A

**TPL\_SUPERLUMT\_INCLUDE\_DIRS** - Path to SuperLU\_MT header files  
SUNDIALS equivalent: **SUPERLUMT\_INCLUDE\_DIR**

**TPL\_SUPERLUMT\_LIBRARIES** - SuperLU\_MT library  
SUNDIALS equivalent: N/A

**TPL\_SUPERLUMT\_THREAD\_TYPE** - SuperLU\_MT library thread type  
SUNDIALS equivalent: **SUPERLUMT\_THREAD\_TYPE**

**USE\_XSDK\_DEFAULTS** - Enable xSDK default configuration settings  
Default: OFF  
SUNDIALS equivalent: N/A  
Note: Enabling xSDK defaults also sets **CMAKE\_BUILD\_TYPE** to Debug

**XSDK\_ENABLE\_FORTRAN** - Enable SUNDIALS Fortran interface  
Default: OFF  
SUNDIALS equivalent: **FCMIX\_ENABLE**

**XSDK\_INDEX\_SIZE** - Integer size (bits) used for indices in SUNDIALS, options are: 32 or 64  
Default: 32  
SUNDIALS equivalent: **SUNDIALS\_INDEX\_SIZE**

**XSDK\_PRECISION** - Precision used in SUNDIALS, options are: **double**, **single**, or **quad**  
Default: **double**  
SUNDIALS equivalent: **SUNDIALS\_PRECISION**

### A.1.3 Configuration examples

The following examples will help demonstrate usage of the CMake configure options. To configure SUNDIALS using the default C and Fortran compilers, and default **mpicc** and **mpif77** parallel compilers, enable compilation of examples, and install libraries, headers, and example sources under subdirectories of `/home/myname/sundials/`, use:

```
% cmake \
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
> -DMPI_ENABLE=ON \
> -DFCMIX_ENABLE=ON \
```

```
> /home/myname/sundials/solverdir
%
% make install
%
```

To disable installation of the examples, use:

```
% cmake \
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
> -DMPI_ENABLE=ON \
> -DFCMIX_ENABLE=ON \
> -DEXAMPLES_INSTALL=OFF \
> /home/myname/sundials/solverdir
%
% make install
%
```

#### A.1.4 Working with external Libraries

The SUNDIALS suite contains many options to enable implementation flexibility when developing solutions. The following are some notes addressing specific configurations when using the supported third party libraries. When building SUNDIALS as a shared library external libraries any used with SUNDIALS must also be build as a shared library or as a static library compiled with the `-fPIC` flag.



##### Building with BLAS

SUNDIALS does not utilize BLAS directly but it may be needed by other external libraries that SUNDIALS can be built with (e.g. LAPACK, PETSc, SuperLU\_MT, etc.). To enable BLAS, set the `BLAS_ENABLE` option to `ON`. If the directory containing the BLAS library is in the `LD_LIBRARY_PATH` environment variable, CMake will set the `BLAS_LIBRARIES` variable accordingly, otherwise CMake will attempt to find the BLAS library in standard system locations. To explicitly tell CMake what libraries to use, the `BLAS_LIBRARIES` variable can be set to the desired library. Example:

```
% cmake \
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
> -DBLAS_ENABLE=ON \
> -DBLAS_LIBRARIES=/myblaspath/lib/libblas.so \
> -DSUPERLUMT_ENABLE=ON \
> -DSUPERLUMT_INCLUDE_DIR=/mysuperlumtpath/SRC
> -DSUPERLUMT_LIBRARY_DIR=/mysuperlumtpath/lib
> /home/myname/sundials/solverdir
%
% make install
%
```

When allowing CMake to automatically locate the LAPACK library, CMake *may* also locate the corresponding BLAS library.



If a working Fortran compiler is not available to infer the Fortran name-mangling scheme, the options `SUNDIALS_F77_FUNC.CASE` and `SUNDIALS_F77_FUNC.UNDERSCORES` *must* be set in order to bypass the check for a Fortran compiler and define the name-mangling scheme. The defaults for these options in earlier versions of SUNDIALS were `lower` and `one` respectively.

### Building with LAPACK

To enable LAPACK, set the `LAPACK_ENABLE` option to `ON`. If the directory containing the LAPACK library is in the `LD_LIBRARY_PATH` environment variable, CMake will set the `LAPACK_LIBRARIES` variable accordingly, otherwise CMake will attempt to find the LAPACK library in standard system locations. To explicitly tell CMake what library to use, the `LAPACK_LIBRARIES` variable can be set to the desired libraries. When setting the LAPACK location explicitly the location of the corresponding BLAS library will also need to be set. Example:



```
% cmake \
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
> -DBLAS_ENABLE=ON \
> -DBLAS_LIBRARIES=/mylapackpath/lib/libblas.so \
> -DLAPACK_ENABLE=ON \
> -DLAPACK_LIBRARIES=/mylapackpath/lib/liblapack.so \
> /home/myname/sundials/solverdir
%
% make install
%
```



When allowing CMake to automatically locate the LAPACK library, CMake *may* also locate the corresponding BLAS library.

If a working Fortran compiler is not available to infer the Fortran name-mangling scheme, the options `SUNDIALS_F77_FUNC_CASE` and `SUNDIALS_F77_FUNC_UNDERSCORES` *must* be set in order to bypass the check for a Fortran compiler and define the name-mangling scheme. The defaults for these options in earlier versions of SUNDIALS were `lower` and `one` respectively.

### Building with KLU

The KLU libraries are part of SuiteSparse, a suite of sparse matrix software, available from the Texas A&M University website: <http://faculty.cse.tamu.edu/davis/suitesparse.html>. SUNDIALS has been tested with SuiteSparse version 4.5.3. To enable KLU, set `KLU_ENABLE` to `ON`, set `KLU_INCLUDE_DIR` to the `include` path of the KLU installation and set `KLU_LIBRARY_DIR` to the `lib` path of the KLU installation. The CMake configure will result in populating the following variables: `AMD_LIBRARY`, `AMD_LIBRARY_DIR`, `BTF_LIBRARY`, `BTF_LIBRARY_DIR`, `COLAMD_LIBRARY`, `COLAMD_LIBRARY_DIR`, and `KLU_LIBRARY`.

### Building with SuperLU\_MT

The SuperLU\_MT libraries are available for download from the Lawrence Berkeley National Laboratory website: [http://crd-legacy.lbl.gov/~xiaoye/SuperLU/#superlu\\_mt](http://crd-legacy.lbl.gov/~xiaoye/SuperLU/#superlu_mt). SUNDIALS has been tested with SuperLU\_MT version 3.1. To enable SuperLU\_MT, set `SUPERLUMT_ENABLE` to `ON`, set `SUPERLUMT_INCLUDE_DIR` to the `SRC` path of the SuperLU\_MT installation, and set the variable `SUPERLUMT_LIBRARY_DIR` to the `lib` path of the SuperLU\_MT installation. At the same time, the variable `SUPERLUMT_THREAD_TYPE` must be set to either `Pthread` or `OpenMP`.

Do not mix thread types when building SUNDIALS solvers. If threading is enabled for SUNDIALS by having either `OPENMP_ENABLE` or `PTHREAD_ENABLE` set to `ON` then SuperLU\_MT should be set to use the same threading type.



### Building with PETSc

The PETSc libraries are available for download from the Argonne National Laboratory website: <http://www.mcs.anl.gov/petsc>. SUNDIALS has been tested with PETSc version 3.7.2. To enable PETSc, set `PETSC_ENABLE` to `ON`, set `PETSC_INCLUDE_DIR` to the `include` path of the PETSc installation, and set the variable `PETSC_LIBRARY_DIR` to the `lib` path of the PETSc installation.

### Building with *hypre*

The *hypre* libraries are available for download from the Lawrence Livermore National Laboratory website: <http://computation.llnl.gov/projects/hypre>. SUNDIALS has been tested with *hypre* version 2.11.1. To enable *hypre*, set `HYPRE_ENABLE` to `ON`, set `HYPRE_INCLUDE_DIR` to the `include` path of the *hypre* installation, and set the variable `HYPRE_LIBRARY_DIR` to the `lib` path of the *hypre* installation.

### Building with CUDA

SUNDIALS CUDA modules and examples have been tested with version 8.0 of the CUDA toolkit. To build them, you need to install the Toolkit and compatible NVIDIA drivers. Both are available for download from the NVIDIA website: <https://developer.nvidia.com/cuda-downloads>. To enable CUDA, set `CUDA_ENABLE` to `ON`. If CUDA is installed in a nonstandard location, you may be prompted to set the variable `CUDA_TOOLKIT_ROOT_DIR` with your CUDA Toolkit installation path. To enable CUDA examples, set `EXAMPLES_ENABLE_CUDA` to `ON`.

### Building with RAJA

RAJA is a performance portability layer developed by Lawrence Livermore National Laboratory and can be obtained from <https://github.com/LLNL/RAJA>. SUNDIALS RAJA modules and examples have been tested with RAJA version 0.3. Building SUNDIALS RAJA modules requires a CUDA-enabled RAJA installation. To enable RAJA, set `CUDA_ENABLE` and `RAJA_ENABLE` to `ON`. If RAJA is installed in a nonstandard location you will be prompted to set the variable `RAJA_DIR` with the path to the RAJA CMake configuration file. To enable building the RAJA examples set `EXAMPLES_ENABLE_RAJA` to `ON`.

#### A.1.5 Testing the build and installation

If SUNDIALS was configured with `EXAMPLES_ENABLE_<language>` options to `ON`, then a set of regression tests can be run after building with the `make` command by running:

```
% make test
```

Additionally, if `EXAMPLES_INSTALL` was also set to `ON`, then a set of smoke tests can be run after installing with the `make install` command by running:

```
% make test_install
```

## A.2 Building and Running Examples

Each of the SUNDIALS solvers is distributed with a set of examples demonstrating basic usage. To build and install the examples, set at least of the `EXAMPLES_ENABLE_<language>` options to `ON`, and set `EXAMPLES_INSTALL` to `ON`. Specify the installation path for the examples with the variable `EXAMPLES_INSTALL_PATH`. CMake will generate `CMakeLists.txt` configuration files (and `Makefile` files if on Linux/Unix) that reference the *installed* SUNDIALS headers and libraries.

Either the `CMakeLists.txt` file or the traditional `Makefile` may be used to build the examples as well as serve as a template for creating user developed solutions. To use the supplied `Makefile` simply run `make` to compile and generate the executables. To use CMake from within the installed example directory, run `cmake` (or `ccmake` to use the GUI) followed by `make` to compile the example code. Note that if CMake is used, it will overwrite the traditional `Makefile` with a new CMake-generated `Makefile`. The resulting output from running the examples can be compared with example output bundled in the SUNDIALS distribution.

NOTE: There will potentially be differences in the output due to machine architecture, compiler versions, use of third party libraries etc.



### A.3 Configuring, building, and installing on Windows

CMake can also be used to build SUNDIALS on Windows. To build SUNDIALS for use with Visual Studio the following steps should be performed:

1. Unzip the downloaded tar file(s) into a directory. This will be the *solverdir*
2. Create a separate *builddir*
3. Open a Visual Studio Command Prompt and cd to *builddir*
4. Run `cmake-gui ../solverdir`
  - (a) Hit Configure
  - (b) Check/Uncheck solvers to be built
  - (c) Change CMAKE\_INSTALL\_PREFIX to *instdir*
  - (d) Set other options as desired
  - (e) Hit Generate
5. Back in the VS Command Window:
  - (a) Run `msbuild ALL_BUILD.vcxproj`
  - (b) Run `msbuild INSTALL.vcxproj`

The resulting libraries will be in the *instdir*. The SUNDIALS project can also now be opened in Visual Studio. Double click on the `ALL_BUILD.vcxproj` file to open the project. Build the whole *solution* to create the SUNDIALS libraries. To use the SUNDIALS libraries in your own projects, you must set the include directories for your project, add the SUNDIALS libraries to your project solution, and set the SUNDIALS libraries as dependencies for your project.

### A.4 Installed libraries and exported header files

Using the CMake SUNDIALS build system, the command

```
% make install
```

will install the libraries under *libdir* and the public header files under *includedir*. The values for these directories are *instdir/lib* and *instdir/include*, respectively. The location can be changed by setting the CMake variable `CMAKE_INSTALL_PREFIX`. Although all installed libraries reside under *libdir/lib*, the public header files are further organized into subdirectories under *includedir/include*.

The installed libraries and exported header files are listed for reference in Table A.1. The file extension *.lib* is typically *.so* for shared libraries and *.a* for static libraries. Note that, in the Tables, names are relative to *libdir* for libraries and to *includedir* for header files.

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

Table A.1: SUNDIALS libraries and header files

SHARED	Libraries	n/a
<i>continued on next page</i>		

<i>continued from last page</i>		
	Header files	sundials/sundials_config.h sundials/sundials_fconfig.h sundials/sundials_types.h sundials/sundials_math.h sundials/sundials_nvector.h sundials/sundials_fnvector.h sundials/sundials_matrix.h sundials/sundials_linearsolver.h sundials/sundials_iterative.h sundials/sundials_direct.h sundials/sundials_dense.h sundials/sundials_band.h sundials/sundials_nonlinearsolver.h sundials/sundials_version.h sundials/sundials_mpi_types.h
NVECTOR_SERIAL	Libraries	libsundials_nvecserial. <i>lib</i> libsundials_fnvecserial.a
	Header files	nvector/nvector_serial.h
NVECTOR_PARALLEL	Libraries	libsundials_nvecparallel. <i>lib</i> libsundials_fnvecparallel.a
	Header files	nvector/nvector_parallel.h
NVECTOR_OPENMP	Libraries	libsundials_nvecopenmp. <i>lib</i> libsundials_fnvecopenmp.a
	Header files	nvector/nvector_openmp.h
NVECTOR_PTHREADS	Libraries	libsundials_nvecpthreads. <i>lib</i> libsundials_fnvecpthreads.a
	Header files	nvector/nvector_pthreads.h
NVECTOR_PARHYP	Libraries	libsundials_nvecparhyp. <i>lib</i>
	Header files	nvector/nvector_parhyp.h
NVECTOR_PETSC	Libraries	libsundials_nvecpetsc. <i>lib</i>
	Header files	nvector/nvector_petsc.h
NVECTOR_CUDA	Libraries	libsundials_nveccuda. <i>lib</i>
	Libraries	libsundials_nvecmpicuda. <i>lib</i>
	Header files	nvector/nvector_cuda.h nvector/nvector_mpicuda.h nvector/cuda/ThreadPartitioning.hpp nvector/cuda/Vector.hpp nvector/cuda/VectorKernels.cuh
NVECTOR_RAJA	Libraries	libsundials_nveccudaraja. <i>lib</i>
	Libraries	libsundials_nveccudampiraja. <i>lib</i>
	Header files	nvector/nvector_raja.h nvector/nvector_mpiraja.h nvector/raja/Vector.hpp
SUNMATRIX_BAND	Libraries	libsundials_sunmatrixband. <i>lib</i> libsundials_fsunmatrixband.a
<i>continued on next page</i>		

<i>continued from last page</i>		
	Header files	sunmatrix/sunmatrix_band.h
SUNMATRIX_DENSE	Libraries	libsundials_sunmatrixdense. <i>lib</i> libsundials_fsunmatrixdense.a
	Header files	sunmatrix/sunmatrix_dense.h
SUNMATRIX_SPARSE	Libraries	libsundials_sunmatrixsparse. <i>lib</i> libsundials_fsunmatrixsparse.a
	Header files	sunmatrix/sunmatrix_sparse.h
SUNLINSOL_BAND	Libraries	libsundials_sunlinsolband. <i>lib</i> libsundials_fsunlinsolband.a
	Header files	sunlinsol/sunlinsol_band.h
SUNLINSOL_DENSE	Libraries	libsundials_sunlinsoldense. <i>lib</i> libsundials_fsunlinsoldense.a
	Header files	sunlinsol/sunlinsol_dense.h
SUNLINSOL_KLU	Libraries	libsundials_sunlinsolklu. <i>lib</i> libsundials_fsunlinsolklu.a
	Header files	sunlinsol/sunlinsol_klu.h
SUNLINSOL_LAPACKBAND	Libraries	libsundials_sunlinsollapackband. <i>lib</i> libsundials_fsunlinsollapackband.a
	Header files	sunlinsol/sunlinsol_lapackband.h
SUNLINSOL_LAPACKDENSE	Libraries	libsundials_sunlinsollapackdense. <i>lib</i> libsundials_fsunlinsollapackdense.a
	Header files	sunlinsol/sunlinsol_lapackdense.h
SUNLINSOL_PCG	Libraries	libsundials_sunlinsolpcg. <i>lib</i> libsundials_fsunlinsolpcg.a
	Header files	sunlinsol/sunlinsol_pcg.h
SUNLINSOL_SPCGGS	Libraries	libsundials_sunlinsolspbcgs. <i>lib</i> libsundials_fsunlinsolspbcgs.a
	Header files	sunlinsol/sunlinsol_spcggs.h
SUNLINSOL_SPGMR	Libraries	libsundials_sunlinsolspfgmr. <i>lib</i> libsundials_fsunlinsolspfgmr.a
	Header files	sunlinsol/sunlinsol_spfgmr.h
SUNLINSOL_SPGMR	Libraries	libsundials_sunlinsolspgmr. <i>lib</i> libsundials_fsunlinsolspgmr.a
	Header files	sunlinsol/sunlinsol_spgmr.h
SUNLINSOL_SPTFQMR	Libraries	libsundials_sunlinsolsptfqmr. <i>lib</i> libsundials_fsunlinsolsptfqmr.a
	Header files	sunlinsol/sunlinsol_sptfqmr.h
SUNLINSOL_SUPERLUMT	Libraries	libsundials_sunlinsolsuperlumt. <i>lib</i> libsundials_fsunlinsolsuperlumt.a
	Header files	sunlinsol/sunlinsol_superlumt.h
SUNNONLINSOL_NEWTON	Libraries	libsundials_sunnonlinsolnewton. <i>lib</i>
<i>continued on next page</i>		



<i>continued from last page</i>			
		libsundials_fsunnonlinsolnewton.a	
	Header files	sunnonlinsol/sunnonlinsol_newton.h	
SUNNONLINSOL_FIXEDPOINT	Libraries	libsundials_sunnonlinsolfixedpoint. <i>lib</i> libsundials_fsunnonlinsolfixedpoint.a	
	Header files	sunnonlinsol/sunnonlinsol_fixedpoint.h	
CVODE	Libraries	libsundials_cvode. <i>lib</i>	libsundials_fcvcde.a
	Header files	cvode/cvode.h	cvode/cvode_impl.h
		cvode/cvode_direct.h	cvode/cvode_ls.h
		cvode/cvode_spils.h	cvode/cvode_bandpre.h
		cvode/cvode_bbdpre.h	
CVODES	Libraries	libsundials_cvodes. <i>lib</i>	
	Header files	cvodes/cvodes.h	cvodes/cvodes_impl.h
		cvodes/cvodes_direct.h	cvodes/cvodes_spils.h
		cvodes/cvodes_bandpre.h	cvodes/cvodes_bbdpre.h
ARKODE	Libraries	libsundials_arkode. <i>lib</i>	libsundials_farkode.a
	Header files	arkode/arkode.h	arkode/arkode_impl.h
		arkode/arkode_ls.h	arkode/arkode_bandpre.h
		arkode/arkode_bbdpre.h	
IDA	Libraries	libsundials_ida. <i>lib</i>	libsundials_fida.a
	Header files	ida/ida.h	ida/ida_impl.h
		ida/ida_direct.h	ida/ida_spils.h
		ida/ida_bbdpre.h	
IDAS	Libraries	libsundials_idas. <i>lib</i>	
	Header files	idas/idas.h	idas/idas_impl.h
		idas/idas_direct.h	idas/idas_spils.h
		idas/idas_bbdpre.h	
KINSOL	Libraries	libsundials_kinsol. <i>lib</i>	libsundials_fkinsol.a
	Header files	kinsol/kinsol.h	kinsol/kinsol_impl.h
		kinsol/kinsol_direct.h	kinsol/kinsol_ls.h
		kinsol/kinsol_spils.h	kinsol/kinsol_bbdpre.h



# Appendix B

## IDA Constants

Below we list all input and output constants used by the main solver and linear solver modules, together with their numerical values and a short description of their meaning.

### B.1 IDA input constants

IDA <b>main solver module</b>		
IDA_NORMAL	1	Solver returns at specified output time.
IDA_ONE_STEP	2	Solver returns after each successful step.
IDA_YA_YDP_INIT	1	Compute $y_a$ and $\dot{y}_d$ , given $y_d$ .
IDA_Y_INIT	2	Compute $y$ , given $\dot{y}$ .
Iterative linear solver module		
PREC_NONE	0	No preconditioning
PREC_LEFT	1	Preconditioning on the left.
MODIFIED_GS	1	Use modified Gram-Schmidt procedure.
CLASSICAL_GS	2	Use classical Gram-Schmidt procedure.

### B.2 IDA output constants

IDA <b>main solver module</b>		
IDA_SUCCESS	0	Successful function return.
IDA_TSTOP_RETURN	1	IDASolve succeeded by reaching the specified stopping point.
IDA_ROOT_RETURN	2	IDASolve succeeded and found one or more roots.
IDA_WARNING	99	IDASolve succeeded but an unusual situation occurred.
IDA_TOO_MUCH_WORK	-1	The solver took <code>mxstep</code> internal steps but could not reach tout.
IDA_TOO_MUCH_ACC	-2	The solver could not satisfy the accuracy demanded by the user for some internal step.
IDA_ERR_FAIL	-3	Error test failures occurred too many times during one internal time step or minimum step size was reached.

IDA_CONV_FAIL	-4	Convergence test failures occurred too many times during one internal time step or minimum step size was reached.
IDA_LINIT_FAIL	-5	The linear solver's initialization function failed.
IDA_LSETUP_FAIL	-6	The linear solver's setup function failed in an unrecoverable manner.
IDA_LSOLVE_FAIL	-7	The linear solver's solve function failed in an unrecoverable manner.
IDA_RES_FAIL	-8	The user-provided residual function failed in an unrecoverable manner.
IDA_REP_RES_FAIL	-9	The user-provided residual function repeatedly returned a recoverable error flag, but the solver was unable to recover.
IDA_RTFUNC_FAIL	-10	The rootfinding function failed in an unrecoverable manner.
IDA_CONSTR_FAIL	-11	The inequality constraints were violated and the solver was unable to recover.
IDA_FIRST_RES_FAIL	-12	The user-provided residual function failed recoverably on the first call.
IDA_LINESEARCH_FAIL	-13	The line search failed.
IDA_NO_RECOVERY	-14	The residual function, linear solver setup function, or linear solver solve function had a recoverable failure, but IDACalcIC could not recover.
IDA_NLS_INIT_FAIL	-15	The nonlinear solver's init routine failed.
IDA_NLS_SETUP_FAIL	-16	The nonlinear solver's setup routine failed.
IDA_MEM_NULL	-20	The <code>ida_mem</code> argument was NULL.
IDA_MEM_FAIL	-21	A memory allocation failed.
IDA_ILL_INPUT	-22	One of the function inputs is illegal.
IDA_NO_MALLOC	-23	The IDA memory was not allocated by a call to <code>IDAInit</code> .
IDA_BAD_EWT	-24	Zero value of some error weight component.
IDA_BAD_K	-25	The $k$ -th derivative is not available.
IDA_BAD_T	-26	The time $t$ is outside the last step taken.
IDA_BAD_DKY	-27	The vector argument where derivative should be stored is NULL.

---

IDALS **linear solver interface**

---

IDALS_SUCCESS	0	Successful function return.
IDALS_MEM_NULL	-1	The <code>ida_mem</code> argument was NULL.
IDALS_LMEM_NULL	-2	The IDALS linear solver has not been initialized.
IDALS_ILL_INPUT	-3	The IDALS solver is not compatible with the current NVECTOR module.
IDALS_MEM_FAIL	-4	A memory allocation request failed.
IDALS_PMEM_NULL	-5	The preconditioner module has not been initialized.
IDALS_JACFUNC_UNRECV	-6	The Jacobian function failed in an unrecoverable manner.
IDALS_JACFUNC_RECVR	-7	The Jacobian function had a recoverable error.
IDALS_SUNMAT_FAIL	-8	An error occurred with the current SUNMATRIX module.
IDALS_SUNLS_FAIL	-9	An error occurred with the current SUNLINSOL module.

# Bibliography

- [1] KLU Sparse Matrix Factorization Library. <http://faculty.cse.tamu.edu/davis/suitesparse.html>.
- [2] SuperLU\_MT Threaded Sparse Matrix Factorization Library. <http://crd-legacy.lbl.gov/~xiaoye/-SuperLU/>.
- [3] D. G. Anderson. Iterative procedures for nonlinear integral equations. *J. Assoc. Comput. Machinery*, 12:547–560, 1965.
- [4] K. E. Brenan, S. L. Campbell, and L. R. Petzold. *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*. SIAM, Philadelphia, Pa, 1996.
- [5] P. N. Brown. A local convergence theory for combined inexact-Newton/finite difference projection methods. *SIAM J. Numer. Anal.*, 24(2):407–434, 1987.
- [6] P. N. Brown and A. C. Hindmarsh. Reduced Storage Matrix Methods in Stiff ODE Systems. *J. Appl. Math. & Comp.*, 31:49–91, 1989.
- [7] P. N. Brown, A. C. Hindmarsh, and L. R. Petzold. Using Krylov Methods in the Solution of Large-Scale Differential-Algebraic Systems. *SIAM J. Sci. Comput.*, 15:1467–1488, 1994.
- [8] P. N. Brown, A. C. Hindmarsh, and L. R. Petzold. Consistent Initial Condition Calculation for Differential-Algebraic Systems. *SIAM J. Sci. Comput.*, 19:1495–1512, 1998.
- [9] P. N. Brown and Y. Saad. Hybrid Krylov Methods for Nonlinear Systems of Equations. *SIAM J. Sci. Stat. Comput.*, 11:450–481, 1990.
- [10] G. D. Byrne. Pragmatic Experiments with Krylov Methods in the Stiff ODE Setting. In J.R. Cash and I. Gladwell, editors, *Computational Ordinary Differential Equations*, pages 323–356, Oxford, 1992. Oxford University Press.
- [11] G. D. Byrne and A. C. Hindmarsh. User Documentation for PODE, An ODE Solver for Parallel Computers. Technical Report UCRL-ID-130884, LLNL, May 1998.
- [12] G. D. Byrne and A. C. Hindmarsh. PODE, An ODE Solver for Parallel Computers. *Intl. J. High Perf. Comput. Apps.*, 13(4):254–365, 1999.
- [13] S. D. Cohen and A. C. Hindmarsh. CVODE, a Stiff/Nonstiff ODE Solver in C. *Computers in Physics*, 10(2):138–143, 1996.
- [14] A. M. Collier, A. C. Hindmarsh, R. Serban, and C.S. Woodward. User Documentation for KINSOL v4.0.0-dev.2. Technical Report UCRL-SM-208116, LLNL, 2018.
- [15] T. A. Davis and P. N. Ekanathan. Algorithm 907: KLU, a direct sparse solver for circuit simulation problems. *ACM Trans. Math. Softw.*, 37(3), 2010.
- [16] R. S. Dembo, S. C. Eisenstat, and T. Steihaug. Inexact Newton Methods. *SIAM J. Numer. Anal.*, 19:400–408, 1982.

- [17] J. W. Demmel, J. R. Gilbert, and X. S. Li. An asynchronous parallel supernodal algorithm for sparse gaussian elimination. *SIAM J. Matrix Analysis and Applications*, 20(4):915–952, 1999.
- [18] J. E. Dennis and R. B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. SIAM, Philadelphia, 1996.
- [19] H. Fang and Y. Saad. Two classes of secant methods for nonlinear acceleration. *Numer. Linear Algebra Appl.*, 16:197–221, 2009.
- [20] R. W. Freund. A Transpose-Free Quasi-Minimal Residual Algorithm for Non-Hermitian Linear Systems. *SIAM J. Sci. Comp.*, 14:470–482, 1993.
- [21] M. R. Hestenes and E. Stiefel. Methods of Conjugate Gradients for Solving Linear Systems. *J. Research of the National Bureau of Standards*, 49(6):409–436, 1952.
- [22] K. L. Hiebert and L. F. Shampine. Implicitly Defined Output Points for Solutions of ODEs. Technical Report SAND80-0180, Sandia National Laboratories, February 1980.
- [23] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward. SUNDIALS, suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Math. Softw.*, (31):363–396, 2005.
- [24] A. C. Hindmarsh and R. Serban. User Documentation for CVODE v4.0.0-dev.2. Technical Report UCRL-SM-208108, LLNL, 2018.
- [25] A. C. Hindmarsh, R. Serban, and A. Collier. Example Programs for IDA v4.0.0-dev.2. Technical Report UCRL-SM-208113, LLNL, 2018.
- [26] A. C. Hindmarsh, R. Serban, and D. R. Reynolds. Example Programs for CVODE v4.0.0-dev.2. Technical report, LLNL, 2018. UCRL-SM-208110.
- [27] A. C. Hindmarsh and A. G. Taylor. PVODE and KINSOL: Parallel Software for Differential and Nonlinear Systems. Technical Report UCRL-ID-129739, LLNL, February 1998.
- [28] C. T. Kelley. *Iterative Methods for Solving Linear and Nonlinear Equations*. SIAM, Philadelphia, 1995.
- [29] X. S. Li. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Trans. Math. Softw.*, 31(3):302–325, September 2005.
- [30] P. A. Lott, H. F. Walker, C. S. Woodward, and U. M. Yang. An accelerated Picard method for nonlinear systems related to variably saturated flow. *Adv. Wat. Resour.*, 38:92–101, 2012.
- [31] Daniel R. Reynolds. Example Programs for ARKODE v3.0.0-dev.2. Technical report, Southern Methodist University, 2018.
- [32] Y. Saad. A flexible inner-outer preconditioned GMRES algorithm. *SIAM J. Sci. Comput.*, 14(2):461–469, 1993.
- [33] 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.
- [34] H. A. Van Der Vorst. Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems. *SIAM J. Sci. Stat. Comp.*, 13:631–644, 1992.
- [35] H. F. Walker and P. Ni. Anderson acceleration for fixed-point iterations. *SIAM Jour. Num. Anal.*, 49(4):1715–1735, 2011.

# Index

- BIG\_REAL, [26](#), [103](#)
- boolean type, [26](#)
- CONSTR\_VEC, [90](#)
- data types
  - Fortran, [79](#)
- eh\_data, [66](#)
- error messages, [39](#)
  - redirecting, [39](#)
  - user-defined handler, [41](#), [66](#)
- FIDA interface module
  - interface to the IDABBDPRE module, [93–94](#)
  - optional input and output, [89](#)
  - rootfinding, [92](#)
  - usage, [82–89](#)
  - user-callable functions, [80–81](#)
  - user-supplied functions, [81](#)
- FIDABANDSETJAC, [85](#)
- FIDABBDINIT, [93](#)
- FIDABBDOPT, [94](#)
- FIDABBREINIT, [94](#)
- FIDABJAC, [85](#)
- FIDACOMMFN, [94](#)
- FIDADENSESETJAC, [85](#)
- FIDADJAC, [84](#)
- FIDADLSINIT, [84](#)
- FIDAEW, [84](#)
- FIDAEWSET, [84](#)
- FIDAFREE, [89](#)
- FIDAGETDKY, [88](#)
- FIDAGETERRWEIGHTS, [89](#)
- FIDAGETESTLOCALERR, [92](#)
- FIDAGLOCFN, [94](#)
- FIDAJTIMES, [86](#), [95](#)
- FIDAJTSETUP, [86](#), [95](#)
- FIDALSINIT, [84](#)
- FIDALSSETJAC, [86](#)
- FIDALSSETPREC, [87](#)
- FIDAMALLOC, [84](#)
- FIDAMALLOC, [83](#)
- FIDAPSET, [87](#)
- FIDAPSOL, [87](#)
- FIDAREINIT, [88](#)
- FIDARESFUN, [82](#)
- FIDASETIIN, [89](#)
- FIDASETRIN, [89](#)
- FIDASETVIN, [89](#)
- FIDASOLVE, [88](#)
- FIDAPARSESETJAC, [86](#)
- FIDASPILSETJAC, [87](#)
- FIDASPILSETPREC, [88](#)
- FIDASPILSINIT, [84](#)
- FIDATOLREINIT, [89](#)
- FSUNBANDLINSOLINIT, [162](#)
- FSUNDENSELINSOLINIT, [160](#)
- FSUNFIXEDPOINTINIT, [222](#)
- FSUNKLUINIT, [170](#)
- FSUNKLUREINIT, [170](#)
- FSUNKLUSETORDERING, [171](#)
- FSUNLAPACKBANDINIT, [167](#)
- FSUNLAPACKDENSEINIT, [164](#)
- FSUNMASSBANDLINSOLINIT, [163](#)
- FSUNMASSDENSELINSOLINIT, [161](#)
- FSUNMASSKLUNIT, [170](#)
- FSUNMASSKLUREINIT, [171](#)
- FSUNMASSKLUSEORDERING, [171](#)
- FSUNMASSLAPACKBANDINIT, [167](#)
- FSUNMASSLAPACKDENSEINIT, [165](#)
- FSUNMASSPCGINIT, [202](#)
- FSUNMASSPCGSETMAXL, [203](#)
- FSUNMASSPCGSETPRECTYPE, [202](#)
- FSUNMASSSPBCGSINIT, [191](#)
- FSUNMASSSPBCGSSETMAXL, [192](#)
- FSUNMASSSPBCGSSETPRECTYPE, [192](#)
- FSUNMASSSPFGMRINIT, [186](#)
- FSUNMASSSPFGMRSETGSTYPE, [186](#)
- FSUNMASSSPFGMRSETMAXRS, [187](#)
- FSUNMASSSPFGMRSETPRECTYPE, [187](#)
- FSUNMASSSPGMRINIT, [180](#)
- FSUNMASSSPGMRSETGSTYPE, [180](#)
- FSUNMASSSPGMRSETMAXRS, [181](#)
- FSUNMASSSPGMRSETPRECTYPE, [181](#)
- FSUNMASSSPTFQMRINIT, [196](#)
- FSUNMASSSPTFQMRSETMAXL, [197](#)

- FSUNMASSSPTFQMRSETPRECTYPE, 197
- FSUNMASSSUPERLUMTINIT, 175
- FSUNMASSUPERLUMTSETORDERING, 176
- FSUNNEWTONINIT, 218
- FSUNPCGINIT, 201
- FSUNPCGSETMAXL, 202
- FSUNPCGSETPRECTYPE, 202
- FSUNSPBCGSINIT, 191
- FSUNSPBCGSSETMAXL, 192
- FSUNSPBCGSSETPRECTYPE, 192
- FSUNSPFGMRINIT, 185
- FSUNSPFGMRSETGSTYPE, 186
- FSUNSPFGMRSETMAXRS, 187
- FSUNSPFGMRSETPRECTYPE, 186
- FSUNSPGMRINIT, 179
- FSUNSPGMRSETGSTYPE, 180
- FSUNSPGMRSETMAXRS, 181
- FSUNSPGMRSETPRECTYPE, 180
- FSUNSPTFQMRINIT, 196
- FSUNSPTFQMRSETMAXL, 197
- FSUNSPTFQMRSETPRECTYPE, 196
- FSUNSUPERLUMTINIT, 175
- FSUNSUPERLUMTSETORDERING, 175
  
- getDevData(N\_Vector v), 121, 123
- getGlobalSize(N\_Vector v), 121, 124
- getHostData(N\_Vector v), 121, 123
- getMPIComm(N\_Vector v), 121, 124
- getSize(N\_Vector v), 121, 123
  
- half-bandwidths, 75
- header files, 27, 74
  
- ID\_VEC, 90
- IDA
  - motivation for writing in C, 1
  - package structure, 19
- IDA linear solver interface
  - IDALS, 35
- IDA linear solver interfaces, 19
- IDA linear solvers
  - header files, 27
  - implementation details, 20
  - NVECTOR compatibility, 25
  - selecting one, 35
- ida/ida.h, 27
- ida/ida\_ls.h, 27
- IDA\_BAD\_DKY, 51
- IDA\_BAD\_EWT, 37
- IDA\_BAD\_K, 51
- IDA\_BAD\_T, 51
- IDA\_CONSTR\_FAIL, 37, 39
- IDA\_CONV\_FAIL, 37, 39
- IDA\_ERR\_FAIL, 39
- IDA\_FIRST\_RES\_FAIL, 37
- IDA\_ILL\_INPUT, 32, 33, 36–38, 42–45, 48–50, 59, 65
- IDA\_LINESEARCH\_FAIL, 37
- IDA\_LINIT\_FAIL, 37, 39
- IDA\_LSETUP\_FAIL, 37, 39
- IDA\_LSOLVE\_FAIL, 37, 39
- IDA\_MEM\_FAIL, 32, 43, 58, 59
- IDA\_MEM\_NULL, 32, 33, 36–38, 41–45, 48–51, 54–60, 65
- IDA\_NO\_MALLOC, 33, 37, 65
- IDA\_NO\_RECOVERY, 37
- IDA\_NORMAL, 38
- IDA\_ONE\_STEP, 38
- IDA\_REP\_RES\_ERR, 39
- IDA\_RES\_FAIL, 37, 39
- IDA\_ROOT\_RETURN, 38
- IDA\_RTFUNC\_FAIL, 39, 67
- IDA\_SUCCESS, 32, 33, 36–38, 41–45, 48–51, 60, 65
- IDA\_TOO\_MUCH\_ACC, 39
- IDA\_TOO\_MUCH\_WORK, 39
- IDA\_TSTOP\_RETURN, 38
- IDA\_WARNING, 66
- IDA\_Y\_INIT, 36
- IDA\_YA\_YDP\_INIT, 36
- IDABBDPRE preconditioner
  - description, 72–73
  - optional output, 77
  - usage, 74–75
  - user-callable functions, 75–77
  - user-supplied functions, 73–74
- IDABBDPrecGetNumGfnEvals, 77
- IDABBDPrecGetWorkSpace, 77
- IDABBDPrecInit, 75
- IDABBDPrecReInit, 76
- IDACalcIC, 36
- IDACreate, 32
- IDADlsGetLastFlag, 64
- IDADlsGetNumJacEvals, 61
- IDADlsGetNumRhsEvals, 61
- IDADlsGetReturnFlagName, 64
- IDADlsGetWorkspace, 61
- IDADlsJacFn, 69
- IDADlsSetJacFn, 46
- IDADlsSetLinearSolver, 35
- IDAErrHandlerFn, 66
- IDAEwtFn, 66
- IDAFree, 31, 32
- IDAGetActualInitStep, 56
- IDAGetConsistentIC, 59
- IDAGetCurrentOrder, 55
- IDAGetCurrentStep, 56
- IDAGetCurrentTime, 56
- IDAGetDky, 51
- IDAGetErrWeights, 57



- IDAGetEstLocalErrors, 57
- IDAGetIntegratorStats, 57
- IDAGetLastLinFlag, 63
- IDAGetLastOrder, 55
- IDAGetLastStep, 56
- IDAGetLinReturnFlagName, 64
- IDAGetLinWorkSpace, 60
- IDAGetNonlinSolvStats, 58
- IDAGetNumBacktrackOps, 59
- IDAGetNumErrTestFails, 55
- IDAGetNumGEvals, 60
- IDAGetNumJacEvals, 61
- IDAGetNumJtimesEvals, 63
- IDAGetNumJTSetupEvals, 63
- IDAGetNumLinConvFails, 62
- IDAGetNumLinIters, 61
- IDAGetNumLinResEvals, 61
- IDAGetNumLinSolvSetups, 55
- IDAGetNumNonlinSolvConvFails, 58
- IDAGetNumNonlinSolvIters, 58
- IDAGetNumPrecEvals, 62
- IDAGetNumPrecSolves, 62
- IDAGetNumResEvals, 54
- IDAGetNumSteps, 54
- IDAGetReturnFlagName, 59
- IDAGetRootInfo, 60
- IDAGetTolScaleFactor, 57
- IDAGetWorkSpace, 53
- IDAInit, 32, 64
- IDALS linear solver interface
  - convergence test, 47
  - Jacobian approximation used by, 45, 46
  - memory requirements, 60
  - optional input, 45–48
  - optional output, 60–64
  - preconditioner setup function, 47, 71
  - preconditioner solve function, 47, 71
- IDALS\_ILL\_INPUT, 35, 48, 76
- IDALS\_LMEM\_NULL, 45–48, 60–63, 76, 77
- IDALS\_MEM\_FAIL, 35, 76
- IDALS\_MEM\_NULL, 35, 45–47, 60–63
- IDALS\_PMEM\_NULL, 77
- IDALS\_SUCCESS, 35, 45–47, 60–63
- IDALS\_SUNLS\_FAIL, 35, 46, 47
- IDALsJacFn, 67
- IDALsJacTimesSetupFn, 70
- IDALsJacTimesVecFn, 69
- IDALsPrecSetupFn, 71
- IDALsPrecSolveFn, 71
- IDAReInit, 64, 65
- IDAResFn, 32, 65
- IDARootFn, 67
- IDARootInit, 37
- IDASetConstraints, 45
- IDASetEpsLin, 47
- IDASetErrFile, 39
- IDASetErrHandlerFn, 41
- IDASetId, 44
- IDASetInitStep, 42
- IDASetJacFn, 45
- IDASetJacTimes, 46
- IDASetLinearSolver, 30, 35, 67, 148, 205
- IDASetLineSearchOffIC, 49
- IDASetMaxBacksIC, 49
- IDASetMaxConvFails, 43
- IDASetMaxErrTestFails, 43
- IDASetMaxNonlinIters, 43
- IDASetMaxNumItersIC, 49
- IDASetMaxNumJacsIC, 49
- IDASetMaxNumSteps, 42
- IDASetMaxNumStepsIC, 48
- IDASetMaxOrd, 41
- IDASetMaxStep, 42
- IDASetNoInactiveRootWarn, 50
- IDASetNonlinConvCoef, 44
- IDASetNonlinConvCoefIC, 48
- IDASetNonLinearSolver, 36
- IDASetNonlinearSolver, 30, 36
- IDASetPreconditioner, 47
- IDASetRootDirection, 50
- IDASetStepToleranceIC, 50
- IDASetStopTime, 43
- IDASetSuppressAlg, 44
- IDASetUserData, 41
- IDASolve, 30, 38
- IDASpilsGetLastFlag, 64
- IDASpilsGetNumConvFails, 62
- IDASpilsGetNumJtimesEvals, 63
- IDASpilsGetNumJTSetupEvals, 63
- IDASpilsGetNumLinIters, 62
- IDASpilsGetNumPrecEvals, 62
- IDASpilsGetNumPrecSolves, 63
- IDASpilsGetNumRhsEvals, 61
- IDASpilsGetReturnFlagName, 64
- IDASpilsGetWorkspace, 61
- IDASpilsJacTimesSetupFn, 70
- IDASpilsJacTimesVecFn, 70
- IDASpilsPrecSetupFn, 72
- IDASpilsPrecSolveFn, 71
- IDASpilsSetEpsLin, 48
- IDASpilsSetJacTimes, 46
- IDASpilsSetLinearSolver, 35
- IDASpilsSetPreconditioner, 47
- IDASStolerances, 33
- IDASVtolerances, 33
- IDAWFtolerances, 33
- INIT\_STEP, 90
- IOUT, 89, 91

itask, 38

Jacobian approximation function

band

use in FIDA, 85

dense

use in FIDA, 84

difference quotient, 45

Jacobian times vector

difference quotient, 46

use in FIDA, 86

user-supplied, 46, 69–70

Jacobian-vector setup

user-supplied, 70–71

sparse

use in FIDA, 85

user-supplied, 45, 67–69

LS\_OFF\_IC, 90

MAX\_CONVFAIL, 90

MAX\_ERRFAIL, 90

MAX\_NITERS, 90

MAX\_NITERS\_IC, 90

MAX\_NJE\_IC, 90

MAX\_NSTEPS, 90

MAX\_NSTEPS\_IC, 90

MAX\_ORD, 90

MAX\_STEP, 90

maxord, 64

memory requirements

IDA solver, 54

IDABBDPRE preconditioner, 77

IDALS linear solver interface, 60

N\_VCloneVectorArray, 98

N\_VCloneVectorArray\_OpenMP, 113

N\_VCloneVectorArray\_Parallel, 111

N\_VCloneVectorArray\_ParHyp, 117

N\_VCloneVectorArray\_Petsc, 119

N\_VCloneVectorArray\_Pthreads, 116

N\_VCloneVectorArray\_Serial, 108

N\_VCloneVectorArrayEmpty, 98

N\_VCloneVectorArrayEmpty\_OpenMP, 113

N\_VCloneVectorArrayEmpty\_Parallel, 111

N\_VCloneVectorArrayEmpty\_ParHyp, 118

N\_VCloneVectorArrayEmpty\_Petsc, 119

N\_VCloneVectorArrayEmpty\_Pthreads, 116

N\_VCloneVectorArrayEmpty\_Serial, 108

N\_VCopyFromDevice\_Cuda, 122

N\_VCopyFromDevice\_Raja, 125

N\_VCopyToDevice\_Cuda, 122

N\_VCopyToDevice\_Raja, 125

N\_VDestroyVectorArray, 98

N\_VDestroyVectorArray\_OpenMP, 113

N\_VDestroyVectorArray\_Parallel, 111

N\_VDestroyVectorArray\_ParHyp, 118

N\_VDestroyVectorArray\_Petsc, 119

N\_VDestroyVectorArray\_Pthreads, 116

N\_VDestroyVectorArray\_Serial, 108

N\_Vector, 27, 97

N\_VGetDeviceArrayPointer\_Cuda, 122

N\_VGetDeviceArrayPointer\_Raja, 125

N\_VGetHostArrayPointer\_Cuda, 122

N\_VGetHostArrayPointer\_Raja, 125

N\_VGetLength\_Cuda, 122

N\_VGetLength\_OpenMP, 114

N\_VGetLength\_Parallel, 111

N\_VGetLength\_Pthreads, 116

N\_VGetLength\_Raja, 125

N\_VGetLength\_Serial, 108

N\_VGetLocalLength\_Parallel, 111

N\_VGetVector\_ParHyp, 117

N\_VGetVector\_Petsc, 119

N\_VMake\_Cuda, 122

N\_VMake\_OpenMP, 113

N\_VMake\_Parallel, 110

N\_VMake\_ParHyp, 117

N\_VMake\_Petsc, 119

N\_VMake\_Pthreads, 116

N\_VMake\_Raja, 124

N\_VMake\_Serial, 108

N\_VNew\_Cuda, 121

N\_VNew\_OpenMP, 113

N\_VNew\_Parallel, 110

N\_VNew\_Pthreads, 115

N\_VNew\_Raja, 124

N\_VNew\_SensWrapper, 215

N\_VNew\_Serial, 108

N\_VNewEmpty\_Cuda, 122

N\_VNewEmpty\_OpenMP, 113

N\_VNewEmpty\_Parallel, 110

N\_VNewEmpty\_ParHyp, 117

N\_VNewEmpty\_Petsc, 119

N\_VNewEmpty\_Pthreads, 115

N\_VNewEmpty\_Raja, 124

N\_VNewEmpty\_SensWrapper, 215

N\_VNewEmpty\_Serial, 108

N\_VPrint\_Cuda, 122

N\_VPrint\_OpenMP, 114

N\_VPrint\_Parallel, 111

N\_VPrint\_ParHyp, 118

N\_VPrint\_Petsc, 119

N\_VPrint\_Pthreads, 116

N\_VPrint\_Raja, 125

N\_VPrint\_Serial, 108

N\_VPrintFile\_Cuda, 122

N\_VPrintFile\_OpenMP, 114

N\_VPrintFile\_Parallel, 111

- N\_VPrintFile.ParHyp, 118
- N\_VPrintFile.Petsc, 119
- N\_VPrintFile.Pthreads, 116
- N\_VPrintFile.Raja, 125
- N\_VPrintFile.Serial, 108
- NLCONV\_COEF, 90
- NLCONV\_COEF\_IC, 90
- NV\_COMM\_P, 110
- NV\_CONTENT\_OMP, 112
- NV\_CONTENT\_P, 109
- NV\_CONTENT\_PT, 115
- NV\_CONTENT\_S, 107
- NV\_DATA\_OMP, 112
- NV\_DATA\_P, 109
- NV\_DATA\_PT, 115
- NV\_DATA\_S, 107
- NV\_GLOBLENGTH\_P, 109
- NV\_Ith\_OMP, 113
- NV\_Ith\_P, 110
- NV\_Ith\_PT, 115
- NV\_Ith\_S, 107
- NV\_LENGTH\_OMP, 112
- NV\_LENGTH\_PT, 115
- NV\_LENGTH\_S, 107
- NV\_LOCLENGTH\_P, 109
- NV\_NUM\_THREADS\_OMP, 112
- NV\_NUM\_THREADS\_PT, 115
- NV\_OWN\_DATA\_OMP, 112
- NV\_OWN\_DATA\_P, 109
- NV\_OWN\_DATA\_PT, 115
- NV\_OWN\_DATA\_S, 107
- NVECTOR module, 97
  - optional input
    - generic linear solver interface, 45–48
    - initial condition calculation, 48–50
    - iterative linear solver, 46–48
    - matrix-based linear solver, 45–46
    - matrix-free linear solver, 46
    - rootfinding, 50–51
    - solver, 39–45
  - optional output
    - band-block-diagonal preconditioner, 77
    - generic linear solver interface, 60–64
    - initial condition calculation, 59
    - interpolated solution, 51
    - solver, 53–59
    - version, 53
- portability, 26
  - Fortran, 79
- Preconditioner setup routine
  - use in FIDA, 87
- Preconditioner solve routine
  - use in FIDA, 87
- preconditioning
  - advice on, 17, 20
  - band-block diagonal, 72
  - setup and solve phases, 20
  - user-supplied, 46–47, 71
- RCONST, 26
- realtype, 26
- reinitialization, 64
- residual function, 65
- Rootfinding, 17, 30, 37, 92
- ROUT, 89, 91
- SM\_COLS\_B, 139
- SM\_COLS\_D, 135
- SM\_COLUMN\_B, 68, 139
- SM\_COLUMN\_D, 68, 135
- SM\_COLUMN\_ELEMENT\_B, 68, 139
- SM\_COLUMNS\_B, 139
- SM\_COLUMNS\_D, 134
- SM\_COLUMNS\_S, 145
- SM\_CONTENT\_B, 139
- SM\_CONTENT\_D, 134
- SM\_CONTENT\_S, 143
- SM\_DATA\_B, 139
- SM\_DATA\_D, 135
- SM\_DATA\_S, 145
- SM\_ELEMENT\_B, 68, 139
- SM\_ELEMENT\_D, 68, 135
- SM\_INDEXPTRS\_S, 145
- SM\_INDEXVALS\_S, 145
- SM\_LBAND\_B, 139
- SM\_LDATA\_B, 139
- SM\_LDATA\_D, 134
- SM\_LDIM\_B, 139
- SM\_NNZ\_S, 69, 145
- SM\_NP\_S, 145
- SM\_ROWS\_B, 139
- SM\_ROWS\_D, 134
- SM\_ROWS\_S, 145
- SM\_SPARSETYPE\_S, 145
- SM\_SUBAND\_B, 139
- SM\_UBAND\_B, 139
- SMALL\_REAL, 26
- step size bounds, 42
- STEP\_TOL\_IC, 90
- STOP\_TIME, 90
- SUNBandLinearSolver, 162
- SUNBandMatrix, 29, 140
- SUNBandMatrix\_Cols, 141
- SUNBandMatrix\_Column, 141
- SUNBandMatrix\_Columns, 140
- SUNBandMatrix\_Data, 141

- SUNBandMatrix.LDim, 141
- SUNBandMatrix.LowerBandwidth, 140
- SUNBandMatrix.Print, 140
- SUNBandMatrix.Rows, 140
- SUNBandMatrix.StoredUpperBandwidth, 140
- SUNBandMatrix.UpperBandwidth, 140
- SUNDenseLinearSolver, 160
- SUNDenseMatrix, 29, 135
- SUNDenseMatrix.Cols, 136
- SUNDenseMatrix.Column, 136
- SUNDenseMatrix.Columns, 136
- SUNDenseMatrix.Data, 136
- SUNDenseMatrix.LData, 136
- SUNDenseMatrix.Print, 135
- SUNDenseMatrix.Rows, 136
- sundials/sundials\_linearsolver.h, 151
- sundials\_nonlinearsolver.h, 27
- sundials\_nvector.h, 27
- sundials\_types.h, 26, 27
- SUNDIALSGetVersion, 53
- SUNDIALSGetVersionNumber, 53
- sunindextype, 26
- SUNKLU, 170
- SUNKLUREInit, 170
- SUNKLUSetOrdering, 170
- SUNLapackBand, 166
- SUNLapackDense, 164
- SUNLineaerSolver, 151
- SUNLinearSolver, 158
- SUNLinearSolver module, 151
- sunlinsol/sunlinsol\_band.h, 27
- sunlinsol/sunlinsol\_dense.h, 27
- sunlinsol/sunlinsol\_klu.h, 27
- sunlinsol/sunlinsol\_lapackband.h, 27
- sunlinsol/sunlinsol\_lapackdense.h, 27
- sunlinsol/sunlinsol\_pcg.h, 28
- sunlinsol/sunlinsol\_spgmgs.h, 28
- sunlinsol/sunlinsol\_spgfmr.h, 27
- sunlinsol/sunlinsol\_spgmr.h, 27
- sunlinsol/sunlinsol\_sptfqmr.h, 28
- sunlinsol/sunlinsol\_superlumt.h, 27
- SUNLinSol\_Band, 35, 162
- SUNLinSol\_Dense, 35, 160
- SUNLinSol\_KLU, 35, 168
- SUNLinSol\_KLUREInit, 169
- SUNLinSol\_KLUSetOrdering, 170
- SUNLinSol\_LapackBand, 35, 166
- SUNLinSol\_LapackDense, 35, 164
- SUNLinSol\_PCG, 35, 200–202
- SUNLinSol\_PCGSetMaxl, 201
- SUNLinSol\_PCGSetPrecType, 201
- SUNLinSol\_SPBCGS, 35, 190, 191
- SUNLinSol\_SPBCGSSetMaxl, 190, 191
- SUNLinSol\_SPBCGSSetPrecType, 190, 191
- SUNLinSol\_SPGFMR, 35, 184–186
- SUNLinSol\_SPGFMRSetGSType, 185
- SUNLinSol\_SPGFMRSetMaxRestarts, 185
- SUNLinSol\_SPGFMRSetPrecType, 184, 185
- SUNLinSol\_SPGMR, 35, 178–180
- SUNLinSol\_SPGMRSetGSType, 179
- SUNLinSol\_SPGMRSetMaxRestarts, 179
- SUNLinSol\_SPGMRSetPrecType, 178, 179
- SUNLinSol\_SPTFQMR, 35, 195, 196
- SUNLinSol\_SPTFQMRSetMaxl, 195, 196
- SUNLinSol\_SPTFQMRSetPrecType, 195, 196
- SUNLinSol\_SuperLUMT, 35, 174
- SUNLinSol\_SuperLUMTSetOrdering, 174, 175
- SUNLinSolFree, 31, 152, 153
- SUNLinSolGetType, 152, 205
- SUNLinSolInitialize, 152
- SUNLinSolLastFlag, 155
- SUNLinSolNumIters, 155
- SUNLinSolResNorm, 155
- SUNLinSolSetATimes, 154
- SUNLinSolSetPreconditioner, 154
- SUNLinSolSetScalingVectors, 154
- SUNLinSolSetup, 152, 153
- SUNLinSolSolve, 152, 153
- SUNLinSolSpace, 155
- SUNMatDestroy, 31
- SUNMatrix, 131
- SUNMatrix module, 131
- SUNNonlinearSolver, 27, 207
- SUNNonlinearSolver module, 207
- SUNNONLINEARSOLVER\_DIRECT, 152
- SUNNONLINEARSOLVER\_FIXEDPOINT, 208
- SUNNONLINEARSOLVER\_ITERATIVE, 152
- SUNNONLINEARSOLVER\_ROOTFIND, 208
- SUNNonlinSol\_FixedPoint, 220, 221
- SUNNonlinSol\_FixedPointSens, 220
- SUNNonlinSol\_Newton, 217
- SUNNonlinSol\_NewtonSens, 217
- SUNNonlinSolFree, 31, 209
- SUNNonlinSolGetCurIter, 211
- SUNNonlinSolGetNumIters, 211
- SUNNonlinSolGetSysFn\_FixedPoint, 220
- SUNNonlinSolGetSysFn\_Newton, 217
- SUNNonlinSolGetType, 208
- SUNNonlinSolInitialize, 208
- SUNNonlinSolLSetupFn, 209
- SUNNonlinSolSetConvTestFn, 210
- SUNNonlinSolSetLSolveFn, 210
- SUNNonlinSolSetMaxIters, 210
- SUNNonlinSolSetSysFn, 209
- SUNNonlinSolSetup, 208
- SUNNonlinSolSolve, 208
- SUNPCG, 201
- SUNPCGSetMaxl, 201

SUNPCGSetPrecType, 201  
SUNSparseFromBandMatrix, 146  
SUNSparseFromDenseMatrix, 145  
SUNSparseMatrix, 29, 145  
SUNSparseMatrix\_Columns, 146  
SUNSparseMatrix\_Data, 147  
SUNSparseMatrix\_IndexPointers, 147  
SUNSparseMatrix\_IndexValues, 147  
SUNSparseMatrix\_NNZ, 69, 147  
SUNSparseMatrix\_NP, 147  
SUNSparseMatrix\_Print, 146  
SUNSparseMatrix\_Realloc, 146  
SUNSparseMatrix\_Reallocate, 146  
SUNSparseMatrix\_Rows, 146  
SUNSparseMatrix\_SparseType, 147  
SUNSPBCGS, 191  
SUNSPBCGSsetMaxl, 191  
SUNSPBCGSSetPrecType, 191  
SUNSPFGMR, 185  
SUNSPFGMRSetGSType, 185  
SUNSPFGMRsetMaxRestarts, 185  
SUNSPFGMRSetPrecType, 185  
SUNSPGMR, 179  
SUNSPGMRSetGSType, 179  
SUNSPGMRsetMaxRestarts, 179  
SUNSPGMRSetPrecType, 179  
SUNSPTFQMR, 196  
SUNSPTFQMRsetMaxl, 196  
SUNSPTFQMRSetPrecType, 196  
SUNSuperLUMT, 174  
SUNSuperLUMTSetOrdering, 174  
SUPPRESS\_ALG, 90

tolerances, 14, 33, 34, 66

UNIT\_ROUNDOFF, 26

User main program

    FIDA usage, 82

    FIDABBD usage, 93

    IDA usage, 28

    IDABBDPRE usage, 74

user\_data, 41, 66, 67, 74

weighted root-mean-square norm, 14

