User Documentation for IDA v5.0.0-dev.0
(SUNDIALS v5.0.0-dev.0)

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

March 28, 2019
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.

Approved for public release; further dissemination unlimited
# Contents

## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ix</td>
<td></td>
</tr>
</tbody>
</table>

## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xi</td>
<td></td>
</tr>
</tbody>
</table>

## 1 Introduction

1.1 Changes from previous versions 
1.2 Reading this User Guide 
1.3 SUNDIALS Release License
  1.3.1 BSD 3-Clause License
  1.3.2 Additional Notice
  1.3.3 SUNDIALS Release Numbers

## 2 Mathematical Considerations

2.1 IVP solution 
2.2 Preconditioning 
2.3 Rootfinding

## 3 Code Organization

3.1 SUNDIALS organization 
3.2 IDA organization

## 4 Using IDA for C Applications

4.1 Access to library and header files 
4.2 Data types
  4.2.1 Floating point types
  4.2.2 Integer types used for vector and matrix indices
4.3 Header files 
4.4 A skeleton of the user’s main program 
4.5 User-callable functions
  4.5.1 IDA initialization and deallocation functions
  4.5.2 IDA tolerance specification functions
  4.5.3 Linear solver interface functions
  4.5.4 Nonlinear solver interface function
  4.5.5 Initial condition calculation function
  4.5.6 Rootfinding initialization function
  4.5.7 IDA solver function
  4.5.8 Optional input functions
    4.5.8.1 Main solver optional input functions
    4.5.8.2 Linear solver interface optional input functions
    4.5.8.3 Initial condition calculation optional input functions
    4.5.8.4 Rootfinding optional input functions
  4.5.9 Interpolated output function 
4.5.10 Optional output functions
6.10.1 NVECTOR_OPENMPDEV accessor macros .............................................. 146
6.10.2 NVECTOR_OPENMPDEV functions ....................................................... 147
6.11 The NVECTOR_TRILINOS implementation ............................................... 151
6.11.1 NVECTOR_TRILINOS functions ............................................................ 151
6.12 The NVECTOR_MANYVECTOR implementation ........................................... 152
6.12.1 NVECTOR_MANYVECTOR structure ......................................................... 153
6.12.2 NVECTOR_MANYVECTOR functions ......................................................... 153
6.13 NVECTOR Examples .................................................................................. 156

7 Description of the SUNMatrix module ............................................................ 161
7.1 The SUNMatrix API ..................................................................................... 161
7.1.1 SUNMatrix functions ............................................................................... 162
7.1.2 SUNMatrix error codes ............................................................................. 163
7.2 Compatibility of SUNMatrix modules ......................................................... 164
7.3 Implementing a custom SUNMatrix ............................................................. 164
7.4 SUNMatrix functions used by IDA ............................................................. 165
7.5 The SUNMatrix_Dense implementation ....................................................... 165
7.5.1 SUNMatrix_Dense accessor macros ......................................................... 166
7.5.2 SUNMatrix_Dense functions ................................................................. 167
7.5.3 SUNMatrix_Dense Fortran interfaces ....................................................... 168
7.6 The SUNMatrix_Band implementation ....................................................... 169
7.6.1 SUNMatrix_Band accessor macros ......................................................... 170
7.6.2 SUNMatrix_Band functions ................................................................. 172
7.6.3 SUNMatrix_Band Fortran interfaces ....................................................... 174
7.7 The SUNMatrix_Sparse implementation ..................................................... 175
7.7.1 SUNMatrix_Sparse accessor macros ....................................................... 176
7.7.2 SUNMatrix_Sparse functions ................................................................. 178
7.7.3 SUNMatrix_Sparse Fortran interfaces ..................................................... 181
7.8 The SUNMatrix_SLUNRloc implementation .............................................. 182
7.8.1 SUNMatrix_SLUNRloc functions ............................................................. 182

8 Description of the SUNLinearSolver module ............................................... 185
8.1 The SUNLinearSolver API ......................................................................... 186
8.1.1 SUNLinearSolver core functions ............................................................ 186
8.1.2 SUNLinearSolver set functions .............................................................. 188
8.1.3 SUNLinearSolver get functions ............................................................. 189
8.1.4 Functions provided by SUNDIALS packages .......................................... 190
8.1.5 SUNLinearSolver return codes .............................................................. 191
8.1.6 The generic SUNLinearSolver module ................................................. 192
8.2 Compatibility of SUNLinearSolver modules ............................................. 193
8.3 Implementing a custom SUNLinearSolver module ..................................... 194
8.3.1 Intended use cases ................................................................................. 194
8.4 IDA SUNLinearSolver interface ............................................................... 195
8.4.1 Lagged matrix information .................................................................... 196
8.4.2 Iterative linear solver tolerance ............................................................. 197
8.5 The SUNLinearSolver_Dense implementation ......................................... 197
8.5.1 SUNLinearSolver_Dense description ..................................................... 197
8.5.2 SUNLinearSolver_Dense functions ....................................................... 198
8.5.3 SUNLinearSolver_Dense Fortran interfaces .......................................... 198
8.5.4 SUNLinearSolver_Dense content .......................................................... 199
8.6 The SUNLinearSolver_Band implementation .......................................... 200
8.6.1 SUNLinearSolver_Band description ...................................................... 200
8.6.2 SUNLinearSolver_Band functions ......................................................... 200
8.6.3 SUNLinearSolver_Band Fortran interfaces ............................................ 201
8.6.4 SUNLinearSolver_Band content ........................................... 202
8.7 The SUNLinearSolver_LapackDense implementation ...................... 202
  8.7.1 SUNLinearSolver_LapackDense description ......................... 203
  8.7.2 SUNLinearSolver_LapackDense functions .......................... 203
  8.7.3 SUNLinearSolver_LapackDense Fortran interfaces ................. 204
  8.7.4 SUNLinearSolver_LapackDense content ........................... 204
8.8 The SUNLinearSolver_LapackBand implementation ........................ 205
  8.8.1 SUNLinearSolver_LapackBand description ......................... 205
  8.8.2 SUNLinearSolver_LapackBand functions .......................... 205
  8.8.3 SUNLinearSolver_LapackBand Fortran interfaces ................. 206
  8.8.4 SUNLinearSolver_LapackBand content ........................... 207
8.9 The SUNLinearSolver_KLU implementation .................................. 207
  8.9.1 SUNLinearSolver_KLU description ................................ 207
  8.9.2 SUNLinearSolver_KLU functions ................................ 208
  8.9.3 SUNLinearSolver_KLU Fortran interfaces ......................... 210
  8.9.4 SUNLinearSolver_KLU content .................................. 212
8.10 The SUNLinearSolver_SuperLUDIST implementation ..................... 213
  8.10.1 SUNLinearSolver_SuperLUDIST description ...................... 213
  8.10.2 SUNLinearSolver_SuperLUDIST functions ......................... 213
  8.10.3 SUNLinearSolver_SuperLUDIST content ......................... 216
8.11 The SUNLinearSolver_SuperLUMT implementation ....................... 217
  8.11.1 SUNLinearSolver_SuperLUMT description ......................... 217
  8.11.2 SUNLinearSolver_SuperLUMT functions .......................... 217
  8.11.3 SUNLinearSolver_SuperLUMT Fortran interfaces ................. 219
  8.11.4 SUNLinearSolver_SuperLUMT content ........................... 220
8.12 The SUNLinearSolver_SPGMR implementation ................................ 221
  8.12.1 SUNLinearSolver_SPGMR description ................................ 221
  8.12.2 SUNLinearSolver_SPGMR functions ................................ 221
  8.12.3 SUNLinearSolver_SPGMR Fortran interfaces ...................... 224
  8.12.4 SUNLinearSolver_SPGMR content ................................ 226
8.13 The SUNLinearSolver_SPFGMR implementation ................................ 227
  8.13.1 SUNLinearSolver_SPFGMR description ................................ 228
  8.13.2 SUNLinearSolver_SPFGMR functions ................................ 228
  8.13.3 SUNLinearSolver_SPFGMR Fortran interfaces ...................... 230
  8.13.4 SUNLinearSolver_SPFGMR content ................................ 233
8.14 The SUNLinearSolver_SPBCGS implementation ................................ 234
  8.14.1 SUNLinearSolver_SPBCGS description ................................ 234
  8.14.2 SUNLinearSolver_SPBCGS functions ................................ 234
  8.14.3 SUNLinearSolver_SPBCGS Fortran interfaces ...................... 236
  8.14.4 SUNLinearSolver_SPBCGS content ................................ 238
8.15 The SUNLinearSolver_SPTFQMR implementation ................................ 239
  8.15.1 SUNLinearSolver_SPTFQMR description ................................ 239
  8.15.2 SUNLinearSolver_SPTFQMR functions ................................ 240
  8.15.3 SUNLinearSolver_SPTFQMR Fortran interfaces ...................... 241
  8.15.4 SUNLinearSolver_SPTFQMR content ................................ 244
8.16 The SUNLinearSolver_PCG implementation ................................ 245
  8.16.1 SUNLinearSolver_PCG description ................................ 245
  8.16.2 SUNLinearSolver_PCG functions ................................ 246
  8.16.3 SUNLinearSolver_PCG Fortran interfaces ......................... 248
  8.16.4 SUNLinearSolver_PCG content ................................ 250
8.17 SUNLinearSolver Examples ................................................ 250
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 ............................................................ 53

5.1 Keys for setting FIDA optional inputs .............................................................. 90
5.2 Description of the FIDA optional output arrays IOUT and ROUT ...................... 91

6.1 Vector Identifications associated with vector kernels supplied with SUNDIALS. .... 100
6.2 Description of the NVECTOR operations .......................................................... 100
6.3 Description of the NVECTOR fused operations ............................................... 103
6.4 Description of the NVECTOR vector array operations ..................................... 104
6.5 Description of the NVECTOR local reduction operations ............................... 107
6.6 List of vector functions usage by IDA code modules ...................................... 110

7.1 Identifiers associated with matrix kernels supplied with SUNDIALS. ................. 162
7.2 Description of the SUNMatrix operations ....................................................... 162
7.3 Description of the SUNMatrix error codes ..................................................... 164
7.4 SUNDIALS matrix interfaces and vector implementations that can be used for each. .. 164
7.5 List of matrix functions usage by IDA code modules ...................................... 165

8.1 Description of the SUNLinearSolver error codes ........................................... 192
8.2 SUNDIALS matrix-based linear solvers and matrix implementations that can be used for each. ................................................................. 193
8.3 List of linear solver function usage in the IDALS interface ............................. 196

9.1 Description of the SUNNonlinearSolver return codes ................................. 259

A.1 SUNDIALS libraries and header files .............................................................. 287
## 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 ................................. 170  
7.2 Diagram of the storage for a compressed-sparse-column matrix ....................... 177  

A.1 Initial ccmake configuration screen ............................................................... 273  
A.2 Changing the \texttt{instdir} ............................................................................ 274
Chapter 1

Introduction

IDA is part of a software family called SUNDIALS: SUite of Nonlinear and Differential/Algebraic equation Solvers [25]. 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 [8, 9], 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 [26, 14] and PVODE [12, 13], and also the nonlinear system solver KINSOL [15].

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) [37], FGMRES (Flexible Generalized Minimum RESidual) [36], Bi-CGSTab (Bi-Conjugate Gradient Stabilized) [38], TFQMR (Transpose-Free Quasi-Minimal Residual) [21], and PCG (Preconditioned Conjugate Gradient) [23] 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-CGFSstab 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 v5.0.0-dev.0

An additional NVector implementation, NVector_ManyVector, was created to support flexible partitioning of solution data among different processing elements (e.g., CPU + GPU) or for multiphysics problems that couple distinct MPI-based simulations together (see Section 6.12 for more details). This implementation is accompanied by additions to user documentation and Sundials examples.

Eleven new optional vector operations have been added to the NVector API to support the new NVector_ManyVector implementation (see Chapter 6 for more details). Two of the operations, NVGetCommunicator and NVGetLength, must be implemented by subvectors that are combined to create an NVector_ManyVector, but are not used outside of this context. The remaining nine operations are optional local reduction operations intended to eliminate unnecessary latency when performing vector reduction operations (norms, etc.) on distributed memory systems. The optional local reduction vector operations are NVDotProdLocal, NVMaxNormLocal, NVMinLocal, NVLinNormLocal, NWSqrSumLocal, NVWSqrSumMaskLocal, NVInvTestLocal, NVConstrMaskLocal, and NVMinQuotientLocal. If an NVector implementation defines any of the local operations as NULL, then the NVector_ManyVector will call standard NVector operations to complete the computation.

A new SunMatrix and SunLinsol implementation was added to facilitate the use of the SuperLU_DIST library with Sundials.

A new operation, SUNMatMatvecSetup, was added to the SunMatrix API. Users who have implemented custom SunMatrix modules will need to at least update their code to set the corresponding ops structure member, matvecsetup, to NULL.

The generic SunMatrix API now defines error codes to be returned by SunMatrix operations. Operations which return an integer flag indicating success/failure may return different values than previously.

Changes in v4.1.0

An additional NVector implementation was added for the Tpetra vector from the Trilinos library to facilitate interoperability between Sundials and Trilinos. This implementation is accompanied by additions to user documentation and Sundials examples.

A bug was fixed where a nonlinear solver object could be freed twice in some use cases.

The EXAMPLES_ENABLE_RAJA CMake option has been removed. The option EXAMPLES_ENABLE_CUDA enables all examples that use CUDA including the RAJA examples with a CUDA back end (if the RAJA NVector is enabled).

The implementation header file ida_impl.h is no longer installed. This means users who are directly manipulating the IDAMem structure will need to update their code to use IDA’s public API.

Python is no longer required to run make test and make test_install.

Changes in v4.0.2

Added information on how to contribute to Sundials and a contributing agreement.

Moved definitions of DLS and SPILS backwards compatibility functions to a source file. The symbols are now included in the IDA library, libsundials_ida.

Changes in v4.0.1

No changes were made in this release.
Changes in v4.0.0

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. This includes the “DIRECT” and “ITERATIVE” types as well as the new “MATRIX_ITERATIVE” type. Details regarding how IDALS utilizes linear solvers of each type as well as discussion regarding intended use cases for user-supplied SUNLINSOL implementations are included in Chapter 8. All IDA example programs and the standalone linear solver examples have been updated to use the unified linear solver interface.

The unified interface for the new IDALS module is very similar to the previous IDADLS and IDASPILS interfaces. 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. Additionally, we 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 follow the naming convention SUNLinSol_* where * is the name of the linear solver. The new names are SUNLinSol_Band, SUNLinSol_Dense, SUNLinSol_KLU, SUNLinSol_LapackBand, SUNLinSol_LapackDense, SUNLinSol_PCQ, SUNLinSol_SPBCGS, SUNLinSol_SPFGMR, SUNLinSol_SPGMR, SUNLinSol_SPTFQMR, and SUNLinSol_SuperLUMT. Solver-specific “set” routine names have been similarly standardized. To minimize challenges in user migration to the new names, 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. All IDA example programs and the standalone linear solver examples have been updated to use the new naming convention.

The SUNBandMatrix constructor has been simplified to remove the storage upper bandwidth argument.

SUNDIALS integrators have been updated to utilize generic nonlinear solver modules defined through the SUNNONLINSOL API. This API will ease the addition of new nonlinear solver options and allow for external or user-supplied nonlinear solvers. The SUNNONLINSOL API and SUNDIALS provided modules are described in Chapter 9 and follow the same object oriented design and implementation used by the nvector, sunmatrix, and SUNLINSOL modules. 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. Note the SUNNONLINSOL_FIXEDPOINT module can optionally utilize 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. Since IDA previously only used an internal implementation of a Newton iteration 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 as needed. While SUNDIALS includes a fixed-point nonlinear solver module, it is not currently supported in IDA. For details on attaching a user-supplied nonlinear solver to IDA see Chapter 4. Additionally, the example program idaRoberts_dns.c explicitly creates an attaches a SUNNONLINSOL_NEWTON object to demonstrate the process of creating and attaching a nonlinear solver module (note this is not necessary in general as IDA uses the SUNNONLINSOL_NEWTON module by default).

Three fused vector operations and seven vector array operations have been added to the nvector API. These optional operations are disabled by default and may be activated by calling vector specific routines after creating an nvector (see Chapter 6 for more details). The new 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 fused operations are N_VLinearCombination, N_VScaleAddMulti, and N_VDotProdMulti and the vector array operations are N_VLinearCombinationVectorArray, N_VScaleVectorArray, N_VConstVectorArray, N_VWrmsNormVectorArray, N_VWrmsNormMaskVectorArray, N_VScaleAddMultiVectorArray, and
N. VLinearCombinationVectorArray. If an NVECTOR implementation defines any of these operations as NULL, then standard NVECTOR operations will automatically be called as necessary to complete the computation.

Multiple updates to NVECTOR_CUDA were made:

- Changed N_VGetLength_Cuda to return the global vector length instead of the local vector length.
- Added N_VGetLocalLength_Cuda to return the local vector length.
- Added N_VGetMPIComm_Cuda to return the MPI communicator used.
- Removed the accessor functions in the namespace suncudavec.
- Changed the N_VMake_Cuda function to take a host data pointer and a device data pointer instead of an N_VectorContent_Cuda object.
- Added the ability to set the cudaStream_t used for execution of the NVECTOR_CUDA kernels. See the function N_VSetCudaStreams_Cuda.
- Added N_VNewManaged_Cuda, N_VMakeManaged_Cuda, and N_VIsManagedMemory_Cuda functions to accommodate using managed memory with the NVECTOR_CUDA.

Multiple changes to NVECTOR_RAJA were made:

- Changed N_VGetLength_Raja to return the global vector length instead of the local vector length.
- Added N_VGetLocalLength_Raja to return the local vector length.
- Added N_VGetMPIComm_Raja to return the MPI communicator used.
- Removed the accessor functions in the namespace suncudavec.

A new NVECTOR implementation for leveraging OpenMP 4.5+ device offloading has been added, NVECTOR_OPENMPDEV. See §6.10 for more details.

Changes in v3.2.1

The changes in this minor release include the following:

- Fixed a bug in the CUDA NVECTOR where the N_VInvTest operation could write beyond the allocated vector data.
- Fixed library installation path for multiarch systems. This fix changes the default library installation path to CMAKE_INSTALL_PREFIX/CMAKE_INSTALL_LIBDIR from CMAKE_INSTALL_PREFIX/lib. CMAKE_INSTALL_LIBDIR is automatically set, but is available as a CMake option that can modified.

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:
1.1 Changes from previous versions

- CMak 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 SUNLINSOL 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_WRmsNormMask and revised the RAJA NVECTOR implementation of N_WRmsNormMask 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 ("JTimes") routine, and where the cost of one JTSetup setup per Newton iteration can be amortized between multiple JTimes 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 boolantype 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 IDASetMaxBacksIC 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 limit 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 

\[ \text{smu} = \min(N-1, \text{mu}+\text{ml}) \]

was changed to

\[ \text{smu} = \text{mu} + \text{ml} \]

to correct an illegal input error for DGBTF/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, RPowI, and RPowR were changed to SUNMIN, SUNMAX, SUNSQR, SUNRabs, SUNRsqrt, SUNRexp, SRpowI, and SUNRpowR, respectively. These names occur in both the solver and in various example programs.

In the FIDA optional input routines FIDASETIN, 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.
1.1 Changes from previous versions

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, y0, 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).

idaspbcg 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 \texttt{iopt} and \texttt{ropt} arrays. Instead, IDA now provides a set of routines (with prefix \texttt{IDASet}) to change the default values for various quantities controlling the solver and a set of extraction routines (with prefix \texttt{IDAGet}) to extract statistics after return from the main solver routine. Similarly, each linear solver module provides its own set of \texttt{Set}- and \texttt{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 \texttt{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 (§7.5), a banded implementation (§7.6) and a sparse implementation (§7.7).

- 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

All SUNDIALS packages are released open source, under the BSD 3-Clause license. The only requirements of the license are preservation of copyright and a standard disclaimer of liability. The full text of the license and an additional notice are provided below and may also be found in the LICENSE and NOTICE files provided with all SUNDIALS packages.

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 BSD 3-Clause License

Copyright (c) 2002-2019, Lawrence Livermore National Security and Southern Methodist University. All rights reserved.

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

* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

* Neither the name of the copyright holder 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 THE COPYRIGHT HOLDER 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.

1.3.2 Additional Notice

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

This work 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.

1.3.3 SUNDIALS Release Numbers

LLNL-CODE-667205 (ARKODE)
UCRL-CODE-155951 (CVODE)
UCRL-CODE-155950 (CVODES)
UCRL-CODE-155952 (IDA)
UCRL-CODE-237203 (IDAS)
LLNL-CODE-665877 (KINSOL)
Chapter 2

Mathematical Considerations

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

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

(2.1)

where \( y, \dot{y}, \text{and} F \) are vectors in \( \mathbb{R}^N \), \( t \) 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 [9]. 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 \( y_d \) but not on any components of \( y_a \). 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 [5]. The method order ranges from 1 to 5, with the BDF of order \( q \) given by the multistep formula

\[ \sum_{i=0}^{q} \alpha_{n,i}y_{n-i} = h_n \dot{y}_n, \]

(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:

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

(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
\[ J[y_{n(m+1)} - y_{n(m)}] = -G(y_{n(m)}), \]  
(2.4)
where \( y_{n(m)} \) is the \( m \)-th approximation to \( y_n \). Here \( J \) is some approximation to the system Jacobian
\[ J = \frac{\partial G}{\partial y} = \frac{\partial F}{\partial y} + \alpha \frac{\partial F}{\partial \dot{y}}, \]
(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 [16, 1], or the thread-enabled SuperLU_MT sparse solver library [31, 18, 3] (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 with or without restarts,
- spfgmres, a scaled preconditioned FGMRES (Flexible Generalized Minimal Residual method) solver with or without restarts,
- 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 a preconditioned Krylov method 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 [7]. For the spils linear solvers with IDA, 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 \( \| \cdot \|_{WRMS} \), 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
\[ W_i = 1/[\text{RTOL} \cdot |y_i| + \text{ATOL}_i]. \]
(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 $\alpha$ 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 $m = 1, 2, \ldots$. The nonlinear solver iteration is halted if $R > 0.9$. The convergence test at the $m$-th iteration is then

$$S||\delta_m|| < 0.33,$$

(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

$$J_{ij} = [F_i(t, y + \sigma_j e_j, \dot{y} + \alpha \sigma_j e_j) - F_i(t, y, \dot{y})]/\sigma_j,$$

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 $\sigma = \sqrt{N}$. As an option, the user can specify a constant factor that is inserted into this expression for $\sigma$. 

2.1 IVP solution
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 (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 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 [5] 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 [7] 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, \( P \approx \frac{\partial F}{\partial y} + \alpha \frac{\partial F}{\partial y} \), 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 \([24]\).

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

\[
\tau = 100 \times U \times (|t_n| + |h|) \quad (U = \text{unit roundoff})
\]

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

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

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
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 [7, 11], 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, IDABBDPRES, 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...
### 3.2 IDA organization

![Diagram of Sundials organization]

**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 SUPERLU_MT 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 [27] 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

- \texttt{libdir/libsundials.ida.lib},
- \texttt{libdir/libsundials_nvec*.lib},

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

- \texttt{indir/include/ida}
- \texttt{indir/include/sundials}
- \texttt{indir/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,

```c
#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,
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 an `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_nonlinear_solver.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 SUNDIALS-provided 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 module, SUNLINSOL_LAPACKDENSE;
  - `sunlinsol/sunlinsol_lapackband.h`, which is used with the LAPACK banded linear solver module, SUNLINSOL_LAPACKBAND;
  - `sunlinsol/sunlinsol_klu.h`, which is used with the KLU sparse linear solver module, SUNLINSOL_KLU;
  - `sunlinsol/sunlinsol_superlumt.h`, which is used with the SUPERLUMT sparse linear solver module, SUNLINSOL_SUPERLUMT;

- **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 [27]), 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 NVVECTOR, 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 $N_{\text{local}}$.

   Note: The variables $N$ and $N_{\text{local}}$ should be of type `sunindextype`.

3. **Set vectors of initial values**

   To set the vectors $y_0$ and $y_0^\prime$ to initial values for $y$ and $\dot{y}$, use the appropriate functions defined by the particular NVVECTOR implementation.

   For native SUNDIALS vector implementations (except the CUDA and RAJA-based ones), use a call of the form $y_0 = N_{\text{VMake}}(\ldots, ydata)$ 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 $y_0 = N_{\text{VNew}}(\ldots)$, and then set its elements by accessing the underlying data with a call of the form `ydata = N_{\text{VGetArrayPointer}}(y0)`. See §6.2-6.5 for details.

   For the HYPRE and PETSC vector wrappers, first create and initialize the underlying vector and then create an NVVECTOR wrapper with a call of the form $y_0 = N_{\text{VMake}}(\ldots, yvec)$, where `yvec` is a HYPRE or PETSC vector. Note that calls like $N_{\text{VNew}}(\ldots)$ and $N_{\text{VGetArrayPointer}}(\ldots)$ are not available for these vector wrappers. See §6.6 and §6.7 for details.
If using either the CUDA- or RAJA-based vector implementations use a call of the form \( y_0 = \text{N}_\text{VMake}(...) \) where \( c \) is a pointer to a \text{suncudavec} or \text{sunrajavec} vector class if this class already exists. Otherwise, create a new vector by making a call of the form \( y_0 = \text{N}_\text{VNew}(...) \), and then set its elements by accessing the underlying data where it is located with a call of the form \( \text{N}_\text{VGetDeviceArrayPointer}(...) \) or \( \text{N}_\text{VGetHostArrayPointer}(...) \). Note that the vector class will allocate memory on both the host and device when instantiated. See §6.8-6.9 for details.

Set the vector \( y_0 \) of initial conditions for \( \dot{y} \) similarly.

4. Create IDA object

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

5. Initialize IDA solver

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

6. Specify integration tolerances

Call \text{IDASSTolerances}(...) or \text{IDAVStolerances}(...) to specify, respectively, a scalar relative tolerance and scalar absolute tolerance, or a scalar relative tolerance and a vector of absolute tolerances. Alternatively, call \text{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 matrix-based linear solver, then a template Jacobian matrix must be created by calling the appropriate constructor function defined by the particular \text{SUNMATRIX} implementation.

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

\[
\text{SUNMatrix J = SUNBandMatrix}(...);
\]

or

\[
\text{SUNMatrix J = SUNDenseMatrix}(...);
\]

or

\[
\text{SUNMatrix J = SUNSparseMatrix}(...);
\]

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 \text{SUNLINSOL} implementation.

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

\[
\text{SUNLinearSolver LS = SUNLinSol*}(...);
\]

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 and §4.5.8 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, yp, 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 yp (which can be the same as the vector yp0 above) will contain ˙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 yp (or y and yp) by calling the appropriate destructor function defined by the NVECTOR implementation:
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.

## 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.

<table>
<thead>
<tr>
<th>Linear Solver</th>
<th>Serial</th>
<th>Parallel (MPI)</th>
<th>OpenMP</th>
<th>pThreads</th>
<th>hypre</th>
<th>PETSc</th>
<th>CUDA</th>
<th>RAJA</th>
<th>User Supp.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dense</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Band</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LapackDense</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LapackBand</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>KLU</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SUPERLUMT</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SPGMR</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SPFGMR</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SPBCGS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SPTFQMR</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>PCG</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>User Supp.</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

### 4.5 User-callable functions

N_VDestroy(yret);

and similarly for ypret.
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**

```c
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**

```c
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 ˙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**

```c
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`.
4.5 User-callable functions

**IDASStolerances**

Call

```c
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

```c
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

```c
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` (IDAEwtFn) 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 $\text{reltol}=10^{-4}$ 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 [27]. 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 0.01 from the actual desired limits on errors. So if you want 0.01% 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 $y_{ret}$ 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 from (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. However, in IDA only left preconditioning is supported. 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

$$\text{SUNLinearSolver } LS = \text{SUNLinSol}_*(...);$$

The current list of such constructor routines includes SUNLinSol_Dense, SUNLinSol_Band, SUNLinSol_LapackDense, SUNLinSol_LapackBand, SUNLinSol_KLU, SUNLinSol_SuperLUMT, SUNLinSol_SPGMR, SUNLinSol_SPGMR, 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

$$\text{flag} = \text{IDASetLinearSolver}(ida\_mem, LS, J);$$

Description The function IDASetLinearSolver attaches a generic SUNLINSOL object $LS$ and corresponding template Jacobian SUNMATRIX object $J$ (if applicable) to IDA, initializing the IDALS linear solver interface.

Arguments

- $ida\_mem$ (void *) pointer to the IDA memory block.
- $LS$ (SUNLinearSolver) SUNLINSOL object to use for solving linear systems of the form (2.4).
- $J$ (SUNMatrix) SUNMATRIX object for used as a template for the Jacobian (or NULL if not applicable).

Return value The return value $\text{flag}$ (of type int) is one of

- IDALS_SUCCESS The IDALS initialization was successful.
- IDALS_MEM_NULL The $ida\_mem$ pointer is NULL.
- IDALS_ILL_INPUT The IDALS interface is not compatible with the $LS$ or $J$ input objects or is incompatible with the current NVECTOR module.
- IDALS_SUNLINS_FAIL A call to the $LS$ object failed.
- IDALS_MEM_FAIL A memory allocation request failed.

Notes If $LS$ is a matrix-based linear solver, then the template Jacobian matrix $J$ will be used in the solve process, so if additional storage is required within the SUNMATRIX 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 SUNMATRIX type in Chapter 7 for further information).

The previous routines IDADlsSetLinearSolver and IDASpilsSetLinearSolver 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.
4.5.4 Nonlinear solver interface function

By default IDA uses the SUNNONLINSOL implementation of Newton’s method defined by the SUNNONLINSOL_ROOTFIND 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. [9].) 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 IDAREInit), 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 IDASolve). 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 flag (of type int) will be one of the following:

- **IDA_SUCCESS**: IDASolve succeeded.
- **IDA_MEM_NULL**: The argument ida_mem was NULL.
- **IDA_NO_MALLOC**: The allocation function IDAInit has not been called.
- **IDA_IILL_INPUT**: One of the input arguments was illegal.
- **IDA_LSETUP_FAIL**: The linear solver's setup function failed in an unrecoverable manner.
- **IDA_LINIT_FAIL**: The linear solver's initialization function failed.
- **IDA_LSOLVE_FAIL**: The linear solver's solve function failed in an unrecoverable manner.
- **IDA_BAD_EWT**: Some component of the error weight vector is zero (illegal), either for the input value of y0 or a corrected value.
- **IDA_FIRST_RES_FAIL**: The user's residual function returned a recoverable error flag on the first call, but IDACalcIC was unable to recover.
- **IDA_RES_FAIL**: The user's residual function returned a nonrecoverable error flag.
- **IDA_NO_RECOVERY**: The user's residual function, or the linear solver's setup or solve function had a recoverable error, but IDACalcIC was unable to recover.
- **IDA_CONSTR_FAIL**: IDACalcIC was unable to find a solution satisfying the inequality constraints.
- **IDA_LINESEARCH_FAIL**: The linesearch algorithm failed to find a solution with a step larger than steptol in weighted RMS norm, and within the allowed number of backtracks.
- **IDA_CONV_FAIL**: IDACalcIC failed to get convergence of the Newton iterations.

Notes All failure return values are negative and therefore a test flag < 0 will trap all IDACalcIC failures.

Note that IDACalcIC will correct the values of y(t0) and \(\dot{y}(t_0)\) which were specified in the previous call to IDAInit or IDAReval. To obtain the corrected values, call IDAGetconsistentIC (see §4.5.10.3).

### 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.

```c
IDARootInit
```

Call `flag = IDARootInit(ida_mem, nrtfn, g);`

Description The function IDARootInit 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
- `ida_mem` (void *) pointer to the IDA memory block returned by IDACreate.
- `nrtfn` (int) is the number of root functions \(g_i\).
- `g` (IDARootFn) is the C function which defines the nrtfn functions \(g_i(t, y, \dot{y})\) whose roots are sought. See §4.6.4 for details.

Return value The return value flag (of type int) is one of
IDA_SUCCESS  The call to IDARootInit was successful.
IDA_MEM_NULL  The ida_mem argument was NULL.
IDA_MEM_FAIL  A memory allocation failed.
IDA_I LL_INPUT  The function g is NULL, but nrtfn > 0.

Notes  If a new IVP is to be solved with a call to IDAREInit, where the new IVP has no rootfinding problem but the prior one did, then call IDARootInit with nrtfn = 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  

```
flag = IDASolve(ida_mem, tout, &tret, yret, ypret, itask);
```

Description  The function IDASolve integrates the DAE over an interval in t.

Arguments  

- ida_mem (void *) pointer to the IDA memory block.
- tout (realtype) the next time at which a computed solution is desired.
- tret (realtype) the time reached by the solver (output).
- yret (N_Vector) the computed solution vector y.
- ypret (N_Vector) the computed solution vector ˙y.
- itask (int) a flag indicating the job of the solver for the next user step. The IDA_NORMAL task is to have the solver take internal steps until it has reached or just passed the user specified tout parameter. The solver then interpolates in order to return approximate values of y(tout) and ˙y(tout). The IDA_ONE_STEP option tells the solver to just take one internal step and return the solution at the point reached by that step.

Return value  

IDASolve returns vectors yret and ypret and a corresponding independent variable value t = tret, such that (yret, ypret) are the computed values of (y(t), ˙y(t)).

In IDA_NORMAL mode with no errors, tret will be equal to tout and yret = y(tout), ypret = ˙y(tout).

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

- IDA_SUCCESS  IDASolve succeeded.
- IDA_TSTOP_RETURN  IDASolve succeeded by reaching the stop point specified through the optional input function IDASetStopTime. See §4.5.8.1 for more information.
- IDA_ROOT_RETURN  IDASolve succeeded and found one or more roots. In this case, tret is the location of the root. If nrtfn > 1, call IDAGetRootInfo to see which gi were found to have a root. See §4.5.10.4 for more information.
- IDA_MEM_NULL  The ida_mem argument was NULL.
- IDA_I LL_INPUT  One of the inputs to IDASolve 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 IDACreate) failed to set the linear solver-specific lsolve field in ida_mem. (d) A root of one of the root functions was found both at a point t and also very near t. In any case, the user should see the printed error message for details.
IDA_TOO MUCH WORK  The solver took mxstep internal steps but could not reach tout.
The default value for mxstep is MXSTEP_DEFAULT = 500.
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 (MXNEF = 10) during
one internal time step or occurred with |h| = h_min.
IDA_CONV_FAIL  Convergence test failures occurred too many times (MXNCF = 10)
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 man-
ner.
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 yret can occupy the same space as the vector y0 of initial conditions that
was passed to IDAInit, and the vector yp ret can occupy the same space as yp0.

In the IDA_ONE_STEP mode, tout is used on the first call only, and only to get the
direction and rough scale of the independent variable.

If a stop time is enabled (through a call to IDASetStopTime), then IDASolve returns
the solution at tstop. Once the integrator returns at a stop time, any future testing for
tstop is disabled (and can be reenabled only through a new call to IDASetStopTime).

All failure return values are negative and therefore a test flag < 0 will trap all IDASolve
failures.

On any error return in which one or more internal steps were taken by IDASolve, the
returned values of tret, yret, and yp ret correspond to the farthest point reached in
the integration. On all other error returns, these values are left unchanged from the
previous IDASolve 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);
Table 4.2: Optional inputs for IDA and IDALS

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

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.
- 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.

\[\text{IDASetErrFile} \]

Call

\[
\text{flag} = \text{IDASetErrFile}(\text{ida_mem}, \text{errfp}, \text{eh_data});
\]

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

Arguments

- ida_mem (void *) pointer to the IDA memory block.
- ehfun (IDAErrHandlerFn) 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.

\[\text{IDASetUserData} \]

Call

\[
\text{flag} = \text{IDASetUserData}(\text{ida_mem}, \text{user_data});
\]

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.
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 ida_mem pointer is NULL.
IDA_ILL_INPUT The input value maxord is ≤ 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 maxord affects the memory requirements for the internal IDA memory block, its value cannot be increased past its previous value.

Call
flag = IDASetMaxNumSteps(ida_mem, mxsteps);

Description
The function IDASetMaxNumSteps specifies the maximum number of steps to be taken by the solver in its attempt to reach the next output time.

Arguments
ida_mem (void *) pointer to the IDA memory block.
mxsteps (long int) maximum allowed number of steps.

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
Passing mxsteps = 0 results in IDA using the default value (500).
Passing mxsteps < 0 disables the test (not recommended).

Call
flag = IDASetInitStep(ida_mem, hin);

Description
The function IDASetInitStep specifies the initial step size.

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

hin (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 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
By default, IDA estimates the initial step as the solution of $\|h\dot{y}\|_{W\text{RMS}} = 1/2$, with an added restriction that $|h| \leq 0.01|t_{\text{out}} - t_{\text{0}}|$.

Call
flag = IDASetMaxStep(ida_mem, hmax);

Description
The function IDASetMaxStep specifies the maximum absolute value of the step size.

Arguments
ida_mem (void *) pointer to the IDA memory block.
hmax (realtype) maximum absolute value of the step size.

Return value
The return value flag (of type int) is one of
### 4.5 User-callable functions

**IDA_SUCCESS**  The optional value has been successfully set.
**IDA_MEM_NULL**  The *ida_mem* pointer is NULL.
**IDA_Ill_INPUT**  Either *hmax* is not positive or it is smaller than the minimum allowable step.

**Notes**  Pass *hmax* = 0 to obtain the default value ∞.

#### IDASetStopTime

**Call**  
```c
flag = IDASetStopTime(ida_mem, tstop);
```

**Description**  The function *IDASetStopTime* specifies the value of the independent variable *t* past which the solution is not to proceed.

**Arguments**  
- *ida_mem*  (*void*) pointer to the *ida* memory block.
- *tstop*  (*realtype*) value of the independent variable past which the solution should not proceed.

**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 *tstop* is not beyond the current *t* value, *t_n*.

**Notes**  
- The default, if this routine is not called, is that no stop time is imposed.
- Once the integrator returns at a stop time, any future testing for *tstop* is disabled (and can be reenabled only through a new call to *IDASetStopTime*).

#### IDASetMaxErrTestFails

**Call**  
```c
flag = IDASetMaxErrTestFails(ida_mem, maxnef);
```

**Description**  The function *IDASetMaxErrTestFails* specifies the maximum number of error test failures in attempting one step.

**Arguments**  
- *ida_mem*  (*void*) pointer to the *ida* memory block.
- *maxnef*  (*int*) maximum number of error test failures allowed on one step (> 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.

**Notes**  The default value is 10.

#### IDASetMaxNonlinIters

**Call**  
```c
flag = IDASetMaxNonlinIters(ida_mem, maxcor);
```

**Description**  The function *IDASetMaxNonlinIters* specifies the maximum number of nonlinear solver iterations at one step.

**Arguments**  
- *ida_mem*  (*void*) pointer to the *ida* memory block.
- *maxcor*  (*int*) maximum number of nonlinear solver iterations allowed on one step (> 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_MEM_FAIL**  The *SUNNONLINSOL* module is NULL.

**Notes**  The default value is 4.
**IDASetMaxConvFails**

Call  
flag = IDASetMaxConvFails(ida_mem, maxncf);

Description  
The function IDASetMaxConvFails specifies the maximum number of nonlinear solver convergence failures at one step.

Arguments  
ida_mem (void *) pointer to the IDA memory block.
maxncf (int) maximum number of allowable nonlinear solver convergence failures on one step (> 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.

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 <= 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 (booleantype) 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. [5] for more on this issue.
4.5 User-callable functions

### IDASetId

**Call**

\[ \text{flag} = \text{IDASetId}(\text{ida}\_\text{mem}, \text{id}) ; \]

**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**

\[ \text{flag} = \text{IDASetConstraints}(\text{ida}\_\text{mem}, \text{constraints}) ; \]

**Description**
The function IDASetConstraints specifies a vector defining inequality constraints for each component of the solution vector \( y \).

**Arguments**
- `ida\_mem` (void *) pointer to the IDA memory block.
- `constraints` (N\_Vector) vector of constraint flags. If `constraints[i]` is
  - 0.0 then no constraint is imposed on \( y_i \).
  - 1.0 then \( y_i \) will be constrained to be \( y_i \geq 0.0 \).
  - -1.0 then \( y_i \) will be constrained to be \( y_i \leq 0.0 \).
  - 2.0 then \( y_i \) will be constrained to be \( y_i > 0.0 \).
  - -2.0 then \( y_i \) will be constrained to be \( y_i < 0.0 \).

**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 constraints vector contains illegal values.

**Notes**
The presence of a non-NULL constraints vector that is not 0.0 in all components will cause constraint checking to be performed. However, a call with 0.0 in all components of `constraints` 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 interface. 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.
Using IDA for C Applications

**IDASetJacFn**

Call

```c
flag = IDASetJacFn(ida_mem, jac);
```

Description

The function `IDASetJacFn` specifies the Jacobian approximation function to be used for a matrix-based solver within the IDALS interface.

Arguments

- `ida_mem` (void *) pointer to the IDA memory block.
- `jac` (IDALsJacFn) user-defined Jacobian approximation function.

Return value

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

- `IDALS_SUCCESS` The optional value has been successfully set.
- `IDALS_MEM_NULL` The `ida_mem` pointer is `NULL`.
- `IDALS_LMEM_NULL` The IDALS linear solver interface has not been initialized.

Notes

This function must be called after the IDALS linear solver interface has been initialized through a call to `IDASetLinearSolver`.

By default, IDALS uses an internal difference quotient function for dense and band matrices. If `NULL` is passed to `jac`, this default function is used. An error will occur if no `jac` is supplied when using other matrix types.

The function type `IDALsJacFn` is described in §4.6.5.

The previous routine `IDADlsSetJacFn` 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.

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

```c
flag = IDASetJacTimes(ida_mem, jsetup, jtimes);
```

Description

The function `IDASetJacTimes` specifies the Jacobian-vector setup and product functions.

Arguments

- `ida_mem` (void *) pointer to the IDA memory block.
- `jsetup` (IDALsJacTimesSetupFn) user-defined function to set up the Jacobian-vector product. Pass `NULL` if no setup is necessary.
- `jtimes` (IDALsJacTimesVecFn) user-defined Jacobian-vector product function.

Return value

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

- `IDALS_SUCCESS` The optional 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.
- `IDALS_SUNLS_FAIL` An error occurred when setting up the system matrix-times-vector routines in the SUNLINSOL object used by the IDALS interface.
4.5 User-callable functions

Notes The default is to use an internal finite difference quotient for jtimes and to omit jtsetup. If NULL is passed to jtimes, these defaults are used. A user may specify non-NULL jtimes and NULL jtsetup inputs.

This function must be called after the IDALS linear solver interface has been initialized through a call to IDASetLinearSolver.

The function type IDALsJacTimesSetupFn is described in §4.6.7.

The function type IDALsJacTimesVecFn is described in §4.6.6.

The previous routine IDASpilsSetJacTimes 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.

Alternately, when using the default difference-quotient approximation to the Jacobian-vector product, the user may specify the factor to use in setting increments for the finite-difference approximation, via a call to IDASetIncrementFactor:

```c
flag = IDASetIncrementFactor(ida_mem, dqincfac);
```

Description The function IDASetIncrementFactor specifies the increment factor to be used in the difference-quotient approximation to the product Jv. Specifically, Jv is approximated via the formula

\[ Jv = \frac{1}{\sigma} [F(t, \tilde{y}, \tilde{y}') - F(t, y, y')], \]

where \( \tilde{y} = y + \sigma v, \tilde{y}' = y' + c_j \sigma v, c_j \) is a BDF parameter proportional to the step size, \( \sigma = \sqrt{N} \) dqincfac, and \( N \) is the number of equations in the DAE system.

Arguments
- `ida_mem` (void *) pointer to the IDA memory block.
- `dqincfac` (realtype) user-specified increment factor (positive).

Return value The return value `flag` (of type `int`) is one of
- IDALS_SUCCESS The optional 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.
- IDALS_ILL_INPUT The specified value of `dqincfac` is \( \leq 0 \).

Notes
- The default value is 1.0.
- This function must be called after the IDALS linear solver interface has been initialized through a call to IDASetLinearSolver.
- The previous routine IDASpilsSetIncrementFactor 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.

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 IDASpilsSetPreconditioner function.

**IDASpilsSetPreconditioner**

**Call**

```c
flag = IDASpilsSetPreconditioner(ida_mem, psetup, psolve);
```

**Description**
The function IDASpilsSetPreconditioner specifies the preconditioner setup and solve functions.

**Arguments**
- `ida_mem` (void *): pointer to the IDA memory block.
- `psetup` (IDALsPrecSetupFn): user-defined function to set up the preconditioner. Pass NULL if no setup is necessary.
- `psolve` (IDALsPrecSolveFn): user-defined preconditioner solve function.

**Return value**
The return value `flag` (of type int) is one of
- IDALS_SUCCESS: The optional values have been successfully set.
- IDALS_MEM_NULL: The `ida_mem` pointer is NULL.
- IDALS_LMEM_NULL: The IDALS linear solver has not been initialized.
- IDALS_SUNLS_FAIL: An error occurred when setting up preconditioning in the SUNLINSOL object used by the IDALS interface.

**Notes**
The default is NULL for both arguments (i.e., no preconditioning).

This function must be called after the IDALS linear solver interface has been initialized through a call to IDASpilsSetLinearSolver.

The function type IDALsPrecSolveFn is described in §4.6.8.

The function type IDALsPrecSetupFn is described in §4.6.9.

The previous routine IDASpilsSetPreconditioner 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.

**IDASpilsSetEpsLin**

**Call**

```c
flag = IDASpilsSetEpsLin(ida_mem, eplifac);
```

**Description**
The function IDASpilsSetEpsLin specifies the factor by which the Krylov linear solver’s convergence test constant is reduced from the nonlinear iteration test constant.

**Arguments**
- `ida_mem` (void *): pointer to the IDA memory block.
- `eplifac` (realtype): linear convergence safety factor ($\geq 0.0$).

**Return value**
The return value `flag` (of type int) is one of
- IDALS_SUCCESS: The optional 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.
- IDALS_SUNLS_FAIL: An error occurred when setting up preconditioning in the SUNLINSOL object used by the IDALS interface.

**Notes**
The default value is 0.05.

This function must be called after the IDALS linear solver interface has been initialized through a call to IDASpilsSetLinearSolver.

If `eplifac` = 0.0 is passed, the default value is used.

The previous routine IDASpilsSetEpsLin 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 `flag = IDASetNonlinConvCoefIC(ida_mem, epiccon);`

Description The function IDASetNonlinConvCoefIC specifies the positive constant in the Newton iteration convergence test within the initial condition calculation.

Arguments `ida_mem` (void *) pointer to the IDA memory block.
`epiccon` (realtype) coefficient in the Newton convergence test (> 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 `epiccon` factor is <= 0.0.

Notes The default value is 0.01 · 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 \( \leq \) `epiccon`, where \( J \) is the system Jacobian.

**IDASetMaxNumStepsIC**

Call `flag = IDASetMaxNumStepsIC(ida_mem, maxnh);`

Description The function IDASetMaxNumStepsIC specifies the maximum number of steps allowed when `icopt=IDA_YA_YDP_INIT` in IDACalcIC, where \( h \) appears in the system Jacobian, \( J = \partial F/\partial y + (1/h)\partial F/\partial \dot{y} \).

Arguments `ida_mem` (void *) pointer to the IDA memory block.
`maxnh` (int) maximum allowed number of values for \( h \).

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` `maxnh` 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 

```c
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 

```c
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 

```c
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` (booleantype) 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`.

**IDASetStepToleranceIC**

Call 

```c
flag = IDASetStepToleranceIC(ida_mem, steptol);
```

Description 
The function `IDASetStepToleranceIC` 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
4.5 User-callable functions

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` The `steptol` tolerance is $\leq 0.0$.

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

**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:
**IDAGetDky**

**Call**

```c
flag = IDAGetDky(ida_mem, t, k, dky);
```

**Description**
The function `IDAGetDky` 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**
- `ida_mem` (void *) pointer to the IDA memory block.
- `t` (realtype) time at which to interpolate.
- `k` (int) integer specifying the order of the derivative of $y$ wanted.
- `dky` (N_Vector) vector containing the interpolated $k$th derivative of $y(t)$.

**Return value**
The return value `flag` (of type int) is one of
- IDA_SUCCESS IDAGetDky succeeded.
- IDA_MEM_NULL The `ida_mem` argument was NULL.
- IDA_BAD_T $t$ is not in the interval $[t_n - h_u, t_n]$.
- IDA_BAD_K $k$ is not one of $\{0, 1, \ldots, k_{last}\}$.
- IDA_BAD_DKY `dky` is NULL.

**Notes**
It is only legal to call the function `IDAGetDky` after a successful return from `IDASolve`. Functions `IDAGetCurrentTime`, `IDAGetLastStep` and `IDAGetLastOrder` (see §4.5.10.2) can be used to access $t_n$, $h_u$ and $k_{last}$.

### 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.

#### 4.5.10.1 SUNDIALS version information

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

**SUNDIALSGetVersion**

**Call**

```c
flag = SUNDIALSGetVersion(version, len);
```

**Description**
The function `SUNDIALSGetVersion` fills a character array with SUNDIALS version information.

**Arguments**
- `version` (char *) character array to hold the SUNDIALS version information.
- `len` (int) allocated length of the `version` character array.
4.5 User-callable functions

Table 4.3: Optional outputs from IDA and IDALS

<table>
<thead>
<tr>
<th>Optional output</th>
<th>Function name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of IDA real and integer workspace</td>
<td>IDAGetWorkSpace</td>
</tr>
<tr>
<td>Cumulative number of internal steps</td>
<td>IDAGetNumSteps</td>
</tr>
<tr>
<td>No. of calls to residual function</td>
<td>IDAGetNumResEvals</td>
</tr>
<tr>
<td>No. of calls to linear solver setup function</td>
<td>IDAGetNumLinSolvSetups</td>
</tr>
<tr>
<td>No. of local error test failures that have occurred</td>
<td>IDAGetNumErrTestFails</td>
</tr>
<tr>
<td>Order used during the last step</td>
<td>IDAGetLastOrder</td>
</tr>
<tr>
<td>Order to be attempted on the next step</td>
<td>IDAGetCurrentOrder</td>
</tr>
<tr>
<td>Order reductions due to stability limit detection</td>
<td>IDAGetNumStabLimOrderReds</td>
</tr>
<tr>
<td>Actual initial step size used</td>
<td>IDAGetActualInitStep</td>
</tr>
<tr>
<td>Step size used for the last step</td>
<td>IDAGetLastStep</td>
</tr>
<tr>
<td>Step size to be attempted on the next step</td>
<td>IDAGetCurrentStep</td>
</tr>
<tr>
<td>Current internal time reached by the solver</td>
<td>IDAGetCurrentTime</td>
</tr>
<tr>
<td>Suggested factor for tolerance scaling</td>
<td>IDAGetTolScaleFactor</td>
</tr>
<tr>
<td>Error weight vector for state variables</td>
<td>IDAGetErrWeights</td>
</tr>
<tr>
<td>Estimated local errors</td>
<td>IDAGetEstLocalErrors</td>
</tr>
<tr>
<td>No. of nonlinear solver iterations</td>
<td>IDAGetNumNonlinSolvIters</td>
</tr>
<tr>
<td>No. of nonlinear convergence failures</td>
<td>IDAGetNumNonlinSolvConvFails</td>
</tr>
<tr>
<td>Array showing roots found</td>
<td>IDAGetRootInfo</td>
</tr>
<tr>
<td>No. of calls to user root function</td>
<td>IDAGetNumGEvals</td>
</tr>
<tr>
<td>Name of constant associated with a return flag</td>
<td>IDAGetReturnFlagName</td>
</tr>
</tbody>
</table>

**IDA initial conditions calculation**

| Number of backtrack operations                                                 | IDAGetNumBacktrackops          |
| Corrected initial conditions                                                    | IDAGetConsistentIC             |

**IDALS linear solver interface**

| 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        |
Return value If successful, SUNDIALSGetVersion returns 0 and version contains the SUNDIALS version information. Otherwise, it returns -1 and version 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 version array are removed.

SUNDIALSGetVersionNumber

Call flag = SUNDIALSGetVersionNumber(&major, &minor, &patch, label, len);

Description The function SUNDIALSGetVersionNumber set integers for the SUNDIALS major, minor, and patch release numbers and fills a character array with the release label if applicable.

Arguments

- major (int) SUNDIALS release major version number.
- minor (int) SUNDIALS release minor version number.
- patch (int) SUNDIALS release patch version number.
- label (char *) character array to hold the SUNDIALS release label.
- len (int) allocated length of the label character array.

Return value If successful, SUNDIALSGetVersionNumber returns 0 and the major, minor, patch, and label values are set. Otherwise, it returns -1 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 label. Any trailing characters in the label 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 flag = IDAGetWorkSpace(ida_mem, &lenrw, &leniw);

Description The function IDAGetWorkSpace returns the IDA real and integer workspace sizes.

Arguments

- ida_mem (void *) pointer to the IDA memory block.
- lenrw (long int) number of real values in the IDA workspace.
- leniw (long int) 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 realltype words, is given by the following:

- base value: lenrw = 55 + (m + 6) * N_r + 3*nrtfn;
- with IDASVtolerances: lenrw = lenrw + N_r;
- with constraint checking (see IDASetConstraints): lenrw = lenrw + N_r;
4.5 User-callable functions

- 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 \( \text{N	extunderscore Vector} \) \( (\approx N) \).

The size of the integer workspace (without distinction between \text{int} and \text{long int} words) is given by:

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

where \( N_i \) is the number of integer words in one \( \text{N	extunderscore Vector} \) \( (= 1 \) for \text{nvector	extunderscore serial} and \( 2 \ast \text{npes} \) for \text{nvector	extunderscore parallel} on \text{npes} processors).

For the default value of \text{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

\[ \text{flag} = \text{IDAGetNumSteps}(\text{ida	extunderscore mem}, \&\text{nsteps}); \]

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

Arguments
- \( \text{ida	extunderscore mem} \) (void *) pointer to the IDA memory block.
- \( \text{nsteps} \) (long int) number of steps taken by IDA.

Return value
The return value \( \text{flag} \) (of type \text{int}) is one of
- IDA\textunderscore SUCCESS The optional output value has been successfully set.
- IDA\textunderscore MEM\textunderscore NULL The \text{ida	extunderscore mem} pointer is NULL.

**IDAGetNumResEvals**

Call

\[ \text{flag} = \text{IDAGetNumResEvals}(\text{ida	extunderscore mem}, \&\text{nrevals}); \]

Description
The function IDAGetNumResEvals returns the number of calls to the user’s residual evaluation function.

Arguments
- \( \text{ida	extunderscore mem} \) (void *) pointer to the IDA memory block.
- \( \text{nrevals} \) (long int) number of calls made to the user’s \text{res} function.

Return value
The return value \( \text{flag} \) (of type \text{int}) is one of
- IDA\textunderscore SUCCESS The optional output value has been successfully set.
- IDA\textunderscore MEM\textunderscore NULL The \text{ida	extunderscore mem} pointer is NULL.

Notes
The \text{nrevals} value returned by IDAGetNumResEvals does not account for calls made to \text{res} from a linear solver or preconditioner module.

**IDAGetNumLinSolvSetups**

Call

\[ \text{flag} = \text{IDAGetNumLinSolvSetups}(\text{ida	extunderscore mem}, \&\text{nlinsetups}); \]

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

Arguments
- \( \text{ida	extunderscore mem} \) (void *) pointer to the IDA memory block.
- \( \text{nlinsetups} \) (long int) number of calls made to the linear solver setup function.

Return value
The return value \( \text{flag} \) (of type \text{int}) is one of
- IDA\textunderscore SUCCESS The optional output value has been successfully set.
- IDA\textunderscore MEM\textunderscore NULL The \text{ida	extunderscore 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 (if from `IDASolve`), or the last value of the artificial step size $h$ (if from `IDACalcIC`).

**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.
4.5 User-callable functions

**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:  
\[ \text{flag} = \text{IDAGetErrWeights}(\text{ida_mem}, \text{eweight}); \]

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:
- \( \text{ida_mem} \) (void *) pointer to the IDA memory block.
- \( \text{eweight} \) (N_Vector) solution error weights at the current time.

Return value: The return value \( \text{flag} \) (of type \( \text{int} \)) 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 \( \text{eweight} \).

**IDAGetEstLocalErrors**

Call:  
\[ \text{flag} = \text{IDAGetEstLocalErrors}(\text{ida_mem}, \text{ele}); \]

Description: The function IDAGetEstLocalErrors returns the estimated local errors.

Arguments:
- \( \text{ida_mem} \) (void *) pointer to the IDA memory block.
- \( \text{ele} \) (N_Vector) estimated local errors at the current time.

Return value: The return value \( \text{flag} \) (of type \( \text{int} \)) 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 \( \text{ele} \). The values returned in \( \text{ele} \) are only valid if IDASolve returned a non-negative value.

The \( \text{ele} \) vector, together with the \( \text{eweight} \) 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 \( \text{eweight}[i]\times\text{ele}[i] \).

**IDAGet Integrator Stats**

Call:  
\[ \text{flag} = \text{IDAGetIntegratorStats}(\text{ida_mem}, \&\text{nsteps}, \&\text{nrevals}, \&\text{nlinsetups}, \&\text{netfails}, \&\text{klast}, \&\text{kcur}, \&\text{hinused}, \&\text{hlast}, \&\text{hcur}, \&\text{tcur}); \]

Description: The function IDAGetIntegratorStats returns the IDA integrator statistics as a group.

Arguments:
- \( \text{ida_mem} \) (void *) pointer to the IDA memory block.
- \( \text{nsteps} \) (long int) cumulative number of steps taken by IDA.
- \( \text{nrevals} \) (long int) cumulative number of calls to the user’s \( \text{res} \) function.
- \( \text{nlinsetups} \) (long int) cumulative number of calls made to the linear solver setup function.
- \( \text{netfails} \) (long int) cumulative number of error test failures.
- \( \text{klast} \) (int) method order used on the last internal step.
- \( \text{kcur} \) (int) method order to be used on the next internal step.
- \( \text{hinused} \) (realtype) actual value of initial step size.
- \( \text{hlast} \) (realtype) step size taken on the last internal step.
- \( \text{hcur} \) (realtype) step size to be attempted on the next internal step.
- \( \text{tcur} \) (realtype) current internal time reached.
4.5 User-callable functions

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

`IDAGetNumBacktrackOps`

Call

```c
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

```c
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

```c
flag = IDAGetRootInfo(ida_mem, rootsfound);
```

Description The function `IDAGetRootInfo` returns an array showing which functions were found to have a root.

Arguments

- `ida_mem` (void *) pointer to the IDA memory block.
- `rootsfound` (int *) array of length `nrtfn` with the indices of the user functions $g_i$ found to have a root. For $i = 0,\ldots,nrtfn - 1$, $\text{rootsfound}[i] \neq 0$ if $g_i$ has a root, and = 0 if not.

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.

Notes Note that, for the components $g_i$ for which a root was found, the sign of `rootsfound[i]` indicates the direction of zero-crossing. A value of +1 indicates that $g_i$ is increasing, while a value of −1 indicates a decreasing $g_i$.

The user must allocate memory for the vector `rootsfound`. 
4.5 User-callable functions

**IDAGetNumGEvals**

Call

\[ \text{flag} = \text{IDAGetNumGEvals}(\text{ida\_mem}, \&\text{ngevals}); \]

Description The function IDAGetNumGEvals returns the cumulative number of calls to the user root function \( g \).

Arguments

- \( \text{ida\_mem} \) (void *) pointer to the IDA memory block.
- \( \text{ngevals} \) (long int) number of calls to the user’s function \( g \) so far.

Return value The return value \( \text{flag} \) (of type int) is one of

- IDA_SUCCESS The optional output value has been successfully set.
- IDA_MEM_NULL The \( \text{ida\_mem} \) pointer is NULL.

**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. lenr\_LS).

**IDAGetLinWorkSpace**

Call

\[ \text{flag} = \text{IDAGetLinWorkSpace}(\text{ida\_mem}, \&\text{lenrwLS}, \&\text{leniwLS}); \]

Description The function IDAGetLinWorkSpace returns the sizes of the real and integer workspaces used by the IDALS linear solver interface.

Arguments

- \( \text{ida\_mem} \) (void *) pointer to the IDA memory block.
- \( \text{lenrwLS} \) (long int) the number of real values in the IDALS workspace.
- \( \text{leniwLS} \) (long int) the number of integer values in the IDALS workspace.

Return value The return value \( \text{flag} \) (of type int) is one of

- IDALS_SUCCESS The optional output value has been successfully set.
- IDALS_MEM_NULL The \( \text{ida\_mem} \) pointer is NULL.
- IDALS_LMEM_NULL The IDALS linear solver has not been initialized.

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

\[ \text{flag} = \text{IDAGetNumJacEvals}(\text{ida\_mem}, \&\text{njevals}); \]

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

Arguments

- \( \text{ida\_mem} \) (void *) pointer to the IDA memory block.
- \( \text{njevals} \) (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

```c
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

```c
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

```c
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.
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
If the IDALS setup function failed (i.e., IDASolve returned IDA SETUP_FAIL) when using the SUNLINSOL.Dense or SUNLINSOL.Band modules, then the value of lsflag is equal to the column index (numbered from one) at which a zero diagonal element was encountered during the LU factorization of the (dense or banded) Jacobian matrix.
If the IDALS setup function failed when using another SUNLINSOL module, then lsflag will be SUNLS PSET_FAIL_UNREC, SUNLS ASET_FAIL_UNREC, or SUNLS PACKAGE_FAIL_UNREC.
If the IDALS solve function failed (IDASolve returned IDA SOLVE_FAIL), lsflag contains the error return flag from the SUNLINSOL object, which will be one of: SUNLS MEM_NULL, indicating that the SUNLINSOL memory is NULL; SUNLS ATIMES_FAIL_UNREC, indicating an unrecoverable failure in the J*v function; SUNLS PSOLVE_FAIL_UNREC, indicating that the preconditioner solve function psolve failed unrecoverably; SUNLS GS_FAIL, indicating a failure in the Gram-Schmidt procedure (generated only in SPGMR or SPFGMR); SUNLS QRSOL_FAIL, indicating that the matrix R was found to be singular during the QR solve phase (SPGMR and SPFGMR only); or SUNLS PACKAGE_FAIL_UNREC, indicating an unrecoverable failure in an external iterative linear solver package.
The previous routines IDADlsGetLastFlag and IDASpilsGetLastFlag 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 User-callable functions

**IDAGetLinReturnFlagName**

Call: \[ \text{name} = \text{IDAGetLinReturnFlagName}(\text{lsflag}); \]

Description: The function `IDAGetLinReturnFlagName` returns the name of the IDALS constant corresponding to `lsflag`.

Arguments: The only argument, of type `long int`, 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 `IDADisGetReturnFlagName` and `IDASpilsGetReturnFlagName` 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: \[ \text{flag} = \text{IDAReInit}(\text{ida_mem}, \text{t0}, \text{y0}, \text{yp0}); \]

Description: The function `IDAReInit` provides required problem specifications and reinitializes IDA.

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

- \( \text{t0} \) (realtype) is the initial value of \( t \).
- \( \text{y0} \) (N_Vector) is the initial value of \( y \).
- \( \text{yp0} \) (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 `IDAReInit` was successful.
IDA_MEM_NULL  The IDA memory block was not initialized through a previous call to IDACreate.

IDA_NO_MALLOC Memory space for the IDA memory block was not allocated through a previous call to IDAInit.

IDA_ILL_INPUT  An input argument to IDAREInit has an illegal value.

Notes  If an error occurred, IDAREInit 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 IDAREFn defined as follows:

```
typedef int (*IDAREFn)(realtype tt, N_Vector yy, N_Vector yp, N_Vector rr, void *user_data);
```

**Definition**  This function computes the problem residual for given values of the independent variable $t$, state vector $y$, and derivative $\dot{y}$.

**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 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 IDAREFn 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 IDAREFn 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 User-supplied functions

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 IDAErrHandlerFn to process any such messages. The function type IDAErrHandlerFn is defined as follows:

```c
typedef void (*IDAErrHandlerFn)(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 IDASetErrHandlerFn.

**Return value**

A IDAErrHandlerFn 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 IDAEwtFn to compute a vector `ewt` containing the multiplicative weights $W_i$ used in the WRMS norm $\| v \|_{WRMS} = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (W_i \cdot v_i)^2}$. These weights will used in place of those defined by Eq. (2.6). The function type IDAEwtFn is defined as follows:

```c
typedef int (*IDAEwtFn)(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.
- `ewt` is the output vector containing the error weights.
- `user_data` is a pointer to user data, the same as the `user_data` parameter passed to IDASetUserData.

**Return value**

An IDAEwtFn function type must return 0 if it successfully set the error weights and −1 otherwise.

**Notes**

Allocation of memory for `ewt` 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**

typedef int (*IDARootFn)(realtype t, N_Vector y, N_Vector yp, 
realtype *gout, void *user_data);

**Purpose**

This function computes a vector-valued function \(g(t, y, \dot{y})\) such that the roots of the \(\text{nrtfn}\) 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 \(y\).
- \(gout\) is the output array, of length \(\text{nrtfn}\), with components \(g_i(t, y, \dot{y})\).
- \(\text{user\_data}\) is a pointer to user data, the same as the \(\text{user\_data}\) parameter passed to \text{IDASetUserData}.

**Return value**

An IDARootFn should return 0 if successful or a non-zero value if an error occurred (in which case the integration is halted and \text{IDASolve} returns \text{IDA_RTFUNC\_FAIL}).

**Notes**

Allocation of memory for \(gout\) 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 \text{SUNMATRIX} object was supplied to \text{IDASetLinearSolver}), the user may provide a function of type \text{IDALSJacFn} defined as follows:

**IDALSJacFn**

**Definition**

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);

**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 \text{SUNMATRIX}), \(J = \partial F/\partial y + cj \partial F/\partial \dot{y}\).
- \(\text{user\_data}\) is a pointer to user data, the same as the \(\text{user\_data}\) parameter passed to \text{IDASetUserData}.

- \(tmp1\)
- \(tmp2\)
- \(tmp3\) are pointers to memory allocated for variables of type \text{N\_Vector} which can be used by \text{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 \text{SUNMATRIX} structure (e.g. number of rows, upper/lower bandwidth, sparsity type) may be obtained through using the implementation-specific \text{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 \(J\).

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 \(\text{ida_mem}\) to \text{user_data} and then use the \text{IDAget\_mem} 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 \(J\) with an approximation to the Jacobian matrix \(J(t, y, \dot{y})\) at the point \((tt, yy, yp)\). The accessor macros \(\text{SM\_ELEMENT\_D}\) and \(\text{SM\_COLUMN\_D}\) allow the user to read and write dense matrix elements without making explicit references to the underlying representation of the SUNMATRIX\_DENSE type. \(\text{SM\_ELEMENT\_D}(J, i, j)\) references the \((i,j)\)-th element of the dense matrix \(J\) (with \(i, j = 0...\text{Neq} - 1\)). 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 \(\text{Neq}\), the Jacobian element \(J_{m,n}\) can be set using the statement \(\text{SM\_ELEMENT\_D}(J, m-1, n-1) = J_{m,n}\). Alternatively, \(\text{SM\_COLUMN\_D}(J, j)\) returns a pointer to the first element of the \(j\)-th column of \(J\) (with \(j = 0...\text{Neq} - 1\)), 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 \(\text{col}_n = \text{SM\_COLUMN\_D}(J, n-1); \text{col}_{m-1} = J_{m,n}\). For large problems, it is more efficient to use \(\text{SM\_COLUMN\_D}\) than to use \(\text{SM\_ELEMENT\_D}\). Note that both of these macros number rows and columns starting from 0. The SUNMATRIX\_DENSE type and accessor macros are documented in §7.5.

banded:

A user-supplied banded Jacobian function must load the \(\text{Neq} \times \text{Neq}\) banded matrix \(J\) with an approximation to the Jacobian matrix \(J(t, y, \dot{y})\) at the point \((tt, yy, yp)\). The accessor macros \(\text{SM\_ELEMENT\_B}, \text{SM\_COLUMN\_B}, \) and \(\text{SM\_COLUMN\_ELEMENT\_B}\) allow the user to read and write banded matrix elements without making specific references to the underlying representation of the SUNMATRIX\_BAND type. \(\text{SM\_ELEMENT\_B}(J, i, j)\) references the \((i,j)\)-th element of the banded matrix \(J\), counting from 0. 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 \(\text{Neq}\) with \((m,n)\) within the band defined by \(\text{mupper}\) and \(\text{mlower}\), the Jacobian element \(J_{m,n}\) can be loaded using the statement \(\text{SM\_ELEMENT\_B}(J, m-1, n-1) = J_{m,n}\). The elements within the band are those with \(-\text{mupper} \leq n-m \leq \text{mlower}\). Alternatively, \(\text{SM\_COLUMN\_B}(J, j)\) returns a pointer to the diagonal element of the \(j\)-th column of \(J\), and if we assign this address to \text{realt}\_type *\text{col}\_j, then the \(i\)-th element of the \(j\)-th column is given by \(\text{SM\_COLUMN\_ELEMENT\_B}(\text{col}\_j, i, j)\), counting from 0. Thus, for \((m,n)\) within the band, \(J_{m,n}\) can be loaded by setting \(\text{col}_n = \text{SM\_COLUMN\_B}(J, n-1); \text{col}_{m-1} = J_{m,n}\). 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 \(\text{col}_n\) can be indexed from \(-\text{mupper}\) to \(\text{mlower}\). For large problems, it is more efficient to use \(\text{SM\_COLUMN\_B}\) and \(\text{SM\_COLUMN\_ELEMENT\_B}\) than to use the \(\text{SM\_ELEMENT\_B}\) 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 §7.6.

sparse:

A user-supplied sparse Jacobian function must load the \(\text{Neq} \times \text{Neq}\) compressed-sparse-column or compressed-sparse-row matrix \(J\) with an approximation to the Jacobian matrix \(J(t, y, \dot{y})\) at the point \((tt, yy, yp)\). Storage for \(J\) already exists on entry to this function, although the user should ensure that sufficient space is allocated in \(J\) 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 \texttt{SM\_NNZ\_S} or the routine \texttt{SUNSparseMatrix\_NNZ}. The SUNMATRIX\_SPARSE type and accessor macros are documented in §7.7.

The previous function type \texttt{IDADlsJacFn} is identical to \texttt{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 \texttt{NULL}-valued SUNMATRIX was supplied to \texttt{IDASetLinearSolver}), the user may provide a function of type \texttt{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.

```c
typedef int (*IDALsJacTimesVecFn)(realtype tt, N\_Vector yy, N\_Vector yp, N\_Vector rr, N\_Vector v, N\_Vector Jv, realtype cj, void *user\_data, N\_Vector tmp1, N\_Vector tmp2);
```

**Definition**

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

**Purpose**

This function computes the product \(Jv\) of the DAE system Jacobian \(J\) (or an approximation to it) and a given vector \(v\), 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 \texttt{user\_data} parameter passed to \texttt{IDASetUserData}.
- \(tmp1\)
- \(tmp2\) are pointers to memory allocated for variables of type N\_Vector which can be used by \texttt{IDALsJacTimesVecFn} 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**

This function must return a value of \(J \ast v\) that uses the \texttt{current} value of \(J\), i.e. as evaluated at the current \((t, y, \dot{y})\).

If the user's \texttt{IDALsJacTimesVecFn} function uses difference quotient approximations, it may need to access quantities not in the call list. These include the current stepsizes, the error weights, etc. To obtain these, the user will need to add a pointer to \texttt{ida\_mem} to \texttt{user\_data} and then use the \texttt{IDAGet\_\*} functions described in §4.5.10.2. The unit roundoff can be accessed as \texttt{UNIT\_ROUNDOFF} defined in \texttt{sundials\_types.h}.

The previous function type \texttt{IDASpilsJacTimesVecFn} is identical to \texttt{IDALsJacTimesVecFn}, 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.
4.6.7 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:

```
typedef int (IDALsJacTimesSetupFn)(realtype tt, N_Vector yy,
                                     N_Vector yp, N_Vector rr,
                                     realtype cj, void *user_data);
```

**Purpose**
This function preprocesses and/or evaluates Jacobian data needed by the Jacobian-times-vector routine.

**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 \( F(t, y, \dot{y}) \).
- `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 `user_data` parameter passed to IDASetUserData.

**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**
Each call to the Jacobian-vector setup function is preceded by a call to the IDAResFn user function with the same \((t, y, \dot{y})\) arguments. Thus, the setup function can use any auxiliary data that is computed and saved during the evaluation of the DAE residual.

If the user’s IDALsJacTimesVecFn 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 `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.

The previous function type IDASpilsJacTimesSetupFn is identical to IDALsJacTimesSetupFn, 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.

4.6.8 Preconditioner solve (iterative linear solvers)

If a user-supplied preconditioner is to be used with a SUNLINSOL solver module, then the user must provide a function to solve the linear system \( Pz = r \) where \( P \) is a left preconditioner matrix which approximates (at least crudely) the Jacobian matrix \( J = \partial F/\partial y + cj \partial F/\partial \dot{y} \). This function must be of type IDALsPrecSolveFn, defined as follows:

```
typedef int (IDALsPrecSolveFn)(realtype tt, N_Vector yy,
                               N_Vector yp, N_Vector rr,
                               N_Vector rvec, N_Vector zvec,
                               realtype cj, realtype delta,
                               void *user_data);
```

**Purpose**
This function solves the preconditioning system \( Pz = r \).
**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 \( F(t, y, \dot{y}) \).
- \( rvec \) is the right-hand side vector \( r \) of the linear system to be solved.
- \( zvec \) 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) ).
- \( delta \) is an input tolerance to be used if an iterative method is employed in the solution. In that case, the residual vector \( Res = r - Pz \) of the system should be made less than \( delta \) in weighted \( l_2 \) norm, i.e., \( \sqrt{\sum_i (Res_i \cdot ewt_i)^2} < delta \). To obtain the \( N\_Vector \ ewt \), call IDAGetErrWeights (see §4.5.10.2).
- \( user\_data \) is a pointer to user data, the same as the \( user\_data \) parameter passed to the function IDASetUserData.

**Return value**
The value to be returned by the preconditioner solve function is a flag indicating whether it was successful. This value should be 0 if successful, positive for a recoverable error (in which case the step will be retried), negative for an unrecoverable error (in which case the integration is halted).

**Notes**
The previous function type IDASpilsPrecSolveFn is identical to IDALsPrecSolveFn, 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.

### 4.6.9 Preconditioner setup (iterative linear solvers)

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

**Definition**
```c
typedef int (*IDALsPrecSetupFn)(realtype tt, N_Vector yy,
                                 N_Vector yp, N_Vector rr,
                                 realtype cj, void *user_data);
```

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

**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 \( F(t, y, \dot{y}) \).
- \( cj \) 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 \( user\_data \) parameter passed to the function IDASetUserData.

**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 IDAResFn user function with the same (\( tt, yy, yp \)) 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 IDALsPrecSetupFn 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 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.

The previous function type IDASpilsPrecSetupFn is identical to IDALsPrecSetupFn, 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 [29] 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 assigned to one of the \( M \) processors to be used to solve the DAE system. The basic idea is to isolate the preconditioning so that it is local to each processor, and also to use a (possibly cheaper) approximate residual function. This requires the definition of a new function \( G(t, y, \dot{y}) \) which approximates the function \( F(t, y, \dot{y}) \) in the definition of the DAE system (2.1). However, the user may set \( G = F \). Corresponding to the domain decomposition, there is a decomposition of the solution vectors \( y \) and \( \dot{y} \) into \( M \) disjoint blocks \( y_m \) and \( \dot{y}_m \), and a decomposition of \( G \) into blocks \( G_m \). The block \( G_m \) depends on \( y_m \) and \( \dot{y}_m \), and also on components of \( y_{m'} \) and \( \dot{y}_{m'} \) associated with neighboring sub-domains (so-called ghost-cell data). Let \( \bar{y}_m \) and \( \dot{\bar{y}}_m \) denote \( y_m \) and \( \dot{y}_m \) (respectively) augmented with those other components on which \( G_m \) depends. Then we have

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

and each of the blocks \( G_m(t, \bar{y}_m, \dot{\bar{y}}_m) \) is uncoupled from the others.

The preconditioner associated with this decomposition has the form

\[
P = \text{diag}[P_1, P_2, \ldots, P_M] \tag{4.2}
\]

where

\[
P_m \approx \partial G_m / \partial y_m + \alpha \partial G_m / \partial \dot{y}_m \tag{4.3}
\]

This matrix is taken to be banded, with upper and lower half-bandwidths \texttt{mudq} and \texttt{mldq} defined as the number of non-zero diagonals above and below the main diagonal, respectively. The difference
quotient approximation is computed using $\text{mudq} + \text{mldq} + 2$ evaluations of $G_m$, but only a matrix of bandwidth $\text{mukeep} + \text{mlkeep} + 1$ is retained.

Neither pair of parameters need be the true half-bandwidths of the Jacobians of the local block of $G$, if smaller values provide a more efficient preconditioner. Such an efficiency gain may occur if the couplings in the DAE system outside a certain bandwidth are considerably weaker than those within the band. Reducing $\text{mukeep}$ and $\text{mlkeep}$ while keeping $\text{mudq}$ and $\text{mldq}$ at their true values, discards the elements outside the narrower band. Reducing both pairs has the additional effect of lumping the outer Jacobian elements into the computed elements within the band, and requires more caution and experimentation.

The solution of the complete linear system

$$ Px = b $$

reduces to solving each of the equations

$$ P_m x_m = b_m $$

and this is done by banded LU factorization of $P_m$ followed by a banded backsolve.

Similar block-diagonal preconditioners could be considered with different treatment of the blocks $P_m$. For example, incomplete LU factorization or an iterative method could be used instead of banded LU factorization.

The idabbdpre module calls two user-provided functions to construct $P$: a required function $\text{Gres}$ (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 $\text{Gcomm}$ (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 $\text{res}$. Both functions take as input the same pointer $\text{user\_data}$ as passed by the user to IDASetUserData and passed to the user’s function $\text{res}$. The user is responsible for providing space (presumably within $\text{user\_data}$) for components of $yy$ and $yp$ that are communicated by $\text{Gcomm}$ from the other processors, and that are then used by $\text{Gres}$, which should not do any communication.

### IDABBDLocalFn

**Definition**

```c
typedef int (*IDABBDLocalFn)(sunindextype Nlocal, realtype tt,
                          N_Vector yy, N_Vector yp, N_Vector gval,
                          void *user_data);
```

**Purpose**

This $\text{Gres}$ function computes $G(t, y, \dot{y})$. It loads the vector $gval$ as a function of $tt$, $yy$, and $yp$.

**Arguments**

- $Nlocal$ is the local vector length.
- $tt$ is the value of the independent variable.
- $yy$ is the dependent variable.
- $yp$ is the derivative of the dependent variable.
- $gval$ is the output vector.
- $\text{user\_data}$ is a pointer to user data, the same as the $\text{user\_data}$ parameter passed to IDASetUserData.

**Return value**

An IDABBDLocalFn 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 $gval$ has already been done, and this data is accessible within $\text{user\_data}$.

The case where $G$ is mathematically identical to $F$ is allowed.
4.7 A parallel band-block-diagonal preconditioner module

IDABBDCommFn

Definition typedef int (*IDABBDCommFn)(sunindextype Nlocal, realtype tt, N_Vector yy, N_Vector yp, void *user_data);

Purpose This Gcomm function performs all inter-processor communications necessary for the execution of the Gres function above, using the input vectors yy and yp.

Arguments Nlocal is the local vector length.
        tt is the value of the independent variable.
        yy is the dependent variable.
        yp is the derivative of the dependent variable.
        user_data is a pointer to user data, the same as the user_data parameter passed to IDASetUserData.

Return value An IDABBDCommFn function type should return 0 to indicate success, 1 for a recoverable error, or -1 for a non-recoverable error.

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

Each call to the Gcomm function is preceded by a call to the residual function res with the same (tt, yy, yp) arguments. Thus Gcomm can omit any communications done by res if relevant to the evaluation of Gres. If all necessary communication was done in res, then Gcomm = NULL can be passed in the call to IDABBDPreInit (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.

[ IDABBDPrecInit ]

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

Description
The function IDABBDPrecInit initializes and allocate (internal) memory for the ID- 
ABBDPRE preconditioner.

Arguments
ida_mem (void *) pointer to the IDA memory block.
Nlocal (sunindextype) local vector dimension.
mudq (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.
mukeep (sunindextype) upper half-bandwidth of the retained banded approximate 
      Jacobian block.
mlkeep (sunindextype) lower half-bandwidth of the retained banded approximate 
      Jacobian block.
dq_rel_yy (realtype) the relative increment in components of y used in the difference 
      quotient approximations. The default is dq_rel_yy = \sqrt{\text{unit roundoff}}, which 
      can be specified by passing dq_rel_yy = 0.0.
Gres (IDABBDLocalFn) the C function which computes the local residual approx- 
      imation G(t, y, \dot{y}).
4.7 A parallel band-block-diagonal preconditioner module

\(G_{\text{comm}}\) (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 \(\text{flag}\) (of type \(\text{int}\)) is one of

- IDALS_SUCCESS The call to IDABBDPrecInit was successful.
- IDALS_MEM_NULL The \(\text{ida\_mem}\) pointer was NULL.
- IDALS_MEM_FAIL A memory allocation request has failed.
- IDALS_LMEM_NULL An IDALS linear solver memory was not attached.
- IDALS_ILL_INPUT The supplied vector implementation was not compatible with the block band preconditioner.

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

The half-bandwidths \(\text{mudq}\) and \(\text{mldq}\) 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 \(\text{mukeep}\) and \(\text{mlkeep}\) 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 \(\text{local\_N}\), \(\text{mukeep}\), or \(\text{mlkeep}\). 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 \(\text{mudq}\) and \(\text{mldq}\) used in the difference-quotient Jacobian approximations, the relative increment \(\text{dq\_rel\_yy}\), or one of the user-supplied functions \(G_{\text{res}}\) and \(G_{\text{comm}}\). 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).

\[
\text{IDABBDPrecReInit}
\]

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

Description The function IDABBDPrecReInit reinitializes the IDABBDPRE preconditioner.

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

- \(\text{mudq}\) (sunindextype) upper half-bandwidth to be used in the difference-quotient Jacobian approximation.
- \(\text{mldq}\) (sunindextype) lower half-bandwidth to be used in the difference-quotient Jacobian approximation.
- \(\text{dq\_rel\_yy}\) (realltype) 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 \(\text{flag}\) (of type \(\text{int}\)) is one of

- IDALS_SUCCESS The call to IDABBDPrecReInit was successful.
- IDALS_MEM_NULL The \(\text{ida\_mem}\) 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 \(\text{mudq}\) or \(\text{mldq}\) is negative or exceeds the value \(\text{Nlocal}-1\), it is replaced by 0 or \(\text{Nlocal}-1\), accordingly.

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

Call

```c
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

```c
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 NVVECTOR 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_spbcs) 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.
### 5.3 FIDA routines

- Interface to the main IDA module
  - FIDAMALLOC interfaces to IDACreate, IDASSetUserData, IDAInit, IDASStolerances, and IDAVtolerances.
  - FIDAREINIT interfaces to IDAREInit and IDASStolerances/IDAVtolerances.
  - FIDASETIN, FIDASETVIN, and FIDASETTRIN interface to IDASet* functions.
  - FIDATOLREINIT interfaces to IDASStolerances/IDAVtolerances.
  - FIDACALCIC interfaces to IDACalcIC.
  - FIDAEWTSET interfaces to IDAWFtolerances.
  - FIDASOLVE interfaces to IDASolve, IDASGet* 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 IDASSetLinearSolver.
  - FIDALSSETEPSLIN interfaces to IDASSetEpsLin.
  - FIDALSSETJAC interfaces to IDASSetJacTimes.
  - FIDALSSETPREC interfaces to IDASSetPreconditioner.
  - FIDADENSESETJAC interfaces to IDASSetJacFn.
  - FIDABANDSETJAC interfaces to IDASSetJacFn.
  - FIDASPARSESETJAC interfaces to IDASSetJacFn.

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

<table>
<thead>
<tr>
<th>FIDA routine</th>
<th>IDA function</th>
<th>IDA type of interface function</th>
</tr>
</thead>
<tbody>
<tr>
<td>(FORTRAN, user-supplied)</td>
<td>(C, interface)</td>
<td></td>
</tr>
<tr>
<td>FIDARESFUN</td>
<td>FIDAREsf</td>
<td>IDAResFn</td>
</tr>
<tr>
<td>FIDAETW</td>
<td>FIDAEwtSet</td>
<td>IDAEwtFn</td>
</tr>
<tr>
<td>FIDADJAC</td>
<td>FIDADenseJac</td>
<td>IDASJacFn</td>
</tr>
<tr>
<td>FIDABJAC</td>
<td>FIDABandJac</td>
<td>IDASJacFn</td>
</tr>
<tr>
<td>FIDASPJAC</td>
<td>FIDASparseJac</td>
<td>IDASJacFn</td>
</tr>
<tr>
<td>FIDAPSOL</td>
<td>FIDAPSol</td>
<td>IDASPrecSolveFn</td>
</tr>
<tr>
<td>FIDAPSET</td>
<td>FIDAPSet</td>
<td>IDASPrecSetupFn</td>
</tr>
<tr>
<td>FIDAJTIMES</td>
<td>FIDAJtimes</td>
<td>IDASJacTimesVecFn</td>
</tr>
<tr>
<td>FIDAJTSETUP</td>
<td>FIDAJTSetup</td>
<td>IDASJacTimesSetupFn</td>
</tr>
</tbody>
</table>

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

```fortran
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

```fortran
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

```fortran
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

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

\[
\begin{align*}
\text{CALL FSUNKLUSETORDERING(...) } \\
\text{CALL FSUNSUPERLUMTSETORDERING(...) } \\
\text{CALL FSUNPCGSETPRECTYPE(...) } \\
\text{CALL FSUNPFQMRSETGSTYPE(...) } \\
\text{CALL FSUNPFQMRSETPRECTYPE(...) } \\
\text{CALL FSUNSPBCGSSETPRECTYPE(...) } \\
\text{CALL FSUNSPBCGSSETMAXL(...) } \\
\text{CALL FSUNSPFGMRSETPRECTYPE(...) } \\
\text{CALL FSUNSPFGMRSETGSTYPE(...) } \\
\text{CALL FSUNSPTFQMRSETPRECTYPE(...) } \\
\text{CALL FSUNSPTFQMRSETMAXL(...) }
\end{align*}
\]

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:

\[
\text{CALL FIDAMALLOC(T0, Y0, YP0, IATOL, RTOL, ATOL, IOUT, ROUT, IPAR, RPAR, IER)}
\]

**Description** This function provides required problem and solution specifications, specifies optional inputs, allocates internal memory, and initializes IDA.

**Arguments**
- **T0** is the initial value of \(t\).
- **Y0** is an array of initial conditions for \(y\).
- **YP0** is an array of initial conditions for \(\dot{y}\).
- **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.
- **RTOL** is the relative tolerance (scalar).
- **ATOL** is the absolute tolerance (scalar or array).
- **IOUT** is an integer array of length at least 21 for integer optional outputs.
- **ROUT** is a real array of length at least 6 for real optional outputs.
- **IPAR** is an integer array of user data which will be passed unmodified to all user-provided routines.
- **RPAR** is a real array of user data which will be passed unmodified to all user-provided routines.

**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 \texttt{IPAR} and \texttt{RPAR} inside a user-provided routine will be propagated to all subsequent calls to such routines.

The optional outputs associated with the main \texttt{IDA} integrator are listed in Table 5.2.

As an alternative to providing tolerances in the call to \texttt{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:

\begin{verbatim}
SUBROUTINE FIDAEWT (Y, EWT, IPAR, RPAR, IER)
  DIMENSION Y(*), EWT(*), IPAR(*), RPAR(*)

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

  If the \texttt{FIDAEWT} routine is provided, then, following the call to \texttt{FIDAMALLOC}, the user must make the call:

  \texttt{CALL FIDAEWTSET (FLAG, IER)}
\end{verbatim}

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

6. \textbf{Set optional inputs}

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

7. \textbf{Linear solver interface specification}

The variable-order, variable-coefficient \texttt{BDF} method used by \texttt{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 \texttt{FIDA} must initialize the \texttt{IDALS} linear solver interface. To attach any \texttt{SUNLINSOL} object (and optional \texttt{SUNMATRIX} object) to \texttt{IDA}, then following calls to initialize the \texttt{SUNLINSOL} (and \texttt{SUNMATRIX}) object(s) in steps 3 and 4 above, the user must make the call:

\begin{verbatim}
CALL FIDALSINIT(IER)
\end{verbatim}

\texttt{IER} is an error return flag set on 0 on success or \texttt{-1} if a memory failure occurred.

The previous routines \texttt{FIDADLSINIT} and \texttt{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.

\textbf{IDALS with dense Jacobian matrix}

As an option when using the \texttt{IDALS} interface with the \texttt{SUNLINSOL_DENSE} or \texttt{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:

\begin{verbatim}
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 \texttt{DJAC}. The vectors \texttt{WK1}, \texttt{WK2}, and \texttt{WK3} of length \texttt{NEQ} are provided as work space for use in \texttt{FIDADJAC}. The input arguments \texttt{T}, \texttt{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:

```plaintext
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.

**idals 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:

```plaintext
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:

```plaintext
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.

**idals 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:

```plaintext
SUBROUTINE FIDASPJAC(T, CJ, Y, YP, R, N, NNZ, JDATA, JINDEXVALS, 
                      & JINDEXPTRS, H, IPAR, RPAR, WK1, WK2, WK3, IER)
```
It must load the \( N \times N \) compressed sparse column [or compressed sparse row] matrix with storage for \( NNZ \) nonzeros, stored in the arrays \textit{JDATA} (nonzero values), \textit{JINDEXVALS} (row [or column] indices for each nonzero), \textit{JINDEXPTRS} (indices for start of each row [or column]), with the Jacobian matrix at the current \((t, y)\) in CSC [or CSR] form (see \texttt{sunmatrix_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; \textit{JDATA}, nonzero values in the Jacobian (of length \( NNZ \)); \textit{JINDEXVALS}, row [or column] indices for each nonzero in Jacobian (of length \( NNZ \)); \textit{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 \texttt{FIDAMALLOC}; \( RPAR \), an array containing real user data that was passed to \texttt{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 \texttt{FIDASPJAC} routine has been provided, then following the call to \texttt{FIDALSINIT}, the following call must be made

\begin{verbatim}
CALL FIDASPARSESETJAC (IER)
\end{verbatim}

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 \texttt{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:

\begin{verbatim}
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 \( v \) is stored in \( V \), and store the product in \( FJV \). On return, set \( IER = 0 \) if \texttt{FIDAJTIMES} was successful, and nonzero otherwise. The vectors \( WK1 \) and \( WK2 \), of length \( NEQ \), are provided as work space for use in \texttt{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 \texttt{FIDAMALLOC}.

If the user’s \texttt{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 \texttt{ROUT(6)}, passed from the calling program to this routine using \texttt{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:

\begin{verbatim}
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 \texttt{idAYP}. It should compute any necessary data for subsequent calls to \texttt{FIDAJTIMES}. On return, set \( IER = 0 \) if \texttt{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 \texttt{FIDAMALLOC}.

To indicate that the \texttt{FIDAJTIMES} and \texttt{FIDAJTSETUP} routines have been provided, then following the call to \texttt{FIDALSINIT}, the following call must be made
\end{verbatim}
CALL FIDALSSETJAC (FLAG, IER)

with FLAG ≠ 0. The return flag 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 IER=0.

**FIDALS 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 \( r = RV \) is input, and store the solution \( z \) in \( ZV \). Here \( P \) is the left preconditioner. 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. On return, set IER = 0 if FIDAPSOL was successful, set IER positive if a recoverable error occurred, and set IER negative if a non-recoverable error occurred.

The arguments \( EWT \) and \( DELTA \) 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 \( DELTA \) in weighted \( \ell_2 \) norm, i.e. \( \sqrt{\sum (\rho_i * EWT[i])^2} < DELTA \). 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 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 \( T, Y, YP, R, \) and \( CJ \) are the current values of \( t, y, \dot{y}, F(t,y,\dot{y}), \) and \( \alpha \), respectively. On return, set IER = 0 if FIDAPSET was successful, set IER positive if a recoverable error occurred, and set IER negative if a non-recoverable error occurred. 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 FIDAPSET 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.

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 \( \text{FLAG} \neq 0 \). The return flag \( \text{IER} \) is 0 if successful, or negative if a memory error occurred. In addition, the user must supply preconditioner routines \text{FIDAPSET} and \text{FIDAPSOL}.

The previous routine \text{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 \text{FIDALSPRETPREC}, the subroutine \text{FIDAPSET} must be provided, even if it is not needed, and it must return \( \text{IER} = 0 \).

8. Correct initial values

Optionally, to correct the initial values \( y \) and/or \( \dot{y} \), make the call

\[
\text{CALL FIDACALCIC (ICOPT, TOUT1, IER)}
\]

(See §2.1 for details.) The arguments are as follows: \( \text{ICOPT} \) is 1 for initializing the algebraic components of \( y \) and differential components of \( \dot{y} \), or 2 for initializing all of \( y \). \( \text{IER} \) is an error return flag, which is 0 for success, or negative for a failure (see \text{IDACalcIC} return values).

9. Problem solution

Carrying out the integration is accomplished by making calls as follows:

\[
\text{CALL FIDASOLVE (TOUT, T, Y, YP, ITASK, IER)}
\]

The arguments are as follows. \( \text{TOUT} \) 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. \( \text{ITASK} \) is a task indicator and should be set to 1 for normal mode (overshoot \( \text{TOUT} \) and interpolate), or to 2 for one-step mode (return after each internal step taken). \( \text{IER} \) 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 \text{IDASolve} returns (see §4.5.7 and §B.2). The current values of the optional outputs are available in \text{IOUT} and \text{ROUT} (see Table 5.2).

10. Additional solution output

After a successful return from \text{FIDASOLVE}, the routine \text{FIDAGETDKY} may be called to get interpolated values of \( y \) or any derivative \( \frac{d^k y}{dt^k} \) for \( k \) not exceeding the current method order, and for any value of \( t \) in the last internal step taken by \text{ida}. The call is as follows:

\[
\text{CALL FIDAGETDKY (T, K, DKY, IER)}
\]

where \( T \) is the input value of \( t \) at which solution derivative is desired, \( K \) is the derivative order, and \( \text{DKY} \) is an array containing the computed vector \( y^{(K)}(t) \) on return. The value of \( T \) must lie between \( \text{TCUR} - \text{HLAST} \) and \( \text{TCUR} \). The value of \( K \) must satisfy \( 0 \leq K \leq \text{QLAST} \). (See the optional outputs for \text{TCUR}, \text{HLAST}, and \text{QLAST}.) The return flag \( \text{IER} \) is set to 0 upon successful return, or to a negative value to indicate an illegal input.

11. Problem reinitialization

To re-initialize the \text{ida} solver for the solution of a new problem of the same size as one already solved, make the following call:

\[
\text{CALL FIDAREINIT (T0, Y0, YPO, IATOL, RTOL, ATOL, IER)}
\]

The arguments have the same names and meanings as those of \text{FIDAMALLOC}. \text{FIDAREINIT} performs the same initializations as \text{FIDAMALLOC}, but does no memory allocation, using instead the existing internal memory created by the previous \text{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: FIDASETIIIN 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 FIDASETIIIN(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 FIDASETIIIN 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)

<table>
<thead>
<tr>
<th>Key</th>
<th>Optional input</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX_ORD</td>
<td>Maximum LMM method order</td>
<td>5</td>
</tr>
<tr>
<td>MAX_NSTEPS</td>
<td>Maximum no. of internal steps before $t_{out}$</td>
<td>500</td>
</tr>
<tr>
<td>MAX_ERRFAIL</td>
<td>Maximum no. of error test failures</td>
<td>10</td>
</tr>
<tr>
<td>MAX_NITERS</td>
<td>Maximum no. of nonlinear iterations</td>
<td>4</td>
</tr>
<tr>
<td>MAX_CONVFAIL</td>
<td>Maximum no. of convergence failures</td>
<td>10</td>
</tr>
<tr>
<td>SUPPRESS_ALG</td>
<td>Suppress alg. vars. from error test (1 = SUNTRUE)</td>
<td>0 (= SUNFALSE)</td>
</tr>
<tr>
<td>MAX_NSTEPS_IC</td>
<td>Maximum no. of steps for IC calc.</td>
<td>5</td>
</tr>
<tr>
<td>MAX_NITERS_IC</td>
<td>Maximum no. of Newton iterations for IC calc.</td>
<td>10</td>
</tr>
<tr>
<td>MAX_NJE_IC</td>
<td>Maximum no. of Jac. evals fo IC calc.</td>
<td>4</td>
</tr>
<tr>
<td>LS_OFF_IC</td>
<td>Turn off line search (1 = SUNTRUE)</td>
<td>0 (= SUNFALSE)</td>
</tr>
</tbody>
</table>

### Real optional inputs (FIDASETRIN)

<table>
<thead>
<tr>
<th>Key</th>
<th>Optional input</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>INIT_STEP</td>
<td>Initial step size</td>
<td>estimated</td>
</tr>
<tr>
<td>MAX_STEP</td>
<td>Maximum absolute step size</td>
<td>$\infty$</td>
</tr>
<tr>
<td>STOP_TIME</td>
<td>Value of $t_{stop}$</td>
<td>undefined</td>
</tr>
<tr>
<td>NLCONV_COEF</td>
<td>Coeff. in the nonlinear conv. test</td>
<td>0.33</td>
</tr>
<tr>
<td>NLCONV_COEF_IC</td>
<td>Coeff. in the nonlinear conv. test for IC calc.</td>
<td>0.0033</td>
</tr>
<tr>
<td>STEP_TOL_IC</td>
<td>Lower bound on Newton step for IC calc.</td>
<td>$\text{uround}^{2/3}$</td>
</tr>
</tbody>
</table>

### Real vector optional inputs (FIDASETVIN)

<table>
<thead>
<tr>
<th>Key</th>
<th>Optional input</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID_VEC</td>
<td>Differential/algebraic component types</td>
<td>undefined</td>
</tr>
<tr>
<td>CONSTR_VEC</td>
<td>Inequality constraints on solution</td>
<td>undefined</td>
</tr>
</tbody>
</table>
Table 5.2: Description of the FIDA optional output arrays IOUT and ROUT

### Integer output array IOUT

<table>
<thead>
<tr>
<th>Index</th>
<th>Optional output</th>
<th>IDA function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LENRW</td>
<td>IDAGetWorkSpace</td>
</tr>
<tr>
<td>2</td>
<td>LENIW</td>
<td>IDAGetWorkSpace</td>
</tr>
<tr>
<td>3</td>
<td>NST</td>
<td>IDAGetNumSteps</td>
</tr>
<tr>
<td>4</td>
<td>NRE</td>
<td>IDAGetNumResEvals</td>
</tr>
<tr>
<td>5</td>
<td>NETF</td>
<td>IDAGetNumErrTestFails</td>
</tr>
<tr>
<td>6</td>
<td>NNCFAILS</td>
<td>IDAGetNonlinSolvConvFails</td>
</tr>
<tr>
<td>7</td>
<td>NNI</td>
<td>IDAGetNumNonlinSolvIters</td>
</tr>
<tr>
<td>8</td>
<td>NSETUPS</td>
<td>IDAGetNumLinSolvSetups</td>
</tr>
<tr>
<td>9</td>
<td>QLAST</td>
<td>IDAGetLastOrder</td>
</tr>
<tr>
<td>10</td>
<td>QCUR</td>
<td>IDAGetCurrentOrder</td>
</tr>
<tr>
<td>11</td>
<td>NBACKTRKOPS</td>
<td>IDAGetNumBacktrackOps</td>
</tr>
<tr>
<td>12</td>
<td>NGE</td>
<td>IDAGetNumGEvals</td>
</tr>
</tbody>
</table>

### IDALS linear solver interface

<table>
<thead>
<tr>
<th>Index</th>
<th>Optional output</th>
<th>IDA function</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>LENRWLS</td>
<td>IDAGetLinWorkSpace</td>
</tr>
<tr>
<td>14</td>
<td>LENIWLS</td>
<td>IDAGetLinWorkSpace</td>
</tr>
<tr>
<td>15</td>
<td>LS_FLAG</td>
<td>IDAGetLastLinFlag</td>
</tr>
<tr>
<td>16</td>
<td>NRELS</td>
<td>IDAGetNumLinResEvals</td>
</tr>
<tr>
<td>17</td>
<td>NJE</td>
<td>IDAGetNumJacEvals</td>
</tr>
<tr>
<td>18</td>
<td>NJITS</td>
<td>IDAGetNumJtimesEvals</td>
</tr>
<tr>
<td>19</td>
<td>NJT</td>
<td>IDAGetNumJtimesEvals</td>
</tr>
<tr>
<td>20</td>
<td>NPE</td>
<td>IDAGetNumPrecEvals</td>
</tr>
<tr>
<td>21</td>
<td>NPS</td>
<td>IDAGetNumPrecSolves</td>
</tr>
<tr>
<td>22</td>
<td>NLI</td>
<td>IDAGetNumLinIters</td>
</tr>
<tr>
<td>23</td>
<td>NCFL</td>
<td>IDAGetNumLinConvFails</td>
</tr>
</tbody>
</table>

### Real output array ROUT

<table>
<thead>
<tr>
<th>Index</th>
<th>Optional output</th>
<th>IDA function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>H0_USED</td>
<td>IDAGetActualInitStep</td>
</tr>
<tr>
<td>2</td>
<td>HLAST</td>
<td>IDAGetLastStep</td>
</tr>
<tr>
<td>3</td>
<td>HCUR</td>
<td>IDAGetCurrentStep</td>
</tr>
<tr>
<td>4</td>
<td>TCUR</td>
<td>IDAGetCurrentTime</td>
</tr>
<tr>
<td>5</td>
<td>TOLFACT</td>
<td>IDAGetTolScaleFactor</td>
</tr>
<tr>
<td>6</td>
<td>UROUND</td>
<td>unit roundoff</td>
</tr>
</tbody>
</table>
To obtain the estimated local errors, following a successful call to \textbf{FIDASOLVE}, make the following call:

\texttt{CALL FIDAGETESTLOCALERR (ELE, IER)}

This computes the \texttt{ELE} array of estimated local errors as of the last step taken. The array \texttt{ELE} must already have been declared by the user. The error return flag \texttt{IER} is zero if successful, and negative if there was a memory error.

### 5.6 Usage of the FIDAROOT interface to rootfinding

The \texttt{FIDAROOT} interface package allows programs written in Fortran to use the rootfinding feature of the \texttt{IDA} solver module. The user-callable functions in \texttt{FIDAROOT}, with the corresponding \texttt{IDA} functions, are as follows:

- \texttt{FIDAROOTINIT} interfaces to \texttt{IDARootInit}.
- \texttt{FIDAROOTINFO} interfaces to \texttt{IDAGetRootInfo}.
- \texttt{FIDAROOTFREE} interfaces to \texttt{IDARootFree}.

Note that, at this time \texttt{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 \texttt{INFO} returned by \texttt{FIDAROOTINFO}).

In order to use the rootfinding feature of \texttt{IDA}, the following call must be made, after calling \texttt{FIDAMALLOC} but prior to calling \texttt{FIDASOLVE}, to allocate and initialize memory for the \texttt{FIDAROOT} module:

\texttt{CALL FIDAROOTINIT (NRTFN, IER)}

The arguments are as follows: \texttt{NRTFN} is the number of root functions. \texttt{IER} is a return completion flag; its values are 0 for success, –1 if the \texttt{IDA} memory was \texttt{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:

\texttt{SUBROUTINE FIDAROOTFN (T, Y, YP, G, IPAR, RPAR, IER)}
\texttt{DIMENSION Y(*), YP(*), G(*), IPAR(*), RPAR(*)}

It must set the \texttt{G} array, of length \texttt{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 \texttt{IPAR} (of integers) and \texttt{RPAR} (of reals) contain user data and are the same as those passed to \texttt{FIDAMALLOC}. Set \texttt{IER} on 0 if successful, or on a non-zero value if an error occurred.

When making calls to \texttt{FIDASOLVE} to solve the DAE system, the occurrence of a root is flagged by the return value \texttt{IER} = 2. In that case, if \texttt{NRTFN} > 1, the functions $g_i$ which were found to have a root can be identified by making the following call:

\texttt{CALL FIDAROOTINFO (NRTFN, INFO, IER)}

The arguments are as follows: \texttt{NRTFN} is the number of root functions. \texttt{INFO} is an integer array of length \texttt{NRTFN} with root information. \texttt{IER} is a return completion flag; its values are 0 for success, negative if there was a memory failure. The returned values of \texttt{INFO}(i) (i = 1, \ldots, \texttt{NRTFN}) are 0 or ±1, such that \texttt{INFO}(i) = +1 if $g_i$ was found to have a root and $g_i$ is increasing, \texttt{INFO}(i) = –1 if $g_i$ was found to have a root and $g_i$ is decreasing, and \texttt{INFO}(i) = 0 otherwise.

The total number of calls made to the root function \texttt{FIDAROOTFN}, denoted \texttt{NGE}, can be obtained from \texttt{IOUT(12)}. If the \texttt{FIDA/IDA} memory block is reinitialized to solve a different problem via a call to \texttt{FIDAREINIT}, then the counter \texttt{NGE} is reset to zero.

To free the memory resources allocated by a prior call to \texttt{FIDAROOTINIT}, make the following call:

\texttt{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 NVVECTOR_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):

<table>
<thead>
<tr>
<th>FIDABBD routine (FORTRAN)</th>
<th>IDA function (C)</th>
<th>IDA function type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIDAGLOCFN</td>
<td>FIDAgloc</td>
<td>IDABBDLocalFn</td>
</tr>
<tr>
<td>FIDACOMMFN</td>
<td>FIDAcfn</td>
<td>IDABBDCommFn</td>
</tr>
<tr>
<td>FIDAJTIMES</td>
<td>FIDAJtimes</td>
<td>IDALSJacTimesVecFn</td>
</tr>
<tr>
<td>FIDAJTSETUP</td>
<td>FIDAJTsSetup</td>
<td>IDALSJacTimesSetupFn</td>
</tr>
</tbody>
</table>

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. NVVECTOR module initialization
3. SUNLINSOL module initialization
   - Initialize one of the iterative SUNLINSOL modules, by calling one of FSUNPCGINIT, FSUNSPBCGSINIT, FSUNSPFGMRINIT, FSUNSPQMRINIT 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. $IDABBDDPRE$ 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 $IDABBDDPRE$ module, make the following call:

\begin{verbatim}
CALL FIDABBDOPT (LENRWBBD, LENIWBBD, NGEBBBD)
\end{verbatim}

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. $LENIWBBD$ 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 $IDABBDDPRE$ preconditioner, then the $IDA$ package can be re-initialized for the second and subsequent problems by calling $FIDAREINIT$, following which a call to $FIDABBDDINIT$ may or may not be needed. If the input arguments are the same, no $FIDABBDDINIT$ call is needed. If there is a change in input arguments other than $MU$ or $ML$, then the user program should make the call

\begin{verbatim}
CALL FIDABBDDREINIT (NLOCAL, MUDQ, MLDQ, DQRELY, IER)
\end{verbatim}

This reinitializes the $IDABBDDPRE$ preconditioner, but without reallocating its memory. The arguments of the $FIDABBDDREINIT$ routine have the same names and meanings as those of $FIDABBDDINIT$. If the value of $MU$ or $ML$ is being changed, then a call to $FIDABBDDINIT$ 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 $FIDABBDD$ module is deallocated automatically by $FIDAFREE$.)

14. User-supplied routines

The following two routines must be supplied for use with the $IDABBDDPRE$ module:

\begin{verbatim}
SUBROUTINE FIDALOCFN (NLOC, T, YLOC, YPLOC, GLOC, IPAR, RPAR, IER)
DIMENSION YLOC(*), YPLOC(*), GLOC(*), IPAR(*), RPAR(*)
\end{verbatim}

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 FIDAJTTIMES 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* (*nvgetcommunicator)(N_Vector);
    sunindextype (*nvgetlength)(N_Vector);
    void (*nvlinearsum)(realtype, N_Vector, realtype, N_Vector, N_Vector);
    void (*nvaddsum)(realtype, N_Vector, realtype, N_Vector, 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);
};
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:

```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, 6.4 and 6.5 list optional fused, vector array and local reduction operations, respectively.

Fused and vector array 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 a fused or vector array operation as NULL, the generic nvector module will automatically call standard vector operations as necessary to complete the desired operation. In all sundials-provided nvector implementations, all fused and
vector array operations are disabled by default. However, these implementations provide additional user-callable functions to enable/disable any or all of the fused and vector array operations. See the following sections for the implementation specific functions to enable/disable operations.

Local reduction operations are similarly intended to reduce parallel communication on distributed memory systems, particularly when NVVECTOR objects are combined together within a NVVECTOR_MANYVECTOR object (see Section 6.12). If a particular NVVECTOR implementation defines a local reduction operation as NULL, the NVVECTOR_MANYVECTOR module will automatically call standard vector reduction operations as necessary to complete the desired operation. All SUNDIALS-provided NVVECTOR implementations include these local reduction operations, which may be used as templates for user-defined NVVECTOR implementations.

Finally, note that the generic NVVECTOR 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

\begin{verbatim}
N_Vector *N_VCloneVectorArray(int count, N_Vector w);
N_Vector *N_VCloneVectorArrayEmpty(int count, N_Vector w);
\end{verbatim}

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

\begin{verbatim}
void N_VDestroyVectorArray(N_Vector *vs, int count);
\end{verbatim}

and whose definition is based on the implementation-specific N_VDestroy operation.

A particular implementation of the NVVECTOR module must:

- Specify the content field of N_Vector.

- Define and implement the vector operations. Note that the names of these routines should be unique to that implementation in order to permit using more than one NVVECTOR 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 NVVECTOR implementation included in SUNDIALS has a unique identifier specified in enumeration and shown in Table 6.1. It is recommended that a user-supplied NVVECTOR implementation use the SUNDIALS_NVEC_CUSTOM identifier.
Table 6.1: Vector Identifications associated with vector kernels supplied with SUNDIALS.

<table>
<thead>
<tr>
<th>Vector ID</th>
<th>Vector type</th>
<th>ID Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUNDIALS_NVEC_SERIAL</td>
<td>Serial</td>
<td>0</td>
</tr>
<tr>
<td>SUNDIALS_NVEC_PARALLEL</td>
<td>Distributed memory parallel (MPI)</td>
<td>1</td>
</tr>
<tr>
<td>SUNDIALS_NVEC_OPENMP</td>
<td>OpenMP shared memory parallel</td>
<td>2</td>
</tr>
<tr>
<td>SUNDIALS_NVEC_PTHREADS</td>
<td>PThreads shared memory parallel</td>
<td>3</td>
</tr>
<tr>
<td>SUNDIALS_NVEC_PARHYP</td>
<td>hypre ParHyp parallel vector</td>
<td>4</td>
</tr>
<tr>
<td>SUNDIALS_NVEC_PETSC</td>
<td>PETSc parallel vector</td>
<td>5</td>
</tr>
<tr>
<td>SUNDIALS_NVEC_CUDA</td>
<td>CUDA parallel vector</td>
<td>6</td>
</tr>
<tr>
<td>SUNDIALS_NVEC_PTHREADS</td>
<td>PThreads shared memory parallel</td>
<td>7</td>
</tr>
<tr>
<td>SUNDIALS_NVEC_CUDA</td>
<td>CUDA parallel vector</td>
<td>8</td>
</tr>
<tr>
<td>SUNDIALS_NVEC_PETSC</td>
<td>PETSc parallel vector</td>
<td>9</td>
</tr>
<tr>
<td>SUNDIALS_NVEC_MANYVECTOR</td>
<td>“ManyVector” vector</td>
<td>10</td>
</tr>
<tr>
<td>SUNDIALS_NVEC_CUSTOM</td>
<td>User-provided custom vector</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 6.2: Description of the NVECTOR operations

<table>
<thead>
<tr>
<th>Name</th>
<th>Usage and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N_VGetVectorID</td>
<td><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.</td>
</tr>
<tr>
<td>N_VClone</td>
<td><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.</td>
</tr>
<tr>
<td>N_VCloneEmpty</td>
<td><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.</td>
</tr>
<tr>
<td>N_VDestroy</td>
<td><code>N_VDestroy(v);</code> Destroys the <code>N_Vector</code> <code>v</code> and frees memory allocated for its internal data.</td>
</tr>
<tr>
<td>N_VSpace</td>
<td><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.</td>
</tr>
<tr>
<td>Name</td>
<td>Usage and Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>N_VGetArrayPointer</td>
<td>vdata = N_VGetArrayPointer(v); Returns a pointer to a realtype array from the N_Vector v. Note that this assumes that the internal data in N_Vector is a contiguous array of realtype. This routine is only used in the solver-specific interfaces to the dense and banded (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.</td>
</tr>
<tr>
<td>N_VSetArrayPointer</td>
<td>N_VSetArrayPointer(vdata, v); Overwrites the data in an N_Vector with a given array of realtype. Note that this assumes that the internal data in N_Vector is a contiguous array of realtype. This routine is only used in the interfaces to the dense (serial) linear solver, hence need not exist in a user-supplied NVVECTOR module for a parallel environment.</td>
</tr>
<tr>
<td>N_VGetCommunicator</td>
<td>N_VGetCommunicator(v); Returns a pointer to the MPI_Comm object associated with the vector (if applicable). For MPI-unaware vector implementations, this should return NULL.</td>
</tr>
<tr>
<td>N_VGetLength</td>
<td>N_VGetLength(v); Returns the global length (number of ‘active’ entries) in the NVVECTOR v. This value should be cumulative across all processes if the vector is used in a parallel environment. If v contains additional storage, e.g., for parallel communication, those entries should not be included.</td>
</tr>
<tr>
<td>N_VLinearSum</td>
<td>N_VLinearSum(a, x, b, y, z); Performs the operation ( z = ax + by ), where ( a ) and ( b ) are realtype scalars and ( x ) and ( y ) are of type N_Vector: ( z_i = ax_i + by_i ), ( i = 0, \ldots, n-1 ).</td>
</tr>
<tr>
<td>N_VConst</td>
<td>N_VConst(c, z); Sets all components of the N_Vector ( z ) to realtype ( c ): ( z_i = c, i = 0, \ldots, n-1 ).</td>
</tr>
<tr>
<td>N_VProd</td>
<td>N_VProd(x, y, z); Sets the N_Vector ( z ) to be the component-wise product of the N_Vector inputs ( x ) and ( y ): ( z_i = x_i y_i, i = 0, \ldots, n-1 ).</td>
</tr>
<tr>
<td>N_VDiv</td>
<td>N_VDiv(x, y, z); Sets the N_Vector ( z ) to be the component-wise ratio of the N_Vector inputs ( x ) and ( y ): ( z_i = x_i / y_i, i = 0, \ldots, 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.</td>
</tr>
</tbody>
</table>
### Usage and Description

<table>
<thead>
<tr>
<th>Name</th>
<th>Usage and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N_VScale</td>
<td>N_VScale(c, x, z); Scales the N_Vector x by the realtype scalar c and returns the result in z: ( z_i = cx_i, \ i = 0, \ldots, n - 1 ).</td>
</tr>
<tr>
<td>N_VAbs</td>
<td>N_VAbs(x, z); Sets the components of the N_Vector z to be the absolute values of the components of the N_Vector x: ( y_i =</td>
</tr>
<tr>
<td>N_VInv</td>
<td>N_VInv(x, z); Sets the components of the N_Vector z to be the inverses of the components of the N_Vector x: ( z_i = 1.0/x_i, \ i = 0, \ldots, n - 1 ). This routine may not check for division by 0. It should be called only with an x which is guaranteed to have all nonzero components.</td>
</tr>
<tr>
<td>N_VAddConst</td>
<td>N_VAddConst(x, b, z); Adds the realtype scalar b to all components of x and returns the result in the N_Vector z: ( z_i = x_i + b, \ i = 0, \ldots, n - 1 ).</td>
</tr>
<tr>
<td>N_VDotProd</td>
<td>d = N_VDotProd(x, y); Returns the value of the ordinary dot product of x and y: ( d = \sum_{i=0}^{n-1} x_i y_i ).</td>
</tr>
<tr>
<td>N_VMaxNorm</td>
<td>m = N_VMaxNorm(x); Returns the maximum norm of the N_Vector x: ( m = \max_i</td>
</tr>
<tr>
<td>N_VWrmsNorm</td>
<td>m = N_VWrmsNorm(x, w) Returns the weighted root-mean-square norm of the N_Vector x with realtype weight vector w: ( m = \sqrt{\frac{\sum_{i=0}^{n-1} (x_i w_i)^2}{n}} ).</td>
</tr>
<tr>
<td>N_VWrmsNormMask</td>
<td>m = N_VWrmsNormMask(x, w, id); Returns the weighted root mean square norm of the N_Vector x with realtype weight vector w built using only the elements of x corresponding to positive elements of the N_Vector id: ( m = \sqrt{\frac{\sum_{i=0}^{n-1} (x_i w_i H(id_i))^2}{n}} ), where ( H(\alpha) = \begin{cases} 1 &amp; \alpha &gt; 0 \ 0 &amp; \alpha \leq 0 \end{cases} ).</td>
</tr>
<tr>
<td>N_VMin</td>
<td>m = N_VMin(x); Returns the smallest element of the N_Vector x: ( m = \min_i x_i ).</td>
</tr>
<tr>
<td>N_VWL2Norm</td>
<td>m = N_VWL2Norm(x, w); Returns the weighted Euclidean ( \ell_2 ) norm of the N_Vector x with realtype weight vector w: ( m = \sqrt{\sum_{i=0}^{n-1} (x_i w_i)^2} ).</td>
</tr>
<tr>
<td>Name</td>
<td>Usage and Description</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>N_VL1Norm</strong></td>
<td>( m = N_VL1Norm(x); ) Returns the ( \ell_1 ) norm of the ( N_Vector ) ( x ): ( m = \sum_{i=0}^{n-1}</td>
</tr>
<tr>
<td><strong>N_VCompare</strong></td>
<td>( N_VCompare(c, x, z); ) Compares the components of the ( N_Vector ) ( x ) to the \textit{realtype} scalar ( c ) and returns an ( N_Vector ) ( z ) such that: ( z_i = 1.0 ) if (</td>
</tr>
<tr>
<td><strong>N_VInvTest</strong></td>
<td>( t = N_VInvTest(x, z); ) Sets the components of the ( N_Vector ) ( z ) to be the inverses of the components of the ( N_Vector ) ( x ), with prior testing for zero values: ( z_i = 1.0/x_i ), ( i = 0, \ldots, n - 1 ). This routine returns a boolean assigned to \textit{SUNTRUE} if all components of ( x ) are nonzero (successful inversion) and returns \textit{SUNFALSE} otherwise.</td>
</tr>
<tr>
<td><strong>N_VConstrMask</strong></td>
<td>( t = N_VConstrMask(c, x, m); ) Performs the following constraint tests: ( x_i &gt; 0 ) if ( c_i = 2 ), ( x_i \geq 0 ) if ( c_i = 1 ), ( x_i \leq 0 ) if ( c_i = -1 ), ( x_i &lt; 0 ) if ( c_i = -2 ). There is no constraint on ( x_i ) if ( c_i = 0 ). This routine returns a boolean assigned to \textit{SUNFALSE} if any element failed the constraint test and assigned to \textit{SUNTRUE} if all passed. It also sets a mask vector ( m ), with elements equal to 1.0 where the constraint test failed, and 0.0 where the test passed. This routine is used only for constraint checking.</td>
</tr>
<tr>
<td><strong>N_VMinQuotient</strong></td>
<td>( \text{minq} = N_VMinQuotient(num, denom); ) This routine returns the minimum of the quotients obtained by term-wise dividing \textit{num} by \textit{denom}. A zero element in \textit{denom} will be skipped. If no such quotients are found, then the large value \textit{BIG_REAL} (defined in the header file \texttt{sundials_types.h}) is returned.</td>
</tr>
</tbody>
</table>

Table 6.3: Description of the NVector fused operations
<table>
<thead>
<tr>
<th>Name</th>
<th>Usage and Description</th>
</tr>
</thead>
</table>
| N_VLinearCombination  | `ier = N_VLinearCombination(nv, c, X, z);`  <br> This routine computes the linear combination of \( n_v \) vectors with \( n \) elements:  
\[
z_i = \sum_{j=0}^{n_v-1} c_j x_{j,i}, \quad i = 0, \ldots, n - 1,
\]
where \( c \) is an array of \( n_v \) scalars (type `realtype*`), \( X \) is an array of \( n_v \) vectors (type `N_Vector*`), and \( z \) is the output vector (type `N_Vector`).<br> If the output vector \( z \) is one of the vectors in \( X \), then it must be the first vector in the vector array. The operation returns 0 for success and a non-zero value otherwise. |
| N_VScaleAddMulti      | `ier = N_VScaleAddMulti(nv, c, x, Y, Z);`  <br> This routine scales and adds one vector to \( n_v \) vectors with \( n \) elements:  
\[
z_{j,i} = c_j x_i + y_{j,i}, \quad j = 0, \ldots, n_v - 1 \quad i = 0, \ldots, n - 1,
\]
where \( c \) is an array of \( n_v \) scalars (type `realtype*`), \( x \) is the vector (type `N_Vector`) to be scaled and added to each vector in the vector array of \( n_v \) vectors \( Y \) (type `N_Vector*`), and \( Z \) (type `N_Vector*`) is a vector array of \( n_v \) output vectors. The operation returns 0 for success and a non-zero value otherwise. |
| N_VDotProdMulti       | `ier = N_VDotProdMulti(nv, x, Y, d);`  <br> This routine computes the dot product of a vector with \( n_v \) other vectors:  
\[
d_j = \sum_{i=0}^{n-1} x_i y_{j,i}, \quad j = 0, \ldots, n_v - 1,
\]
where \( d \) (type `realtype*`) is an array of \( n_v \) scalars containing the dot products of the vector \( x \) (type `N_Vector`) with each of the \( n_v \) vectors in the vector array \( Y \) (type `N_Vector*`). The operation returns 0 for success and a non-zero value otherwise. |
Table 6.4: Description of the NVECTOR vector array operations

<table>
<thead>
<tr>
<th>Name</th>
<th>Usage and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N_VLinearSumVectorArray</td>
<td>ier = N_VLinearSumVectorArray(nv, a, X, b, Y, Z); This routine computes the linear sum of two vector arrays containing ( n_v ) vectors of ( n ) elements:</td>
</tr>
<tr>
<td></td>
<td>[ z_{j,i} = ax_{j,i} + by_{j,i}, \quad i = 0, \ldots, n - 1 \quad j = 0, \ldots, n_v - 1, ]</td>
</tr>
<tr>
<td></td>
<td>where ( a ) and ( b ) are realtype scalars and ( X, Y, ) and ( Z ) are arrays of ( n_v ) vectors (type N_Vector*). The operation returns 0 for success and a non-zero value otherwise.</td>
</tr>
<tr>
<td>N_VScaleVectorArray</td>
<td>ier = N_VScaleVectorArray(nv, c, X, Z); This routine scales each vector of ( n ) elements in a vector array of ( n_v ) vectors by a potentially different constant:</td>
</tr>
<tr>
<td></td>
<td>[ z_{j,i} = c x_{j,i}, \quad i = 0, \ldots, n - 1 \quad j = 0, \ldots, n_v - 1, ]</td>
</tr>
<tr>
<td></td>
<td>where ( c ) is an array of ( n_v ) scalars (type realtype*) and ( X ) and ( Z ) are arrays of ( n_v ) vectors (type N_Vector*). The operation returns 0 for success and a non-zero value otherwise.</td>
</tr>
<tr>
<td>N_VConstVectorArray</td>
<td>ier = N_VConstVectorArray(nv, c, X); This routine sets each element in a vector of ( n ) elements in a vector array of ( n_v ) vectors to the same value:</td>
</tr>
<tr>
<td></td>
<td>[ z_{j,i} = c, \quad i = 0, \ldots, n - 1 \quad j = 0, \ldots, n_v - 1, ]</td>
</tr>
<tr>
<td></td>
<td>where ( c ) is a realtype scalar and ( X ) is an array of ( n_v ) vectors (type N_Vector*). The operation returns 0 for success and a non-zero value otherwise.</td>
</tr>
</tbody>
</table>
### Description of the NVECTOR module

<table>
<thead>
<tr>
<th>Name</th>
<th>Usage and Description</th>
</tr>
</thead>
</table>
| N_VWrmsNormVectorArray | \( ier = \text{N\_VWrmsNormVectorArray}(nv, X, W, m); \)  
This routine computes the weighted root mean square norm of \( nv \) vectors with \( n \) elements:
\[
m_j = \left( \frac{1}{n} \sum_{i=0}^{n-1} (x_{j,i}w_{j,i})^2 \right)^{1/2}, \quad j = 0,\ldots,n_v - 1,
\]
where \( m \) (type \text{realtype*}) contains the \( n_v \) norms of the vectors in the vector array \( X \) (type \text{N\_Vector*}) with corresponding weight vectors \( W \) (type \text{N\_Vector*}). The operation returns 0 for success and a non-zero value otherwise. |
| N_VWrmsNormMaskVectorArray | \( ier = \text{N\_VWrmsNormMaskVectorArray}(nv, X, W, id, m); \)  
This routine computes the masked weighted root mean square norm of \( nv \) vectors with \( n \) elements:
\[
m_j = \left( \frac{1}{n} \sum_{i=0}^{n-1} (x_{j,i}w_{j,i}H(id_i))^2 \right)^{1/2}, \quad j = 0,\ldots,n_v - 1,
\]
where \( H(id_i) = 1 \) for \( id_i > 0 \) and is zero otherwise, \( m \) (type \text{realtype*}) contains the \( n_v \) norms of the vectors in the vector array \( X \) (type \text{N\_Vector*}) with corresponding weight vectors \( W \) (type \text{N\_Vector*}) and mask vector \( id \) (type \text{N\_Vector}). The operation returns 0 for success and a non-zero value otherwise. |
| N_VScaleAddMultiVectorArray | \( ier = \text{N\_VScaleAddMultiVectorArray}(nv, ns, c, X, YY, ZZ); \)  
This routine scales and adds a vector in a vector array of \( nv \) vectors to the corresponding vector in \( ns \) vector arrays:
\[
z_{j,i} = \sum_{k=0}^{n_s-1} c_k x_{k,j,i}, \quad i = 0,\ldots,n - 1 \quad j = 0,\ldots,n_v - 1,
\]
where \( c \) is an array of \( n_s \) scalars (type \text{realtype*}), \( X \) is a vector array of \( n_v \) vectors (type \text{idN\_Vector*}) to be scaled and added to the corresponding vector in each of the \( n_s \) vector arrays in the array of vector arrays \( YY \) (type \text{N\_Vector**}) and stored in the output array of vector arrays \( ZZ \) (type \text{N\_Vector**}). The operation returns 0 for success and a non-zero value otherwise. |
N_VLinearCombinationVectorArray

This routine computes the linear combination of \( n_s \) vector arrays containing \( n_v \) vectors with \( n \) elements:

\[
z_{j,i} = \sum_{k=0}^{n_s-1} c_k x_{k,j,i}, \quad i = 0, \ldots, n - 1 \quad j = 0, \ldots, n_v - 1,
\]

where \( c \) is an array of \( n_s \) scalars (type \text{realtype} *), \( XX \) (type \text{N_Vector}**) is an array of \( n_s \) vector arrays each containing \( n_v \) vectors to be summed into the output vector array of \( n_v \) vectors \( Z \) (type \text{N_Vector}**). If the output vector array \( Z \) is one of the vector arrays in \( XX \), then it must be the first vector array in \( XX \). The operation returns 0 for success and a non-zero value otherwise.

Table 6.5: Description of the NVECTOR local reduction operations

<table>
<thead>
<tr>
<th>Name</th>
<th>Usage and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N_VDotProdLocal</td>
<td>( d = \text{N_VDotProdLocal}(x, y); ) ( d = \sum_{i=0}^{n_{local}-1} x_i y_i, ) ( n_{local} ) corresponds to the number of components in the vector on this MPI task (or ( n_{local} = n ) for MPI-unaware applications).</td>
</tr>
<tr>
<td>N_VMaxNormLocal</td>
<td>( m = \text{N_VMaxNormLocal}(x); ) ( m = \max_{0 \leq i &lt; n_{local}}</td>
</tr>
<tr>
<td>Name</td>
<td>Usage and Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>N_VMinLocal</td>
<td>This routine computes the smallest element of the MPI task-local portion of the N_Vector $x$: [ m = \min_{0 \leq i &lt; n_{local}} x_i, ] where $n_{local}$ corresponds to the number of components in the vector on this MPI task (or $n_{local} = n$ for MPI-unaware applications).</td>
</tr>
<tr>
<td>N_VL1NormLocal</td>
<td>This routine computes the MPI task-local portion of the $\ell_1$ norm of the N_Vector $x$: [ n = \sum_{i=0}^{n_{local}-1}</td>
</tr>
<tr>
<td>N_VWSqrSumLocal</td>
<td>This routine computes the MPI task-local portion of the weighted squared sum of the N_Vector $x$ with weight vector $w$: [ s = \sum_{i=0}^{n_{local}-1} (x_iw_i)^2, ] where $n_{local}$ corresponds to the number of components in the vector on this MPI task (or $n_{local} = n$ for MPI-unaware applications).</td>
</tr>
<tr>
<td>N_VWSqrSumMaskLocal</td>
<td>This routine computes the MPI task-local portion of the weighted squared sum of the N_Vector $x$ with weight vector $w$ built using only the elements of $x$ corresponding to positive elements of the N_Vector $id$: [ m = \sum_{i=0}^{n_{local}-1} (x_iw_iH(id_i))^2, ] where $H(\alpha) = \begin{cases} 1 &amp; \alpha &gt; 0 \ 0 &amp; \alpha \leq 0 \end{cases}$ and $n_{local}$ corresponds to the number of components in the vector on this MPI task (or $n_{local} = n$ for MPI-unaware applications).</td>
</tr>
</tbody>
</table>
6.1 NVECTOR functions used by IDA

In Table 6.6 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 IDABBBDPRE 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.

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.

### Table 6.6: NVECTOR functions used by IDA

<table>
<thead>
<tr>
<th>Name</th>
<th>Usage and Description</th>
</tr>
</thead>
</table>
| N_VInvTestLocal | \( t = N_{\text{VInvTestLocal}}(x); \)
Sets the MPI task-local components of the N_Vector \( z \) to be the inverses of the components of the N_Vector \( x \), with prior testing for zero values:
\[
z_i = \frac{1.0}{x_i}, \quad i = 0, \ldots, n_{\text{local}} - 1,
\]
where \( n_{\text{local}} \) corresponds to the number of components in the vector on this MPI task (or \( n_{\text{local}} = n \) for MPI-unaware applications). This routine returns a boolean assigned to SUNTRUE if all task-local components of \( x \) are nonzero (successful inversion) and returns SUNFALSE otherwise. |
| N_VConstrMaskLocal | \( t = N_{\text{VConstrMaskLocal}}(c,x,m); \)
Performs the following constraint tests:
\[
x_i > 0 \quad \text{if} \quad c_i = 2,
\]
\[
x_i \geq 0 \quad \text{if} \quad c_i = 1,
\]
\[
x_i \leq 0 \quad \text{if} \quad c_i = -1,
\]
\[
x_i < 0 \quad \text{if} \quad c_i = -2, \text{and}
\]
no test \quad \text{if} \quad c_i = 0,
for all MPI task-local components of the vectors. This routine returns a boolean assigned to SUNFALSE if any task-local element failed the constraint test and assigned to SUNTRUE if all passed. It also sets a mask vector \( m \), with elements equal to 1.0 where the constraint test failed, and 0.0 where the test passed. This routine is used only for constraint checking. |
| N_VMinQuotientLocal | \( \text{minq} = N_{\text{VMinQuotientLocal}}(\text{num},\text{denom}); \)
This routine returns the minimum of the quotients obtained by term-wise dividing \( \text{num} \) by \( \text{denom} \), for all MPI task-local components of the vectors. A zero element in \( \text{denom} \) will be skipped. If no such quotients are found, then the large value BIG_REAL (defined in the header file sundials_types.h) is returned. |
Table 6.6: List of vector functions usage by IDA code modules

<table>
<thead>
<tr>
<th>Function</th>
<th>IDA</th>
<th>IDALS</th>
<th>IDABBPR</th>
<th>FIDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>N_VGetVectorID</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>N_VClone</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>N_VCloneEmpty</td>
<td>1</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VDestroy</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>N_VSpace</td>
<td>✓</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VGetArrayPointer</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>N_VSetArrayPointer</td>
<td>1</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VLinearSum</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VConst</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VProd</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VDiv</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VScale</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VAbs</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VInv</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VAddConst</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VDotProd</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VMaxNorm</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VWrmsNorm</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VMin</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VMinQuotient</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VConstrMask</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VWrmsNormMask</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VCompare</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VLinearCombination</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VScaleAddMulti</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VDotProdMulti</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VLinearSumVectorArray</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VScaleVectorArray</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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, N_VInvTest, N_VGetCommunicator, and N_VGetLength are not used by IDA. Therefore a user-supplied NVeCTOR module for IDA could omit these functions.

6.2 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.

```c
struct _N_VectorContent_Serial {
  sunindextype length;
```
6.2 The NVECTOR_SERIAL implementation

```c
boolantype 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.

### 6.2.1 NVECTOR_SERIAL accessor macros

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:
  ```c
  #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:
  ```c
  #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:
  ```c
  #define NV_Ith_S(v,i) ( NV_DATA_S(v)[i] )
  ```

### 6.2.2 NVECTOR_SERIAL functions

The NVECTOR_SERIAL module defines serial implementations of all vector operations listed in Tables 6.2, 6.3, 6.4 and 6.5. Their names are obtained from those in these tables by appending the suffix _Serial (e.g. N_VDestroy_Serial). All the standard vector operations listed in 6.2 with the suffix _Serial appended are callable via the FORTRAN 2003 interface by prepending an ‘F’ (e.g. FN_VDestroy_Serial).

The module NVECTOR_SERIAL provides the following additional user-callable routines:
Description of the NVECTOR module

N_VNew_Serial
Prototype  N_Vector N_VNew_Serial(sunindextype vec_length);
Description  This function creates and allocates memory for a serial N_Vector. Its only argument is the vector length.
F2003 Name  This function is callable as FN_VNew_Serial when using the Fortran 2003 interface module.

N_VNewEmpty_Serial
Prototype  N_Vector N_VNewEmpty_Serial(sunindextype vec_length);
Description  This function creates a new serial N_Vector with an empty (NULL) data array.
F2003 Name  This function is callable as FN_VNewEmpty_Serial when using the Fortran 2003 interface module.

N_VMake_Serial
Prototype  N_Vector N_VMake_Serial(sunindextype vec_length, realtype *v_data);
Description  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.)
F2003 Name  This function is callable as FN_VMake_Serial when using the Fortran 2003 interface module.

N_VCloneVectorArray_Serial
Prototype  N_Vector *N_VCloneVectorArray_Serial(int count, N_Vector w);
Description  This function creates (by cloning) an array of count serial vectors.

N_VCloneVectorArrayEmpty_Serial
Prototype  N_Vector *N_VCloneVectorArrayEmpty_Serial(int count, N_Vector w);
Description  This function creates (by cloning) an array of count serial vectors, each with an empty (NULL) data array.

N_VDestroyVectorArray_Serial
Prototype  void N_VDestroyVectorArray_Serial(N_Vector *vs, int count);
Description  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.

N_VPrint_Serial
Prototype  void N_VPrint_Serial(N_Vector v);
Description  This function prints the content of a serial vector to stdout.
F2003 Name  This function is callable as FN_VPrint_Serial when using the Fortran 2003 interface module.
The NVECTOR_SERIAL implementation

**N_VPrintFile_Serial**
Prototype: \texttt{void N\_VPrintFile\_Serial(N\_Vector v, FILE \*outfile);}  
Description: This function prints the content of a serial vector to \texttt{outfile}.

By default all fused and vector array operations are disabled in the NVECTOR_SERIAL module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with \texttt{N\_VNew\_Serial}, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using \texttt{N\_VClone}. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with \texttt{N\_VNew\_Serial} will have the default settings for the NVECTOR_SERIAL module.

**N_VEnableFusedOps_Serial**
Prototype: \texttt{int N\_VEnableFusedOps\_Serial(N\_Vector v, boolean type tf);}  
Description: This function enables (\texttt{SUNTRUE}) or disables (\texttt{SUNFALSE}) all fused and vector array operations in the serial vector. The return value is 0 for success and -1 if the input vector or its \texttt{ops} structure are \texttt{NULL}.

**N_VEnableLinearCombination_Serial**
Prototype: \texttt{int N\_VEnableLinearCombination\_Serial(N\_Vector v, boolean type tf);}  
Description: This function enables (\texttt{SUNTRUE}) or disables (\texttt{SUNFALSE}) the linear combination fused operation in the serial vector. The return value is 0 for success and -1 if the input vector or its \texttt{ops} structure are \texttt{NULL}.

**N_VEnableScaleAddMulti_Serial**
Prototype: \texttt{int N\_VEnableScaleAddMulti\_Serial(N\_Vector v, boolean type tf);}  
Description: This function enables (\texttt{SUNTRUE}) or disables (\texttt{SUNFALSE}) the scale and add a vector to multiple vectors fused operation in the serial vector. The return value is 0 for success and -1 if the input vector or its \texttt{ops} structure are \texttt{NULL}.

**N_VEnableDotProdMulti_Serial**
Prototype: \texttt{int N\_VEnableDotProdMulti\_Serial(N\_Vector v, boolean type tf);}  
Description: This function enables (\texttt{SUNTRUE}) or disables (\texttt{SUNFALSE}) the multiple dot products fused operation in the serial vector. The return value is 0 for success and -1 if the input vector or its \texttt{ops} structure are \texttt{NULL}.

**N_VEnableLinearSumVectorArray_Serial**
Prototype: \texttt{int N\_VEnableLinearSumVectorArray\_Serial(N\_Vector v, boolean type tf);}  
Description: This function enables (\texttt{SUNTRUE}) or disables (\texttt{SUNFALSE}) the linear sum operation for vector arrays in the serial vector. The return value is 0 for success and -1 if the input vector or its \texttt{ops} structure are \texttt{NULL}.

**N_VEnableScaleVectorArray_Serial**
Prototype: \texttt{int N\_VEnableScaleVectorArray\_Serial(N\_Vector v, boolean type tf);}  
Description: This function enables (\texttt{SUNTRUE}) or disables (\texttt{SUNFALSE}) the scale operation for vector arrays in the serial vector. The return value is 0 for success and -1 if the input vector or its \texttt{ops} structure are \texttt{NULL}.
Description of the NVVECTOR module

N_VEnableConstVectorArray_Serial
Prototype int N_VEnableConstVectorArray_Serial(N_Vector v, booleantype tf);
Description This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the serial vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableWrmsNormVectorArray_Serial
Prototype int N_VEnableWrmsNormVectorArray_Serial(N_Vector v, booleantype tf);
Description This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the serial vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableWrmsNormMaskVectorArray_Serial
Prototype int N_VEnableWrmsNormMaskVectorArray_Serial(N_Vector v, booleantype tf);
Description This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the serial vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableScaleAddMultiVectorArray_Serial
Prototype int N_VEnableScaleAddMultiVectorArray_Serial(N_Vector v, booleantype tf);
Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the serial vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableLinearCombinationVectorArray_Serial
Prototype int N_VEnableLinearCombinationVectorArray_Serial(N_Vector v, booleantype tf);
Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the serial vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

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 NVVECTOR_SERIAL implementation that have more than one N_Vector argument do not check for consistent internal representation of these vectors. It is the user’s responsibility to ensure that such routines are called with N_Vector arguments that were all created with the same internal representations.
6.2.3 NVECTOR_SERIAL Fortran interfaces

The nvector_serial module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

FORTRAN 2003 interface module

The fnvector_serial_mod FORTRAN module defines interfaces to all nvector_serial C functions using the intrinsic iso_c_binding module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading ‘F’. For example, the function N_VNew_Serial is interfaced as FN_VNew_Serial.

The FORTRAN 2003 NVECTOR_SERIAL interface module can be accessed with the use statement, i.e. use fnvector_serial_mod, and linking to the library lib sundials_fnvectorserial_mod.lib in addition to the C library. For details on where the library and module file fnvector_serial_mod.mod are installed see Appendix A. We note that the module is accessible from the FORTRAN 2003 SUNDIALS integrators without separately linking to the lib sundials_fnvectorserial_mod library.

FORTRAN 77 interface functions

For solvers that include a FORTRAN 77 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.

6.3 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 lib sundials_nvecparallel.lib where .lib is typically .so for shared libraries and .a for static libraries.

6.3.1 NVECTOR_PARALLEL accessor macros

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_len = NV_LOCLENGTH_P(v) sets v_len 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_len = NV_GLOBLENGTH_P(v) sets v_len 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] )

6.3.2 NVECTOR_PARALLEL functions

The NVECTOR_PARALLEL module defines parallel implementations of all vector operations listed in Tables 6.2, 6.3, 6.4, and 6.5. Their names are obtained from those in these tables by appending the suffix _Parallel (e.g. N_VDestroy_Parallel). The module NVECTOR_PARALLEL provides the following additional user-callable routines:

N_VNew_Parallel
Prototype N_Vector N_VNew_Parallel(MPI_Comm comm, sunindextype local_length, sunindextype global_length);
Description This function creates and allocates memory for a parallel vector.

N_VNewEmpty_Parallel
Prototype N_Vector N_VNewEmpty_Parallel(MPI_Comm comm, sunindextype local_length, sunindextype global_length);
Description This function creates a new parallel N_Vector with an empty (NULL) data array.
6.3 The NVECTOR_PARALLEL implementation

**NVMake_Parallel**

Prototype: \( \text{N}_\text{Vector} \text{NVMake_Parallel}(\text{MPI_Commm} \text{ comm}, \text{sunixindexe}_\text{type} \text{ local}_\text{length}, \text{sunixindexe}_\text{type} \text{ global}_\text{length}, \text{realt}_\text{ype} *v\text{data}); \)

Description: This function creates and allocates memory for a parallel vector with user-provided data array. This function does not allocate memory for \(v\text{data} \) itself.

**NVCloneVectorArray_Parallel**

Prototype: \( \text{N}_\text{Vector} *\text{NVCloneVectorArray_Parallel}(\text{int} \text{ count}, \text{N}_\text{Vector} \text{ w}); \)

Description: This function creates (by cloning) an array of \(\text{count}\) parallel vectors.

**NVCloneVectorArrayEmpty_Parallel**

Prototype: \( \text{N}_\text{Vector} *\text{NVCloneVectorArrayEmpty_Parallel}(\text{int} \text{ count}, \text{N}_\text{Vector} \text{ w}); \)

Description: This function creates (by cloning) an array of \(\text{count}\) parallel vectors, each with an empty (NULL) data array.

**NVDestroyVectorArray_Parallel**

Prototype: \( \text{void NVDestroyVectorArray_Parallel}(\text{N}_\text{Vector} *\text{vs}, \text{int} \text{ count}); \)

Description: This function frees memory allocated for the array of \(\text{count}\) variables of type \(\text{N}_\text{Vector}\) created with \(\text{NVCloneVectorArray_Parallel}\) or with \(\text{NVCloneVectorArrayEmpty_Parallel}\).

**NGetLocalLength_Parallel**

Prototype: \( \text{sunixindexe}_\text{type} \text{NGetLocalLength_Parallel}(\text{N}_\text{Vector} \text{ v}); \)

Description: This function returns the local vector length.

**NPrint_Parallel**

Prototype: \( \text{void NPrint_Parallel}(\text{N}_\text{Vector} \text{ v}); \)

Description: This function prints the local content of a parallel vector to \(\text{stdout}\).

**NPrintFile_Parallel**

Prototype: \( \text{void NPrintFile_Parallel}(\text{N}_\text{Vector} \text{ v}, \text{FILE} *\text{outfile}); \)

Description: This function prints the local content of a parallel vector to \(\text{outfile}\).

By default all fused and vector array operations are disabled in the NVECTOR_PARALLEL module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with \(\text{NNew_Parallel}\), enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using \(\text{NClone}\) with that vector. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with \(\text{NNew_Parallel}\) will have the default settings for the NVECTOR_PARALLEL module.
N_VEnableFusedOps_Parallel
Prototype int N_VEnableFusedOps_Parallel(N_Vector v, booleantype tf);
Description This function enables (SUNTRUE) or disables (SUNFALSE) all fused and vector array operations in the parallel vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableLinearCombination_Parallel
Prototype int N_VEnableLinearCombination_Parallel(N_Vector v, booleantype tf);
Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused operation in the parallel vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableScaleAddMulti_Parallel
Prototype int N_VEnableScaleAddMulti_Parallel(N_Vector v, booleantype tf);
Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the parallel vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableDotProdMulti_Parallel
Prototype int N_VEnableDotProdMulti_Parallel(N_Vector v, booleantype tf);
Description This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the parallel vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableLinearSumVectorArray_Parallel
Prototype int N_VEnableLinearSumVectorArray_Parallel(N_Vector v, booleantype tf);
Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the parallel vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableScaleVectorArray_Parallel
Prototype int N_VEnableScaleVectorArray_Parallel(N_Vector v, booleantype tf);
Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the parallel vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableConstVectorArray_Parallel
Prototype int N_VEnableConstVectorArray_Parallel(N_Vector v, booleantype tf);
Description This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the parallel vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.
6.3 The NVECTOR_PARALLEL implementation

### N_VEnableWrmsNormVectorArray_Parallel

**Prototype**

```c
int N_VEnableWrmsNormVectorArray_Parallel(N_Vector v, booleantype tf);
```

**Description**

This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the WRMS norm operation for vector arrays in the parallel vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

### N_VEnableWrmsNormMaskVectorArray_Parallel

**Prototype**

```c
int N_VEnableWrmsNormMaskVectorArray_Parallel(N_Vector v, booleantype tf);
```

**Description**

This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the masked WRMS norm operation for vector arrays in the parallel vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

### N_VEnableScaleAddMultiVectorArray_Parallel

**Prototype**

```c
int N_VEnableScaleAddMultiVectorArray_Parallel(N_Vector v, booleantype tf);
```

**Description**

This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the scale and add a vector array to multiple vector arrays operation in the parallel vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

### N_VEnableLinearCombinationVectorArray_Parallel

**Prototype**

```c
int N_VEnableLinearCombinationVectorArray_Parallel(N_Vector v, booleantype tf);
```

**Description**

This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear combination operation for vector arrays in the parallel vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

**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 rather than 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.

### 6.3.3 NVECTOR_PARALLEL Fortran interfaces

For solvers that include a FORTRAN 77 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.

### 6.4 The NVVECTOR_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 NVVECTOR using OpenMP, called NVVECTOR_OPENMP, and an implementation using Pthreads, called NVVECTOR_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 NVVECTOR implementation provided with SUNDIALS, NVVECTOR_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.

```c
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 FORTRAN module file to use when using the FORTRAN 2003 interface to this module is `fnvector_openmp_mod.mod`.

#### 6.4.1 NVVECTOR_OPENMP accessor macros

The following macros are provided to access the content of an NVVECTOR_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:
  ```c
  #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:
  ```c
  #define NV_OWN_DATA_OMP(v) ( NV_CONTENT_OMP(v)->own_data )
  #define NV_DATA_OMP(v) ( NV_CONTENT_OMP(v)->data )
  ```
### 6.4 The NVECTOR_OPENMP implementation

```c
#define NV_LENGTH_OMP(v) ( NVCONTENT_OMP(v)->length )
#define NV_NUM_THREADS_OMP(v) ( NVCONTENT_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:

  ```c
  #define NV_Ith_OMP(v, i) ( NV_DATA_OMP(v)[i] )
  ```

### 6.4.2 NVECTOR_OPENMP functions

The NVECTOR_OPENMP module defines OpenMP implementations of all vector operations listed in Tables 6.2, 6.3, 6.4, and 6.5. Their names are obtained from those in these tables by appending the suffix _OpenMP_ (e.g. N_VDestroy_OpenMP). All the standard vector operations listed in 6.2 with the suffix _OpenMP_ appended are callable via the FORTRAN 2003 interface by prepending an ‘F’ (e.g. FN_VDestroy_OpenMP).

The module NVECTOR_OPENMP provides the following additional user-callable routines:

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Prototype</th>
<th>Description</th>
<th>F2003 Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>N_New_OpenMP</strong></td>
<td>N_Vector N_New_OpenMP(sunindextype vec_length, int num_threads)</td>
<td>This function creates and allocates memory for a OpenMP N_Vector. Arguments are the vector length and number of threads.</td>
<td>This function is callable as FN_New_OpenMP when using the Fortran 2003 interface module.</td>
</tr>
<tr>
<td><strong>N_NewEmpty_OpenMP</strong></td>
<td>N_Vector N_NewEmpty_OpenMP(sunindextype vec_length, int num_threads)</td>
<td>This function creates a new OpenMP N_Vector with an empty (NULL) data array.</td>
<td>This function is callable as FN_NewEmpty_OpenMP when using the Fortran 2003 interface module.</td>
</tr>
<tr>
<td><strong>N_Make_OpenMP</strong></td>
<td>N_Vector N_Make_OpenMP(sunindextype vec_length, realtype *v_data, int num_threads);</td>
<td>This function creates and allocates memory for a OpenMP vector with user-provided data array. This function does not allocate memory for v_data itself.</td>
<td>This function is callable as FN_Make_OpenMP when using the Fortran 2003 interface module.</td>
</tr>
<tr>
<td><strong>N_CloneVectorArray_OpenMP</strong></td>
<td>N_Vector *N_CloneVectorArray_OpenMP(int count, N_Vector w)</td>
<td>This function creates (by cloning) an array of count OpenMP vectors.</td>
<td></td>
</tr>
</tbody>
</table>
**N_VCloneVectorArrayEmpty_OpenMP**

Prototype: `N_Vector *N_VCloneVectorArrayEmpty_OpenMP(int count, N_Vector w)`

Description: This function creates (by cloning) an array of `count` OpenMP vectors, each with an empty (NULL) data array.

**N_VDestroyVectorArray_OpenMP**

Prototype: `void N_VDestroyVectorArray_OpenMP(N_Vector *vs, int count)`

Description: 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`.

**N_VPrint_OpenMP**

Prototype: `void N_VPrint_OpenMP(N_Vector v)`

Description: This function prints the content of an OpenMP vector to stdout.

F2003 Name: This function is callable as `FN_VPrint_OpenMP` when using the Fortran 2003 interface module.

**N_VPrintFile_OpenMP**

Prototype: `void N_VPrintFile_OpenMP(N_Vector v, FILE *outfile)`

Description: This function prints the content of an OpenMP vector to `outfile`.

By default all fused and vector array operations are disabled in the `nvector_openmp` module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with `N_VNew_OpenMP`, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using `N_VClone`. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with `N_VNew_OpenMP` will have the default settings for the `nvector_openmp` module.

**N_VEnableFusedOps_OpenMP**

Prototype: `int N_VEnableFusedOps_OpenMP(N_Vector v, booleantype tf)`

Description: This function enables (`SUNTRUE`) or disables (`SUNFALSE`) all fused and vector array operations in the OpenMP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

**N_VEnableLinearCombination_OpenMP**

Prototype: `int N_VEnableLinearCombination_OpenMP(N_Vector v, booleantype tf)`

Description: This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear combination fused operation in the OpenMP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

**N_VEnableScaleAddMulti_OpenMP**

Prototype: `int N_VEnableScaleAddMulti_OpenMP(N_Vector v, booleantype tf)`

Description: This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the scale and add a vector to multiple vectors fused operation in the OpenMP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.
6.4 The NVECTOR_OPENMP implementation

**N_VEnableDotProdMulti_OpenMP**
Prototype: `int N_VEnableDotProdMulti_OpenMP(N_Vector v, booleantype tf)`
Description: This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the OpenMP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

**N_VEnableLinearSumVectorArray_OpenMP**
Prototype: `int N_VEnableLinearSumVectorArray_OpenMP(N_Vector v, booleantype tf)`
Description: This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the OpenMP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

**N_VEnableScaleVectorArray_OpenMP**
Prototype: `int N_VEnableScaleVectorArray_OpenMP(N_Vector v, booleantype tf)`
Description: This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the OpenMP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

**N_VEnableConstVectorArray_OpenMP**
Prototype: `int N_VEnableConstVectorArray_OpenMP(N_Vector v, booleantype tf)`
Description: This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the OpenMP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

**N_VEnableWrmsNormVectorArray_OpenMP**
Prototype: `int N_VEnableWrmsNormVectorArray_OpenMP(N_Vector v, booleantype tf)`
Description: This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the OpenMP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

**N_VEnableWrmsNormMaskVectorArray_OpenMP**
Prototype: `int N_VEnableWrmsNormMaskVectorArray_OpenMP(N_Vector v, booleantype tf)`
Description: This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the OpenMP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

**N_VEnableScaleAddMultiVectorArray_OpenMP**
Prototype: `int N_VEnableScaleAddMultiVectorArray_OpenMP(N_Vector v, booleantype tf)`
Description: This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the OpenMP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.
Description of the NVECTOR module

**N_VEnableLinearCombinationVectorArray_OpenMP**

**Prototype**

```c
int N_VEnableLinearCombinationVectorArray_OpenMP(N_Vector v,
                                                 booleantype tf)
```

**Description**
This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the OpenMP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

**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.

### 6.4.3 NVECTOR_OPENMP Fortran interfaces

The NVECTOR_OPENMP module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

**FORTRAN 2003 interface module**

The `nvctor_openmp_mod` FORTRAN module defines interfaces to most NVECTOR_OPENMP C functions using the intrinsic `iso_c_binding` module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading 'F'. For example, the function `N_VNew_OpenMP` is interfaced as `FN_VNew_OpenMP`.

The FORTRAN 2003 NVECTOR_OPENMP interface module can be accessed with the `use` statement, i.e. `use fnvector_openmp_mod`, and linking to the library `libsundials_fnvectoropenmp_mod.lib` in addition to the C library. For details on where the library and module file `fnvector_openmp_mod.mod` are installed see Appendix A.

**FORTRAN 77 interface functions**

For solvers that include a FORTRAN 77 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.

### 6.5 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).

```c
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.

### 6.5.1 NVECTOR_PTHREADS accessor macros

The following macros are provided to access the content of an nvector_pthreads vector. The suffix _PT in the names denotes the Pthreads version.

- **NVCONTENT_PT**
  - This routine gives access to the contents of the Pthreads vector N_Vector.
  - The assignment \( v_{\text{cont}} = \text{NVCONTENT}_{\text{PT}}(v) \) sets \( v_{\text{cont}} \) to be a pointer to the Pthreads N_Vector content structure.
  - Implementation:
    ```c
    #define NVCONTENT_PT(v) ( (N_VectorContent_Pthreads)(v->content) )
    ```

- **NVOWN_DATA_PT, NVDATA_PT, NVLENGTH_PT, NVNUM_THREADS_PT**
  - These macros give individual access to the parts of the content of a Pthreads N_Vector.
  - The assignment \( v_{\text{data}} = \text{NVDATA}_{\text{PT}}(v) \) sets \( v_{\text{data}} \) to be a pointer to the first component of the data for the N_Vector \( v \). The assignment \( \text{NVDATA}_{\text{PT}}(v) = v_{\text{data}} \) sets the component array of \( v \) to be \( v_{\text{data}} \) by storing the pointer \( v_{\text{data}} \).
  - The assignment \( v_{\text{len}} = \text{NVLENGTH}_{\text{PT}}(v) \) sets \( v_{\text{len}} \) to be the length of \( v \). On the other hand, the call \( \text{NVLENGTH}_{\text{PT}}(v) = v_{\text{len}} \) sets the length of \( v \) to be \( v_{\text{len}} \).
  - The assignment \( v_{\text{num}} = \text{NVSUM_THREADS}_{\text{PT}}(v) \) sets \( v_{\text{num}} \) to be the number of threads from \( v \). On the other hand, the call \( \text{NVNUM_THREADS}_{\text{PT}}(v) = v_{\text{num}} \) sets the number of threads for \( v \) to be \( v_{\text{num}} \).
  - Implementation:
    ```c
    #define NVOWN_DATA_PT(v) ( NVCONTENT_PT(v)->own_data )
    #define NVDATA_PT(v) ( NVCONTENT_PT(v)->data )
    #define NVLENGTH_PT(v) ( NVCONTENT_PT(v)->length )
    #define NVNUM_THREADS_PT(v) ( NVCONTENT_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 \( r = \text{NV_Ith}_{\text{PT}}(v,i) \) sets \( r \) to be the value of the \( i \)-th component of \( v \). The assignment \( \text{NV_Ith}_{\text{PT}}(v,i) = r \) 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:
    ```c
    #define NV_Ith_PT(v,i) ( NV_DATA_PT(v)[i] )
    ```
6.5.2 NVECTOR_PTHREADS functions

The nvector_pthreads module defines Pthreads implementations of all vector operations listed in Tables 6.2, 6.3, 6.4, and 6.5. Their names are obtained from those in these tables by appending the suffix _Pthreads (e.g. N_VDestroy_Pthreads). All the standard vector operations listed in 6.2 are callable via the FORTRAN 2003 interface by prepending an ‘F’ (e.g. FN_VDestroy_Pthreads). The module nvector_pthreads provides the following additional user-callable routines:

**N_VNew_Pthreads**

Prototype: `N_Vector N_VNew_Pthreads(sunindextype vec_length, int num_threads)`

Description: This function creates and allocates memory for a Pthreads N_Vector. Arguments are the vector length and number of threads.

F2003 Name: This function is callable as FN_VNew_Pthreads when using the Fortran 2003 interface module.

**N_VNewEmpty_Pthreads**

Prototype: `N_Vector N_VNewEmpty_Pthreads(sunindextype vec_length, int num_threads)`

Description: This function creates a new Pthreads N_Vector with an empty (NULL) data array.

F2003 Name: This function is callable as FN_VNewEmpty_Pthreads when using the Fortran 2003 interface module.

**N_VMake_Pthreads**

Prototype: `N_Vector N_VMake_Pthreads(sunindextype vec_length, realtype *v_data, int num_threads)`

Description: 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.

F2003 Name: This function is callable as FN_VMake_Pthreads when using the Fortran 2003 interface module.

**N_VCloneVectorArray_Pthreads**

Prototype: `N_Vector *N_VCloneVectorArray_Pthreads(int count, N_Vector w)`

Description: This function creates (by cloning) an array of count Pthreads vectors.

**N_VCloneVectorArrayEmpty_Pthreads**

Prototype: `N_Vector *N_VCloneVectorArrayEmpty_Pthreads(int count, N_Vector w)`

Description: This function creates (by cloning) an array of count Pthreads vectors, each with an empty (NULL) data array.

**N_VDestroyVectorArray_Pthreads**

Prototype: `void N_VDestroyVectorArray_Pthreads(N_Vector *vs, int count)`

Description: 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.
6.5 The NVVECTOR_PTHREADS implementation

**N_VPrint_Pthreads**
Prototype: void N_VPrint_Pthreads(N_Vector v)
Description: This function prints the content of a Pthreads vector to stdout.
F2003 Name: This function is callable as FN_VPrint_Pthreads when using the Fortran 2003 interface module.

**N_VPrintFile_Pthreads**
Prototype: void N_VPrintFile_Pthreads(N_Vector v, FILE *outfile)
Description: This function prints the content of a Pthreads vector to outfile.
By default all fused and vector array operations are disabled in the NVVECTOR_PTHREADS module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with N_VNew_Pthreads, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using N_VClone. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enabled/disabled options as the vector they are cloned from while vectors created with N_VNew_Pthreads will have the default settings for the NVVECTOR_PTHREADS module.

**N_VEnableFusedOps_Pthreads**
Prototype: int N_VEnableFusedOps_Pthreads(N_Vector v, booleantype tf)
Description: This function enables (SUNTRUE) or disables (SUNFALSE) all fused and vector array operations in the Pthreads vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

**N_VEnableLinearCombination_Pthreads**
Prototype: int N_VEnableLinearCombination_Pthreads(N_Vector v, booleantype tf)
Description: This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused operation in the Pthreads vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

**N_VEnableScaleAddMulti_Pthreads**
Prototype: int N_VEnableScaleAddMulti_Pthreads(N_Vector v, booleantype tf)
Description: This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the Pthreads vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

**N_VEnableDotProdMulti_Pthreads**
Prototype: int N_VEnableDotProdMulti_Pthreads(N_Vector v, booleantype tf)
Description: This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the Pthreads vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

**N_VEnableLinearSumVectorArray_Pthreads**
Prototype: int N_VEnableLinearSumVectorArray_Pthreads(N_Vector v, booleantype tf)
Description: This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the Pthreads vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.
Description of the NVECTOR module

**N_VEnableScaleVectorArray_Pthreads**

Prototype: int N_VEnableScaleVectorArray_Pthreads(N_Vector v, booleantype tf)

Description: This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the Pthreads vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

**N_VEnableConstVectorArray_Pthreads**

Prototype: int N_VEnableConstVectorArray_Pthreads(N_Vector v, booleantype tf)

Description: This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the Pthreads vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

**N_VEnableWrmsNormVectorArray_Pthreads**

Prototype: int N_VEnableWrmsNormVectorArray_Pthreads(N_Vector v, booleantype tf)

Description: This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the Pthreads vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

**N_VEnableWrmsNormMaskVectorArray_Pthreads**

Prototype: int N_VEnableWrmsNormMaskVectorArray_Pthreads(N_Vector v, booleantype tf)

Description: This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the Pthreads vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

**N_VEnableScaleAddMultiVectorArray_Pthreads**

Prototype: int N_VEnableScaleAddMultiVectorArray_Pthreads(N_Vector v, booleantype tf)

Description: This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the Pthreads vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

**N_VEnableLinearCombinationVectorArray_Pthreads**

Prototype: int N_VEnableLinearCombinationVectorArray_Pthreads(N_Vector v, booleantype tf)

Description: This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the Pthreads vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

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 NVVECTOR_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.

6.5.3 NVVECTOR_PTHREADS Fortran interfaces

The NVVECTOR_PTHREADS module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

FORTRAN 2003 interface module

The nvvector_pthreads_mod FORTRAN module defines interfaces to most NVVECTOR_PTHREADS C functions using the intrinsic iso_c_binding module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading ‘F’. For example, the function NVNew_Pthreads is interfaced as FN_NVNew_Pthreads.

The FORTRAN 2003 NVVECTOR_PTHREADS interface module can be accessed with the use statement, i.e. use fnvector_pthreads_mod, and linking to the library libsundials_fnvectorpthreads_mod.lib in addition to the C library. For details on where the library and module file fnvector_pthreads_mod.mod are installed see Appendix A.

FORTRAN 77 interface functions

For solvers that include a FORTRAN interface module, the NVVECTOR_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.

6.6 The NVVECTOR_PARHYP implementation

The NVVECTOR_PARHYP implementation of the NVVECTOR module provided with SUNDIALS is a wrapper around hypre’s ParVector class. Most of the vector kernels simply call hypre 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 HYPRE_ParVector, an MPI communicator, and a boolean flag own_parvector indicating ownership of the hypre parallel vector object x.

struct _N_VectorContent_ParHyp {
  sunindextype local_length;
  sunindextype global_length;
  booleantype own_parvector;
  MPI_Comm comm;
  HYPRE_ParVector x;
};

The header file to include when using this module is nvvector_parhyp.h. The installed module library to link to is libsundials_nvvecparhyp.lib where .lib is typically .so for shared libraries and .a for static libraries.

Unlike native SUNDIALS vector types, NVVECTOR_PARHYP does not provide macros to access its member variables. Note that NVVECTOR_PARHYP requires SUNDIALS to be built with MPI support.
6.6.1 NVECTOR_PARHYP functions

The NVECTOR_PARHYP module defines implementations of all vector operations listed in Tables 6.2, 6.3, 6.4, and 6.5, except for \texttt{N_VSetArrayPointer} and \texttt{N_VGetArrayPointer}, because accessing raw vector data is handled by low-level hypre 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 hypre vector first, and then use hypre methods to access the data. Usage examples of NVECTOR_PARHYP are provided in the \texttt{cvAdvDiff\_non\_ph.c} example program for CVODE [28] and the \texttt{ark_diurnal\_kry\_ph.c} example program for ARKODE [35].

The names of parhyp methods are obtained from those in Tables 6.2, 6.3, 6.4, and 6.5 by appending the suffix \texttt{ParHyp} (e.g. \texttt{N_VDestroy\_ParHyp}). The module NVECTOR_PARHYP provides the following additional user-callable routines:

\begin{table}
\centering
\begin{tabular}{l}
\textbf{N_VNewEmpty\_ParHyp} \\
Prototype \hspace{1cm} \texttt{N\_Vector N\_VNewEmpty\_ParHyp(MPI\_Comm comm, sunindextype local\_length, sunindextype global\_length)} \\
Description \hspace{1cm} This function creates a new parhyp \texttt{N\_Vector} with the pointer to the hypre vector set to NULL. \\
\end{tabular}
\end{table}

\begin{table}
\centering
\begin{tabular}{l}
\textbf{N_VMMake\_ParHyp} \\
Prototype \hspace{1cm} \texttt{N\_Vector N\_VMake\_ParHyp(HYPRE\_ParVector x)} \\
Description \hspace{1cm} This function creates an \texttt{N\_Vector} wrapper around an existing hypre parallel vector. It does not allocate memory for \texttt{x} itself. \\
\end{tabular}
\end{table}

\begin{table}
\centering
\begin{tabular}{l}
\textbf{N_VGetVector\_ParHyp} \\
Prototype \hspace{1cm} \texttt{HYPRE\_ParVector N\_VGetVector\_ParHyp(N\_Vector v)} \\
Description \hspace{1cm} This function returns the underlying hypre vector. \\
\end{tabular}
\end{table}

\begin{table}
\centering
\begin{tabular}{l}
\textbf{N_VCloneVectorArray\_ParHyp} \\
Prototype \hspace{1cm} \texttt{N\_Vector \*N\_VCloneVectorArray\_ParHyp(int count, N\_Vector w)} \\
Description \hspace{1cm} This function creates (by cloning) an array of count parallel vectors. \\
\end{tabular}
\end{table}

\begin{table}
\centering
\begin{tabular}{l}
\textbf{N_VCloneVectorArrayEmpty\_ParHyp} \\
Prototype \hspace{1cm} \texttt{N\_Vector \*N\_VCloneVectorArrayEmpty\_ParHyp(int count, N\_Vector w)} \\
Description \hspace{1cm} This function creates (by cloning) an array of count parallel vectors, each with an empty (NULL) data array. \\
\end{tabular}
\end{table}

\begin{table}
\centering
\begin{tabular}{l}
\textbf{N_VDestroyVectorArray\_ParHyp} \\
Prototype \hspace{1cm} \texttt{void N\_VDestroyVectorArray\_ParHyp(N\_Vector \*vs, int count)} \\
Description \hspace{1cm} This function frees memory allocated for the array of count variables of type \texttt{N\_Vector} created with \texttt{N\_VCloneVectorArray\_ParHyp} or with \texttt{N\_VCloneVectorArrayEmpty\_ParHyp}. \\
\end{tabular}
\end{table}

\begin{table}
\centering
\begin{tabular}{l}
\textbf{N_VPrint\_ParHyp} \\
Prototype \hspace{1cm} \texttt{void N\_VPrint\_ParHyp(N\_Vector v)} \\
Description \hspace{1cm} This function prints the local content of a parhyp vector to stdout. \\
\end{tabular}
\end{table}
6.6 The NV\textsc{vector\_parhyp} implementation

**N\textsc{vprintfile\_parhyp}**

Prototype: `void N\textsc{vprintfile\_parhyp}(N\textsc{vector} v, FILE *outfile)`

Description: This function prints the local content of a parhyp vector to `outfile`. By default all fused and vector array operations are disabled in the NV\textsc{vector\_parhyp} module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with \textsc{n\_vmake\_parhyp}, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using \textsc{n\_vclone}. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with \textsc{n\_vmake\_parhyp} will have the default settings for the NV\textsc{vector\_parhyp} module.

**N\textsc{venablefusedops\_parhyp}**

Prototype: `int N\textsc{venablefusedops\_parhyp}(N\textsc{vector} v, booleantype tf)`

Description: This function enables (`SUN\textsc{true}`) or disables (`SUN\textsc{false}`) all fused and vector array operations in the parhyp vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

**N\textsc{venablelinearcombination\_parhyp}**

Prototype: `int N\textsc{venablelinearcombination\_parhyp}(N\textsc{vector} v, booleantype tf)`

Description: This function enables (`SUN\textsc{true}`) or disables (`SUN\textsc{false}`) the linear combination fused operation in the parhyp vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

**N\textsc{venablescaleaddmulti\_parhyp}**

Prototype: `int N\textsc{venablescaleaddmulti\_parhyp}(N\textsc{vector} v, booleantype tf)`

Description: This function enables (`SUN\textsc{true}`) or disables (`SUN\textsc{false}`) the scale and add a vector to multiple vectors fused operation in the parhyp vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

**N\textsc{venabledotprodmulti\_parhyp}**

Prototype: `int N\textsc{venabledotprodmulti\_parhyp}(N\textsc{vector} v, booleantype tf)`

Description: This function enables (`SUN\textsc{true}`) or disables (`SUN\textsc{false}`) the multiple dot products fused operation in the parhyp vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

**N\textsc{venablelinearsumvectorarray\_parhyp}**

Prototype: `int N\textsc{venablelinearsumvectorarray\_parhyp}(N\textsc{vector} v, booleantype tf)`

Description: This function enables (`SUN\textsc{true}`) or disables (`SUN\textsc{false}`) the linear sum operation for vector arrays in the parhyp vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

**N\textsc{venablescalevectorarray\_parhyp}**

Prototype: `int N\textsc{venablescalevectorarray\_parhyp}(N\textsc{vector} v, booleantype tf)`
Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the parhyp vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

Prototype int N_VEnableConstVectorArray_ParHyp(N_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the parhyp vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

Prototype int N_VEnableWrmsNormVectorArray_ParHyp(N_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the parhyp vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

Prototype int N_VEnableWrmsNormMaskVectorArray_ParHyp(N_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the parhyp vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

Prototype int N_VEnableScaleAddMultiVectorArray_ParHyp(N_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the parhyp vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

Prototype int N_VEnableLinearCombinationVectorArray_ParHyp(N_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the parhyp vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

Notes

- When there is a need to access components of an N_Vector_ParHyp, v, it is recommended to extract the hypre vector via \texttt{x_vec = N_VGetVector_ParHyp(v)} and then access components using appropriate hypre functions.

- \texttt{N_VNewEmpty_ParHyp}, \texttt{N_VMake_ParHyp}, and \texttt{N_VCloneVectorArrayEmpty_ParHyp} set the field \texttt{own_parvector} to SUNFALSE. \texttt{N_VDestroy_ParHyp} and \texttt{N_VDestroyVectorArray_ParHyp} will not attempt to delete an underlying hypre vector for any \texttt{N_Vector} with \texttt{own_parvector} set to SUNFALSE. In such a case, it is the user’s responsibility to delete the underlying vector.
6.7 The NVECTOR_PETSC implementation

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.

6.7 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 lib sundials_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.

6.7.1 NVECTOR_PETSC functions

The NVECTOR_PETSC module defines implementations of all vector operations listed in Tables 6.2, 6.3, 6.4, and 6.5, 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 [27].

The names of vector operations are obtained from those in Tables 6.2, 6.3, 6.4, and 6.5 by appending the suffix _Petsc (e.g. N_VDestroy_Petsc). The module NVECTOR_PETSC provides the following additional user-callable routines:

- **N_VNewEmpty_Petsc**
  - Prototype: `N_Vector N_VNewEmpty_Petsc(MPI_Comm comm, sunindextype local_length, sunindextype global_length)`
  - Description: 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_VMake_Petsc**
  - Prototype: `N_Vector N_VMake_Petsc(Vec *pvec)`
  - Description: 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_VGetVector_Petsc**
  - Prototype: `Vec *N_VGetVector_Petsc(N_Vector v)`
  - Description: This function returns a pointer to the underlying PETSc vector.
Description of the NVECTOR module

**N_VCloneVectorArray_Petsc**
Prototype: `N_Vector *N_VCloneVectorArray_Petsc(int count, N_Vector w)`
Description: This function creates (by cloning) an array of `count` NVECTOR_PETSC vectors.

**N_VCloneVectorArrayEmpty_Petsc**
Prototype: `N_Vector *N_VCloneVectorArrayEmpty_Petsc(int count, N_Vector w)`
Description: This function creates (by cloning) an array of `count` NVECTOR_PETSC vectors, each with pointers to PETSc vectors set to (NULL).

**N_VDestroyVectorArray_Petsc**
Prototype: `void N_VDestroyVectorArray_Petsc(N_Vector *vs, int count)`
Description: 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`.

**N_VPrint_Petsc**
Prototype: `void N_VPrint_Petsc(N_Vector v)`
Description: This function prints the global content of a wrapped PETSc vector to stdout.

**N_VPrintFile_Petsc**
Prototype: `void N_VPrintFile_Petsc(N_Vector v, const char fname[])`
Description: This function prints the global content of a wrapped PETSc vector to `fname`.

By default all fused and vector array operations are disabled in the NVECTOR_PETSC module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with `N_VMMake_Petsc`, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using `N_VClone`. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with `N_VMMake_Petsc` will have the default settings for the NVECTOR_PETSC module.

**N_VEnableFusedOps_Petsc**
Prototype: `int N_VEnableFusedOps_Petsc(N_Vector v, boolantype tf)`
Description: This function enables (`SUNTRUE`) or disables (`SUNFALSE`) all fused and vector array operations in the PETSc vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

**N_VEnableLinearCombination_Petsc**
Prototype: `int N_VEnableLinearCombination_Petsc(N_Vector v, boolantype tf)`
Description: This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear combination fused operation in the PETSc vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.
6.7 The NVECTOR_PETSC implementation

**N_VEnableScaleAddMulti_Petsc**

Prototype: `int N_VEnableScaleAddMulti_Petsc(N_Vector v, booleantype tf)`

Description: This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the petsc vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

**N_VEnableDotProdMulti_Petsc**

Prototype: `int N_VEnableDotProdMulti_Petsc(N_Vector v, booleantype tf)`

Description: This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the petsc vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

**N_VEnableLinearSumVectorArray_Petsc**

Prototype: `int N_VEnableLinearSumVectorArray_Petsc(N_Vector v, booleantype tf)`

Description: This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the petsc vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

**N_VEnableScaleVectorArray_Petsc**

Prototype: `int N_VEnableScaleVectorArray_Petsc(N_Vector v, booleantype tf)`

Description: This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the petsc vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

**N_VEnableConstVectorArray_Petsc**

Prototype: `int N_VEnableConstVectorArray_Petsc(N_Vector v, booleantype tf)`

Description: This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the petsc vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

**N_VEnableWrmsNormVectorArray_Petsc**

Prototype: `int N_VEnableWrmsNormVectorArray_Petsc(N_Vector v, booleantype tf)`

Description: This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the petsc vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

**N_VEnableWrmsNormMaskVectorArray_Petsc**

Prototype: `int N_VEnableWrmsNormMaskVectorArray_Petsc(N_Vector v, booleantype tf)`

Description: This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the petsc vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.
### Description of the NVECTOR module

**N_VEnableScaleAddMultiVectorArray_Petsc**

**Prototype**

```c
int N_VEnableScaleAddMultiVectorArray_Petsc(N_Vector v, booleantype tf)
```

**Description**

This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the PETSc vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

**N_VEnableLinearCombinationVectorArray_Petsc**

**Prototype**

```c
int N_VEnableLinearCombinationVectorArray_Petsc(N_Vector v, booleantype tf)
```

**Description**

This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the PETSc vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

**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.

### 6.8 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 the namespace `suncudavec` manages the vector data layout:

```cpp
template <class T, class I>
class Vector {
  I size_;  // size of the vector
  I mem_size_;  // memory size
  I global_size_;  // global size
  T* h_vec_;    // host memory
  T* d_vec_;    // device memory
  ThreadPartitioning<T, I>* partStream_;  // partitioning for streams
  ThreadPartitioning<T, I>* partReduce_;  // partitioning for reductions
  bool ownPartitioning_;  // own partitioning
  bool ownData_;  // own data
  bool managed_mem_;  // managed memory
  SUNMPI_Comm comm_;  // MPI communicator
    ...
};
```
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, a boolean flag that signals if the vector owns
the thread partitioning, a boolean flag that signals if the vector owns the data, a boolean flag that
signals if managed memory is used for the data arrays, and the MPI communicator. The class Vector
inherits from the empty structure

struct _N_VectorContent_Cuda {};

to interface the C++ class with the NVECTOR C code. Due to the rapid progress of CUDA development,
we expect that the suncudadec::Vector class will change frequently in future SUNDIALS releases. The
code is structured so that it can tolerate significant changes in the suncudadec::Vector class without
requiring changes to the user API.

When instantiated with N_VNew_Cuda, the class Vector will allocate memory on both the host and
the device. Alternatively, a user can provide host and device data arrays by using the N_VMake_Cuda
constructor. To use CUDA managed memory, the constructors N_VNewManaged_Cuda and
N_VMakeManaged_Cuda are provided. Details on each of these constructors are provided below.

The NVECTOR_CUDA module can be utilized for single-node parallelism or in a distributed context
with MPI. In the single-node case the header file to include nvector_cuda.h and the library to
link to is libsundials_nveccuda.lib. In the a distributed setting the header file to include is
nvector_mpicuda.h and the library to link to is libsundials_nvecmpicuda.lib. The extension,
.lib, is typically .so for shared libraries and .a for static libraries. 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.

### 6.8.1 NVECTOR_CUDA functions

Unlike other native SUNDIALS vector types, NVECTOR_CUDA does not provide macros to access its
member variables. Instead, user should use the accessor functions:

**N_VGetLocalLength_Cuda**

Prototype: sunindextype N_VGetLocalLength_Cuda(N_Vector v)

Description: This function returns the local length of the vector.

Note: This function is for use in a distributed context and is defined in the header
nvector_mpicuda.h and the library to link to is libsundials_nvecmpicuda.lib.

**N_VGetHostArrayPointer_Cuda**

Prototype: realtype *N_VGetHostArrayPointer_Cuda(N_Vector v)

Description: This function returns a pointer to the vector data on the host.

**N_VGetDeviceArrayPointer_Cuda**

Prototype: realtype *N_VGetDeviceArrayPointer_Cuda(N_Vector v)

Description: This function returns a pointer to the vector data on the device.

**N_VGetMPIComm_Cuda**

Prototype: MPI_Comm N_VGetMPIComm_Cuda(N_Vector v)

Description: This function returns the MPI communicator for the vector.

Note: This function is for use in a distributed context and is defined in the header
nvector_mpicuda.h and the library to link to is libsundials_nvecmpicuda.lib.
Description of the NVECTOR module

**N_VIsManagedMemory_Cuda**

Prototype:  
`boolantype *N_VIsManagedMemory_Cuda(N_Vector v)`

Description:  
This function returns a boolean flag indicating if the vector data is allocated in managed memory or not.

The NVECTOR_CUDA module defines implementations of all vector operations listed in Tables 6.2, 6.3, 6.4 and 6.5, 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 [28].

The names of vector operations are obtained from those in Tables 6.2, 6.3, 6.4, and 6.5 by appending the suffix _Cuda (e.g. `N_VDestroy_Cuda`). The module NVECTOR_CUDA provides the following functions:

**N_VNew_Cuda**

Single-node usage

Prototype:  
`N_Vector N_VNew_Cuda(sunindextype length)`

Description:  
This function creates and allocates memory for a CUDA N_Vector. The vector data array is allocated on both the host and device. In the single-node setting, the only input is the vector length. This constructor is defined in the header nvector_cuda.h and the library to link to is libsundials_nveccuda.lib.

Distributed-memory parallel usage

Prototype:  
`N_Vector N_VNew_Cuda(MPI_Comm comm, sunindextype local_length, sunindextype global_length)`

Description:  
This function creates and allocates memory for a CUDA N_Vector. The vector data array is allocated on both the host and device. When used in a distributed context with MPI, the arguments are the MPI communicator, the local vector length, and the global vector length. This constructor is defined in the header nvector_mpicuda.h and the library to link to is libsundials_nvecmpicuda.lib.

**N_VNewManaged_Cuda**

Single-node usage

Prototype:  
`N_Vector N_VNewManaged_Cuda(sunindextype length)`

Description:  
This function creates and allocates memory for a CUDA N_Vector on a single node. The vector data array is allocated in managed memory. In the single-node setting, the only input is the vector length. This constructor is defined in the header nvector_cuda.h and the library to link to is libsundials_nveccuda.lib.

Distributed-memory parallel usage

Prototype:  
`N_Vector N_VNewManaged_Cuda(MPI_Comm comm, sunindextype local_length, sunindextype global_length)`

Description:  
This function creates and allocates memory for a CUDA N_Vector on a single node. The vector data array is allocated in managed memory. When used in a distributed context with MPI, the arguments are the MPI communicator, the local vector length, and the global vector length. This constructor is defined in the header nvector_mpicuda.h and the library to link to is libsundials_nvecmpicuda.lib.
6.8 The NVECTOR_CUDA implementation

N_VNewEmpty_Cuda
Prototype N_Vector N_VNewEmpty_Cuda()
Description 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_VMake_Cuda
Single-node usage
Prototype N_Vector N_VMake_Cuda(sunindextype length, realtype *h_vdata, realtype *d_vdata)
Description This function creates an NVECTOR_CUDA with user-supplied vector data arrays h_vdata and d_vdata. This function does not allocate memory for data itself. In the single-node setting, the inputs are the vector length, the host data array, and the device data. This constructor is defined in the header nvector_cuda.h and the library to link to is libsundials_nveccuda.lib.

Distributed-memory parallel usage
Prototype N_Vector N_VMake_Cuda(MPI_Comm comm, sunindextype local_length, sunindextype global_length, realtype *h_vdata, realtype *d_vdata)
Description This function creates an NVECTOR_CUDA with user-supplied vector data arrays h_vdata and d_vdata. This function does not allocate memory for data itself. When used in a distributed context with MPI, the arguments are the MPI communicator, the local vector length, the global vector length, the host data array, and the device data array. This constructor is defined in the header nvector_mpicuda.h and the library to link to is libsundials_nvecmpicuda.lib.

N_VMakeManaged_Cuda
Single-node usage
Prototype N_Vector N_VMakeManaged_Cuda(sunindextype length, realtype *vdata)
Description This function creates an NVECTOR_CUDA with a user-supplied managed memory data array. This function does not allocate memory for data itself. In the single-node setting, the inputs are the vector length and the managed data array. This constructor is defined in the header nvector_cuda.h and the library to link to is libsundials_nveccuda.lib.

Distributed-memory parallel usage
Prototype N_Vector N_VMakeManaged_Cuda(MPI_Comm comm, sunindextype local_length, sunindextype global_length, realtype *vdata)
Description This function creates an NVECTOR_CUDA with a user-supplied managed memory data array. This function does not allocate memory for data itself. When used in a distributed context with MPI, the arguments are the MPI communicator, the local vector length, the global vector length, the managed data array. This constructor is defined in the header nvector_mpicuda.h and the library to link to is libsundials_nvecmpicuda.lib.

The module NVECTOR_CUDA also provides the following user-callable routines:

N_VSetCudaStream_Cuda
Prototype void N_VSetCudaStream_Cuda(N_Vector v, cudaStream_t *stream)
Description of the NVECTOR module

Description
This function sets the CUDA stream that all vector kernels will be launched on. By default an NVECTOR_CUDA uses the default CUDA stream.

Note: All vectors used in a single instance of a SUNDIALS solver must use the same CUDA stream, and the CUDA stream must be set prior to solver initialization. Additionally, if manually instantiating the stream and reduce ThreadPartitioning of a suncudavec::Vector, ensure that they use the same CUDA stream.

N_VCopyToDevice_Cuda
Prototype
void N_VCopyToDevice_Cuda(N_Vector v)
Description
This function copies host vector data to the device.

N_VCopyFromDevice_Cuda
Prototype
void N_VCopyFromDevice_Cuda(N_Vector v)
Description
This function copies vector data from the device to the host.

N_VPrint_Cuda
Prototype
void N_VPrint_Cuda(N_Vector v)
Description
This function prints the content of a CUDA vector to stdout.

N_VPrintFile_Cuda
Prototype
void N_VPrintFile_Cuda(N_Vector v, FILE *outfile)
Description
This function prints the content of a CUDA vector to outfile.

By default all fused and vector array operations are disabled in the NVECTOR_CUDA module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with N_VNew_Cuda, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using N_VClone. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with N_VNew_Cuda will have the default settings for the NVECTOR_CUDA module.

N_VEnableFusedOps_Cuda
Prototype
int N_VEnableFusedOps_Cuda(N_Vector v, booleantype tf)
Description
This function enables (SUNTRUE) or disables (SUNFALSE) all fused and vector array operations in the CUDA vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableLinearCombination_Cuda
Prototype
int N_VEnableLinearCombination_Cuda(N_Vector v, booleantype tf)
Description
This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused operation in the CUDA vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.
6.8 The NVECTOR_CUDA implementation

**N_VEnableScaleAddMulti_Cuda**

Prototype: `int N_VEnableScaleAddMulti_Cuda(N_Vector v, booleantype tf)`

Description: This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the CUDA vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

**N_VEnableDotProdMulti_Cuda**

Prototype: `int N_VEnableDotProdMulti_Cuda(N_Vector v, booleantype tf)`

Description: This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the CUDA vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

**N_VEnableLinearSumVectorArray_Cuda**

Prototype: `int N_VEnableLinearSumVectorArray_Cuda(N_Vector v, booleantype tf)`

Description: This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the CUDA vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

**N_VEnableScaleVectorArray_Cuda**

Prototype: `int N_VEnableScaleVectorArray_Cuda(N_Vector v, booleantype tf)`

Description: This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the CUDA vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

**N_VEnableConstVectorArray_Cuda**

Prototype: `int N_VEnableConstVectorArray_Cuda(N_Vector v, booleantype tf)`

Description: This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the CUDA vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

**N_VEnableWrmsNormVectorArray_Cuda**

Prototype: `int N_VEnableWrmsNormVectorArray_Cuda(N_Vector v, booleantype tf)`

Description: This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the CUDA vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

**N_VEnableWrmsNormMaskVectorArray_Cuda**

Prototype: `int N_VEnableWrmsNormMaskVectorArray_Cuda(N_Vector v, booleantype tf)`

Description: This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the CUDA vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.
**N_VEnableScaleAddMultiVectorArray_Cuda**

**Prototype**

```c
int N_VEnableScaleAddMultiVectorArray_Cuda(N_Vector v, booleantype tf)
```

**Description**

This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the CUDA vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

**N_VEnableLinearCombinationVectorArray_Cuda**

**Prototype**

```c
int N_VEnableLinearCombinationVectorArray_Cuda(N_Vector v, booleantype tf)
```

**Description**

This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the CUDA vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

**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.

## 6.9 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 OpenACC. These backends are not used in this SUNDIALS release. Class `Vector` in namespace `sunrajavec` manages the vector data layout:

```cpp
template <class T, class I>
class Vector {
    I size_;  
    I mem_size_;  
    I global_size_;  
    T* h_vec_;  
    T* d_vec_;  
    SUNMPI_Comm comm_;  
    ...  
};
```

The class members are: vector size (length), size of the vector data memory block, the global vector size (length), pointers to vector data on the host and on the device, and the MPI communicator. The class `Vector` inherits from an empty structure

```cpp
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 \texttt{sunravec::Vector} class without requiring changes to the user API.

The \texttt{NVVECTOR\_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 \texttt{nvvector\_raja.h}. The header file to include when using this module in the distributed case is \texttt{nvvector\_mpiraja.h}. The installed module libraries to link to are \texttt{libsundials\_nvecraja.lib} in the single-node case, or \texttt{libsundials\_nvecmpicudaraja.lib} in the distributed case. Only one 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.

### 6.9.1 \texttt{NVVECTOR\_RAJA} functions

Unlike other native SUNDIALS vector types, \texttt{NVVECTOR\_RAJA} does not provide macros to access its member variables. Instead, user should use the accessor functions:

#### \texttt{N\_VGetLocalLength\_Raja}

**Prototype**

\begin{verbatim}
sunindextype N_VGetLocalLength\_Raja(N\_Vector v)
\end{verbatim}

**Description**

This function returns the local length of the vector.

Note: This function is for use in a \textit{distributed context} and is defined in the header \texttt{nvvector\_mpiraja.h} and the library to link to is \texttt{libsundials\_nvecmpicudaraja.lib}.

#### \texttt{N\_VGetHostArrayPointer\_Raja}

**Prototype**

\begin{verbatim}
realtype \*N\_VGetHostArrayPointer\_Raja(N\_Vector v)
\end{verbatim}

**Description**

This function returns a pointer to the vector data on the host.

#### \texttt{N\_VGetDeviceArrayPointer\_Raja}

**Prototype**

\begin{verbatim}
realtype \*N\_VGetDeviceArrayPointer\_Raja(N\_Vector v)
\end{verbatim}

**Description**

This function returns a pointer to the vector data on the device.

#### \texttt{N\_VGetMPIComm\_Raja}

**Prototype**

\begin{verbatim}
MPI\_Comm N\_VGetMPIComm\_Raja(N\_Vector v)
\end{verbatim}

**Description**

This function returns the MPI communicator for the vector.

Note: This function is for use in a \textit{distributed context} and is defined in the header \texttt{nvvector\_mpiraja.h} and the library to link to is \texttt{libsundials\_nvecmpicudaraja.lib}.

The \texttt{NVVECTOR\_RAJA} module defines the implementations of all vector operations listed in Tables 6.2, 6.3, 6.4, and 6.5, except for \texttt{N\_VDotProdMulti}, \texttt{N\_VWrmsNormVectorArray}, and \texttt{N\_VWrmsNormMaskVectorArray} as support for arrays of reduction vectors is not yet supported in RAJA. These function will be added to the \texttt{NVVECTOR\_RAJA} implementation in the future. Additionally the vector operations \texttt{N\_VGetArrayPointer} and \texttt{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 \texttt{NVVECTOR\_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 \texttt{NVVECTOR\_RAJA} are provided in some example programs for \texttt{CVODE} [28].

The names of vector operations are obtained from those in Tables 6.2, 6.3, 6.4, and 6.5 by appending the suffix \texttt{\_Raja} (e.g. \texttt{N\_VDestroy\_Raja}). The module \texttt{NVVECTOR\_RAJA} provides the following additional user-callable routines:
**N_VNew_Raja**

*Single-node usage*

**Prototype**

\[ \text{N\_Vector N\_VNew\_Raja(sunindextype length)} \]

**Description**

This function creates and allocates memory for a CUDA N\_Vector. The vector data array is allocated on both the host and device. In the *single-node* setting, the only input is the vector length. This constructor is defined in the header `nvectordaraja.h` and the library to link to is `libsundials_nvectordaraja.lib`.

**Distributed-memory parallel usage**

**Prototype**

\[ \text{N\_Vector N\_VNew\_Raja(MPI\_Comm comm, sunindextype local_length, sunindextype global_length)} \]

**Description**

This function creates and allocates memory for a CUDA N\_Vector. The vector data array is allocated on both the host and device. When used in a *distributed* context with MPI, the arguments are the MPI communicator, the local vector length, and the global vector length. This constructor is defined in the header `nvectormpiraja.h` and the library to link to is `libsundials_nvvecmpidaraja.lib`.

**N_VNewEmpty_Raja**

**Prototype**

\[ \text{N\_Vector N\_VNewEmpty\_Raja()} \]

**Description**

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_VMake_Raja**

**Prototype**

\[ \text{N\_Vector N\_VMake\_Raja(N\_VectorContent\_Raja c)} \]

**Description**

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_VCopyToDevice_Raja**

**Prototype**

\[ \text{realtype \*N\_VCopyToDevice\_Raja(N\_Vector v)} \]

**Description**

This function copies host vector data to the device.

**N_VCopyFromDevice_Raja**

**Prototype**

\[ \text{realtype \*N\_VCopyFromDevice\_Raja(N\_Vector v)} \]

**Description**

This function copies vector data from the device to the host.

**N_VPrint_Raja**

**Prototype**

\[ \text{void N\_VPrint\_Raja(N\_Vector v)} \]

**Description**

This function prints the content of a RAJA vector to `stdout`. 
6.9 The NVVECTOR_RAJA implementation

**N_VPrintFile_Raja**
Prototype: `void N_VPrintFile_Raja(N_Vector v, FILE *outfile)`
Description: This function prints the content of a RAJA vector to `outfile`.

By default all fused and vector array operations are disabled in the NVVECTOR_RAJA module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with `N_VNew_Raja`, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using `N_VClone`. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with `N_VNew_Raja` will have the default settings for the NVVECTOR_RAJA module.

**N_VEnableFusedOps_Raja**
Prototype: `int N_VEnableFusedOps_Raja(N_Vector v, booleantype tf)`
Description: This function enables (SUNTRUE) or disables (SUNFALSE) all fused and vector array operations in the RAJA vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are NULL.

**N_VEnableLinearCombination_Raja**
Prototype: `int N_VEnableLinearCombination_Raja(N_Vector v, booleantype tf)`
Description: This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused operation in the RAJA vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are NULL.

**N_VEnableScaleAddMulti_Raja**
Prototype: `int N_VEnableScaleAddMulti_Raja(N_Vector v, booleantype tf)`
Description: This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the RAJA vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are NULL.

**N_VEnableLinearSumVectorArray_Raja**
Prototype: `int N_VEnableLinearSumVectorArray_Raja(N_Vector v, booleantype tf)`
Description: This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the RAJA vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are NULL.

**N_VEnableScaleVectorArray_Raja**
Prototype: `int N_VEnableScaleVectorArray_Raja(N_Vector v, booleantype tf)`
Description: This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the RAJA vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are NULL.

**N_VEnableConstVectorArray_Raja**
Prototype: `int N_VEnableConstVectorArray_Raja(N_Vector v, booleantype tf)`
Description  This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the RAJA vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

Prototype  int N_VEnableScaleAddMultiVectorArray_Raja(N_Vector v, booleantype tf)

Description  This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the RAJA vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

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 NVVECTOR_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.

6.10 The NVVECTOR_OPENMPDEV implementation

In situations where a user has access to a device such as a GPU for offloading computation, SUNDIALS provides an NVVECTOR implementation using OpenMP device offloading, called NVVECTOR_OPENMPDEV.

The NVVECTOR_OPENMPDEV implementation defines the content field of the N_Vector to be a structure containing the length of the vector, a pointer to the beginning of a contiguous data array on the host, a pointer to the beginning of a contiguous data array on the device, and a boolean flag own_data which specifies the ownership of host and device data arrays.

```c
struct _N_VectorContent_OpenMPDEV {
    sunindextype length;
    booleantype own_data;
    realtype *host_data;
    realtype *dev_data;
};
```

The header file to include when using this module is nvector_openmpdev.h. The installed module library to link to is libsundials_nvecopenmpdev.lib where .lib is typically .so for shared libraries and .a for static libraries.

6.10.1 NVVECTOR_OPENMPDEV accessor macros

The following macros are provided to access the content of an NVVECTOR_OPENMPDEV vector.
6.10 The NVECTOR OPENMPDEV implementation

- **NV_CONTENT_OMPDEV**

  This routine gives access to the contents of the NVECTOR_OPENMPDEV vector N_Vector.
  
  The assignment \( v\_cont = NV\_CONTENT\_OMPDEV(v) \) sets \( v\_cont \) to be a pointer to the NVECTOR OPENMPDEV N_Vector content structure.

  Implementation:
  
  ```c
  #define NV_CONTENT_OMPDEV(v) ( N_VectorContent_OpenMPDEV)(v->content) 
  ``

- **NV_OWN_DATA_OMPDEV, NV_DATA_HOST_OMPDEV, NV_DATA_DEV_OMPDEV, NV_LENGTH_OMPDEV**

  These macros give individual access to the parts of the content of an NVECTOR_OPENMPDEV N_Vector.

  The assignment \( v\_data = NV\_DATA\_HOST\_OMPDEV(v) \) sets \( v\_data \) to be a pointer to the first component of the data on the host for the N_Vector \( v \). The assignment \( NV\_DATA\_HOST\_OMPDEV(v) = v\_data \) sets the host component array of \( v \) to be \( v\_data \) by storing the pointer \( v\_data \).

  The assignment \( v\_dev\_data = NV\_DATA\_DEV\_OMPDEV(v) \) sets \( v\_dev\_data \) to be a pointer to the first component of the data on the device for the N_Vector \( v \). The assignment \( NV\_DATA\_DEV\_OMPDEV(v) = v\_dev\_data \) sets the device component array of \( v \) to be \( v\_dev\_data \) by storing the pointer \( v\_dev\_data \).

  The assignment \( v\_len = NV\_LENGTH\_OMPDEV(v) \) sets \( v\_len \) to be the length of \( v \). On the other hand, the call \( NV\_LENGTH\_OMPDEV(v) = len\_v \) sets the length of \( v \) to be \( len\_v \).

  Implementation:
  
  ```c
  #define NV_OWN_DATA_OMPDEV(v) ( NV_CONTENT_OMPDEV(v)->own_data )
  #define NV_DATA_HOST_OMPDEV(v) ( NV_CONTENT_OMPDEV(v)->host_data )
  #define NV_DATA_DEV_OMPDEV(v) ( NV_CONTENT_OMPDEV(v)->dev_data )
  #define NV_LENGTH_OMPDEV(v) ( NV_CONTENT_OMPDEV(v)->length )
  ```

### 6.10.2 NVECTOR_OPENMPDEV functions

The NVECTOR_OPENMPDEV module defines OpenMP device offloading implementations of all vector operations listed in Tables 6.2, 6.3, 6.4, and 6.5, except for NVGetArrayPointer and NVSetArrayPointer. As such, this vector cannot be used with the SUNDIALS Fortran interfaces, nor with the SUNDIALS direct solvers and preconditioners. It also provides methods for copying from the host to the device and vice versa.

The names of vector operations are obtained from those in Tables 6.2, 6.3, 6.4, and 6.5 by appending the suffix _OpenMPDEV_ (e.g. NVDestroy_OpenMPDEV). The module NVECTOR_OPENMPDEV provides the following additional user-callable routines:

**N_VNew_OpenMPDEV**

Prototype: \( \text{N\_Vector N\_VNew\_OpenMPDEV}(\text{sunindextype vec\_length}) \)

Description: This function creates and allocates memory for an NVECTOR_OPENMPDEV N_Vector.

**N_VNewEmpty_OpenMPDEV**

Prototype: \( \text{N\_Vector N\_VNewEmpty\_OpenMPDEV}(\text{sunindextype vec\_length}) \)

Description: This function creates a new NVECTOR_OPENMPDEV N_Vector with an empty (NULL) host and device data arrays.
**N_VMake_OpenMPDEV**
Prototype: `N_Vector N_VMake_OpenMPDEV(sunindextype vec_length, realtype *h_vdata, realtype *d_vdata)`
Description: This function creates an NVVECTOR_OPENMPDEV vector with user-supplied vector data arrays `h_vdata` and `d_vdata`. This function does not allocate memory for data itself.

**N_VCloneVectorArray_OpenMPDEV**
Prototype: `N_Vector *N_VCloneVectorArray_OpenMPDEV(int count, N_Vector w)`
Description: This function creates (by cloning) an array of `count` NVVECTOR_OPENMPDEV vectors.

**N_VCloneVectorArrayEmpty_OpenMPDEV**
Prototype: `N_Vector *N_VCloneVectorArrayEmpty_OpenMPDEV(int count, N_Vector w)`
Description: This function creates (by cloning) an array of `count` NVVECTOR_OPENMPDEV vectors, each with an empty (NULL) data array.

**N_VDestroyVectorArray_OpenMPDEV**
Prototype: `void N_VDestroyVectorArray_OpenMPDEV(N_Vector *vs, int count)`
Description: This function frees memory allocated for the array of `count` variables of type `N_Vector` created with `N_VCloneVectorArray_OpenMPDEV` or with `N_VCloneVectorArrayEmpty_OpenMPDEV`.

**N_VGetHostArrayPointer_OpenMPDEV**
Prototype: `realtype *N_VGetHostArrayPointer_OpenMPDEV(N_Vector v)`
Description: This function returns a pointer to the host data array.

**N_VGetDeviceArrayPointer_OpenMPDEV**
Prototype: `realtype *N_VGetDeviceArrayPointer_OpenMPDEV(N_Vector v)`
Description: This function returns a pointer to the device data array.

**N_VPrint_OpenMPDEV**
Prototype: `void N_VPrint_OpenMPDEV(N_Vector v)`
Description: This function prints the content of an NVVECTOR_OPENMPDEV vector to `stdout`.

**N_VPrintFile_OpenMPDEV**
Prototype: `void N_VPrintFile_OpenMPDEV(N_Vector v, FILE *outfile)`
Description: This function prints the content of an NVVECTOR_OPENMPDEV vector to `outfile`.

**N_VCopyToDevice_OpenMPDEV**
Prototype: `void N_VCopyToDevice_OpenMPDEV(N_Vector v)`
Description: This function copies the content of an NVVECTOR_OPENMPDEV vector’s host data array to the device data array.
### NVECTOR_OPENMPDEV Implementation

#### N_copyFromDevice_OpenMPDEV

**Prototype:** `void N_VCopyFromDevice_OpenMPDEV(N_Vector v)`

**Description:** This function copies the content of an NVECTOR_OPENMPDEV vector’s device data array to the host data array.

By default all fused and vector array operations are disabled in the NVECTOR_OPENMPDEV module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with `N_VNew_OpenMPDEV`, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using `N_VClone`. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with `N_VNew_OpenMPDEV` will have the default settings for the NVECTOR_OPENMPDEV module.

#### N_EnableFusedOps_OpenMPDEV

**Prototype:** `int N_VEnableFusedOps_OpenMPDEV(N_Vector v, booleantype tf)`

**Description:** This function enables (SUNTRUE) or disables (SUNFALSE) all fused and vector array operations in the NVECTOR_OPENMPDEV vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

#### N_EnableLinearCombination_OpenMPDEV

**Prototype:** `int N_VEnableLinearCombination_OpenMPDEV(N_Vector v, booleantype tf)`

**Description:** This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused operation in the NVECTOR_OPENMPDEV vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

#### N_EnableScaleAddMulti_OpenMPDEV

**Prototype:** `int N_VEnableScaleAddMulti_OpenMPDEV(N_Vector v, booleantype tf)`

**Description:** This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the NVECTOR_OPENMPDEV vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

#### N_EnableDotProdMulti_OpenMPDEV

**Prototype:** `int N_VEnableDotProdMulti_OpenMPDEV(N_Vector v, booleantype tf)`

**Description:** This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the NVECTOR_OPENMPDEV vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

#### N_EnableLinearSumVectorArray_OpenMPDEV

**Prototype:** `int N_VEnableLinearSumVectorArray_OpenMPDEV(N_Vector v, booleantype tf)`

**Description:** This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the NVECTOR_OPENMPDEV vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.
### N_VEnableScaleVectorArray_OpenMPDEV

**Prototype**
```c
int N_VEnableScaleVectorArray_OpenMPDEV(N_Vector v, booleantype tf)
```

**Description**
This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the nvector_openmpdev vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

### N_VEnableConstVectorArray_OpenMPDEV

**Prototype**
```c
int N_VEnableConstVectorArray_OpenMPDEV(N_Vector v, booleantype tf)
```

**Description**
This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the nvector_openmpdev vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

### N_VEnableWrmsNormVectorArray_OpenMPDEV

**Prototype**
```c
int N_VEnableWrmsNormVectorArray_OpenMPDEV(N_Vector v, booleantype tf)
```

**Description**
This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the nvector_openmpdev vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

### N_VEnableWrmsNormMaskVectorArray_OpenMPDEV

**Prototype**
```c
int N_VEnableWrmsNormMaskVectorArray_OpenMPDEV(N_Vector v, booleantype tf)
```

**Description**
This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the nvector_openmpdev vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

### N_VEnableScaleAddMultiVectorArray_OpenMPDEV

**Prototype**
```c
int N_VEnableScaleAddMultiVectorArray_OpenMPDEV(N_Vector v, booleantype tf)
```

**Description**
This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the nvector_openmpdev vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

### N_VEnableLinearCombinationVectorArray_OpenMPDEV

**Prototype**
```c
int N_VEnableLinearCombinationVectorArray_OpenMPDEV(N_Vector v, booleantype tf)
```

**Description**
This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the nvector_openmpdev vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

### Notes
- When looping over the components of an N_Vector v, it is most efficient to first obtain the component array via h_data = NV_DATA_HOST_OPENMPDEV(v) for the host array or d_data = NV_DATA_DEV_OPENMPDEV(v) for the device array and then access h_data[i] or d_data[i] within the loop.
6.11 The NVeCtor_TRILINOS implementation

- When accessing individual components of an N_Vector v on the host remember to first copy the array back from the device with NV_CopyFromDevice_OpenMPDEV(v) to ensure the array is up to date.

- NVNewEmpty_OpenMPDEV, NVMake_OpenMPDEV, and NVCloneVectorArrayEmpty_OpenMPDEV set the field own_data = SUNFALSE. NV_Destroy_OpenMPDEV and NV_DestroyVectorArray_OpenMPDEV 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_OpenMPDEV implementation that have more than one N_Vector argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with N_Vector arguments that were all created with the same internal representations.

6.11 The NVeCtor_TRILINOS implementation

The NVeCtor_TRILINOS module is an NVeCtor wrapper around the Trilinos Tpetra vector. The interface to Tpetra is implemented in the Sundials::TpetraVectorInterface class. This class simply stores a reference counting pointer to a Tpetra vector and inherits from an empty structure

struct _N_VectorContent_Trilinos {};

to interface the C++ class with the NVeCtor C code. A pointer to an instance of this class is kept in the content field of the N_Vector object, to ensure that the Tpetra vector is not deleted for as long as the N_Vector object exists.

The Tpetra vector type in the Sundials::TpetraVectorInterface class is defined as:

typedef Tpetra::Vector<realtype, sunindextype, sunindextype> vector_type;

The Tpetra vector will use the Sundials-specified realtype as its scalar type, and it will use sunindextype as the global and the local ordinal types. This type definition will use Tpetra’s default node type. Available Kokkos node types in Trilinos 12.14 release are serial (single thread), OpenMP, Pthread, and CUDA. The default node type is selected when building the Kokkos package. For example, the Tpetra vector will use a CUDA node if Tpetra was built with CUDA support and the CUDA node was selected as the default when Tpetra was built.

The header file to include when using this module is nvector_trilinos.h. The installed module library to link to is lib sundials_nvectrilinos. lib where . lib is typically . so for shared libraries and . a for static libraries.

6.11.1 NVeCtor_TRILINOS functions

The NVeCtor_TRILINOS module defines implementations of all vector operations listed in Tables 6.2, 6.5, and 6.5, except for NV_GetArrayPointer and NV_SetArrayPointer. As such, this vector cannot be used with Sundials Fortran interfaces, nor with the Sundials direct solvers and preconditioners. When access to raw vector data is needed, it is recommended to extract the Trilinos Tpetra vector first, and then use Tpetra vector methods to access the data. Usage examples of NVeCtor_TRILINOS are provided in example programs for IDA [27].

The names of vector operations are obtained from those in Tables 6.2, 6.5, and 6.5 by appending the suffix _Trilinos (e.g. NV_Destroy_Trilinos). Vector operations call existing Tpetra::Vector methods when available. Vector operations specific to Sundials are implemented as standalone functions in the namespace Sundials::TpetraVector, located in the file SundialsTpetraVectorKernels.hpp. The module NVeCtor_TRILINOS provides the following additional user-callable functions:

- NV_GetVector_Trilinos

  This C++ function takes an N_Vector as the argument and returns a reference counting pointer to the underlying Tpetra vector. This is a standalone function defined in the global namespace.
Teuchos::RCP<vector_type> N_VGetVector_Trilinos(N_Vector v);

- **N_VMake_Trilinos**
  This C++ function creates and allocates memory for an \texttt{NVector\_Trilinos} wrapper around a user-provided Tpetra vector. This is a standalone function defined in the global namespace.

  
  \[ \text{N\_Vector N\_VMake\_Trilinos(Teuchos::RCP<vector\_type> v);} \]

**Notes**

- The template parameter \texttt{vector\_type} should be set as:
  
  ```cpp
typedef Sundials::TpetraVectorInterface::vector\_type vector\_type
  ```
  
  This will ensure that data types used in Tpetra vector match those in \texttt{SUNDIALS}.

- When there is a need to access components of an \texttt{N\_Vector\_Trilinos}, \( v \), it is recommended to extract the Trilinos vector object via \( x\_vec = N\_VGetVector\_Trilinos(v) \) and then access components using the appropriate Trilinos functions.

- The functions \texttt{N\_VDestroy\_Trilinos} and \texttt{N\_VDestroyVectorArray\_Trilinos} only delete the \texttt{N\_Vector} wrapper. The underlying Tpetra vector object will exist for as long as there is at least one reference to it.

6.12 The NV\textsc{ECTOR\_MANYVECTOR} implementation

The \texttt{NV\_VECTOR\_MANYVECTOR} implementation of the \texttt{NV\_VECTOR} module provided with \texttt{SUNDIALS} is designed to facilitate problems with an inherent data partitioning for the solution vector. These data partitions are entirely user-defined, through construction of distinct \texttt{NV\_VECTOR} modules for each component, that are then combined together to form the \texttt{NV\_VECTOR\_MANYVECTOR}. We envision three generic use cases for this implementation:

A. \textit{Heterogeneous computational architectures (serial or parallel)}: for users who wish to partition data on a node between different computing resources, they may create architecture-specific subvectors for each partition. For example, a user could create one MPI-parallel component based on \texttt{NV\_VECTOR\_PARALLEL}, another single-node component for GPU accelerators based on \texttt{NV\_VECTOR\_CUDA}, and another threaded single-node component based on \texttt{NV\_VECTOR\_OPENMP}.

B. \textit{Process-based multiphysics decompositions (parallel)}: for users who wish to combine separate simulations together, e.g., where one subvector resides on one subset of MPI processes, while another subvector resides on a different subset of MPI processes, and where the user has created a MPI \texttt{intercommunicator} to connect these distinct process sets together.

C. \textit{Structure of arrays (SOA) data layouts (serial or parallel)}: for users who wish to create separate subvectors for each solution component, e.g., in a Navier-Stokes simulation they could have separate subvectors for density, velocities and pressure, which are combined together into a single \texttt{NV\_VECTOR\_MANYVECTOR} for the overall “solution”.

We note that the above use cases are not mutually exclusive, and the \texttt{NV\_VECTOR\_MANYVECTOR} implementation should support arbitrary combinations of these cases.

The \texttt{NV\_VECTOR\_MANYVECTOR} implementation is designed to work with any \texttt{NV\_VECTOR} subvectors that implement the minimum \textit{required} set of operations, however significant performance benefits may be obtained when subvectors additionally implement the optional local reduction operations listed in Table 6.5.

Additionally, \texttt{NV\_VECTOR\_MANYVECTOR} sets no limit on the number of subvectors that may be attached (aside from the limitations of using \texttt{sunindex\_type} for indexing, and standard per-node memory limitations). However, while this ostensibly supports subvectors with one entry each (i.e., one
subvector for each solution entry), we anticipate that this extreme situation will hinder performance due to non-stride-one memory accesses and increased function call overhead. We therefore recommend a relatively coarse partitioning of the problem, although actual performance will likely be problem-dependent.

As a final note, in the coming years we plan to introduce additional algebraic solvers and time integration modules that will leverage the problem partitioning enabled by `nvector_manyvector`. However, even at present we anticipate that users will be able to leverage such data partitioning in their problem-defining ODE right-hand side, DAE residual, or nonlinear solver residual functions.

### 6.12.1 `nvector_manyvector` structure

The `nvector_manyvector` implementation defines the `content` field of `N_Vector` to be a structure containing the MPI communicator (or `SUNMPI_Comm_NULL` if running in serial), the number of subvectors comprising the ManyVector, the global length of the ManyVector (including all subvectors on all MPI tasks), a pointer to the beginning of the array of subvectors, and a boolean flag `own_data` indicating ownership of the subvectors that populate `subvec_array`.

```c
struct _N_VectorContent_ManyVector {
    SUNMPI_Comm comm; /* overall MPI communicator */
    sunindextype num_subvectors; /* number of vectors attached */
    sunindextype global_length; /* overall manyvector length */
    N_Vector* subvec_array; /* pointer to N_Vector array */
    booleantype own_data; /* flag indicating data ownership */
};
```

The header file to include when using this module is `nvector_manyvector.h`. The installed module library to link against is `libsundials_nvecmanyvector.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

**Note:** If SUNDIALS is configured with MPI enabled, then the ManyVector library will be built for the parallel use case and `SUNMPI_Comm` is set to `MPI_Comm`. As such an MPI-aware compiler will become necessary even in single node uses of the ManyVector library. If SUNDIALS is configured with MPI disabled, then the ManyVector library is built for the single-node (serial) use case and `SUNMPI_Comm` is set to `int`. As such, users need not include `mpi.h`, nor must executables be built with an MPI-aware compiler. For more details see Appendix A.

### 6.12.2 `nvector_manyvector` functions

The `nvector_manyvector` module implements all vector operations listed in Tables 6.2, 6.3, 6.4, and 6.5, except for `N_VGetArrayPointer`, `N_VSetArrayPointer`, `N_VScaleAddMultiVectorArray`, and `N_VLinearCombinationVectorArray`. As such, this vector cannot be used with the SUNDIALS Fortran interfaces, nor with the SUNDIALS direct solvers and preconditioners. Instead, the `nvector_manyvector` module provides functions to access subvectors, whose data may in turn be accessed according to their `nvector` implementations.

The names of vector operations are obtained from those in Tables 6.2, 6.3, 6.4, and 6.5 by appending the suffix `ManyVector` (e.g. `N_VDestroy_ManyVector`). The module `nvector_manyvector` provides the following additional user-callable routines:

```c
N_VNew_ManyVector
```

**Prototype**

```c
N_Vector N_VNew_ManyVector(sunindextype num_subvectors,
                           N_Vector *vec_array);
```

**Description**

This function creates a ManyVector from a set of existing `nvector` objects, under the requirement that all MPI-aware subvectors use the same MPI communicator (this is checked internally). If none of the subvectors are MPI-aware, then this may equivalently
be used to describe data partitioning within a single node. We note that this routine is designed to support use cases A and C above.

This routine will copy all N_Vector pointers from the input vec_array, so the user may modify/free that pointer array after calling this function. However, this routine does not allocate any new subvectors, so the underlying NVECTOR objects themselves should not be destroyed before the ManyVector that contains them.

Upon successful completion, the new ManyVector is returned; otherwise this routine returns NULL (e.g., if two MPI-aware subvectors use different MPI communicators).

**N_VMake_ManyVector**

Prototype: `N_Vector N_VMake_ManyVector(void *comm, sunindextype num_subvectors, N_Vector *vec_array);`

Description: This function creates a ManyVector from a set of existing nvector objects, and a user-created MPI communicator that “connects” these subvectors. Any MPI-aware subvectors may use different MPI communicators than the input comm. We note that this routine is designed to support any combination of the use cases above.

The input comm should be the memory reference to this user-created MPI communicator. We note that since many MPI implementations define MPI_COMM_WORLD to be a specific integer value (that has no memory reference), users who wish to supply MPI_COMM_WORLD to this routine should first duplicate this to a specific MPI_Comm variable before passing in the reference, e.g.,

```c
MPI_Comm comm;
N_Vector x;
MPI_Comm_dup(MPI_COMM_WORLD, &comm);
x = N_VMake_ManyVector(&comm, ...);
```

This routine will internally call MPI_Comm_dup to create a copy of the input comm, so the user-supplied comm argument need not be retained after the call to N_VMake_ManyVector. If all subvectors are MPI-unaware, then the input comm argument should be NULL, although in this case, it would be simpler to call N_VNew_ManyVector instead.

This routine will copy all N_Vector pointers from the input vec_array, so the user may modify/free that pointer array after calling this function. However, this routine does not allocate any new subvectors, so the underlying NVECTOR objects themselves should not be destroyed before the ManyVector that contains them.

Upon successful completion, the new ManyVector is returned; otherwise this routine returns NULL (e.g., if the input vec_array is NULL).

**N_VGetSubvector_ManyVector**

Prototype: `N_Vector N_VGetSubvector_ManyVector(N_Vector v, sunindextype vec_num);`

Description: This function returns the vec_num subvector from the nvector array.

**N_VGetNumSubvectors_ManyVector**

Prototype: `sunindextype N_VGetNumSubvectors_ManyVector(N_Vector v);`

Description: This function returns the overall number of subvectors in the ManyVector object.

By default all fused and vector array operations are disabled in the NVECTOR_MANYVECTOR module, except for N_VWrmsNormVectorArray and N_VWrmsNormMaskVectorArray, that are enabled by default. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with N_VNew_ManyVector or N_VMake_ManyVector, enable/disable the desired
operations for that vector with the functions below, and create any additional vectors from that vector using \texttt{N_VClone}. This guarantees that the new vectors will have the same operations enabled/disabled, since cloned vectors inherit those configuration options from the vector they are cloned from, while vectors created with \texttt{N_VNew\_ManyVector} and \texttt{N_VMake\_ManyVector} will have the default settings for the \texttt{NVECTOR\_MANYVECTOR} module. We note that these routines do not call the corresponding routines on subvectors, so those should be set up as desired before attaching them to the ManyVector in \texttt{N_VNew\_ManyVector} or \texttt{N_VMake\_ManyVector}.

\textbf{\texttt{N_VEnableFusedOps\_ManyVector}}

Prototype \texttt{int N_VEnableFusedOps\_ManyVector(N\_Vector v, bool type tf);}

Description This function enables (\texttt{SUNTRUE}) or disables (\texttt{SUNFALSE}) all fused and vector array operations in the ManyVector. The return value is 0 for success and -1 if the input vector or its \texttt{ops} structure are \texttt{NULL}.

\textbf{\texttt{N_VEnableLinearCombination\_ManyVector}}

Prototype \texttt{int N_VEnableLinearCombination\_ManyVector(N\_Vector v, bool type tf);}

Description This function enables (\texttt{SUNTRUE}) or disables (\texttt{SUNFALSE}) the linear combination fused operation in the ManyVector. The return value is 0 for success and -1 if the input vector or its \texttt{ops} structure are \texttt{NULL}.

\textbf{\texttt{N_VEnableScaleAddMulti\_ManyVector}}

Prototype \texttt{int N_VEnableScaleAddMulti\_ManyVector(N\_Vector v, bool type tf);}

Description This function enables (\texttt{SUNTRUE}) or disables (\texttt{SUNFALSE}) the scale and add a vector to multiple vectors fused operation in the ManyVector. The return value is 0 for success and -1 if the input vector or its \texttt{ops} structure are \texttt{NULL}.

\textbf{\texttt{N_VEnableDotProdMulti\_ManyVector}}

Prototype \texttt{int N_VEnableDotProdMulti\_ManyVector(N\_Vector v, bool type tf);}

Description This function enables (\texttt{SUNTRUE}) or disables (\texttt{SUNFALSE}) the multiple dot products fused operation in the ManyVector. The return value is 0 for success and -1 if the input vector or its \texttt{ops} structure are \texttt{NULL}.

\textbf{\texttt{N_VEnableLinearSumVectorArray\_ManyVector}}

Prototype \texttt{int N_VEnableLinearSumVectorArray\_ManyVector(N\_Vector v, bool type tf);}

Description This function enables (\texttt{SUNTRUE}) or disables (\texttt{SUNFALSE}) the linear sum operation for vector arrays in the ManyVector. The return value is 0 for success and -1 if the input vector or its \texttt{ops} structure are \texttt{NULL}.

\textbf{\texttt{N_VEnableScaleVectorArray\_ManyVector}}

Prototype \texttt{int N_VEnableScaleVectorArray\_ManyVector(N\_Vector v, bool type tf);}

Description This function enables (\texttt{SUNTRUE}) or disables (\texttt{SUNFALSE}) the scale operation for vector arrays in the ManyVector. The return value is 0 for success and -1 if the input vector or its \texttt{ops} structure are \texttt{NULL}. 

\textbf{\texttt{N_VEnableScaleVectorArray\_ManyVector}}
**N_VEnableConstVectorArray_ManyVector**

**Prototype**

```c
int N_VEnableConstVectorArray_ManyVector(N_Vector v, booleantype tf);
```

**Description**

This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the ManyVector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

**N_VEnableWrmsNormVectorArray_ManyVector**

**Prototype**

```c
int N_VEnableWrmsNormVectorArray_ManyVector(N_Vector v, booleantype tf);
```

**Description**

This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the ManyVector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

**N_VEnableWrmsNormMaskVectorArray_ManyVector**

**Prototype**

```c
int N_VEnableWrmsNormMaskVectorArray_ManyVector(N_Vector v, booleantype tf);
```

**Description**

This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the ManyVector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

**Notes**

- **N_VNew_ManyVector** and **N_VMake_ManyVector** set the field `own_data = SUNFALSE`. **N_VDestroy_ManyVector** will not attempt to call **N_VDestroy** on any subvectors contained in the subvector array for any **N_Vector** with `own_data` set to SUNFALSE. In such a case, it is the user’s responsibility to deallocate the subvectors.

- To maximize efficiency, arithmetic vector operations in the **NVECTOR_MANYVECTOR** 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 subvector representations.

### 6.13 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_VGetLength**: Compares self-reported length to calculated length.
• Test_N_VGetCommunicator: Compares self-reported communicator to the one used in constructor; or for MPI-unaware vectors it ensures that NULL is reported.

• 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 NVWrmsNormMaskVectorArray Case 1a: Create a vector of known values, find and validate the weighted root mean square norm using all elements except one.

• Test NVWrmsNormMaskVectorArray Case 1b: Create a vector array of three vectors of known values, find and validate the weighted root mean square norm of each using all elements except one.

• Test NVScaleAddMultiVectorArray Case 1a: \( y = a \times + y \)

• Test NVScaleAddMultiVectorArray Case 1b: \( z = a \times + y \)

• Test NVScaleAddMultiVectorArray Case 2a: \( Y[j][0] = a[j] \times[0] + Y[j][0] \)

• Test NVScaleAddMultiVectorArray Case 2b: \( Z[j][0] = a[j] \times[0] + Y[j][0] \)

• Test NVScaleAddMultiVectorArray Case 3a: \( Y[0][i] = a[0] \times[i] + Y[0][i] \)

• Test NVScaleAddMultiVectorArray Case 3b: \( Z[0][i] = a[0] \times[i] + Y[0][i] \)

• Test NVScaleAddMultiVectorArray Case 4a: \( Y[j][i] = a[j] \times[i] + Y[j][i] \)

• Test NVScaleAddMultiVectorArray Case 4b: \( Z[j][i] = a[j] \times[i] + Y[j][i] \)

• Test NVLLinearCombinationVectorArray Case 1a: \( x = a \times \)

• Test NVLLinearCombinationVectorArray Case 1b: \( z = a \times \)

• Test NVLLinearCombinationVectorArray Case 2a: \( x = a \times + b \times y \)

• Test NVLLinearCombinationVectorArray Case 2b: \( z = a \times + b \times y \)

• Test NVLLinearCombinationVectorArray Case 3a: \( x = a \times + b \times y + c \times z \)

• Test NVLLinearCombinationVectorArray Case 3b: \( w = a \times + b \times y + c \times z \)

• Test NVLLinearCombinationVectorArray Case 4a: \( X[0][i] = c[0] \times[0][i] \)

• Test NVLLinearCombinationVectorArray Case 4b: \( Z[i] = c[0] \times[0][i] \)

• Test NVLLinearCombinationVectorArray Case 5a: \( X[0][i] = c[0] \times[0][i] + c[1] \times[1][i] \)

• Test NVLLinearCombinationVectorArray Case 5b: \( Z[i] = c[0] \times[0][i] + c[1] \times[1][i] \)

• Test NVLLinearCombinationVectorArray Case 6a: \( X[0][i] = X[0][i] + c[1] \times[1][i] + c[2] \times[2][i] \)

• Test NVLLinearCombinationVectorArray Case 6b: \( X[0][i] = c[0] \times[0][i] + c[1] \times[1][i] + c[2] \times[2][i] \)

• Test NVLLinearCombinationVectorArray Case 6c: \( Z[i] = c[0] \times[0][i] + c[1] \times[1][i] + c[2] \times[2][i] \)

• Test NVDotProdLocal: Calculate MPI task-local portion of the dot product of two vectors.

• Test NVMaxNormLocal: Create vector with known values, find and validate the MPI task-local portion of the max norm.

• Test NVMinLocal: Create vector, find and validate the MPI task-local min.

• Test NVL1NormLocal: Create vector, find and validate the MPI task-local portion of the L1 norm.

• Test NVWSqrSumLocal: Create vector of known values, find and validate the MPI task-local portion of the weighted squared sum of two vectors.
• **Test_NVW_SqrSumMaskLocal**: Create vector of known values, find and validate the MPI task-local portion of the weighted squared sum of two vectors, using all elements except one.

• **Test_NVInvTestLocal**: Test the MPI task-local portion of $z[i] = 1 / x[i]$

• **Test_NVConstrMaskLocal**: Test the MPI task-local portion of the mask of vector $x$ with vector $c$.

• **Test_NVMinQuotientLocal**: Fill two vectors with known values. Calculate and validate the MPI task-local minimum quotient.
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.

7.1 The SUNMatrix API

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 (*matvecsetup)(SUNMatrix);
    int (*matvec)(SUNMatrix, N_Vector, N_Vector);
    int (*space)(SUNMatrix, long int*, long int*);
};
Table 7.1: Identifiers associated with matrix kernels supplied with SUNDIALS.

<table>
<thead>
<tr>
<th>Matrix ID</th>
<th>Matrix type</th>
<th>ID Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUNMATRIX_DENSE</td>
<td>Dense $M \times N$ matrix</td>
<td>0</td>
</tr>
<tr>
<td>SUNMATRIX_BAND</td>
<td>Band $M \times M$ matrix</td>
<td>1</td>
</tr>
<tr>
<td>SUNMATRIX_SPARSE</td>
<td>Sparse (CSR or CSC) $M \times N$ matrix</td>
<td>2</td>
</tr>
<tr>
<td>SUNMATRIX_SLUNRLOC</td>
<td>Adapter for the SuperLU_DIST SuperMatrix</td>
<td>3</td>
</tr>
<tr>
<td>SUNMATRIX_CUSTOM</td>
<td>User-provided custom matrix</td>
<td>4</td>
</tr>
</tbody>
</table>

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 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:

```c
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.

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 SUNMATRIXCUSTOM identifier.

### 7.1.1 SUNMatrix functions

The generic SUNMatrix object defines the following set of operations:

Table 7.2: Description of the SUNMatrix operations

<table>
<thead>
<tr>
<th>Name</th>
<th>Usage and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUNMatGetID</td>
<td>id = SUNMatGetID(A); Returns the type identifier for the matrix $A$. It is used to determine the matrix implementation type (e.g. dense, banded, sparse,...) from the abstract SUNMatrix interface. This is used to assess compatibility with SUNDIALS-provided linear solver implementations. Returned values are given in the Table 7.1.</td>
</tr>
<tr>
<td>SUNMatClone</td>
<td>B = SUNMatClone(A); Creates a new SUNMatrix of the same type as an existing matrix $A$ and sets the ops field. It does not copy the matrix, but rather allocates storage for the new matrix.</td>
</tr>
</tbody>
</table>
7.1 The SUNMatrix API

<table>
<thead>
<tr>
<th>Name</th>
<th>Usage and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUNMatDestroy</td>
<td>SUNMatDestroy(A); Destroys the SUNMatrix A and frees memory allocated for its internal data.</td>
</tr>
<tr>
<td>SUNMatSpace</td>
<td>ier = SUNMatSpace(A, &amp;lrw, &amp;liw); Returns the storage requirements for the matrix A. lrw is a long int containing the number of realtype words and liw is a long int containing the number of integer words. The return value is of type int and is a SUNMatrix error code denoting success/failure of the operation. 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 SUNMATRIX module if that information is not of interest.</td>
</tr>
<tr>
<td>SUNMatZero</td>
<td>ier = SUNMatZero(A); Performs the operation ( A_{ij} = 0 ) for all entries of the matrix ( A ). The return value is of type int and is a SUNMatrix error code denoting success/failure of the operation.</td>
</tr>
<tr>
<td>SUNMatCopy</td>
<td>ier = SUNMatCopy(A,B); Performs the operation ( B_{ij} = A_{i,j} ) for all entries of the matrices ( A ) and ( B ). The return value is of type int and is a SUNMatrix error code denoting success/failure of the operation.</td>
</tr>
<tr>
<td>SUNMatScaleAdd</td>
<td>ier = SUNMatScaleAdd(c, A, B); Performs the operation ( A = cA + B ). The return value is of type int and is a SUNMatrix error code denoting success/failure of the operation.</td>
</tr>
<tr>
<td>SUNMatScaleAddI</td>
<td>ier = SUNMatScaleAddI(c, A); Performs the operation ( A = cA + I ). The return value is of type int and is a SUNMatrix error code denoting success/failure of the operation.</td>
</tr>
<tr>
<td>SUNMatMatvecSetup</td>
<td>ier = SUNMatMatvecSetup(A); Performs any setup necessary to perform a matrix-vector product. The return value is of type int and is a SUNMatrix error code denoting success/failure of the operation. It is useful for SUNMatrix implementations which need to prepare the matrix itself, or communication structures before performing the matrix-vector product.</td>
</tr>
<tr>
<td>SUNMatMatvec</td>
<td>ier = SUNMatMatvec(A, x, y); Performs the matrix-vector product operation, ( y = Ax ). It should only be called with vectors ( x ) and ( y ) that are compatible with the matrix ( A ) – both in storage type and dimensions. The return value is of type int and is a SUNMatrix error code denoting success/failure of the operation.</td>
</tr>
</tbody>
</table>

7.1.2 SUNMatrix error codes

The functions provided to SUNMATRIX modules within the SUNDIALS-provided SUNMATRIX implementations utilize a common set of return codes, shown in Table 7.3. These adhere to a common pattern: 0 indicates success, and a negative value indicates a failure. The actual values of each error code are primarily to provide additional information to the user in case of a failure.
### Description of the SUNMatrix error codes

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUNMAT_SUCCESS</td>
<td>0</td>
<td>successful call or converged solve</td>
</tr>
<tr>
<td>SUNMAT_ILL_INPUT</td>
<td>-1</td>
<td>an illegal input has been provided to the function</td>
</tr>
<tr>
<td>SUNMAT_MEM_FAIL</td>
<td>-2</td>
<td>failed memory access or allocation</td>
</tr>
<tr>
<td>SUNMAT_OPERATION_FAIL</td>
<td>-3</td>
<td>a SUNMatrix operation returned nonzero</td>
</tr>
<tr>
<td>SUNMAT_MATVEC_SETUP_REQUIRED</td>
<td>-4</td>
<td>the SUNMatMatvecSetup routine needs to be called before calling SUNMatMatvec</td>
</tr>
</tbody>
</table>

### 7.2 Compatibility of SUNMatrix modules

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 7.4 we show the matrix interfaces available as SUNMATRIX modules, and the compatible vector implementations.

<table>
<thead>
<tr>
<th>Matrix Interface</th>
<th>Dense</th>
<th>Band</th>
<th>Sparse</th>
<th>SLUNRloc</th>
<th>User supplied</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial (MPI)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>OpenMP</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>pThreads</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>hypre Vec.</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>PETSC Vec.</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CUDA</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>RAJA</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

### 7.3 Implementing a custom SUNMatrix

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.
7.4 SUNMatrix functions used by IDA

In Table 7.5, 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 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 7.5: List of matrix functions usage by IDA code modules

<table>
<thead>
<tr>
<th>Function</th>
<th>IDALS</th>
<th>IDABBDPRE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUNMatGetID</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>SUNMatDestroy</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>SUNMatZero</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SUNMatSpace</td>
<td>†</td>
<td></td>
</tr>
</tbody>
</table>

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.

7.5 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:

```c
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 ≤ i < M and 0 ≤ 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 ≤ i < M and 0 ≤ 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_sunnmatrixdense` module library.

### 7.5.1 SUNMatrix_Dense accessor macros

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 \( A_{\text{cont}} = \text{SM\_CONTENT\_D}(A) \) sets \( A_{\text{cont}} \) to be a pointer to the dense SUNMatrix content structure.

  Implementation:
  ```c
  #define SM_CONTENT_D(A) ( (SUNMatrixContent_Dense)(A->content) )
  ```

- **SM_ROWS_D, SM_COLUMNS_D, and SM_LDATA_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 \( A_{\text{rows}} = \text{SM\_ROWS\_D}(A) \) sets \( A_{\text{rows}} \) to be the number of rows in the matrix \( A \). Similarly, the assignment \( \text{SM\_COLUMNS\_D}(A) = A_{\text{cols}} \) sets the number of columns in \( A \) to equal \( A_{\text{cols}} \).

  Implementation:
  ```c
  #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 \( A_{\text{data}} = \text{SM\_DATA\_D}(A) \) sets \( A_{\text{data}} \) to be a pointer to the first component of the data array for the dense SUNMatrix \( A \). The assignment \( \text{SM\_DATA\_D}(A) = A_{\text{data}} \) sets the data array of \( A \) to be \( A_{\text{data}} \) by storing the pointer \( A_{\text{data}} \).

  Similarly, the assignment \( A_{\text{cols}} = \text{SM\_COLS\_D}(A) \) sets \( A_{\text{cols}} \) to be a pointer to the array of column pointers for the dense SUNMatrix \( A \). The assignment \( \text{SM\_COLS\_D}(A) = A_{\text{cols}} \) sets the column pointer array of \( A \) to be \( A_{\text{cols}} \) by storing the pointer \( A_{\text{cols}} \).

  Implementation:
  ```c
  #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 \( \text{col}_j = \text{SM\_COLUMN\_D}(A,j) \) sets \( \text{col}_j \) 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 \( \text{SM\_COLUMN\_D}(A,j) \) is `realtype *`. The pointer returned by the call \( \text{SM\_COLUMN\_D}(A,j) \) can be treated as an array which is indexed from 0 to \( M - 1 \).

  The assignments \( \text{SM\_ELEMENT\_D}(A,i,j) = a_{ij} \) and \( a_{ij} = \text{SM\_ELEMENT\_D}(A,i,j) \) 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:

```c
#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] )
```

### 7.5.2 SUNMatrix_Dense functions

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). All the standard matrix operations listed in 7.2 with the suffix _Dense appended are callable via the FORTRAN 2003 interface by prepending an `F` (e.g. FSUNMatCopy_Dense).

The module SUNMATRIX_DENSE provides the following additional user-callable routines:

**SUNDenseMatrix**

Prototype: `SUNMatrix SUNDenseMatrix(sunindextype M, sunindextype N)`

Description: 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.

F2003 Name: This function is callable as FSUNDenseMatrix when using the Fortran 2003 interface module.

**SUNDenseMatrix_Print**

Prototype: `void SUNDenseMatrix_Print(SUNMatrix A, FILE* outfile)`

Description: 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.

**SUNDenseMatrix_Rows**

Prototype: `sunindextype SUNDenseMatrix_Rows(SUNMatrix A)`

Description: This function returns the number of rows in the dense SUNMatrix.

F2003 Name: This function is callable as FSUNDenseMatrix_Rows when using the Fortran 2003 interface module.

**SUNDenseMatrix_Columns**

Prototype: `sunindextype SUNDenseMatrix_Columns(SUNMatrix A)`

Description: This function returns the number of columns in the dense SUNMatrix.

F2003 Name: This function is callable as FSUNDenseMatrix_Columns when using the Fortran 2003 interface module.

**SUNDenseMatrix_LData**

Prototype: `sunindextype SUNDenseMatrix_LData(SUNMatrix A)`

Description: This function returns the length of the data array for the dense SUNMatrix.

F2003 Name: This function is callable as FSUNDenseMatrix_LData when using the Fortran 2003 interface module.
Description of the SUNMatrix module

**SUNDenseMatrix_Data**
Prototype realtype* SUNDenseMatrix_Data(SUNMatrix A)
Description This function returns a pointer to the data array for the dense SUNMatrix.
F2003 Name This function is callable as FSUNDenseMatrix_Data when using the Fortran 2003 interface module.

**SUNDenseMatrix_Cols**
Prototype realtype** SUNDenseMatrix_Cols(SUNMatrix A)
Description This function returns a pointer to the cols array for the dense SUNMatrix.

**SUNDenseMatrix_Column**
Prototype realtype* SUNDenseMatrix_Column(SUNMatrix A, sunindextype j)
Description This function returns a pointer to the first entry of the jth column of the dense SUNMatrix. The resulting pointer should be indexed over the range 0 to M − 1.
F2003 Name This function is callable as FSUNDenseMatrix_Column when using the Fortran 2003 interface module.

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 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.

7.5.3 SUNMatrix_Dense Fortran interfaces

The sunmatrix_dense module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

**FORTRAN 2003 interface module**

The fsunmatrix_dense_mod FORTRAN module defines interfaces to most sunmatrix_dense C functions using the intrinsic iso_c_binding module which provides a standardized mechanism for interop-erating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading ‘F’. For example, the function SUNDenseMatrix is interfaced as FSUNDenseMatrix.

The FORTRAN 2003 SUNMATRIX_DENSE interface module can be accessed with the use statement, i.e. use fsunmatrix_dense_mod, and linking to the library libsundials_fsunmatrixdense_mod.lib in
addition to the C library. For details on where the library and module file `fsunmatrix_dense_mod.mod` are installed see Appendix A. We note that the module is accessible from the FORTRAN 2003 SUNDIALS integrators without separately linking to the `libsundials_fsunmatrixdense_mod` library.

**FORTRAN 77 interface functions**

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.

### 7.6 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:

```c
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 \( s_{\text{mu}} - \mu - 1 \) give access to extra storage elements required by the LU decomposition function. Finally, \( \text{cols}[j][i-j+s_{\text{mu}}] \) 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.

### 7.6.1 SUNMatrix_Band accessor macros

The following macros are provided to access the content of a SUNMATRIX_BAND matrix. The prefix \( \text{SM} \) in the names denotes that these macros are for SUNMatrix implementations, and the suffix \( \_B \) denotes that these are specific to the banded version.

- **SM\_CONTENT\_B**

  This routine gives access to the contents of the banded SUNMatrix.

  The assignment \( \text{A\_cont} = \text{SM\_CONTENT\_B}(\text{A}) \) sets \( \text{A\_cont} \) to be a pointer to the banded SUNMatrix content structure.
7.6 The SUNMatrix Band implementation

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)->ml )
#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 x N band matrix A, where 0 ≤ i, j ≤ N − 1. The location (i,j) should further satisfy j−mu ≤ i ≤ j+ml.
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 x N band matrix A, 0 ≤ j ≤ 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 ml.
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 ≤ i ≤ j+ml.

Implementation:
#define SM_COLUMN_B(A,j) ( ((SM_CONTENT_B(A)->cols)[j])+SM SUBAND_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 SUBAND_B(A) )
7.6.2 SUNMatrix_Band functions

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). All the standard matrix operations listed in 7.2 with the suffix _Band appended are callable via the FORTRAN 2003 interface by prepending an 'F' (e.g. FSUNMatCopy_Band).

The module SUNMatrix_Band provides the following additional user-callable routines:

**SUNBandMatrix**
Prototype SUNMatrix SUNBandMatrix(sunindextype N, sunindextype mu, sunindextype ml)
Description This constructor function creates and allocates memory for a banded SUNMatrix. Its arguments are the matrix size, N, and the upper and lower half-bandwidths of the matrix, mu and ml. The stored upper bandwidth is set to mu+ml to accommodate subsequent factorization in the SUNLINSOL_BAND and SUNLINSOL_LAPACKBAND modules.
F2003 Name This function is callable as FSUNBandMatrix when using the Fortran 2003 interface module.

**SUNBandMatrixStorage**
Prototype SUNMatrix SUNBandMatrixStorage(sunindextype N, sunindextype mu, sunindextype ml, sunindextype smu)
Description 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.

*Note: it is strongly recommended that users call the default constructor, SUNBandMatrix, in all standard use cases. This advanced constructor is used internally within SUNDIALS solvers, and is provided to users who require banded matrices for non-default purposes.*

**SUNBandMatrix_Print**
Prototype void SUNBandMatrix_Print(SUNMatrix A, FILE* outfile)
Description 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.

**SUNBandMatrix_ROWS**
Prototype sunindextype SUNBandMatrix_ROWS(SUNMatrix A)
Description This function returns the number of rows in the banded SUNMatrix.
F2003 Name This function is callable as FSUNBandMatrix_ROWS when using the Fortran 2003 interface module.
### SUNBandMatrix::Columns

**Prototype**

sunindextype SUNBandMatrix::Columns(SUNMatrix A)

**Description**

This function returns the number of columns in the banded SUNMatrix.

**F2003 Name**

This function is callable as FSUNBandMatrix::Columns when using the Fortran 2003 interface module.

### SUNBandMatrix::LowerBandwidth

**Prototype**

sunindextype SUNBandMatrix::LowerBandwidth(SUNMatrix A)

**Description**

This function returns the lower half-bandwidth of the banded SUNMatrix.

**F2003 Name**

This function is callable as FSUNBandMatrix::LowerBandwidth when using the Fortran 2003 interface module.

### SUNBandMatrix::UpperBandwidth

**Prototype**

sunindextype SUNBandMatrix::UpperBandwidth(SUNMatrix A)

**Description**

This function returns the upper half-bandwidth of the banded SUNMatrix.

**F2003 Name**

This function is callable as FSUNBandMatrix::UpperBandwidth when using the Fortran 2003 interface module.

### SUNBandMatrix::StoredUpperBandwidth

**Prototype**

sunindextype SUNBandMatrix::StoredUpperBandwidth(SUNMatrix A)

**Description**

This function returns the stored upper half-bandwidth of the banded SUNMatrix.

**F2003 Name**

This function is callable as FSUNBandMatrix::StoredUpperBandwidth when using the Fortran 2003 interface module.

### SUNBandMatrix::LDim

**Prototype**

sunindextype SUNBandMatrix::LDim(SUNMatrix A)

**Description**

This function returns the length of the leading dimension of the banded SUNMatrix.

**F2003 Name**

This function is callable as FSUNBandMatrix::LDim when using the Fortran 2003 interface module.

### SUNBandMatrix::Data

**Prototype**

realtype* SUNBandMatrix::Data(SUNMatrix A)

**Description**

This function returns a pointer to the data array for the banded SUNMatrix.

**F2003 Name**

This function is callable as FSUNBandMatrix::Data when using the Fortran 2003 interface module.

### SUNBandMatrix::Cols

**Prototype**

realtype** SUNBandMatrix::Cols(SUNMatrix A)

**Description**

This function returns a pointer to the cols array for the banded SUNMatrix.
Description of the SUNMatrix module

**SUNBandMatrix_Column**

Prototype: `realtype* SUNBandMatrix_Column(SUNMatrix A, sunindextype j)`

Description: 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 $ml$.

F2003 Name: This function is callable as `FSUNBandMatrix_Column` when using the Fortran 2003 interface module.

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.

7.6.3 SUNMatrix_Band Fortran interfaces

The SUNMATRIX_BAND module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

FORTRAN 2003 interface module

The `fsunmatrix_band_mod` FORTRAN module defines interfaces to most SUNMATRIX_BAND C functions using the intrinsic iso_c_binding module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading ‘F’. For example, the function SUNBandMatrix is interfaced as `FSUNBandMatrix`.

The FORTRAN 2003 SUNMATRIX_BAND interface module can be accessed with the `use` statement, i.e. `use fsunmatrix_band_mod`, and linking to the library `libsundials_fsunmatrixband_mod.lib` in addition to the C library. For details on where the library and module file `fsunmatrix_band_mod.mod` are installed see Appendix A. We note that the module is accessible from the FORTRAN 2003 SUNDIALS integrators without separately linking to the `libsundials_fsunmatrixband_mod` library.

FORTRAN 77 interface functions

For solvers that include a FORTRAN interface module, the SUNMATRIX_BAND module also includes the FORTRAN-callable function `FSUNBandMatInit(code, N, mu, ml, 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`, and `ml` 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, ier)` initializes this SUNMATRIX_BAND module for storing the mass matrix.
7.7 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:

```c
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 on 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

```c
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

```c
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

```c
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.

### 7.7.1 SUNMatrix_Sparse accessor macros

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.
7.7 The SUNMatrix_Sparse implementation

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.
Implementation:

#define SM_CONTENT_S(A) ( (SUNMatrixContent_Sparse)(A->content) )

- 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 \texttt{A\_rows = SM\_ROWS\_S(A)} sets \texttt{A\_rows} to be the number of rows in the matrix \texttt{A}. Similarly, the assignment \texttt{SM\_COLUMNS\_S(A) = A\_cols} sets the number of columns in \texttt{A} to equal \texttt{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 \texttt{A\_data = SM\_DATA\_S(A)} sets \texttt{A\_data} to be a pointer to the first component of the data array for the sparse SUNMatrix \texttt{A}. The assignment \texttt{SM\_DATA\_S(A) = A\_data} sets the data array of \texttt{A} to be \texttt{A\_data} by storing the pointer \texttt{A\_data}.

Similarly, the assignment \texttt{A\_indexvals = SM\_INDEXVALS\_S(A)} sets \texttt{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 \texttt{A}. The assignment \texttt{A\_indexptrs = SM\_INDEXPTRS\_S(A)} sets \texttt{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 )

7.7.2 SUNMatrix_Sparse functions

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). All the standard matrix operations listed in 7.2 with the suffix \_Sparse appended are callable via the FORTRAN 2003 interface by prepending an ‘F’ (e.g. FSUNMatCopy\_Sparse).

The module SUNMATRIX\_SPARSE provides the following additional user-callable routines:

<table>
<thead>
<tr>
<th>Prototype</th>
<th>Description</th>
<th>F2003 Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUNMatrix SUNSparseMatrix(sunindextype M, sunindextype N, sunindextype NNZ, int sparsetype)</td>
<td>This function creates and allocates memory for a sparse SUNMatrix. Its arguments are the number of rows and columns of the matrix, \texttt{M} and \texttt{N}, the maximum number of nonzeros to be stored in the matrix, \texttt{NNZ}, and a flag \texttt{sparsetype} indicating whether to use CSR or CSC format (valid arguments are \texttt{CSR}_MAT or \texttt{CSC}_MAT).</td>
<td>This function is callable as FSUNSparseMatrix when using the Fortran 2003 interface module.</td>
</tr>
</tbody>
</table>
7.7 The SUNMatrix_Sparse implementation

**SUNSparseFromDenseMatrix**

Prototype:  
```c
SUNMatrix SUNSparseFromDenseMatrix(SUNMatrix A, realtype droptol,
    int sparsetype);
```

Description:  
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.

F2003 Name:  
This function is callable as `FSUNSparseFromDenseMatrix` when using the Fortran 2003 interface module.

**SUNSparseFromBandMatrix**

Prototype:  
```c
SUNMatrix SUNSparseFromBandMatrix(SUNMatrix A, realtype droptol,
    int sparsetype);
```

Description:  
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.

F2003 Name:  
This function is callable as `FSUNSparseFromBandMatrix` when using the Fortran 2003 interface module.

**SUNSparseMatrix_Realloc**

Prototype:  
```c
int SUNSparseMatrix_Realloc(SUNMatrix A)
```

Description:  
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).

F2003 Name:  
This function is callable as `FSUNSparseMatrix_Realloc` when using the Fortran 2003 interface module.

**SUNSparseMatrix_Reallocate**

Prototype:  
```c
int SUNSparseMatrix_Reallocate(SUNMatrix A, sunindextype NNZ)
```

Description:  
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).

F2003 Name:  
This function is callable as `FSUNSparseMatrix_Reallocate` when using the Fortran 2003 interface module.
Description of the SUNMatrix module

**SUNSparseMatrix_Print**

Prototype: `void SUNSparseMatrix_Print(SUNMatrix A, FILE* outfile)`

Description: 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.

**SUNSparseMatrix_Rows**

Prototype: `sunindextype SUNSparseMatrix_Rows(SUNMatrix A)`

Description: This function returns the number of rows in the sparse SUNMatrix.

F2003 Name: This function is callable as `FSUNSparseMatrix_Rows` when using the Fortran 2003 interface module.

**SUNSparseMatrix_Columns**

Prototype: `sunindextype SUNSparseMatrix_Columns(SUNMatrix A)`

Description: This function returns the number of columns in the sparse SUNMatrix.

F2003 Name: This function is callable as `FSUNSparseMatrix_Columns` when using the Fortran 2003 interface module.

**SUNSparseMatrix_NNZ**

Prototype: `sunindextype SUNSparseMatrix_NNZ(SUNMatrix A)`

Description: This function returns the number of entries allocated for nonzero storage for the sparse SUNMatrix.

F2003 Name: This function is callable as `FSUNSparseMatrix_NNZ` when using the Fortran 2003 interface module.

**SUNSparseMatrix_NP**

Prototype: `sunindextype SUNSparseMatrix_NP(SUNMatrix A)`

Description: 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.

F2003 Name: This function is callable as `FSUNSparseMatrix_NP` when using the Fortran 2003 interface module.

**SUNSparseMatrix_SparseType**

Prototype: `int SUNSparseMatrix_SparseType(SUNMatrix A)`

Description: This function returns the storage type (`CSR_MAT` or `CSC_MAT`) for the sparse SUNMatrix.

F2003 Name: This function is callable as `FSUNSparseMatrix_SparseType` when using the Fortran 2003 interface module.

**SUNSparseMatrix_Data**

Prototype: `realtype* SUNSparseMatrix_Data(SUNMatrix A)`

Description: This function returns a pointer to the data array for the sparse SUNMatrix.

F2003 Name: This function is callable as `FSUNSparseMatrix_Data` when using the Fortran 2003 interface module.
7.7 The SUNMatrix_Sparse implementation

**SUNSparseMatrix_IndexValues**

Prototype: `sunindextype* SUNSparseMatrix_IndexValues(SUNMatrix A)`

Description: 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.

F2003 Name: This function is callable as `FSUNSparseMatrix_IndexValues` when using the Fortran 2003 interface module.

**SUNSparseMatrix_IndexPointers**

Prototype: `sunindextype* SUNSparseMatrix_IndexPointers(SUNMatrix A)`

Description: 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.

F2003 Name: This function is callable as `FSUNSparseMatrix_IndexPointers` when using the Fortran 2003 interface module.

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.

### 7.7.3 SUNMatrix_Sparse Fortran interfaces

The `sunmatrix_sparse` module provides a Fortran 2003 module as well as Fortran 77 style interface functions for use from Fortran applications.

**FORTRAN 2003 interface module**

The `fsunmatrix_sparse_mod` Fortran module defines interfaces to most `sunmatrix_sparse` C functions using the intrinsic `iso_c_binding` module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading 'F'. For example, the function `SUNSparseMatrix` is interfaced as `FSUNSparseMatrix`.

The Fortran 2003 `sunmatrix_sparse` interface module can be accessed with the `use` statement, i.e. `use fsunmatrix_sparse_mod`, and linking to the library `libsundials_fsunmatrixsparse_mod.lib` in addition to the C library. For details on where the library and module file `fsunmatrix_sparse_mod.mod` are installed see Appendix A. We note that the module is accessible from the FORTRAN 2003 SUNDIALS integrators without separately linking to the `libsundials_fsunmatrixsparse_mod.mod` library.

**FORTRAN 77 interface functions**

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.
7.8 The SUNMatrix_SLUNRloc implementation

The SUNMATRIX_SLUNRLOC implementation of the SUNMATRIX module provided with SUNDIALS is an adapter for the SuperMatrix structure provided by the SuperLU_DIST sparse matrix factorization and solver library written by X. Sherry Li [2, 22, 32, 33]. It is designed to be used with the SUNLINSOL_SUPERLUDIST linear solver discussed in Section 8.10. To this end, it defines the content field of SUNMatrix to be the following structure:

```
struct _SUNMatrixContent_SLUNRloc {
  bool own_data;
  gridinfo_t *grid;
  sunindextype *row_to_proc;
  pdgsmv_comm_t *gsmv_comm;
  SuperMatrix *A_super;
  SuperMatrix *ACS_super;
};
```

A more complete description of the this content field is given below:

- **own_data** - a flag which indicates if the SUNMatrix is responsible for freeing `A_super`
- **grid** - pointer to the SuperLU_DIST structure that stores the 2D process grid
- **row_to_proc** - a mapping between the rows in the matrix and the process it resides on; will be NULL until the SUNMatMatvecSetup routine is called
- **gsmv_comm** - pointer to the SuperLU_DIST structure that stores the communication information needed for matrix-vector multiplication; will be NULL until the SUNMatMatvecSetup routine is called
- **A_super** - pointer to the underlying SuperLU_DIST SuperMatrix with Stype = SLU_NR_loc, Dtype = SLU_D, Mtype = SLU_GE; must have the full diagonal present to be used with SUNMatScaleAddI routine
- **ACS_super** - a column-sorted version of the matrix needed to perform matrix-vector multiplication; will be NULL until the routine SUNMatMatvecSetup routine is called

The header file to include when using this module is `sunmatrix/sunmatrix_slunrloc.h`. The installed module library to link to is `libsundials_sunmatrixslunrloc.lib` where .lib is typically .so for shared libraries and .a for static libraries.

7.8.1 SUNMatrix_SLUNRloc functions

The module SUNMATRIX_SLUNRLOC provides the following user-callable routines:

```c
SUNMatrix_SLUNRloc
Call
A = SUNMatrix_SLUNRloc(Asuper, grid);
Description
The function SUNMatrix_SLUNRloc creates and allocates memory for a SUNMATRIX_SLUNRLOC object.
Arguments
Asuper (SuperMatrix*) a fully-allocated SuperLU_DIST SuperMatrix that the SUNMatrix will wrap; must have Stype = SLU_NR_loc, Dtype = SLU_D, Mtype = SLU_GE to be compatible
grid (gridinfo_t*) the initialized SuperLU_DIST 2D process grid structure
Return value
a SUNMatrix object if Asuper is compatible else NULL
Notes
```
The SUNMatrix SLUNRloc implementation

**SUNMatrix_SLUNRloc_print**

- **Call**: SUNMatrix_SLUNRloc_print(A, fp);
- **Description**: The function SUNMatrix_SLUNRloc_print prints the underlying SuperMatrix content.
- **Arguments**:
  - A (SUNMatrix) the matrix to print
  - fp (FILE) the file pointer used for printing
- **Return value**: void
- **Notes**

**SUNMatrix_SLUNRloc_SuperMatrix**

- **Call**: Asuper = SUNMatrix_SLUNRloc_SuperMatrix(A);
- **Description**: The function SUNMatrix_SLUNRloc_SuperMatrix provides access to the underlying SuperLU_DIST SuperMatrix of A.
- **Arguments**:
  - A (SUNMatrix) the matrix to access
- **Return value**: SuperMatrix*
- **Notes**

**SUNMatrix_SLUNRloc_ProcessGrid**

- **Call**: grid = SUNMatrix_SLUNRloc_ProcessGrid(A);
- **Description**: The function SUNMatrix_SLUNRloc_ProcessGrid provides access to the SuperLU_DIST gridinfo_t structure associated with A.
- **Arguments**:
  - A (SUNMatrix) the matrix to access
- **Return value**: gridinfo_t*
- **Notes**

**SUNMatrix_SLUNRloc_OwnData**

- **Call**: does_own_data = SUNMatrix_SLUNRloc_OwnData(A);
- **Description**: The function SUNMatrix_SLUNRloc_OwnData returns true if the SUNMatrix object is responsible for freeing A_super, otherwise it returns false.
- **Arguments**:
  - A (SUNMatrix) the matrix to access
- **Return value**: boolean
- **Notes**

The SUNMATRIX_SLUNRLOC module defines implementations of all generic SUNMatrix operations listed in Table 7.2:

- SUNMatGetID_SLUNRloc - returns SUNMATRIX_SLUNRLOC
- SUNMatClone_SLUNRloc
- SUNMatDestroy_SLUNRloc
- SUNMatSpace_SLUNRloc - this only returns information for the storage within the matrix interface, i.e. storage for row_to_proc
- SUNMatZero_SLUNRloc
- SUNMatCopy_SLUNRloc
• **SUNMatScaleAdd** - performs $A = cA + B$, but $A$ and $B$ must have the same sparsity pattern
• **SUNMatScaleAddI** - performs $A = cA + I$, but the diagonal of $A$ must be present
• **SUNMatMatvecSetup** - initializes the SuperLU_DIST parallel communication structures needed to perform a matrix-vector product; only needs to be called before the first call to **SUNMatMatvec** or if the matrix changed since the last setup
• **SUNMatMatvec**

The SUNMATRIX module requires that the complete diagonal, i.e. nonzeros and zeros, is present in order to use the SUNMatScaleAddI operation.
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 iterative (matrix-based or matrix-free) 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 also 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

$$
\tilde{A}\tilde{x} = \tilde{b}
$$

where

$$
\begin{align*}
\tilde{A} &= S_1 P_1^{-1} A P_2^{-1} S_2^{-1}, \\
\tilde{b} &= S_1 P_1^{-1} b, \\
\tilde{x} &= S_2 P_2 x,
\end{align*}
$$

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 $P_2 x$. 
The scaling matrices are chosen so that $S_1P_1^{-1}b$ and $S_2P_2x$ have dimensionless components. If preconditioning is done on the left only ($P_2 = I$), by a matrix $P$, then $S_2$ must be a scaling for $x$, while $S_1$ is a scaling for $P^{-1}b$, and so may also be taken as a scaling for $x$. Similarly, if preconditioning is done on the right only ($P_1 = I$ and $P_2 = P$), then $S_1$ must be a scaling for $b$, while $S_2$ is a scaling for $Px$, and may also be taken as a scaling for $b$.

SUNdIALS packages request that iterative linear solvers stop based on the 2-norm of the scaled preconditioned residual meeting a prescribed tolerance

$$\left\| \tilde{b} - \tilde{A}\tilde{x} \right\|_2 < \text{tol}.$$ 

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 (see §8.4.2 for more details). In this case, they instead request that iterative linear solvers stop based on the criteria

$$\left\| P_1^{-1}b - P_1^{-1}Ax \right\|_2 < \text{tol}.$$ 

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.

For users interested in providing their own SUNLINSOL module, the following section presents the SUNLINSOL API and its implementation beginning with the definition of SUNLINSOL functions in sections 8.1.1 – 8.1.3. This is followed by the definition of functions supplied to a linear solver implementation in section 8.1.4. A table of linear solver return codes is given in section 8.1.5. The SUNLinearSolver type and the generic SUNLINSOL module are defined in section 8.1.6. The section 8.2 discusses compatibility between the SUNDIALS-provided SUNLINSOL modules and SUNMATRIX modules. Section 8.3 lists the requirements for supplying a custom SUNLINSOL module and discusses some intended use cases. Users wishing to supply their own SUNLINSOL module are encouraged to use the SUNLINSOL implementations provided with SUNDIALS as a template for supplying custom linear solver modules. The SUNLINSOL functions required by this SUNDIALS package as well as other package specific details are given in section 8.4. The remaining sections of this chapter present the SUNLINSOL modules provided with SUNDIALS.

### 8.1 The SUNLinearSolver API

The SUNLINSOL API defines several linear solver operations that enable SUNDIALS packages to utilize any SUNLINSOL implementation that provides the required functions. These functions can be divided into three categories. The first are the core linear solver functions. The second group of functions consists of set routines to supply the linear solver with functions provided by the SUNDIALS time integrators and to modify solver parameters. The final group consists of get routines for retrieving linear solver statistics. All of these functions are defined in the header file sundials/sundials_linear solver.h.

#### 8.1.1 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.
8.1 The SUNLinearSolver API

**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, iterative, or matrix-iterative) from the abstract `SUNLinearSolver` interface.

Arguments

LS (SUNLinearSolver) a SUNLINSOL object.

Return value

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

- **SUNLINEARSOLVER_DIRECT** – 0, the SUNLINSOL module requires a matrix, and computes an “exact” solution to the linear system defined by that matrix.
- **SUNLINEARSOLVER_ITERATIVE** – 1, the SUNLINSOL module does not require a matrix (though one may be provided), and computes an inexact solution to the linear system using a matrix-free iterative algorithm. That is, it solves the linear system defined by the package-supplied `ATimes` routine (see `SUNLinSolSetATimes` below), even if that linear system differs from the one encoded in the matrix object (if one is provided). As the solver computes the solution only inexact (or may diverge), the linear solver should check for solution convergence/accuracy as appropriate.
- **SUNLINEARSOLVER_MATRIX_ITERATIVE** – 2, the SUNLINSOL module requires a matrix, and computes an inexact solution to the linear system defined by that matrix using an iterative algorithm. That is, it solves the linear system defined by the matrix object even if that linear system differs from that encoded by the package-supplied `ATimes` routine. As the solver computes the solution only inexact (or may diverge), the linear solver should check for solution convergence/accuracy as appropriate.

Notes

See section 8.3.1 for more information on intended use cases corresponding to the linear solver type.

**SUNLinSolInitialize**

Call
```
retval = SUNLinSolInitialize(LS);
```

Description

The `required` function `SUNLinSolInitialize` performs linear solver initialization (assuming 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.

**SUNLinSolSetup**

Call
```
retval = SUNLinSolSetup(LS, A);
```

Description

The `required` function `SUNLinSolSetup` performs any linear solver setup needed, based on an updated system `SUNMatrix` `A`. 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

LS (SUNLinearSolver) a SUNLINSOL object.

A (SUNMatrix) 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.
Description of the SUNLinearSolver module

SUNLinSolSolve
Call
retval = SUNLinSolSolve(LS, A, x, b, tol);
Description The required function SUNLinSolSolve solves a linear system \( Ax = b \).
Arguments
LS (SUNLinearSolver) a SUNLINSOL object.
A (SUNMatrix) a SUNMATRIX object.
x (N_Vector) a NVECTOR object containing the initial guess for the solution of the linear system, and the solution to the linear system upon return.
b (N_Vector) a NVECTOR object containing the linear system right-hand side.
tol (realtype) 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
Direct solvers: can ignore the tol argument.
Matrix-free solvers: (those that identify as SUNLINEARSOLVER_ITERATIVE) can ignore the SUNMATRIX input \( A \), and should instead rely on the matrix-vector product function supplied through the routine SUNLinSolSetATimes.
Iterative solvers: (those that identify as SUNLINEARSOLVER_ITERATIVE or SUNLINEARSOLVER_MATRIX_ITERATIVE) should attempt to solve to the specified tolerance tol in a weighted 2-norm. If the solver does not support scaling then it should just use a 2-norm.

SUNLinSolFree
Call
retval = SUNLinSolFree(LS);
Description The optional function SUNLinSolFree frees memory allocated by the linear solver.
Arguments
LS (SUNLinearSolver) a SUNLINSOL object.
Return value This should return zero for a successful call and a negative value for a failure.

8.1.2 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
retval = SUNLinSolSetATimes(LS, A_data, ATimes);
Description The function SUNLinSolSetATimes is required for matrix-free linear solvers; otherwise it is optional.
This routine provides an ATimesFn function pointer, as well as a void* 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.
Arguments
LS (SUNLinearSolver) a SUNLINSOL object.
A_data (void*) data structure passed to ATimes.
ATimes (ATimesFn) function pointer implementing the matrix-vector product routine.
8.1 The SUNLinearSolver API

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.

**SUNLinSolSetPreconditioner**

Call `retval = SUNLinSolSetPreconditioner(LS, Pdata, Pset, Psol);`

Description The optional function SUNLinSolSetPreconditioner provides PSetupFn and PSolveFn function pointers that implement the preconditioner solves $P_1^{-1}$ and $P_2^{-1}$ from equations (8.1)-(8.2). This routine will be called by a SUNDIALS package, which will provide translation between the generic Pset and Psol calls and the package- or user-supplied routines.

Arguments
- `LS` (SUNLinearSolver) a SUNLINSOL object.
- `Pdata` (void*) data structure passed to both Pset and Psol.
- `Pset` (PSetupFn) function pointer implementing the preconditioner setup.
- `Psol` (PSolveFn) function pointer implementing the preconditioner solve.

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.

**SUNLinSolSetScalingVectors**

Call `retval = SUNLinSolSetScalingVectors(LS, s1, s2);`

Description The optional function SUNLinSolSetScalingVectors provides left/right scaling vectors for the linear system solve. Here, $s_1$ and $s_2$ are NVECTOR of positive scale factors containing the diagonal of the matrices $S_1$ and $S_2$ from equations (8.1)-(8.2), respectively. Neither of these vectors need to be tested for positivity, and a NULL argument for either indicates that the corresponding scaling matrix is the identity.

Arguments
- `LS` (SUNLinearSolver) a SUNLINSOL object.
- `s1` (N_Vector) diagonal of the matrix $S_1$
- `s2` (N_Vector) diagonal of the matrix $S_2$

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.

### 8.1.3 SUNLinearSolver get functions

The following get functions allow SUNDIALS packages to retrieve results from a 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

**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
Description of the SUNLinearSolver module

**SUNLinSolResid**

Call

```c
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 set to NULL in the implementation.

**SUNLinSolLastFlag**

Call

```c
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

**SUNLinSolSpace**

Call

```c
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.
- `lrw` (long int*) the number of realtype words stored by the linear solver.
- `liw` (long int*) 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.

### 8.1.4 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

```c
typedef int (*ATimesFn)(void *A_data, N_Vector v, N_Vector z);
```

Purpose

These functions compute the action of a matrix on a vector, performing the operation \( z = Av \). Memory for \( z \) should already be allocated prior to calling this function. The vector \( v \) should be left unchanged.
8.1 The SUNLinearSolver API

Arguments

A_data is a pointer to client data, the same as that supplied to SUNLinSolSetATimes.

v is the input vector to multiply.

z is the output vector computed.

Return value

This routine should return 0 if successful and a non-zero value if unsuccessful.

PSetupFn

Definition

typedef int (*PSetupFn)(void *P_data)

Purpose

These functions set up any requisite problem data in preparation for calls to the corresponding PSolveFn.

Arguments

P_data is a pointer to client data, the same pointer as that supplied to the routine SUNLinSolSetPreconditioner.

Return value

This routine should return 0 if successful and a non-zero value if unsuccessful.

PSolveFn

Definition

typedef int (*PSolveFn)(void *P_data, N_Vector r, N_Vector z, realtype tol, int lr)

Purpose

These functions solve the preconditioner equation \( Pz = r \) for the vector \( z \). Memory for \( z \) should already be allocated prior to calling this function. The parameter P_data is a pointer to any information about \( P \) which the function needs in order to do its job (set up by the corresponding PSetupFn). The parameter lr is input, and indicates whether \( P \) is to be taken as the left preconditioner or the right preconditioner: \( lr = 1 \) for left and \( lr = 2 \) for right. If preconditioning is on one side only, \( lr \) 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.

8.1.5 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 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUNLS_SUCCESS</td>
<td>0</td>
<td>successful call or converged solve</td>
</tr>
<tr>
<td>SUNLS_MEM_NULL</td>
<td>-1</td>
<td>the memory argument to the function is NULL</td>
</tr>
<tr>
<td>SUNLS_ILL_INPUT</td>
<td>-2</td>
<td>an illegal input has been provided to the function</td>
</tr>
<tr>
<td>SUNLS_MEM_FAIL</td>
<td>-3</td>
<td>failed memory access or allocation</td>
</tr>
<tr>
<td>SUNLS_ATIMES_FAIL_UNREC</td>
<td>-4</td>
<td>an unrecoverable failure occurred in the ATimes routine</td>
</tr>
<tr>
<td>SUNLS_PSET_FAIL_UNREC</td>
<td>-5</td>
<td>an unrecoverable failure occurred in the Pset routine</td>
</tr>
<tr>
<td>SUNLS_PSOVLE_FAIL_UNREC</td>
<td>-6</td>
<td>an unrecoverable failure occurred in the Psolve routine</td>
</tr>
<tr>
<td>SUNLS_PACKAGE_FAIL_UNREC</td>
<td>-7</td>
<td>an unrecoverable failure occurred in an external linear solver package</td>
</tr>
<tr>
<td>SUNLS_GS_FAIL</td>
<td>-8</td>
<td>a failure occurred during Gram-Schmidt orthogonalization</td>
</tr>
<tr>
<td>SUNLS_QRSOL_FAIL</td>
<td>-9</td>
<td>a singular R matrix was encountered in a QR factorization</td>
</tr>
<tr>
<td>SUNLS_RES_REduced</td>
<td>1</td>
<td>an iterative solver reduced the residual, but did not converge to the desired tolerance</td>
</tr>
<tr>
<td>SUNLS_CONV_FAIL</td>
<td>2</td>
<td>an iterative solver did not converge (and the residual was not reduced)</td>
</tr>
<tr>
<td>SUNLS_ATIMES_FAIL_REC</td>
<td>3</td>
<td>a recoverable failure occurred in the ATimes routine</td>
</tr>
<tr>
<td>SUNLS_PSET_FAIL_REC</td>
<td>4</td>
<td>a recoverable failure occurred in the Pset routine</td>
</tr>
<tr>
<td>SUNLS_PSOVLE_FAIL_REC</td>
<td>5</td>
<td>a recoverable failure occurred in the Psolve routine</td>
</tr>
<tr>
<td>SUNLS_PACKAGE_FAIL_REC</td>
<td>6</td>
<td>a recoverable failure occurred in an external linear solver package</td>
</tr>
<tr>
<td>SUNLS_QRFACFact_FAIL</td>
<td>7</td>
<td>a singular matrix was encountered during a QR factorization</td>
</tr>
<tr>
<td>SUNLS_LUFACT_FAIL</td>
<td>8</td>
<td>a singular matrix was encountered during a LU factorization</td>
</tr>
</tbody>
</table>

8.1.6 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

```c
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

```c
struct _generic_SUNLinearSolver_Ops {
  SUNLinearSolver_Type (*gettype)(SUNLinearSolver);
```
8.2 Compatibility of SUNLinearSolver modules

The generic SUNLINSOL module defines and implements the linear solver operations defined in Sections 8.1.1-8.1.3. 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:

```c
int SUNLinSolInitialize(SUNLinearSolver S)
{
    return ((int) S->ops->initialize(S));
}
```

### 8.2 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>
<thead>
<tr>
<th>Linear Solver Interface</th>
<th>Dense Matrix</th>
<th>Banded Matrix</th>
<th>Sparse Matrix</th>
<th>SLUNRloc Matrix</th>
<th>User Supplied</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dense</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Band</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LapackDense</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LapackBand</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>KLU</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SuperLU_DIST</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SUPERLUMT</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

continued on next page
8.3 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 section 8.4 to determine which SUNLINSOL operations are required for this SUNDIALS package.

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.

8.3.1 Intended use cases

The SUNLINSOL (and SUNMATRIX) APIs are designed to require a minimal set of routines to ease interfacing with custom or third-party linear solver libraries. External solvers provide similar routines with the necessary functionality and thus will require minimal effort to wrap within custom SUNMATRIX and SUNLINSOL implementations. Sections 7.4 and 8.4 include a list of the required set of routines that compatible SUNMATRIX and SUNLINSOL implementations must provide. As SUNDIALS packages utilize generic SUNLINSOL modules allowing for user-supplied SUNLinearSolver implementations, there exists a wide range of possible linear solver combinations. Some intended use cases for both the SUNDIALS-provided and user-supplied SUNLINSOL modules are discussed in the following sections.

Direct linear solvers

Direct linear solver modules require a matrix and compute an ‘exact’ solution to the linear system defined by the matrix. Multiple matrix formats and associated direct linear solvers are supplied with SUNDIALS through different SUNMATRIX and SUNLINSOL implementations. SUNDIALS packages strive to amortize the high cost of matrix construction by reusing matrix information for multiple nonlinear iterations. As a result, each package’s linear solver interface recomputes Jacobian information as infrequently as possible.

Alternative matrix storage formats and compatible linear solvers that are not currently provided by, or interfaced with, SUNDIALS can leverage this infrastructure with minimal effort. To do so, a user must implement custom SUNMATRIX and SUNLINSOL wrappers for the desired matrix format and/or linear solver following the APIs described in Chapters 7 and 8. This user-supplied SUNLINSOL module must then self-identify as having SUNLINEARSOLVER_DIRECT type.
Matrix-free iterative linear solvers

Matrix-free iterative linear solver modules do not require a matrix and compute an inexact solution to the linear system defined by the package-supplied ATimes routine. SUNDIALS supplies multiple scaled, preconditioned iterative linear solver (SPILS) SUNLINSOL modules that support scaling to allow users to handle non-dimensionalization (as best as possible) within each SUNDIALS package and retain variables and define equations as desired in their applications. For linear solvers that do not support left/right scaling, the tolerance supplied to the linear solver is adjusted to compensate (see section 8.4.2 for more details); however, this use case may be non-optimal and cannot handle situations where the magnitudes of different solution components or equations vary dramatically within a single problem.

To utilize alternative linear solvers that are not currently provided by, or interfaced with, SUNDIALS a user must implement a custom SUNLINSOL wrapper for the linear solver following the API described in Chapter 8. This user-supplied SUNLINSOL module must then self-identify as having SUNLINEARSOLVER_ITERATIVE type.

Matrix-based iterative linear solvers (reusing $A$)

Matrix-based iterative linear solver modules require a matrix and compute an inexact solution to the linear system defined by the matrix. This matrix will be updated infrequently and reused across multiple solves to amortize cost of matrix construction. As in the direct linear solver case, only wrappers for the matrix and linear solver in SUNMATRIX and SUNLINSOL implementations need to be created to utilize a new linear solver. This user-supplied SUNLINSOL module must then self-identify as having SUNLINEARSOLVER_MATRIX_ITERATIVE type.

At present, SUNDIALS has one example problem that uses this approach for wrapping a structured-grid matrix, linear solver, and preconditioner from the hypre library that may be used as a template for other customized implementations (see examples/arkode/CXX_parhyp/ark_heat2D_hypre.cpp).

Matrix-based iterative linear solvers (current $A$)

For users who wish to utilize a matrix-based iterative linear solver module where the matrix is purely for preconditioning and the linear system is defined by the package-supplied ATimes routine, we envision two current possibilities.

The preferred approach is for users to employ one of the SUNDIALS spils SUNLINSOL implementations (SUNLINSOL_SPGMR, SUNLINSOL_SPPGMRES, SUNLINSOL_SPCG, SUNLINSOL_SPTQMR, or SUNLINSOL_PC) as the outer solver. The creation and storage of the preconditioner matrix, and interfacing with the corresponding linear solver, can be handled through a package’s preconditioner ‘setup’ and ‘solve’ functionality (see §4.5.8.2) without creating SUNMATRIX and SUNLINSOL implementations. This usage mode is recommended primarily because the SUNDIALS-provided spils modules support the scaling as described above.

A second approach supported by the linear solver APIs is as follows. If the SUNLINSOL implementation is matrix-based, self-identifies as having SUNLINEARSOLVER_ITERATIVE type, and also provides a non-NULL SUNLinSolSetATimes routine, then each SUNDIALS package will call that routine to attach its package-specific matrix-vector product routine to the SUNLINSOL object. The SUNDIALS package will then call the SUNLINSOL-provided SUNLinSolSetup routine (infrequently) to update matrix information, but will provide current matrix-vector products to the SUNLINSOL implementation through the package-supplied ATimesFn routine.

8.4 IDA SUNLinearSolver interface

Table 8.3 below lists the SUNLINSOL module linear solver functions used within the IDALS interface. 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. Note:

1. Although IDALS does not call SUNLinSolLastFlag directly, this routine is available for users to query linear solver issues directly.

2. Although IDALS does not call SUNLinSolFree directly, this routine should be available for users to call when cleaning up from a simulation.

Table 8.3: List of linear solver function usage in the IDALS interface

<table>
<thead>
<tr>
<th>Function</th>
<th>DIRECT</th>
<th>ITERATIVE</th>
<th>MATRIX-ITERATIVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUNLinSolGetType</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SUNLinSolSetATimes</td>
<td>†</td>
<td>✓</td>
<td>†</td>
</tr>
<tr>
<td>SUNLinSolSetPreconditioner</td>
<td>†</td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td>SUNLinSolSetScalingVectors</td>
<td>†</td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td>SUNLinSolInitialize</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SUNLinSolSetup</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SUNLinSolSolve</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SUNLinSolNumIters</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SUNLinSolResid</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>^SUNLinSolLastFlag</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>^SUNLinSolFree</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUNLinSolSpace</td>
<td>†</td>
<td>†</td>
<td>†</td>
</tr>
</tbody>
</table>

Since there are a wide range of potential SUNLINSOL use cases, the following subsections describe some details of the IDALS interface, in the case that interested users wish to develop custom SUNLINSOL modules.

8.4.1 Lagged matrix information

If the SUNLINSOL object self-identifies as having type SUNLINEARSOLVER_DIRECT or SUNLINEARSOLVER_MATRIX_ITERATIVE, then the SUNLINSOL object solves a linear system defined by a SUNMATRIX object. CVLS will update the matrix information infrequently according to the strategies outlined in §2.1. When solving a linear system \( J \bar{x} = b \), it is likely that the value \( \bar{\alpha} \) used to construct \( J \) differs from the current value of \( \alpha \) in the BDF method, since \( J \) is updated infrequently. Therefore, after calling the SUNLINSOL-provided SUNLinSolSolve routine, we test whether \( \alpha/\bar{\alpha} \neq 1 \), and if this is the case we scale the solution \( \bar{x} \) to obtain the desired linear system solution \( x \) via

\[
x = \frac{2}{1 + \alpha/\bar{\alpha}} \bar{x}.
\]  

(8.3)

For values of \( \alpha/\bar{\alpha} \) that are “close” to 1, this rescaling approximately solves the original linear system.
8.4.2 Iterative linear solver tolerance

If the sunlinsol object self-identifies as having type SUNLINEARSOLVER_ITERATIVE or SUNLINEARSOLVER_MATRIX_ITERATIVE then idals will set the input tolerance delta as described in §2.1. However, if the iterative linear solver does not support scaling matrices (i.e., the SUNLinSolSetScalingVectors routine is NULL), then idals will attempt to adjust the linear solver tolerance to account for this lack of functionality. To this end, the following assumptions are made:

1. All solution components have similar magnitude; hence the error weight vector W used in the WRMS norm (see §2.1) should satisfy the assumption

\[ W_i \approx W_{\text{mean}}, \quad \text{for} \quad i = 0, \ldots, n - 1. \]

2. The sunlinsol object uses a standard 2-norm to measure convergence.

Since ida uses identical left and right scaling matrices, \( S_1 = S_2 = S = \text{diag}(W) \), then the linear solver convergence requirement is converted as follows (using the notation from equations (8.1)-(8.2)):

\[
\left\| \tilde{b} - \tilde{A} \tilde{x} \right\|_2 < \text{tol} \\
\Leftrightarrow \left\| SP^{-1}_1 b - SP^{-1}_1 Ax \right\|_2 < \text{tol} \\
\Leftrightarrow \sum_{i=0}^{n-1} [W_i (P^{-1}_1(b - Ax))^i]^2 < \text{tol}^2 \\
\Leftrightarrow W_{\text{mean}}^2 \sum_{i=0}^{n-1} [(P^{-1}_1(b - Ax))^i]^2 < \text{tol}^2 \\
\Leftrightarrow \sum_{i=0}^{n-1} [(P^{-1}_1(b - Ax))^i]^2 < \left( \frac{\text{tol}}{W_{\text{mean}}} \right)^2 \\
\Leftrightarrow \left\| P^{-1}_1(b - Ax) \right\|_2 < \frac{\text{tol}}{W_{\text{mean}}}
\]

Therefore the tolerance scaling factor

\[ W_{\text{mean}} = \|W\|_2/\sqrt{n} \]

is computed and the scaled tolerance \( \text{delta} = \text{tol}/W_{\text{mean}} \) is supplied to the sunlinsol object.

8.5 The SUNLinearSolver_Dense implementation

This section describes the sunlinsol implementation for solving dense linear systems. The SUNLINEARSOLVER_DENSE module is designed to be used with the corresponding SUNMATRIX_DENSE matrix type, and one of the serial or shared-memory NVVECTOR implementations (NVVECTOR_SERIAL, NVVECTOR_OPENMP, or NVVECTOR_PTHREADS).

To access the SUNLINEARSOLVER_DENSE module, include the header file sunlinsol/sunlinsol_dense.h. We note that the SUNLINEARSOLVER_DENSE module is accessible from SUNDIALS packages without separately linking to the libsundials_sunlinsoldense module library.

8.5.1 SUNLinearSolver_Dense description

This solver is constructed to perform the following operations:

- The “setup” call performs a LU factorization with partial (row) pivoting (\( O(N^3) \) 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 (\( O(N^2) \) cost).
8.5.2 SUNLinearSolver_Dense functions

The SUNLINSOL_DENSE module provides the following user-callable constructor for creating a SUNLinearSolver object.

**Call**

\[ \text{LS} = \text{SUNLinSol_Dense}(y, A); \]

**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.

**Deprecated Name**
For backward compatibility, the wrapper function SUNDenseLinearSolver with identical input and output arguments is also provided.

**F2003 Name**
This function is callable as FSUNLinSol_Dense when using the Fortran 2003 interface module.

The SUNLINSOL_DENSE module defines implementations of all “direct” linear solver operations listed in Sections 8.1.1 – 8.1.3:

- **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 \( N \), \( \text{last\_flag} \), and \( \text{pivots} \).
- **SUNLinSolFree_Dense**

All of the listed operations are callable via the FORTRAN 2003 interface module by prepending an ‘F’ to the function name.

8.5.3 SUNLinearSolver_Dense Fortran interfaces

The SUNLINSOL_DENSE module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.
8.5 The SUNLinearSolver_Dense implementation

FORTRAN 2003 interface module

The fsunlinsol_dense_mod FORTRAN module defines interfaces to all SUNLINSOL_DENSE C functions using the intrinsic iso_c_binding module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading ‘F’. For example, the function SUNLinSol_Dense is interfaced as FSUNLinSol_Dense.

The FORTRAN 2003 SUNLINSOL_DENSE interface module can be accessed with the use statement, i.e. use fsunlinsol_dense_mod, and linking to the library libsundials_fsunlinsoldense_mod.lib in addition to the C library. For details on where the library and module file fsunlinsol_dense_mod.mod are installed see Appendix A. We note that the module is accessible from the FORTRAN 2003 SUNDIALS integrators without separately linking to the libsundials_fsunlinsoldense_mod library.

FORTRAN 77 interface functions

For solvers that include a FORTRAN 77 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 None

Return value ier is an 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.

8.5.4 SUNLinearSolver_Dense content

The SUNLINSOL_DENSE module defines the content field of a SUNLinearSolver as 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.

### 8.6 The SUNLinearSolver_Band implementation

This section describes the SUNLINSOL implementation for solving banded linear systems. The SUNLINSOL_BAND module is designed to be used with the corresponding SUNMATRIX_BAND matrix type, and one of the serial or shared-memory NVVECTOR implementations (NVVECTOR_SERIAL, NVVECTOR_OPENMP, or NVVECTOR_PTHREADS).

To access the SUNLINSOL_BAND module, include the header file `sunlinsol/sunlinsol_band.h`. We note that the SUNLINSOL_BAND module is accessible from SUNDIALS packages *without* separately linking to the `libsundials_sunlinsolband` module library.

#### 8.6.1 SUNLinearSolver_Band description

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 \( \mu_u \) and lower bandwidth \( \mu_l \), then the upper triangular factor \( U \) can have upper bandwidth as big as \( \mu_u = \text{MIN}(N-1, \mu_u+\mu_l) \). The lower triangular factor \( L \) has lower bandwidth \( \mu_l \).

#### 8.6.2 SUNLinearSolver_Band functions

The SUNLINSOL_BAND module provides the following user-callable constructor for creating a SUNLinearSolver object.

```c
SUNLinSol_Band
Call
LS = SUNLinSol_Band(y, A);
```

**Description** The function `SUNLinSol_Band` creates and allocates memory for a band SUNLinearSolver object.

**Arguments**
- `y` (*N_Vector*) a template for cloning vectors needed within the solver
- `A` (*SUNMatrix*) a SUNMATRIX_BAND 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 NVVECTOR and SUNMATRIX implementations. These are currently limited to the SUNMATRIX_BAND matrix type and the NVVECTOR_SERIAL, NVVECTOR_OPENMP, and NVVECTOR_PTHREADS 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 $A$ is allocated with appropriate upper bandwidth storage for the $LU$ factorization.

**Deprecated Name** For backward compatibility, the wrapper function SUNBandLinearSolver with identical input and output arguments is also provided.

**F2003 Name** This function is callable as FSUNLinSolBand when using the Fortran 2003 interface module.

The SUNLINSOL_BAND module defines band implementations of all “direct” linear solver operations listed in Sections 8.1.1 – 8.1.3:

- 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

All of the listed operations are callable via the FORTRAN 2003 interface module by prepending an ‘F’ to the function name.

### 8.6.3 SUNLinearSolver_Band Fortran interfaces

The SUNLINSOL_BAND module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

#### FORTRAN 2003 interface module

The fsunlinsol_band_mod FORTRAN module defines interfaces to all SUNLINSOL_BAND C functions using the intrinsic iso_c_binding module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading ‘F’. For example, the function SUNLinSol_Band is interfaced as FSUNLinSol_Band.

The FORTRAN 2003 SUNLINSOL_BAND interface module can be accessed with the use statement, i.e. use fsunlinsol_band_mod, and linking to the library libsundials_fsunlinsolband_mod.lib in addition to the C library. For details on where the library and module file fsunlinsol_band_mod.mod are installed see Appendix A. We note that the module is accessible from the FORTRAN 2003 SUNDIALS integrators without separately linking to the libsundials_fsunlinsolband_mod library.

#### FORTRAN 77 interface functions

For solvers that include a FORTRAN 77 interface module, the SUNLINSOL_BAND module also includes a Fortran-callable function for creating a SUNLinearSolver object.

```
FSUNBANDLINSOLINIT
```

**Call**

FSUNBANDLINSOLINIT(code, ier)

**Description**
The function FSUNBANDLINSOLINIT can be called for Fortran programs to create a band 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_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  None

Return value  ier is an 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.

8.6.4 SUNLinearSolver_Band content

The SUNLINSOL_BAND module defines the content field of a SUNLinearSolver as the following structure:

```c
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.

8.7 The SUNLinearSolver_LapackDense implementation

This section describes the SUNLINSOL implementation for solving dense linear systems with LAPACK. The SUNLINSOL_LAPACKDENSE module 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).

To access the SUNLINSOL_LAPACKDENSE module, include the header file

```
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 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 64-bit integers for the sunindextype.

8.7.1 SUNLinearSolver_LapackDense description

This solver is constructed to perform the following operations:

- The “setup” call performs a LU factorization with partial (row) pivoting ($O(N^3)$ 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 ($O(N^2)$ cost).

8.7.2 SUNLinearSolver_LapackDense functions

The SUNLINSOL_LAPACKDENSE module provides the following user-callable constructor for creating a SUNLinearSolver object.

```c
SUNLinSol_LapackDense
```

Call

```
LS = SUNLinSol_LapackDense(y, A);
```

Description

The function SUNLinSol_LapackDense creates and allocates memory for a LAPACK-based, 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.

Deprecated Name

For backward compatibility, the wrapper function SUNLapackDense with identical input and output arguments is also provided.

The SUNLINSOL_LAPACKDENSE module defines dense implementations of all “direct” linear solver operations listed in Sections 8.1.1 – 8.1.3:

- 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

### 8.7.3 SUNLinearSolver_LapackDense Fortran interfaces

For solvers that include a FORTRAN 77 interface module, the SUNLINSOL_LAPACKDENSE module also includes a Fortran-callable function for creating a SUNLinearSolver object.

**FSUNLAPACKDENSEINIT**

**Call**

\[ \text{FSUNLAPACKDENSEINIT}(\text{code}, \text{ier}) \]

**Description**

The function **FSUNLAPACKDENSEINIT** can be called for Fortran programs to create a LAPACK-based dense SUNLinearSolver object.

**Arguments**

- \( \text{code} \) (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).

**Return value**

\( \text{ier} \) 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 NVVECTOR and SUNMATRIX 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**

\[ \text{FSUNMASSLAPACKDENSEINIT}(\text{ier}) \]

**Description**

The function **FSUNMASSLAPACKDENSEINIT** can be called for Fortran programs to create a LAPACK-based, dense SUNLinearSolver object for mass matrix linear systems.

**Arguments**

None

**Return value**

\( \text{ier} \) is an 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 NVVECTOR and SUNMATRIX mass-matrix objects have been initialized.

### 8.7.4 SUNLinearSolver_LapackDense content

The SUNLINSOL_LAPACKDENSE module defines the **content** field of a SUNLinearSolver as 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,
- \( \text{pivots} \) - index array for partial pivoting in LU factorization,
- \( \text{last_flag} \) - last error return flag from internal function evaluations.
8.8 The SUNLinearSolver_LapackBand implementation

This section describes the SUNLINSOL implementation for solving banded linear systems with LAPACK. The SUNLINSOL_LAPACKBAND module 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).

To access the SUNLINSOL_LAPACKBAND module, include the header file sunlinsol/sunlinsol_lapackband.h. The installed module library to link to is lib sundials_sunlinsollapackband.lib where .lib is typically .so for shared libraries and .a for static libraries.

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 64-bit integers for the sunindextype.

8.8.1 SUNLinearSolver_LapackBand description

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 \( \mu \), then the upper triangular factor \( U \) can have upper bandwidth as big as \( \text{smu} = \min(N-1, \mu+\mu) \). The lower triangular factor \( L \) has lower bandwidth \( \mu \).

8.8.2 SUNLinearSolver_LapackBand functions

The SUNLINSOL_LAPACKBAND module provides the following user-callable constructor for creating a SUNLinearSolver object.

```
SUNLinSol_LapackBand
Call LS = SUNLinSol_LapackBand(y, A);
Description The function SUNLinSol_LapackBand creates and allocates memory for a LAPACK-based, band SUNLinearSolver object.
Arguments y (N_Vector) a template for cloning vectors needed within the solver
A (SUNMatrix) a SUNMATRIX_BAND 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_Band 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.

Additionally, this routine will verify that the input matrix A is allocated with appropriate upper bandwidth storage for the LU factorization.

Deprecated Name

For backward compatibility, the wrapper function SUNLapackBand with identical input and output arguments is also provided.

The SUNLINSOL_LAPACKBAND module defines band implementations of all “direct” linear solver operations listed in Sections 8.1.1 – 8.1.3:

- 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

8.8.3 SUNLinearSolver_LapackBand Fortran interfaces

For solvers that include a Fortran 77 interface module, the SUNLINSOL_LAPACKBAND module also includes a Fortran-callable function for creating a SUNLinearSolver object.

```fortran
FSUNLAPACKDENSEINIT
```

Call  

FSUNLAPACKBANDINIT(code, ier)

Description

The function FSUNLAPACKBANDINIT can be called for Fortran programs to create a LAPACK-based band 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_LAPACKBAND module includes a Fortran-callable function for creating a SUNLinearSolver mass matrix solver object.
8.9 The SUNLinearSolver_KLU implementation

**FSUNMASSLAPACKBANDINIT**

Call: `FSUNMASSLAPACKBANDINIT(ier)`

**Description**
The function `FSUNMASSLAPACKBANDINIT` can be called for Fortran programs to create a LAPACK-based, band `SUNLinearSolver` object for mass matrix linear systems.

**Arguments**
None

**Return value**
`ier` is an `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.

8.8.4 SUNLinearSolver_LapackBand content

The `sunlinsol_lapackband` module defines the `content` field of a `SUNLinearSolver` as the following structure:

```c
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.

8.9 The SUNLinearSolver_KLU implementation

This section describes the `sunlinsol` implementation for solving sparse linear systems with KLU. The `sunlinsol_klu` module 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`).

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 `sunlinsol_klu` module is a `sunlinsol` wrapper for the KLU sparse matrix factorization and solver library written by Tim Davis [1, 16]. 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.

8.9.1 SUNLinearSolver_KLU description

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.

### 8.9.2 SUNLinearSolver_KLU functions

The SUNLINSOL_KLU module provides the following user-callable constructor for creating a SUNLinearSolver object.

```c
SUNLinSol_KLU
```

**Call**

```c
LS = SUNLinSol_KLU(y, A);
```

**Description**

The function SUNLinSol_KLU creates and allocates memory for a KLU-based SUNLinearSolver 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

**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_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.

**Deprecation Name**

For backward compatibility, the wrapper function SUNKLU with identical input and output arguments is also provided.

**F2003 Name**

This function is callable as FSUNLinSol_KLU when using the Fortran 2003 interface module.

The SUNLINSOL_KLU module defines implementations of all “direct” linear solver operations listed in Sections 8.1.1 – 8.1.3:

- SUNLinSolGetType_KLU
8.9 The SUNLinearSolver_KLU implementation

- **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**

All of the listed operations are callable via the FORTRAN 2003 interface module by prepending an ‘F’ to the function name.

The SUNLIN_SOL_KLU module also defines the following additional user-callable functions.

```
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 nonzeroes has changed or if the structure of the linear system has changed which would require a new symbolic (and numeric factorization).
Arguments      LS         (SUNLinearSolver) a template for cloning vectors needed within the solver
                A          (SUNMatrix) a SUNMATRIX_SPARSE matrix template for cloning matrices needed within the solver
                nnz        (sunindextype) the new number of nonzero in the matrix
                reinit_type (int) flag governing the level of reinitialization. The allowed values are:
                              • SUNKLU_REINIT_FULL – The Jacobian matrix will be destroyed and a new one will be allocated based on the `nnz` value passed to this call. New symbolic and numeric factorizations will be completed at the next solver setup.
                              • SUNKLU_REINIT_PARTIAL – Only symbolic and numeric factorizations will be completed. It is assumed that the Jacobian size has not exceeded the size of `nnz` given in the sparse matrix provided to the original constructor routine (or the previous SUNLinSol_KLUReInit call).
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 (relocation of the sparse matrix failed) or SUNLS_SUCCESS.
Notes         This routine will perform consistency checks to ensure that it is called with consistent NVECTORS and SUNMATRIX implementations. These are currently limited to the SUNMATRIX_SPARSE matrix type (using either CSR or CSC storage formats) and the NVECTORS_SERIAL, NVECTORS_OPENMP, and NVECTORS_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.
```
Description of the SUNLinearSolver module

Deprecated Name  For backward compatibility, the wrapper function SUNKLUREInit with identical input and output arguments is also provided.

F2003 Name  This function is callable as FSUNLinSol_KLUREInit when using the Fortran 2003 interface module.

SUNLinSol_KLSetOrdering

Call  retval = SUNLinSol_KLSetOrdering(LS, ordering);

Description  This function sets the ordering used by KLU for reducing fill in the linear solve.

Arguments  

- **LS** (SUNLinearSolver) the SUNLINSOL_KLU object
- **ordering** (int) flag indicating the reordering algorithm to use, the options are:
  0  AMD,
  1  COLAMD, and
  2  the natural ordering.

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 choice), or SUNLS_SUCCESS.

Deprecated Name  For backward compatibility, the wrapper function SUNKLUSetOrdering with identical input and output arguments is also provided.

F2003 Name  This function is callable as FSUNLinSol_KLUSetOrdering when using the Fortran 2003 interface module.

8.9.3  SUNLinearSolver_KLU Fortran interfaces

The SUNLINSOL_KLU module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

FORTRAN 2003 interface module

The fsunlinsol_klu_mod FORTRAN module defines interfaces to all SUNLINSOL_KLU C functions using the intrinsic iso_c_binding module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading ‘F’. For example, the function SUNLinSol_klu is interfaced as FSUNLinSol_klu.

The FORTRAN 2003 SUNLINSOL_KLU interface module can be accessed with the use statement, i.e. use fsunlinsol_klu_mod, and linking to the library libsundials_fsunlinsolklu_mod.lib in addition to the C library. For details on where the library and module file fsunlinsol_klu_mod.mod are installed see Appendix A.

FORTRAN 77 interface functions

For solvers that include a FORTRAN 77 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 \texttt{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.

\textbf{FSUNMASSKLUINIT}

Call \texttt{FSUNMASSKLUINIT(ier)}

Description The function \texttt{FSUNMASSKLUINIT} can be called for Fortran programs to create a KLU-based SUNLinearSolver object for mass matrix linear systems.

Arguments None

Return value \texttt{ier} is a \texttt{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:

\textbf{FSUNKLUREINIT}

Call \texttt{FSUNKLUREINIT(code, nnz, reinit_type, ier)}

Description The function \texttt{FSUNKLUREINIT} can be called for Fortran programs to re-initialize a SUNLINSOL_KLU object.

Arguments \texttt{code} (\texttt{int*}) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).

\texttt{nnz} (\texttt{sunindextype*}) the new number of nonzeros in the matrix

\texttt{reinit_type} (\texttt{int*}) flag governing the level of reinitialization. The allowed values are:

1 – The Jacobian matrix will be destroyed and a new one will be allocated based on the \texttt{nnz} value passed to this call. New symbolic and numeric factorizations will be completed at the next solver setup.

2 – Only symbolic and numeric factorizations will be completed. It is assumed that the Jacobian size has not exceeded the size of \texttt{nnz} given in the sparse matrix provided to the original constructor routine (or the previous SUNLinSol_KLUREInit call).

Return value \texttt{ier} is a \texttt{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_KLUREInit for complete further documentation of this routine.

\textbf{FSUNMASSKLUREINIT}

Call \texttt{FSUNMASSKLUREINIT(nnz, reinit_type, ier)}

Description The function \texttt{FSUNMASSKLUREINIT} can be called for Fortran programs to re-initialize a SUNLINSOL_KLU object for mass matrix linear systems.

Arguments The arguments are identical to \texttt{FSUNKLUREINIT} above, except that \texttt{code} is not needed since mass matrix linear systems only arise in ARKODE.

Return value \texttt{ier} is a \texttt{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_KLUREInit for complete further documentation of this routine.
**Description of the SUNLinearSolver module**

---

### FSUNKLUSETORDERING

**Call**

FSUNKLUSETORDERING(code, ordering, ier)

**Description**
The function FSUNKLUSETORDERING can be called for Fortran programs to change the reordering algorithm used by KLU.

**Arguments**
- code (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).
- ordering (int*) flag indication the reordering algorithm to use. Options include:
  - 0 AMD,
  - 1 COLAMD, and
  - 2 the natural ordering.

The default is 1 for COLAMD.

**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.

### FSUNMASSKLUSETORDERING

**Call**

FSUNMASSKLUSETORDERING(ier)

**Description**
The function FSUNMASSKLUSETORDERING 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 FSUNKLUSETORDERING 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_KLUSetOrdering for complete further documentation of this routine.

---

### 8.9.4 SUNLinearSolver_KLU content

The SUNLINSOL_KLU module defines the content field of a SUNLinearSolver as the following structure:

```c
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).
8.10 The SUNLinearSolver_SuperLUDIST implementation

The SuperLU_DIST implementation of the sunlinsol module provided with SUNDIALS, sunlinsol_superludist, is designed to be used with the corresponding sunmatrix_superludist matrix type, and one of the serial, threaded or parallel nvector implementations (nvector_serial, nvector_openmp, nvector_pthreads, nvector_parallel, or nvector_parhyp).

The header file to include when using this module is sunlinsol/sunlinsol_superludist.h. The installed module library to link to is libsundials_sunlinsolsuperludist.lib where .lib is typically .so for shared libraries and .a for static libraries.

8.10.1 SUNLinearSolver_SuperLUDIST description

The sunlinsol_superludist module is a sunlinsol adapter for the SuperLU_DIST sparse matrix factorization and solver library written by X. Sherry Li [2, 22, 32, 33]. The package uses a SPMD parallel programming model and multithreading to enhance efficiency in distributed-memory parallel environments with multicore nodes and possibly GPU accelerators. It uses MPI for communication, OpenMP for threading, and CUDA for GPU support. In order to use the sunlinsol_superludist interface to SuperLU_DIST, it is assumed that SuperLU_DIST has been installed on the system prior to installation of SUNDIALS, and that SUNDIALS has been configured appropriately to link with SuperLU_DIST (see Appendix A for details). Additionally, the adapter only supports double-precision calculations, and therefore cannot be compiled if SUNDIALS is configured to use single or extended precision. Moreover, since the SuperLU_DIST library may be installed to support either 32-bit or 64-bit integers, it is assumed that the SuperLU_DIST library is installed using the same integer size as SUNDIALS.

The SuperLU_DIST library provides many options to control how a linear system will be solved. These options may be set by a user on an instance of the superlu_dist_options_t struct, and then it may be provided as an argument to the sunlinsol_superludist constructor. The sunlinsol_superludist module will respect all options set except for Fact – this option is necessarily modified by the sunlinsol_superludist module in the setup and solve routines.

Since the linear systems that arise within the context of SUNDIALS calculations will typically have identical sparsity patterns, the sunlinsol_superludist module is constructed to perform the following operations:

- The first time that the “setup” routine is called, it sets the SuperLU_DIST option Fact to DOFACT so that a subsequent call to the “solve” routine will perform a symbolic factorization, followed by an initial numerical factorization before continuing to solve the system.

- On subsequent calls to the “setup” routine, it sets the SuperLU_DIST option Fact to SamePattern so that a subsequent call to “solve” will perform factorization assuming the same sparsity pattern as prior, i.e. it will reuse the column permutation vector.

- If “setup” is called prior to the “solve” routine, then the “solve” routine will perform a symbolic factorization, followed by an initial numerical factorization before continuing to the sparse triangular solves, and, potentially, iterative refinement. If “setup” is not called prior, “solve” will skip to the triangular solve step. We note that in this solve SuperLU_DIST operates on the native data arrays for the right-hand side and solution vectors, without requiring costly data copies.

8.10.2 SUNLinearSolver_SuperLUDIST functions

The sunlinsol_superludist module defines implementations of all “direct” linear solver operations listed in Sections 8.1.1-8.1.3:

- SUNLinSolGetType_SuperLUDIST
- SUNLinSolInitialize_SuperLUDIST – this sets the first_factorize flag to 1 and resets the internal SuperLU_DIST statistics variables.
• **SUNLinSolSetup_SuperLUDIST** – this sets the appropriate SuperLU_DIST options so that a subsequent solve will perform a symbolic and numerical factorization before proceeding with the triangular solves.

• **SUNLinSolSolve_SuperLUDIST** – this calls the SuperLU_DIST solve routine to perform factorization (if the setup routine was called prior) and then use the $LU$ factors to solve the linear system.

• **SUNLinSolLastFlag_SuperLUDIST**

• **SUNLinSolSpace_SuperLUDIST** – 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 SuperLU_DIST documentation.

• **SUNLinSolFree_SuperLUDIST**

In addition, the module SUNLINSOL_SUPERLUDIST provides the following user-callable routines:

```c
SUNLinSol_SuperLUDIST
Call LS = SUNLinSol_SuperLUDIST(y, A, grid, lu, scaleperm, solve, stat, options);
Description The function SUNLinSol_SuperLUDIST creates and allocates memory for a SUNLINSOL_SUPERLUDIST object.
Arguments y (N_Vector) a template for cloning vectors needed within the solver
A (SUNMatrix) a SUNMATRIX_SUNRLOC matrix template for cloning matrices needed within the solver
grid (gridinfo_t*)
lu (LUstruct_t*)
scaleperm (ScalePermstruct_t*)
solve (SOLVEstruct_t*)
stat (SuperLUStat_t*)
options (superlu_dist_options_t*)
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 SuperLU_DIST 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_SUNRLOC matrix type and the NVECTOR_SERIAL, NVECTOR_PARALLEL, NVECTOR_PARHY, 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 grid, lu, scaleperm, solve, and options arguments are not checked and are passed directly to SuperLU_DIST routines.
Some struct members of the options argument are modified internally by the SUNLINSOL_SUPERLUDIST solver. Specifically the member Fact, is modified in the setup and solve routines.
```

```c
SUNLinSol_SuperLUDIST_GetBerr
Call realtype berr = SUNLinSol_SuperLUDIST_GetBerr(LS);
Description The function SUNLinSol_SuperLUDIST_GetBerr returns the componentwise relative backward error of the computed solution.
```
8.10 The SUNLinearSolver_SuperLUDIST implementation

Arguments  LS (SUNLinearSolver) the SUNLINSOL_SUPERLUDIST object
Return value  realtype

Notes

**SUNLinSol_SuperLUDIST_GetGridinfo**
Call  gridinfo_t *grid = SUNLinSol_SuperLUDIST_GetGridinfo(LS);
Description  The function SUNLinSol_SuperLUDIST_GetGridinfo returns the SuperLU_DIST structure that contains the 2D process grid.
Arguments  LS (SUNLinearSolver) the SUNLINSOL_SUPERLUDIST object
Return value  gridinfo_t*

Notes

**SUNLinSol_SuperLUDIST_GetLUstruct**
Call  LUstruct_t *lu = SUNLinSol_SuperLUDIST_GetLUstruct(LS);
Description  The function SUNLinSol_SuperLUDIST_GetLUstruct returns the SuperLU_DIST structure that contains the distributed L and U factors.
Arguments  LS (SUNLinearSolver) the SUNLINSOL_SUPERLUDIST object
Return value  LUstruct_t*

Notes

**SUNLinSol_SuperLUDIST_GetSuperLUOptions**
Call  superlu_dist_options_t *opts = SUNLinSol_SuperLUDIST_GetSuperLUOptions(LS);
Description  The function SUNLinSol_SuperLUDIST_GetSuperLUOptions returns the SuperLU_DIST structure that contains the options which control how the linear system is factorized and solved.
Arguments  LS (SUNLinearSolver) the SUNLINSOL_SUPERLUDIST object
Return value  superlu_dist_options_t*

Notes

**SUNLinSol_SuperLUDIST_GetScalePermstruct**
Call  ScalePermstruct_t *sp = SUNLinSol_SuperLUDIST_GetScalePermstruct(LS);
Description  The function SUNLinSol_SuperLUDIST_GetScalePermstruct returns the SuperLU_DIST structure that contains the vectors that describe the transformations done to the matrix, A.
Arguments  LS (SUNLinearSolver) the SUNLINSOL_SUPERLUDIST object
Return value  ScalePermstruct_t*

Notes
Description of the SUNLinearSolver module

```c
SUNLinSol_SuperLUDIST_GetSOLVEstruct
```

Call

```
SOLVEstruct_t *solve = SUNLinSol_SuperLUDIST_GetSOLVEstruct(LS);
```

Description

The function `SUNLinSol_SuperLUDIST_GetSOLVEstruct` returns the SuperLU_DIST structure that contains information for communication during the solution phase.

Arguments

`LS` (SUNLinearSolver) the SUNLINSOL_SUPERLUDIST object

Return value

`SOLVEstruct_t`

Notes

```c
SUNLinSol_SuperLUDIST_GetSuperLUStat
```

Call

```
SuperLUStat_t *stat = SUNLinSol_SuperLUDIST_GetSuperLUStat(LS);
```

Description

The function `SUNLinSol_SuperLUDIST_GetSuperLUStat` returns the SuperLU_DIST structure that stores information about runtime and flop count.

Arguments

`LS` (SUNLinearSolver) the SUNLINSOL_SUPERLUDIST object

Return value

`SuperLUStat_t`

Notes

### 8.10.3 SUNLinearSolver_SuperLUDIST content

The SUNLINSOL_SUPERLUDIST module defines the `content` field of a SUNLinearSolver to be the following structure:

```c
struct _SUNLinearSolverContent_SuperLUDIST {
    bool Haye first_factorize;
    long int last_flag;
    realtype berr;
    gridinfo_t *grid;
    LUstruct_t *lu;
    superlu_dist_options_t *options;
    ScalePermstruct_t *scaleperm;
    SOLVEstruct_t *solve;
    SuperLUStat_t *stat;
    sunindextype N;
};
```

These entries of the `content` field contain the following information:

- **first_factorize** - flag indicating whether the factorization has ever been performed,
- **last_flag** - last error return flag from calls to internal routines,
- **berr** - the componentwise relative backward error of the computed solution,
- **grid** - pointer to the SuperLU_DIST structure that stores the 2D process grid,
- **lu** - pointer to the SuperLU_DIST structure that stores the distributed L and U factors,
- **options** - pointer to SuperLU_DIST options structure,
- **scaleperm** - pointer to the SuperLU_DIST structure that stores vectors describing the transformations done to the matrix, A,
- **solve** - pointer to the SuperLU_DIST solve structure,
- **stat** - pointer to the SuperLU_DIST structure that stores information about runtime and flop count,
- **N** - the number of equations in the system
8.11 The SUNLinearSolver_SuperLUMT implementation

This section describes the SUNLINSOL implementation for solving sparse linear systems with SuperLU_MT. The SuperLUMT module is designed to be used with the corresponding SUNMATRIX_SPARSE matrix type, and one of the serial or shared-memory NVVECTOR implementations (NVVECTOR_SERIAL, NVVECTOR_OPENMP, or NVVECTOR_PTHREADS). While these are compatible, it is not recommended to use a threaded vector module with SUNLINSOL_SUPERLUMT unless it is the NVVECTOR_OPENMP module and the SuperLUMT library has also been compiled with OpenMP.

The header file to include when using this module is sunlinsol/sunlinsol_superlumt.h. The installed module library to link to is libsundials_sunlinsolsuperlumt.lib where .lib is typically .so for shared libraries and .a for static libraries.

The SUNLINSOL_SUPERLUMT module is a SUNLINSOL wrapper for the SUPERLUMT sparse matrix factorization and solver library written by X. Sherry Li [3, 31, 18]. 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 realltype 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.

8.11.1 SUNLinearSolver_SuperLUMT description

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 $A^T \ast A$, 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.

8.11.2 SUNLinearSolver_SuperLUMT functions

The module SUNLINSOL_SUPERLUMT provides the following user-callable constructor for creating a SUNLinearSolver object.

```
SUNLinSol_SuperLUMT
Call
LS = SUNLinSol_SuperLUMT(y, A, num_threads);
Description
The function SUNLinSol_SuperLUMT creates and allocates memory for a SuperLU_MT-based SUNLinearSolver 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.

Deprecated Name For backward compatibility, the wrapper function SUNSuperLUMT with identical input and output arguments is also provided.

The SUNLINSOL\_SUPERLUMT module defines implementations of all “direct” linear solver operations listed in Sections 8.1.1 – 8.1.3:

- 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

The SUNLINSOL\_SUPERLUMT module also defines the following additional user-callable function.

```
SUNLinSol\_SuperLUMTSetOrdering
```

Call `retval = SUNLinSol\_SuperLUMTSetOrdering(LS, ordering);`

Description This function sets the ordering used by SUPERLUMT for reducing fill in the linear solve.

Arguments

| LS | (SUNLinearSolver) the SUNLINSOL\_SUPERLUMT object |
| ordering | (int) a flag indicating the ordering algorithm to use, the 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 SUNLS_MEM_NULL (S is NULL), SUNLS_ILL_INPUT (invalid ordering choice), or SUNLS_SUCCESS.

Deprecated Name For backward compatibility, the wrapper function SUNSuperLUMTSetOrdering with identical input and output arguments is also provided.

8.11.3 SUNLinearSolver_SuperLUMT Fortran interfaces

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(code, num_threads, ier)

Description The function FSUNSUPERLUMTINIT 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).
num_threads (int*) desired number of threads (OpenMP or Pthreads, depending on how SUPERLUMT was installed) to use during the factorization steps.

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_SUPERLUMT module includes a Fortran-callable function for creating a SUNLinearSolver mass matrix solver object.

**FSUNMASSSUPERLUMTINIT**

Call 
FSUNMASSSUPERLUMTINIT(num_threads, ier)

Description The function FSUNMASSSUPERLUMTINIT can be called for Fortran programs to create a SuperLU_MT-based SUNLinearSolver object for mass matrix linear systems.

Arguments num_threads (int*) desired number of threads (OpenMP or Pthreads, depending on how SUPERLUMT was installed) to use during the factorization steps.

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_SuperLUMTSetOrdering routine also supports Fortran interfaces for the system and mass matrix solvers:

**FSUNSUPERLUMTSETORDERING**

Call 
FSUNSUPERLUMTSETORDERING(code, ordering, ier)

Description The function FSUNSUPERLUMTSETORDERING can be called for Fortran programs to update the ordering algorithm in a SUNLINSOL_SUPERLUMT 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).
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 $\text{ier}$ is a $\text{int}$ 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.

\textbf{FSUNMASSUPERLUMTSETORDERING}

Call \texttt{FSUNMASSUPERLUMTSETORDERING(ordering, ier)}

Description The function \texttt{FSUNMASSUPERLUMTSETORDERING} can be called for Fortran programs to update the ordering algorithm in a SUNLINSOL\_SUPERLUMT object for mass matrix linear systems.

Arguments $\text{ordering}$ (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 $\text{ier}$ is a $\text{int}$ 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.

\section*{8.11.4 SUNLinearSolver\_SuperLUMT content}

The SUNLINSOL\_SUPERLUMT module defines the \textit{content} field of a SUNLinearSolver as the following structure:

\begin{verbatim}
struct _SUNLinearSolverContent_SuperLUMT {
    long int last_flag;
    int first_factorize;
    Gstat_t *Gstat;
    sunindextype *perm_r, *perm_c;
    sunindextype N;
    int num_threads;
    realtype diag_pivot_thresh;
    int ordering;
    superlumt_options_t *options;
};
\end{verbatim}

These entries of the \textit{content} field contain the following information:

$\text{last_flag}$ - last error return flag from internal function evaluations,
$\text{first_factorize}$ - 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.

8.12 The SUNLinearSolver_SPGMR implementation

This section describes the SUNLINSOL implementation of the SPGMR (Scaled, Preconditioned, Generalized Minimum Residual [37]) iterative linear solver. The SUNLINSOL_SPGMR module is designed to be compatible with any NVVECTOR implementation 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.

To access the SUNLINSOL_SPGMR module, include the header file sunlinsol/sunlinsol_spgmr.h. We note that the SUNLINSOL_SPGMR module is accessible from SUNDIALS packages without separately linking to the libsundials_sunlinsolspgmr module library.

8.12.1 SUNLinearSolver_SPGMR description

This solver is constructed to perform the following operations:

- During construction, the xcor and vtemp arrays are cloned from a template NVVECTOR 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.

8.12.2 SUNLinearSolver_SPGMR functions

The SUNLINSOL_SPGMR module provides the following user-callable constructor for creating a SUNLinearSolver object.

\[\text{SUNLinSol_SPGMR} \]

Call: \( \text{LS = SUNLinSol_SPGMR(y, pretype, maxl)}; \)
Description: The function SUNLinSol_SPGMR creates and allocates memory for a SPGMR SUNLinearSolver object.
Arguments:
- \( y \) (N_Vector) a template for cloning vectors needed within the solver
- \( \text{pretype} \) (int) flag indicating the desired type of preconditioning, allowed values are:
  - PREC_NONE (0)
  - PREC_LEFT (1)
  - PREC_RIGHT (2)
Description of the SUNLinearSolver module

- PREC_BOTH (3)

Any other integer input will result in the default (no preconditioning).

\[
\text{maxl} \quad \text{(int)} \text{ the number of Krylov basis vectors to use. Values} \leq 0 \text{ will result in the default value (5).}
\]

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_SPGMR object to use any of the preconditioning options with these solvers, this use mode is not supported and may result in inferior performance.

Deprecated Name: For backward compatibility, the wrapper function SUNSPGMR with identical input and output arguments is also provided.

F2003 Name: This function is callable as FSUNLinSol_SPGMR when using the Fortran 2003 interface module.

The SUNLINSOL_SPGMR module defines implementations of all “iterative” linear solver operations listed in Sections 8.1.1 – 8.1.3:

- 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

All of the listed operations are callable via the FORTRAN 2003 interface module by prepending an ‘F’ to the function name.

The SUNLINSOL_SPGMR module also defines the following additional user-callable functions.
Call \[ \text{retval} = \text{SUNLinSol::SPGMRSetPrecType}(\text{LS}, \text{pretype}); \]

Description The function `SUNLinSol::SPGMRSetPrecType` updates the type of preconditioning to use in the `SUNLINSOL::SPGMR` object.

Arguments
- `LS` (`SUNLinearSolver`) the `SUNLINSOL::SPGMR` object to update
- `pretype` (`int`) flag indicating the desired type of preconditioning, allowed values match those discussed in `SUNLinSol::SPGMR`.

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`.

Deprecated Name For backward compatibility, the wrapper function `SUNSPGMRSetPrecType` with identical input and output arguments is also provided.

F2003 Name This function is callable as `FSUNLinSol::SPGMRSetPrecType` when using the Fortran 2003 interface module.

Call \[ \text{retval} = \text{SUNLinSol::SPGMRSetGSType}(\text{LS}, \text{gstype}); \]

Description The function `SUNLinSol::SPGMRSetGSType` sets the type of Gram-Schmidt orthogonalization to use in the `SUNLINSOL::SPGMR` object.

Arguments
- `LS` (`SUNLinearSolver`) the `SUNLINSOL::SPGMR` object to update
- `gstype` (`int`) flag indicating the desired orthogonalization algorithm; allowed values are:
  - `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 `gstype`), `SUNLS::MEM_NULL` (`S` is NULL) or `SUNLS::SUCCESS`.

Deprecated Name For backward compatibility, the wrapper function `SUNSPGMRSetGSType` with identical input and output arguments is also provided.

F2003 Name This function is callable as `FSUNLinSol::SPGMRSetGSType` when using the Fortran 2003 interface module.

Call \[ \text{retval} = \text{SUNLinSol::SPGMRSetMaxRestarts}(\text{LS}, \text{maxrs}); \]

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`.

Deprecated Name For backward compatibility, the wrapper function `SUNSPGMRSetMaxRestarts` with identical input and output arguments is also provided.

F2003 Name This function is callable as `FSUNLinSol::SPGMRSetMaxRestarts` when using the Fortran 2003 interface module.
8.12.3 SUNLinearSolver_SPGMR Fortran interfaces

The SUNLINSOL_SPGMR module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

FORTRAN 2003 interface module

The _fsunlinsol_spgrm_mod_ FORTRAN module defines interfaces to all SUNLINSOL_SPGMR C functions using the intrinsic _iso_c_binding_ module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading ‘F’. For example, the function _SUNLinSol_SPGMR_ is interfaced as _FSUNLinSol_SPGMR_.

The FORTRAN 2003 SUNLINSOL_SPGMR interface module can be accessed with the _use_ statement, i.e. _use fsunlinsol_spgrm_mod_, and linking to the library _libsundials_fsunlinsolspgmr_mod.lib_ in addition to the C library. For details on where the library and module file _fsunlinsol_spgrm_mod.mod_ are installed see Appendix A. We note that the module is accessible from the FORTRAN 2003 SUNDIALS integrators _without_ separately linking to the _libsundials_fsunlinsolspgmr_mod_ library.

FORTRAN 77 interface functions

For solvers that include a FORTRAN 77 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 an _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 orthogonaliation 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  
ierr is an 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 orthogonaliation 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  
ierr is an 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  
ierr is an 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(pretype, ier)

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 code is not needed since mass matrix linear systems only arise in ARKODE.

Return value  
ierr is an 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.
226 Description of the SUNLinearSolver module

FSUNSPGMRSETMAXRS

Call

FSUNSPGMRSETMAXRS(code, maxrs, ier)

Description

The function FSUNSPGMRSETMAXRS can be called for Fortran programs to change the maximum number of restarts allowed for SPGMR.

Arguments

code (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).

maxrs (int*) maximum allowed number of restarts.

Return value

ier is an 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_SPGMRSetMaxRestarts for complete further documentation of this routine.

FSUNMASSSPGMRSETMAXRS

Call

FSUNMASSSPGMRSETMAXRS(maxrs, ier)

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 code is not needed since mass matrix linear systems only arise in ARKODE.

Return value

ier is an 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_SPGMRSetMaxRestarts for complete further documentation of this routine.

8.12.4 SUNLinearSolver_SPGMR content

The SUNLINSOL_SPGMR module defines the content field of a SUNLinearSolver as the following structure:

```c
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),
- **num restarts**: 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 $v_1,...,v_{maxl+1}$, stored in $V[0],...,V[maxl]$. Each $v_i$ is a vector of type nvector.,
- **Hes**: the $(maxl+1) \times maxl$ Hessenberg matrix. It is stored row-wise so that the $(i,j)$th element is given by $Hes[i][j]$.,
- **givens**: a length $2*maxl$ array which represents the Givens rotation matrices that arise in the GMRES algorithm. These matrices are $F_0, F_1,..., F_j$, where

$$
F_i = \begin{bmatrix}
1 & & & & \\
& \ddots & & & \\
& & 1 & -s_i & c_i \\
& & s_i & c_i & \\
& & & \ddots & \\
& & & & 1
\end{bmatrix},
$$

are represented in the givens vector as $givens[0] = c_0, givens[1] = s_0, givens[2] = c_1, givens[3] = s_1, \ldots givens[2j] = c_j, givens[2j+1] = s_j$.,
- **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.

### 8.13 The SUNLinearSolver_SPFGMR implementation

This section describes the SUNLINSOL implementation of the SPFGMR (Scaled, Preconditioned, Flexible, Generalized Minimum Residual [36]) iterative linear solver. The SUNLINSOL_SPFGMR module is designed to be compatible with any NVECTOR implementation that supports a minimal subset of operations ($NVClone, NVDotProd, NVScale, NVLinearSum, NVProd, NConst, NDiv, and NDestroy$). When using Classical Gram-Schmidt, the optional function $NVDotProdMulti$ may be supplied for increased efficiency. Unlike the other Krylov iterative linear solvers supplied with SUNDIALS, SPFGMR is specifically designed to work with a changing preconditioner (e.g. from an iterative method).

To access the SUNLINSOL_SPFGMR module, include the header file sunlinsol/sunlinsol_spfgmr.h. We note that the SUNLINSOL_SPFGMR module is accessible from SUNDIALS packages without separately linking to the libsundials_sunlinsolspfgmr module library.
8.13.1 SUNLinearSolver_SPFGMR description

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.

8.13.2 SUNLinearSolver_SPFGMR functions

The SUNLINSOL_SPFGMR module provides the following user-callable constructor for creating a SUNLinearSolver object.

```c
SUNLinSol_SPFGMR
Call LS = SUNLinSol_SPFGMR(y, pretype, maxl);
Description The function SUNLinSol_SPFGMR creates and allocates memory for a SPFGMR SUNLinearSolver object.
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 Krylov basis vectors to use. Values ≤ 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_SPFGMR object to use any of the preconditioning options with these solvers, this use mode is not supported and may result in inferior performance.
F2003 Name This function is callable as FSUNLinSol_SPFGMR when using the Fortran 2003 interface module.
SUNSPFGMR The SUNLINSOL_SPFGMR module defines implementations of all “iterative” linear solver operations listed in Sections 8.1.1 – 8.1.3.
8.13 The SUNLinearSolver_SPFGMR implementation

- 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

All of the listed operations are callable via the FORTRAN 2003 interface module by prepending an ‘F’ to the function name.

The SUNLINSOL_SPFGMR module also defines the following additional user-callable functions.

### SUNLinSol_SPFGMRSetPrecType

**Call**

`retval = SUNLinSol_SPFGMRSetPrecType(LS, pretype);`

**Description**
The function `SUNLinSol_SPFGMRSetPrecType` updates the type of preconditioning to use in the SUNLINSOL_SPFGMR object.

**Arguments**
- `LS` (SUNLinearSolver) the SUNLINSOL_SPFGMR object to update
- `pretype` (int) flag indicating the desired type of preconditioning, allowed values match those discussed in `SUNLinSol_SPFGMR`.

**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`.

**Deprecated Name**
For backward compatibility, the wrapper function `SUNSPFGMRSetPrecType` with identical input and output arguments is also provided.

**F2003 Name**
This function is callable as `FSUNLinSol_SPFGMRSetPrecType` when using the FORTRAN 2003 interface module.

### SUNLinSol_SPFGMRSetGSType

**Call**

`retval = SUNLinSol_SPFGMRSetGSType(LS, gstype);`

**Description**
The function `SUNLinSol_SPFGMRSetsGSType` sets the type of Gram-Schmidt orthogonalization to use in the SUNLINSOL_SPFGMR object.

**Arguments**
- `LS` (SUNLinearSolver) the SUNLINSOL_SPFGMR object to update
- `gstype` (int) flag indicating the desired orthogonalization algorithm; allowed values are:
  - `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.

Deprecated Name: For backward compatibility, the wrapper function SUNSPFGMRSetGSType with identical input and output arguments is also provided.

F2003 Name: This function is callable as FSUNLinSol_SPFGMRSetGSType when using the Fortran 2003 interface module.

---

8.13.3 SUNLinearSolver_SPFGMR Fortran interfaces

The SUNLINSOL_SPFGMR module provides a FORTAN 2003 module as well as FORTAN 77 style interface functions for use from FORTRAN applications.

FORTAN 2003 interface module

The fsunlinsol_spfmgmr_mod FORTAN module defines interfaces to all SUNLINSOL_SPFGMR C functions using the intrinsic iso_c_binding module which provides a standardized mechanism for interop-erating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading ‘F’. For example, the function SUNLinSol_SPFGMR is interfaced as FSUNLinSol_SPFGMR.

The FORTRAN 2003 SUNLINSOL_SPFGMR interface module can be accessed with the use statement, i.e. use fsunlinsol_spfmgmr_mod, and linking to the library libsundials_sunlinsolspfmgmr_mod.lib in addition to the C library. For details on where the library and module file fsunlinsol_spfmgmr_mod.mod are installed see Appendix A. We note that the module is accessible from the FORTRAN 2003 SUNDIALS integrators without separately linking to the libsundials_sunlinsolspfmgmr_mod library.

FORTAN 77 interface functions

For solvers that include a FORTAN 77 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 an 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 an 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 an 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 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.SPFGMRSetPrecType for complete further documentation of this routine.

**FSUNMASSSPFGMRSETPRECTYPE**

Call

FSUNMASSSPFGMRSETPRECTYPE(pretype, ier)

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 `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.SPFGMRSetPrecType for complete further documentation of this routine.

**FSUNSPFGMRSETMAXRS**

Call

FSUNSPFGMRSETMAXRS(code, maxrs, ier)

Description
The function `FSUNSPFGMRSETMAXRS` can be called for Fortran programs to change the maximum number of restarts allowed for SPFGMR.

Arguments
- `code` (int*) is an integer input specifying the solver id (1 for cvode, 2 for ida, 3 for kinsol, and 4 for arkode).
- `maxrs` (int*) maximum allowed number of restarts.

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.SPFGMRSetMaxRestarts for complete further documentation of this routine.

**FSUNMASSSPFGMRSETMAXRS**

Call

FSUNMASSSPFGMRSETMAXRS(maxrs, ier)

Description
The function `FSUNMASSSPFGMRSETMAXRS` can be called for Fortran programs to change the maximum number of restarts allowed for mass matrix linear systems.

Arguments
- The arguments are identical to `FSUNSPFGMRSETMAXRS` 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.SPFGMRSetMaxRestarts for complete further documentation of this routine.
8.13.4 SUNLinearSolver_SPFGMR content

The sunlinsol_spfgmr module defines the content field of a SUNLinearSolver as the following structure:

```c
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 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 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 $v_1, \ldots, v_{maxl+1}$, stored in $V[0], \ldots, V[maxl]$. Each $v_i$ is a vector of type NVECTOR,
- `Z` - the array of preconditioned Krylov basis vectors $z_1, \ldots, z_{maxl+1}$, stored in $Z[0], \ldots, Z[maxl]$. Each $z_i$ is a vector of type NVECTOR,
- `Hes` - the $(maxl + 1) \times maxl$ Hessenberg matrix. It is stored row-wise so that the $(i,j)$th element is given by $Hes[i][j]$,
234 Description of the SUNLinearSolver module

givens - a length $2\times \text{maxl}$ array which represents the Givens rotation matrices that arise in the FGMRES algorithm. These matrices are $F_0, F_1, \ldots, F_j$, where

$$
F_i = \begin{bmatrix}
1 & \cdots & 1 \\
\vdots & \ddots & \vdots \\
-\frac{c_i}{s_i} & \cdots & \frac{1}{s_i}
\end{bmatrix},
$$

are represented in the givens vector as $\text{givens}[0] = c_0, \text{givens}[1] = s_0, \text{givens}[2] = c_1, \text{givens}[3] = s_1, \ldots \text{givens}[2j] = c_j, \text{givens}[2j+1] = s_j$.

xcor - a vector which holds the scaled, preconditioned correction to the initial guess,
yg - a length $(\text{maxl}+1)$ array of \texttt{realtype} values used to hold “short” vectors (e.g. $y$ and $g$),
vtemp - temporary vector storage.

8.14 The SUNLinearSolver_SPBCGS implementation

This section describes the SUNLINSOL implementation of the SPBCGS (Scaled, Preconditioned, Bi-Conjugate Gradient, Stabilized [38]) iterative linear solver. The SUNLINSOL_SPBCGS module is designed to be compatible with any nvector implementation that supports a minimal subset of operations ($\text{N_VClone}$, $\text{N_VDotProd}$, $\text{N_VScale}$, $\text{N_VLinearSum}$, $\text{N_VProd}$, $\text{N_VDiv}$, and $\text{N_VDestroy}$). Unlike the SPGMR and SPFGMR algorithms, SPBCGS requires a fixed amount of memory that does not increase with the number of allowed iterations.

To access the SUNLINSOL_SPBCGS module, include the header file \texttt{sunlinsol/sunlinsol_spbcgs.h}. We note that the SUNLINSOL_SPBCGS module is accessible from SUNDIALS packages \textit{without} separately linking to the \texttt{libsundials_sunlinsolspbcgs} module library.

8.14.1 SUNLinearSolver_SPBCGS description

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 \texttt{ATimes}, \texttt{PSetup}, and \texttt{Psolve} function pointers and $s_1$ and $s_2$ scaling vectors.
- In the “initialize” call, the solver parameters are checked for validity.
- In the “setup” call, any non-\texttt{NULL} \texttt{PSetup} function is called. Typically, this is provided by the SUNDIALS solver itself, that translates between the generic \texttt{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.

8.14.2 SUNLinearSolver_SPBCGS functions

The SUNLINSOL_SPBCGS module provides the following user-callable constructor for creating a SUNLinearSolver object.
SUNLinSol_SPBCGS

Call
LS = SUNLinSol_SPBCGS(y, pretype, maxl);

Description
The function SUNLinSol_SPBCGS creates and allocates memory for a SPBCGS SUNLinearSolver object.

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 ≤ 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.

Deprecated Name
For backward compatibility, the wrapper function SUNSPBCGS with identical input and output arguments is also provided.

F2003 Name
This function is callable as FSUNLinSol_SPBCGS when using the Fortran 2003 interface module.

The SUNDIALS_SPBCGS module defines implementations of all “iterative” linear solver operations listed in Sections 8.1.1 – 8.1.3:

- SUNLinSolGetType_SPBCGS
- SUNLinSolInitialize_SPBCGS
- SUNLinSolSetATimes_SPBCGS
- SUNLinSolSetPreconditioner_SPBCGS
- SUNLinSolSetScalingVectors_SPBCGS
- SUNLinSolSetup_SPBCGS
- SUNLinSolSolve_SPBCGS
- SUNLinSolNumIters_SPBCGS
- SUNLinSolResNorm_SPBCGS
- SUNLinSolResid_SPBCGS
- SUNLinSolLastFlag_SPBCGS
- SUNLinSolSpace_SPBCGS
236 Description of the SUNLinearSolver module

- **SUNLinSolFree_SPBCGS**

All of the listed operations are callable via the FORTRAN 2003 interface module by prepending an ‘F’ to the function name.

The SUNLINSOL_SPBCGS module also defines the following additional user-callable functions.

```fortran
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.
Deprecated Name For backward compatibility, the wrapper function SUNSPBCGSSetPrecType with identical input and output arguments is also provided.
F2003 Name This function is callable as FSUNLinSol_SPBCGSSetPrecType when using the Fortran 2003 interface module.
```

```fortran
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 ≤ 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.
Deprecated Name For backward compatibility, the wrapper function SUNSPBCGSSetMaxl with identical input and output arguments is also provided.
F2003 Name This function is callable as FSUNLinSol_SPBCGSSetMaxl when using the Fortran 2003 interface module.
```

8.14.3 SUNLinearSolver_SPBCGS Fortran interfaces

The SUNLINSOL_SPBCGS module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

**FORTRAN 2003 interface module**

The fsunlinsol_spbcgs_mod FORTRAN module defines interfaces to all SUNLINSOL_SPBCGS C functions using the intrinsic iso_c_binding module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading ‘F’. For example, the function SUNLinSol_SPBCGS is interfaced as FSUNLinSol_SPBCGS.

The FORTRAN 2003 SUNLINSOL_SPBCGS interface module can be accessed with the use statement, i.e. use fsunlinsol_spbcgs_mod, and linking to the library libsundials_fsunlinsolspbcgs_mod.lib in addition to the C library. For details on where the library and module file fsunlinsol_spbcgs_mod.mod are installed see Appendix A. We note that the module is accessible
from the FORTRAN 2003 SUNDIALS integrators without separately linking to the 
libsundials_fsunlinsolspbcgs_mod library.

FORTRAN 77 interface functions

For solvers that include a FORTRAN 77 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 an 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(code, pretype, ier)
```

Description 
The function FSUNSPBCGSSETPRECTYPE 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_SPBCGSSetPrecType for complete further documentation of this routine.
### Description of the SUNLinearSolver module

**FSUNMASSSPBCGSSETPRECTYPE**

**Call**

FSUNMASSSPBCGSSETPRECTYPE(pretype, ier)

**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 code is not needed since mass matrix linear systems only arise in ARKODE.

**Return value**
ien is an 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_SPBCGSSetPrecType for complete further documentation of this routine.

**FSUNSPBCGSSETMAXL**

**Call**

FSUNSPBCGSSETMAXL(code, maxl, ier)

**Description**
The function FSUNSPBCGSSETMAXL 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**
ien is an 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_SPBCGSSetMaxl for complete further documentation of this routine.

**FSUNMASSSPBCGSSETMAXL**

**Call**

FSUNMASSSPBCGSSETMAXL(maxl, ier)

**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 code is not needed since mass matrix linear systems only arise in ARKODE.

**Return value**
ien is an 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_SPBCGSSetMaxl for complete further documentation of this routine.

### 8.14.4 SUNLinearSolver_SPBCGS content

The SUNLINSOL_SPBCGS module defines the content field of a SUNLinearSolver as the following structure:

```c
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),
umiters - 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 NVVECTOR which holds the current scaled, preconditioned linear system residual,
r_star - a NVVECTOR which holds the initial scaled, preconditioned linear system residual,
p, q, u, Ap, vtemp - NVECTORS used for workspace by the SPBCGS algorithm.

8.15 The SUNLinearSolver_SPTFQMR implementation

This section describes the SUNLINSOL implementation of the SPTFQMR (Scaled, Preconditioned, Transpose-Free Quasi-Minimum Residual [21]) iterative linear solver. The SUNLINSOL_SPTFQMR module is designed to be compatible with any NVVECTOR implementation 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.

To access the SUNLINSOL_SPTFQMR module, include the header file sunlinsol/sunlinsol_sptfqmr.h. We note that the SUNLINSOL_SPTFQMR module is accessible from SUNDIALS packages without separately linking to the libsundials_sunlinsolsptfqmr module library.

8.15.1 SUNLinearSolver_SPTFQMR description

This solver is constructed to perform the following operations:

- During construction all NVVECTOR solver data is allocated, with vectors cloned from a template NVVECTOR 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.

8.15.2 SUNLinearSolver_SPTFQMR functions

The SUNLINSOL_SPTFQMR module provides the following user-callable constructor for creating a SUNLinearSolver object.

[SUNLinSol_SPTFQMR]
Call LS = SUNLinSol_SPTFQMR(y, pretype, maxl);
Description The function SUNLinSol_SPTFQMR creates and allocates memory for a SPTFQMR SUNLinearSolver object.
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 ≤ 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_SPTFQMR object to use any of the preconditioning options with these solvers, this use mode is not supported and may result in inferior performance.
Deprecated Name For backward compatibility, the wrapper function SUNSPTFQMR with identical input and output arguments is also provided.
F2003 Name This function is callable as FSUNLinSol_SPTFQMR when using the Fortran 2003 interface module.

The SUNLINSOL_SPTFQMR module defines implementations of all “iterative” linear solver operations listed in Sections 8.1.1 – 8.1.3:

• SUNLinSolGetType_SPTFQMR
• SUNLinSolInitialize_SPTFQMR
• SUNLinSolSetATimes_SPTFQMR
• SUNLinSolSetPreconditioner_SPTFQMR
• SUNLinSolSetScalingVectors_SPTFQMR
8.15 The SUNLinearSolver_SPTFQMR implementation

- SUNLinSolSetup_SPTFQMR
- SUNLinSolSolve_SPTFQMR
- SUNLinSolNumIters_SPTFQMR
- SUNLinSolResNorm_SPTFQMR
- SUNLinSolResid_SPTFQMR
- SUNLinSolLastFlag_SPTFQMR
- SUNLinSolSpace_SPTFQMR
- SUNLinSolFree_SPTFQMR

All of the listed operations are callable via the FORTRAN 2003 interface module by prepending an ‘F’
to the function name.

The SUNLINSOL_SPTFQMR module also defines the following additional user-callable functions.

**SUNLinSol_SPTFQMRSetPrecType**

Call

```fortran
retval = SUNLinSol_SPTFQMRSetPrecType(LS, pretype);
```

Description

The function SUNLinSol_SPTFQMRSetPrecType updates the type of preconditioning

to use in the SUNLINSOL_SPTFQMR object.

Arguments

- LS (SUNLinearSolver) the SUNLINSOL_SPTFQMR object to update
- pretype (int) flag indicating the desired type of preconditioning, allowed values
  match those discussed in SUNLinSol_SPTFQMR.

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.

Deprecated Name For backward compatibility, the wrapper function SUNSPTFQMRSetPrecType with
identical input and output arguments is also provided.

F2003 Name This function is callable as FSUNLinSol_SPTFQMRSetPrecType when using the Fortran
2003 interface module.

**SUNLinSol_SPTFQMRSetMaxl**

Call

```fortran
retval = SUNLinSol_SPTFQMRSetMaxl(LS, maxl);
```

Description

The function SUNLinSol_SPTFQMRSetMaxl updates the number of linear solver iterations

to allow.

Arguments

- LS (SUNLinearSolver) the SUNLINSOL_SPTFQMR object to update
- maxl (int) flag indicating the number of iterations to allow; values ≤ 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.

F2003 Name This function is callable as FSUNLinSol_SPTFQMRSetMaxl when using the Fortran 2003
interface module.

SUNSPTFQMRSetMaxl

8.15.3 SUNLinearSolver_SPTFQMR Fortran interfaces

The SUNLINSOL_SPTFGMR module provides a FORTRAN 2003 module as well as FORTRAN 77 style
interface functions for use from FORTRAN applications.
FORTRAN 2003 interface module

The fsunlinsol_sptfqr_mod FORTRAN module defines interfaces to all SUNLINSOL_SPTFQR C functions using the intrinsic iso_c_binding module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading ‘F’. For example, the function SUNLinSol_SPTFQMR is interfaced as FSUNLinSol_SPTFQMR.

The FORTRAN 2003 SUNLINSOL_SPTFQR interface module can be accessed with the use statement, i.e. use fsunlinsol_sptfqr_mod, and linking to the library libsunlinsolsptfqr_mod.lib in addition to the C library. For details on where the library and module file fsunlinsol_sptfqr_mod.mod are installed see Appendix A. We note that the module is accessible from the FORTRAN 2003 SUNDIALS integrators without separately linking to the libsunlinsolsptfqr_mod library.

FORTRAN 77 interface functions

For solvers that include a FORTRAN 77 interface module, the SUNLINSOL_SPTFQR module also includes a Fortran-callable function for creating a SUNLinearSolver object.

FSUNSPSTFQMRINIT

Call FSUNSPSTFQMRINIT(code, pretype, maxl, ier)

Description The function FSUNSPSTFQMRINIT 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 forIDA, 3 forKINSOL, 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 NVVECTOR 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.

FSUNMASSSPSTFQMRINIT

Call FSUNMASSSPSTFQMRINIT(pretype, maxl, ier)

Description The function FSUNMASSSPSTFQMRINIT 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 an 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 NVVECTOR 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.
Section 8.15 The SUNLinearSolver_SPTFQMR implementation

**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 indicating the type of preconditioning to use.

**Return value**
- ier is an 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(pretype, 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 an 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 an 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 an 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.
8.15.4 SUNLinearSolver_SPTFQMR content

The SUNLINSOL_SPTFQMR module defines the content field of a SUNLinearSolver as the following structure:

```c
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.
8.16 The SUNLinearSolver_PCG implementation

This section describes the SUNLINSOL implementation of the PCG (Preconditioned Conjugate Gradient [23]) iterative linear solver. The SUNLINSOL_PCG module is designed to be compatible with any nvector implementation 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.

To access the SUNLINSOL_PC module, include the header file sunlinsol/sunlinsol_pcg.h. We note that the SUNLINSOL_PC module is accessible from SUNDIALS packages without separately linking to the libsundials_sunlinsolpcg module library.

8.16.1 SUNLinearSolver_PCG description

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

\[
\tilde{A}\tilde{x} = \tilde{b}
\]  

(8.4)

where

\[
\tilde{A} = SP^{-1}AP^{-1}S,
\]

\[
\tilde{b} = SP^{-1}b,
\]

\[
\tilde{x} = S^{-1}Px.
\]  

(8.5)

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:

\[
\|\tilde{b} - \tilde{A}\tilde{x}\|_2 < \delta
\]

\[
\Leftrightarrow
\]

\[
\|SP^{-1}b - SP^{-1}Ax\|_2 < \delta
\]

\[
\Leftrightarrow
\]

\[
\|P^{-1}b - P^{-1}Ax\|_S < \delta
\]

where \( \|v\|_S = \sqrt{v^TSv} \), with an input tolerance \( \delta \).

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 the 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.

8.16.2 SUNLinearSolver_PCG functions

The SUNLINSOL_PCG module provides the following user-callable constructor for creating a SUNLinearSolver object.

```
SUNLinSol_PCG
```

Call

```c
LS = SUNLinSol_PCG(y, pretype, maxl);
```

Description

The function SUNLinSol_PCG creates and allocates memory for a PCG SUNLinearSolver object.

Arguments

- `y` (N_Vector) a template for cloning vectors needed within the solver
- `pretype` (int) flag indicating whether to use preconditioning. Since the PCG algorithm is designed to only support symmetric preconditioning, any of the `pretype` inputs PREC_LEFT (1), PREC_RIGHT (2), or PREC_BOTH (3) will result in use of the symmetric preconditioner; any other integer input will result in the default (no preconditioning).
- `maxl` (int) the number of linear iterations to allow; values ≤ 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.

Although some SUNDIALS solvers are designed to only work with left preconditioning (IDA and IDAS) and others with only right preconditioning (KINSOL), PCG should only be used with these packages when the linear systems are known to be symmetric. 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.

Deprecated Name

For backward compatibility, the wrapper function SUNPCG with identical input and output arguments is also provided.

F2003 Name

This function is callable as FSUNLinSol_PCG when using the Fortran 2003 interface module.

The SUNLINSOL_PCG module defines implementations of all “iterative” linear solver operations listed in Sections 8.1.1 – 8.1.3:

- SUNLinSolGetType_PCG
- SUNLinSolInitialize_PCG
- SUNLinSolSetATimes_PCG
8.16 The SUNLinearSolver_PCG implementation

- 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

All of the listed operations are callable via the FORTRAN 2003 interface module by prepending an ‘F’ to the function name.

The SUNLINSOL_PCG module also defines the following additional user-callable functions.

**SUNLinSol_PCGSetPrecType**

Call: `retval = SUNLinSol_PCGSetPrecType(LS, pretype);`

Description: The function `SUNLinSol_PCGSetPrecType` updates the flag indicating use of preconditioning in the `SUNLINSOL_PCG` object.

Arguments: `LS` (SUNLinearSolver) the `SUNLINSOL_PCG` object to update

`pretype` (int) flag indicating use of preconditioning, allowed values match those discussed in `SUNLinSol_PCG`.

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`.

Deprecated Name: For backward compatibility, the wrapper function `SUNPCGSetPrecType` with identical input and output arguments is also provided.

F2003 Name: This function is callable as `FSUNLinSol_PCGSetPrecType` when using the Fortran 2003 interface module.

**SUNLinSol_PCGSetMaxl**

Call: `retval = SUNLinSol_PCGSetMaxl(LS, maxl);`

Description: The function `SUNLinSol_PCGSetMaxl` updates the number of linear solver iterations to allow.

Arguments: `LS` (SUNLinearSolver) the `SUNLINSOL_PCG` object to update

`maxl` (int) flag indicating the number of iterations to allow; values ≤ 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`.

Deprecated Name: For backward compatibility, the wrapper function `SUNPCGSetMaxl` with identical input and output arguments is also provided.

F2003 Name: This function is callable as `FSUNLinSol_PCGSetMaxl` when using the Fortran 2003 interface module.
8.16.3 SUNLinearSolver_PCG Fortran interfaces

The SUNLINSOL_PCG module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

FORTRAN 2003 interface module

The fsunlinsol_pcg_mod FORTRAN module defines interfaces to all SUNLINSOL_PCG C functions using the intrinsic iso_c_binding module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading ‘F’. For example, the function SUNLinSol_PCG is interfaced as FSUNLinSol_PCG.

The FORTRAN 2003 SUNLINSOL_PCG interface module can be accessed with the use statement, i.e. use fsunlinsol_pcg_mod, and linking to the library libsundials_fsunlinsolpcg_mod.lib in addition to the C library. For details on where the library and module file fsunlinsol_pcg_mod.mod are installed see Appendix A. We note that the module is accessible from the FORTRAN 2003 SUNDIALS integrators without separately linking to the libsundials_fsunlinsolpcg_mod library.

FORTRAN 77 interface functions

For solvers that include a FORTRAN 77 interface module, the SUNLINSOL_PCG module also includes a Fortran-callable function for creating a SUNLinearSolver object.

**FSUNPCGINIT**

Call FSUNPCGINIT(code, pretype, maxl, ier)

Description The function FSUNPCGINIT can be called for Fortran programs to create a SUNLINSOL_PCG 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_PCG.

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 SUNLINSOL_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 an 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.
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 indicating the type of preconditioning to use.

**Return value**
- `ier` is an 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.

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 an 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.

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 an 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.

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 `code` is not needed since mass matrix linear systems only arise in `arkode`.

**Return value**
- `ier` is an 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.
8.16.4 SUNLinearSolver_PCG content

The SUNLINSOL_PCG module defines the content field of a SUNLinearSolver as the following structure:

```c
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 NVECTORS which holds the preconditioned linear system residual,
- `p, z, Ap` - NVECTORS used for workspace by the PCG algorithm.

8.17 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*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.
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 (SUNNonlinsSolGetType) and solve the nonlinear system (SUNNonlinSolSolve). The remaining three functions for nonlinear solver initialization (SUNNonlinSolInitialization), setup (SUNNonlinSolSetup), and destruction (SUNNonlinSolFree) are optional.
Description of the SUNNonlinearSolver module

**SUNNonlinSolGetType**

Call

\[ \text{type} = \text{SUNNonlinSolGetType}(\text{NLS}); \]

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

\[ \text{retval} = \text{SUNNonlinSolInitialize}(\text{NLS}); \]

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

\[ \text{retval} = \text{SUNNonlinSolSetup}(\text{NLS}, \text{y}, \text{mem}); \]

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

\[ \text{retval} = \text{SUNNonlinSolSolve}(\text{NLS}, \text{y0}, \text{y}, \text{w}, \text{tol}, \text{callLSetup}, \text{mem}); \]

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.
9.1 The SUNNonlinearSolver API

```
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.

```c
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 SUNNonlinSolSetLSetupFn 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.
Description of the SUNNonlinearSolver module

**SUNNonlinSolSetLSolveFn**

Call: \( \text{retval} = \text{SUNNonlinSolSetLSolveFn} (\text{NLS}, \text{LSolveFn}) \);

Description: The *optional* function `SUNNonlinSolSetLSolveFn` is called by SUNDIALS integrators to provide the nonlinear solver with access to its linear solver solve function.

Arguments:
- \( \text{NLS} \) (SUNNonlinearSolver) a SUNNONLINSOL object
- \( \text{LSolveFn} \) (SUNNonlinSolLSolveFn) a wrapper function to the SUNDIALS integrator’s linear solver solve function. See section 9.1.4 for the definition of SUNNonlinSolLSolveFn.

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 SUNNonlinLSolveFn 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 set this operation to NULL.

**SUNNonlinSolSetConvTestFn**

Call: \( \text{retval} = \text{SUNNonlinSolSetConvTestFn} (\text{NLS}, \text{CTestFn}) \);

Description: The *optional* function `SUNNonlinSolSetConvTestFn` 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:
- \( \text{NLS} \) (SUNNonlinearSolver) a SUNNONLINSOL object.
- \( \text{CTestFn} \) (SUNNonlineSolConvTestFn) a SUNDIALS integrator’s nonlinear solver convergence test function. See section 9.1.4 for the definition of SUNNonlinSolConvTestFn.

Return value: The return value `retval` (of type int) should be zero for a successful call, and a negative value for a failure.

Notes: SUNNONLINSOL implementations utilizing their own convergence test criteria may set this function to NULL.

**SUNNonlinSolSetMaxIters**

Call: \( \text{retval} = \text{SUNNonlinSolSetMaxIters} (\text{NLS}, \text{maxiters}) \);

Description: The *optional* function `SUNNonlinSolSetMaxIters` 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:
- \( \text{NLS} \) (SUNNonlinearSolver) a SUNNONLINSOL object.
- \( \text{maxiters} \) (int) the maximum number of nonlinear iterations.

Return value: The return value `retval` (of type int) should be zero for a successful call, and a negative value for a failure (e.g., \( \text{maxiters} < 1 \)).

9.1.3 SUNNonlinearSolver get functions

The following get functions allow SUNDIALS integrators to retrieve nonlinear solver statistics. The routines to get the current total number of iterations (`SUNNonlinSolGetNumIters`) and number of convergence failures (`SUNNonlinSolGetNumConvFails`) are optional. The routine to get 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.
9.1 The SUNNonlinearSolver API

**SUNNonlinSolGetNumIters**

Call: `retval = SUNNonlinSolGetNumIters(NLS, numiters);`

Description: The `optional` function `SUNNonlinSolGetNumIters` 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:
- `NLS` (SUNNonlinearSolver) a SUNNONLINSOL object
- `numiters` (long int*) the total number of nonlinear solver iterations.

Return value: The return value `retval` (of type int) should be zero for a successful call, and a negative value for a failure.

**SUNNonlinSolGetCurIter**

Call: `retval = SUNNonlinSolGetCurIter(NLS, iter);`

Description: The function `SUNNonlinSolGetCurIter` returns the iteration index of the current nonlinear solve. This function is `required` when using SUNDIALS integrator-provided convergence tests or when using a SUNLINSOL spies linear solver; otherwise it is `optional`.

Arguments:
- `NLS` (SUNNonlinearSolver) a SUNNONLINSOL object
- `iter` (int*) the nonlinear solver iteration in the current solve starting from zero.

Return value: The return value `retval` (of type int) should be zero for a successful call, and a negative value for a failure.

**SUNNonlinSolGetNumConvFails**

Call: `retval = SUNNonlinSolGetNumConvFails(NLS, nconvfails);`

Description: The `optional` function `SUNNonlinSolGetNumConvFails` returns the total number of nonlinear solver convergence failures. This may be called by the SUNDIALS integrator to store the nonlinear solver statistics, but may also be called by the user.

Arguments:
- `NLS` (SUNNonlinearSolver) a SUNNONLINSOL object
- `nconvfails` (long int*) the total number of nonlinear solver convergence failures.

Return value: The return value `retval` (of type int) 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: `typedef int (*SUNNonlinSolSysFn)(N_Vector y, N_Vector F, void* mem);`

Purpose: These functions evaluate the nonlinear system $F(y)$ for SUNNONLINEARSOLVER_ROOTFIND type modules or $G(y)$ for SUNNONLINEARSOLVER_FIXEDPOINT type modules. Memory for $F$ must be allocated prior to calling this function. The vector $y$ must be left unchanged.

Arguments:
- $y$ is the state vector at which the nonlinear system should be evaluated.
- $F$ is the output vector containing $F(y)$ or $G(y)$, depending on the solver type.
mem is 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.

**SUNNonlinSolLSsetupFn**

**Definition**

typedef int (*SUNNonlinSolLSsetupFn)(N_Vector y, N_Vector F,
   bool b jbad,
   bool b jcur, void* mem);

**Purpose**

These functions are wrappers to the SUNDIALS integrator’s function for setting up linear solves with SUNLINSOL modules.

**Arguments**

- **y** is the state vector at which the linear system should be setup.
- **F** is the value of the nonlinear system function at y.
- **jbad** is an input indicating whether the nonlinear solver believes that A has gone stale (SUNTRUE) or not (SUNFALSE).
- **jcur** is an output indicating whether the routine has updated the Jacobian A (SUNTRUE) or not (SUNFALSE).
- **mem** is 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.

**Notes**

The SUNNonlinSolLSsetupFn 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**

typedef int (*SUNNonlinSolLSolveFn)(N_Vector y, N_Vector b, void* mem);

**Purpose**

These functions are wrappers to the SUNDIALS integrator’s function for solving linear systems with SUNLINSOL modules.

**Arguments**

- **y** is the input vector containing the current nonlinear iteration.
- **b** contains the right-hand side vector for the linear solve on input and the solution to the linear system on output.
- **mem** is 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.

**Notes**

The SUNNonlinSolLSolveFn 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**

typedef int (*SUNNonlinSolConvTestFn)(SUNNonlinearSolver NLS, N_Vector y,
   N_Vector del, realtype tol,
   N_Vector ewt, void* mem);

**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.
9.1 The SUNNonlinearSolver API

Arguments

**NLS** is the SUNNONLINSOL object.

**y** is the current nonlinear iterate.

**del** is the difference between the current and prior nonlinear iterates.

**tol** is the nonlinear solver tolerance.

**ewt** is the weight vector used in computing weighted norms.

**mem** is the SUNDIALS integrator memory structure.

Return value

The return value of this routine will be a negative value if an unrecoverable error occurred or one of the following:

- **SUN-NLS_SUCCESS**: 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 **ewt**. 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUN-NLS_SUCCESS</td>
<td>0</td>
<td>successful call or converged solve</td>
</tr>
<tr>
<td>SUN-NLS_CONTINUE</td>
<td>1</td>
<td>the nonlinear solver is not converged, keep iterating</td>
</tr>
<tr>
<td>SUN-NLS_CONV_RECVR</td>
<td>2</td>
<td>the nonlinear solver appears to be diverging, try to recover</td>
</tr>
<tr>
<td>SUN-NLS_MEM_NULL</td>
<td>-1</td>
<td>a memory argument is NULL</td>
</tr>
<tr>
<td>SUN-NLS_MEM_FAIL</td>
<td>-2</td>
<td>a memory access or allocation failed</td>
</tr>
<tr>
<td>SUN-NLS_ILL_INPUT</td>
<td>-3</td>
<td>an illegal input option was provided</td>
</tr>
</tbody>
</table>

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:

```c
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
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:

```c
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 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 \( n \) vectors of length \( n \), \( N_VLinearSum(a,x,b,y,z) \), is
computed as
\[
z_{j,i} = ax_{j,i} + by_{j,i}, \quad i = 0, \ldots, n - 1, \quad j = 0, \ldots, n_v - 1.
\]

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
\[
m = \max_j \left( \frac{1}{n} \sum_{i=0}^{n-1} (x_{j,i}w_{j,i})^2 \right)^{\frac{1}{2}}
\]

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_NewEmpty_SensWrapper**

Call: \(w = N_{VNewEmpty_SensWrapper}(count)\);

Description: The function \(N_{VNewEmpty_SensWrapper}\) creates an empty nvector_senswrapper wrapper with space for \(count\) vectors.

Arguments: count (int) the number of vectors the wrapper will contain.

Return value: The return value \(w\) (of type N_Vector) will be a nvector object if the constructor exits successfully, otherwise \(w\) will be NULL.

**N_NewSensWrapper**

Call: \(w = N_{VNewSensWrapper}(count, y)\);

Description: The function \(N_{VNewSensWrapper}\) creates an nvector_senswrapper wrapper containing \(count\) vectors cloned from \(y\).

Arguments: count (int) the number of vectors the wrapper will contain.
\(y\) (N_Vector) the template vectors to use in creating the vector wrapper.

Return value: The return value \(w\) (of type N_Vector) will be a nvector object if the constructor exits successfully, otherwise \(w\) will be NULL.

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;
    bool 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.Owner.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 module, include the header file sunnonlinsol/sunnonlinsol_newton.h. We note that the SUNNONLINSOL 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

\[ F(y) = 0 \]  

given an initial guess \( y^{(0)} \), Newton’s method computes a series of approximate solutions

\[ y^{(m+1)} = y^{(m)} + \delta^{(m+1)} \]  

where \( m \) is the Newton iteration index, and the Newton update \( \delta^{(m+1)} \) is the solution of the linear system

\[ A(y^{(m)})\delta^{(m+1)} = -F(y^{(m)}) , \]  

in which \( A \) is the Jacobian matrix

\[ A \equiv \frac{\partial F}{\partial y}. \]  

Depending on the linear solver used, the SUNNONLINSOL.Newton module will employ either a Modified Newton method, or an Inexact Newton method [6, 10, 17, 19, 30]. 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 SUNNonlinSolLSetupFn 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 SUNNonlinSolLSetupFn function in two instances:
9.2 The SUNNonlinearSolver_Newton implementation

(a) when requested by the integrator (the input callLSetSetup 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.

```
# SUNNonlinearSolver_Newton
Call NLS = SUNNonlinSol_Newton(y);
Description The function SUNNonlinSol_Newton creates a SUNNonlinearSolver object for use with Sundials integrators to solve nonlinear systems of the form $F(y) = 0$ using Newton's method.
Arguments y (N_Vector) a template for cloning vectors needed within the solver.
Return value The return value NLS (of type SUNNonlinearSolver) will be a SUNNONLINSOL object if the constructor exits successfully, otherwise NLS will be NULL.
F2003 Name This function is callable as FSUNNonlinSol_Newton when using the Fortran 2003 interface module.
```

```
# SUNNonlinearSolver_NewtonSens
Call NLS = SUNNonlinSol_NewtonSens(count, y);
Description The function SUNNonlinSol_NewtonSens creates a SUNNonlinearSolver 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 count (int) the number of vectors in the nonlinear solve. When integrating a system containing $N_s$ sensitivities the value of count is:

- $N_s+1$ if using a simultaneous corrector approach.
- $N_s$ if using a staggered corrector approach.

y (N_Vector) a template for cloning vectors needed within the solver.
Return value The return value NLS (of type SUNNonlinearSolver) will be a SUNNONLINSOL object if the constructor exits successfully, otherwise NLS will be NULL.
F2003 Name This function is callable as FSUNNonlinSol_NewtonSens when using the Fortran 2003 interface module.
```

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: 
```
retval = SUNNonlinSolGetSysFn_Newton(NLS, SysFn);
```

Description: The function `SUNNonlinSolGetSysFn_Newton` returns the residual function that defines the nonlinear system.

Arguments:  
- `NLS` (SUNNonlinearSolver) a SUNNONLINSOL object  
- `SysFn` (SUNNonlinSolSysFn*) the function defining the nonlinear system.

Return value: The return value `retval` (of type `int`) 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 SUNNONLINSOL\_NEWTON module. We note that SUNNONLINSOL\_NEWTON will not leverage the results from any user calls to `SysFn`.

F2003 Name: This function is callable as `FSUNNonlinSolGetSysFn_Newton` when using the Fortran 2003 interface module.

### 9.2.3 SUNNonlinearSolver\_Newton Fortran interfaces

The SUNNONLINSOL\_NEWTON module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

**FORTRAN 2003 interface module**

The `fsunnonlinsol_newton_mod` FORTRAN module defines interfaces to all SUNNONLINSOL\_NEWTON C functions using the intrinsic `iso_c_binding` module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading ‘F’. For example, the function `SUNNonlinSol_Newton` is interfaced as `FSUNNonlinSol_Newton`.

The FORTRAN 2003 SUNNONLINSOL\_NEWTON interface module can be accessed with the `use` statement, i.e. `use fsunnonlinsol_newton_mod`, and linking to the library `libsundials_fsunnonlinsolnewton_mod.lib` in addition to the C library. For details on where the library and module file `fsunnonlinsol_newton_mod.mod` are installed see Appendix A. We note that the module is accessible from the FORTRAN 2003 SUNDIALS integrators *without* separately linking to the `libsundials_fsunnonlinsolnewton_mod` library.

**FORTRAN 77 interface functions**

For SUNDIALS integrators that include a FORTRAN 77 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, 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.
9.2.4 SUNNonlinearSolver_Newton content

The SUNNONLINSOL_NEWTON module defines the content field of a SUNNonlinearSolver as the following structure:

```c
struct _SUNNonlinearSolverContent_Newton {
    SUNNonlinSolSysFn   Sys;
    SUNNonlinSolLSetupFn LSetup;
    SUNNonlinSolLSolveFn LSolve;
    SUNNonlinSolConvTestFn CTest;

    N_Vector          delta;
    booleantype       jcur;
    int               curiter;
    int               maxiters;
    long int          niter;
    long int          nconvfails;
};
```

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
- **niter** - the total number of nonlinear iterations across all solves.
- **nconvfails** - the total number of nonlinear convergence failures across all solves.

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

\[ G(y) = y \]  \hspace{1cm} (9.5)

given an initial guess \( y^{(0)} \), the fixed point iteration computes a series of approximate solutions

\[ y^{(n+1)} = G(y^{(n)}) \]  \hspace{1cm} (9.6)

where \( n \) is the iteration index. The convergence of this iteration may be accelerated using Anderson’s method [4, 39, 20, 34]. With Anderson acceleration using subspace size \( m \), the series of approximate solutions can be formulated as the linear combination

\[ y^{(n+1)} = \sum_{i=0}^{m} \alpha_i^{(n)} G(y^{(n-m+i)}) \]  \hspace{1cm} (9.7)
where \( m_n = \min\{m, n\} \) and the factors

\[ \alpha^{(n)} = (\alpha_0^{(n)}, \ldots, \alpha_{m_n}^{(n)}) \]  

solve the minimization problem \( \min_\alpha \| F_n \alpha^T \|_2 \) under the constraint that \( \sum_{i=0}^{m_n} \alpha_i = 1 \) where

\[ F_n = (f_{n-m_n}, \ldots, f_n) \]  

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 [39], the SUNNONLINSOL\_FIXEDPOINT implementation computes the series of approximate solutions as

\[ y^{(n+1)} = G(y^{(n)}) - \sum_{i=0}^{m_n-1} \gamma_i^{(n)} \Delta g_{n-m_n+i} \]  

with \( \Delta g_i = G(y^{(i+1)}) - G(y^{(i)}) \) and where the factors

\[ \gamma^{(n)} = (\gamma_0^{(n)}, \ldots, \gamma_{m_n-1}^{(n)}) \]  

solve the unconstrained minimization problem \( \min_\gamma \| f_n - \Delta F_n \gamma^T \|_2 \) where

\[ \Delta F_n = (\Delta f_{n-m_n}, \ldots, \Delta f_{n-1}) \]  

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.

| Call | NLS = SUNNonlinSol\_FixedPoint(y, m); |
| Description | The function SUNNonlinSol\_FixedPoint creates a SUNNonlinearSolver object for use with SUNDIALS integrators to solve nonlinear systems of the form \( G(y) = y \). |
| Arguments | y (N\_Vector) a template for cloning vectors needed within the solver m (int) the number of acceleration vectors to use |
| Return value | The return value NLS (of type SUNNonlinearSolver) will be a SUNNONLINSOL object if the constructor exits successfully, otherwise NLS will be NULL. |
| F2003 Name | This function is callable as FSUNNonlinSol\_FixedPoint when using the Fortran 2003 interface module. |
9.3 The SUNNonlinearSolver_FixedPoint implementation

SUNNonlinSol_FixedPointSens
Call
\[ \text{NLS} = \text{SUNNonlinSol.FixedPointSens(count, y, m);} \]

Description
The function SUNNonlinSol_FixedPointSens creates a SUNNonlinearSolver object for use with SUNDIALS sensitivity enabled integrators (CVODES and IDAS) to solve nonlinear systems of the form \( G(y) = y \).

Arguments
- **count** (int) the number of vectors in the nonlinear solve. When integrating a system containing \( N_s \) sensitivities the value of **count** is:
  - \( N_s+1 \) if using a simultaneous corrector approach.
  - \( N_s \) if using a staggered corrector approach.
- **y** (N_Vector) a template for cloning vectors needed within the solver.
- **m** (int) the number of acceleration vectors to use.

Return value
The return value **NLS** (of type SUNNonlinearSolver) will be a SUNNONLINSOL object if the constructor exits successfully, otherwise **NLS** will be NULL.

F2003 Name
This function is callable as FSUNNonlinSol_FixedPointSens when using the Fortran 2003 interface module.

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
\[ \text{retval} = \text{SUNNonlinSolGetSysFn_FixedPoint(NLS, SysFn);} \]

Description
The function SUNNonlinSolGetSysFn_FixedPoint returns the fixed-point function that defines the nonlinear system.

Arguments
- **NLS** (SUNNonlinearSolver) a SUNNONLINSOL object
- **SysFn** (SUNNonlinSolSysFn*) the function defining the nonlinear system.

Return value
The return value **retval** (of type int) 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 SUNNONLINSOL_FIXEDPOINT module. We note that SUNNONLINSOL_FIXEDPOINT will not leverage the results from any user calls to **SysFn**.

F2003 Name
This function is callable as FSUNNonlinSolGetSysFn_FixedPoint when using the Fortran 2003 interface module.

9.3.3 SUNNonlinearSolver_FixedPoint Fortran interfaces

The SUNNONLINSOL_FIXEDPOINT module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.
FORTRAN 2003 interface module

The fsunnonlinsol_fixedpoint_mod FORTRAN module defines interfaces to all SUNNONLINSOL_FIXEDPOINT C functions using the intrinsic iso_c_binding module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading ‘F’. For example, the function SUNNonlinSol_FixedPoint is interfaced as FSUNNonlinSol_FixedPoint.

The FORTRAN 2003 SUNNONLINSOL_FIXEDPOINT interface module can be accessed with the use statement, i.e. use fsunnonlinsol_fixedpoint_mod, and linking to the library lib sundials fsunnonlinsol_fixedpoint_mod.lib in addition to the C library. For details on where the library and module file fsunnonlinsol_fixedpoint_mod.mod are installed see Appendix A. We note that the module is accessible from the FORTRAN 2003 SUNDIALS integrators without separately linking to the lib sundials fsunnonlinsol_fixedpoint_mod library.

FORTRAN 77 interface functions

For SUNDIALS integrators that include a FORTRAN 77 interface, the SUNNONLINSOL_FIXEDPOINT module also includes a Fortran-callable function for creating a SUNNonlinearSolver object.

```fortran
FSUNFIXEDPOINTINIT
Call FSUNFIXEDPOINTINIT(code, m, ier);
Description The function FSUNFIXEDPOINTINIT can be called for Fortran programs to create a SUNNonlinearSolver object for use with SUNDIALS integrators to solve nonlinear systems of the form G(y) = y.
Arguments code (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, and 4 for ARKODE).
   m   (int*) is an integer input specifying the number of acceleration vectors.
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.
```

9.3.4 SUNNonlinearSolver_FixedPoint content

The SUNNONLINSOL_FIXEDPOINT module defines the content field of a SUNNonlinearSolver as the following structure:

```c
 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;
long int nconvfails;
};

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.
nconvfails - the total number of nonlinear convergence failures 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 $m \times m$)
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
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/CMAKE_INSTALL_LIBDIR, with instdir and CMAKE_INSTALL_LIBDIR 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 CMaké-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](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:

```bash
% 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
A.1 CMake-based installation

- 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.

![Default configuration screen](image)

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 instdir 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:
274 SUNDIALS Package Installation Procedure

Figure A.2: Changing the instdir for SUNDIALS and corresponding examples

% make

To install SUNDIALS in the installation directory specified in the configuration, simply run:

% make install

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 provided below. Note that the default values shown are for a typical configuration on a Linux system and are provided as illustration only.
A.1 CMake-based installation

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 CMAKE_INSTALL_LIBDIR of CMAKE_INSTALL_PREFIX, respectively.

CMAKE_INSTALL_LIBDIR - Library installation directory
   Default:
   Note: This is the directory within CMAKE_INSTALL_PREFIX that the SUNDIALS libraries will be installed under. The default is automatically set based on the operating system using the GNUInstallDirs CMake module.

Fortran_INSTALL_MODDIR - Fortran module installation directory
   Default: fortran

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 unless Trilinos_ENABLE is ON.

EXAMPLES_ENABLE_F77 - Build the SUNDIALS Fortran77 examples
   Default: ON (if F77_INTERFACE_ENABLE is ON)
A.1 CMake-based installation

**EXAMPLES_ENABLE_F90** - Build the SUNDIALS Fortran90/Fortran2003 examples
Default: ON (if F77_INTERFACE_ENABLE or F2003_INTERFACE_ENABLE is ON)

**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.

**F77_INTERFACE_ENABLE** - Enable Fortran-C support via the Fortran 77 interfaces
Default: OFF

**F2003_INTERFACE_ENABLE** - Enable Fortran-C support via the Fortran 2003 interfaces
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 informa-
tion 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. This will build the parallel NVector and the MPI-aware version
of the ManyVector library.
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) and Fortran-C support is enabled (F77_INTERFACE_ENABLE or F2003_INTERFACE_ENABLE is 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

OPENMP_DEVICE_ENABLE - Enable OpenMP device offloading (build the OpenMPDEV nvector) if supported by the provided compiler.
 Default: OFF

SKIP_OPENMP_DEVICE_CHECK - advanced option - Skip the check done to see if the OpenMP provided by the compiler supports OpenMP device offloading.
 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 option - 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

SUPERLU_DIST_ENABLE - Enable SuperLU_DIST support
Default: OFF
Note: See additional information on building with SuperLU_DIST enabled in A.1.4.

SUPERLU_DIST_INCLUDE_DIR - Path to SuperLU_DIST header files (typically SRC directory)

SUPERLU_DIST_LIBRARY_DIR - Path to SuperLU_DIST installed library files

SUPERLU_DIST_LIBRARIES - Semi-colon separated list of libraries needed for SuperLU_DIST

SUPERLU_DIST_OpenMP - Enable SUNDIALS support for SuperLU_DIST built with OpenMP
Default: OFF
Note: SuperLU_DIST must be built with OpenMP support for this option to function properly. Additionally the environment variable OMP_NUM_THREADS must be set to the desired number of threads.

SUPERLUMT_ENABLE - Enable SUPERLUMT support
Default: OFF
Note: See additional information on building with SUPERLUMT 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

Trilinos_ENABLE - Enable Trilinos support (build the Tpetra NVECTOR).
Default: OFF

Trilinos_DIR - Path to the Trilinos install directory.
Default:

TRILINOS_INTERFACE_C_COMPILER - Advanced option - Set the C compiler for building the Trilinos interface (i.e., nvector_trilinos and the examples that use it).
Default: The C compiler exported from the found Trilinos installation if USE_XSDK_DEFAULTS=OFF. CMAKE_C_COMPILER or MPI_C_COMPILER if USE_XSDK_DEFAULTS=ON.
Note: It is recommended to use the same compiler that was used to build the Trilinos library.
TRILINOS_INTERFACE_C_COMPILER_FLAGS - advanced option - Set the C compiler flags for Trilinos interface (i.e., NVECTOR_TRILINOS and the examples that use it).
Default: The C compiler flags exported from the found Trilinos installation if USE_XSDK_DEFAULTS=OFF.
CMAKE_C_FLAGS if USE_XSDK_DEFAULTS=ON.
Note: It is recommended to use the same flags that were used to build the Trilinos library.

TRILINOS_INTERFACE_CXX_COMPILER - advanced option - Set the C++ compiler for building Trilinos interface (i.e., NVECTOR_TRILINOS and the examples that use it).
Default: The C++ compiler exported from the found Trilinos installation if USE_XSDK_DEFAULTS=OFF.
CMAKE_CXX_COMPILER or MPI_CXX_COMPILER if USE_XSDK_DEFAULTS=ON.
Note: It is recommended to use the same compiler that was used to build the Trilinos library.

TRILINOS_INTERFACE_CXX_COMPILER_FLAGS - advanced option - Set the C++ compiler flags for Trilinos interface (i.e., NVECTOR_TRILINOS and the examples that use it).
Default: The C++ compiler flags exported from the found Trilinos installation if USE_XSDK_DEFAULTS=OFF.
CMAKE_CXX_FLAGS if USE_XSDK_DEFAULTS=ON.
Note: It is recommended to use the same flags that were used to build the Trilinos library.

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
A.1 CMake-based installation

TPL_ENABLE_LAPACK - Enable LAPACK support
  Default: OFF
  SUNDIALS equivalent: LAPACK_ENABLE

TPL_ENABLE_SUPERLUDIST - Enable SuperLU_DIST support
  Default: OFF
  SUNDIALS equivalent: SUPERLUDIST_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_SUPERLUDIST_INCLUDE_DIRS - Path to SuperLU_DIST header files
  SUNDIALS equivalent: SUPERLUDIST_INCLUDE_DIR

TPL_SUPERLUDIST_LIBRARIES - Semi-colon separated list of libraries needed for SuperLU_DIST including the SuperLU_DIST library itself
  SUNDIALS equivalent: SUPERLUDIST_LIBRARIES

TPL_SUPERLUDIST_OPENMP - Enable SUNDIALS support for SuperLU_DIST built with OpenMP
  SUNDIALS equivalent: SUPERLUDIST_OPENMP

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 interfaces
  Default: OFF
  SUNDIALS equivalent: F77_INTERFACE_ENABLE/F2003_INTERFACE_ENABLE
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:

```bash
% 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:

```bash
% 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:

```bash
% cmake \
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
```

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 CMake-based installation

> -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 5.3.0. 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. 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 SuperLU_DIST

The SuperLU_DIST libraries are available for download from the Lawrence Berkeley National Laboratory website: http://crd-legacy.lbl.gov/~xiaoye/SuperLU/#superlu_dist. SUNDIALS has been tested with SuperLU_DIST greater than 6.1. To enable SuperLU_DIST, set SUPERLUDIST_ENABLE to ON, set SUPERLUDIST_INCLUDE_DIR to the include directory of the SuperLU_DIST installation (typically SRC), and set the variable SUPERLUDIST_LIBRARY_DIR to the path to library directory of the SuperLU_DIST installation (typically lib). At the same time, the variable SUPERLUDIST_LIBRARIES must be set to a semi-colon separated list of other libraries SuperLU_DIST depends on. For example, if SuperLU_DIST was built with LAPACK, then include the LAPACK library in this list. If SuperLU_DIST was built with OpenMP support, then you may set SUPERLUDIST_OPENMP to ON to utilize the OpenMP functionality of SuperLU_DIST.

Do not mix thread types when building SUNDIALS solvers. If threading is enabled for SUNDIALS by having PTHREAD_ENABLE set to ON then SuperLU_DIST should not be set to use OpenMP.

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 3.10.3. 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 2.14.0. 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 9.0 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 0.6. 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_CUDA to ON.

**Building with Trilinos**

Trilinos is a suite of numerical libraries developed by Sandia National Laboratories. It can be obtained at https://github.com/trilinos/Trilinos. SUNDIALS Trilinos modules and examples have been tested with Trilinos version 12.14. To enable Trilinos, set Trilinos_ENABLE to ON. If Trilinos is installed in a nonstandard location you will be prompted to set the variable Trilinos_DIR with the path to the Trilinos CMake configuration file. It is desirable to build the Trilinos vector interface with same compiler and options that were used to build Trilinos. CMake will try to find the correct compiler settings automatically from the Trilinos configuration file. If that is not successful, the compilers and options can be manually set with the following CMake variables:

- Trilinos_INTERFACE_C_COMPILER
- Trilinos_INTERFACE_C_COMPILER_FLAGS
- Trilinos_INTERFACE_CXX_COMPILER
- Trilinos_INTERFACE_CXX_COMPILER_FLAGS

**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/CMAKE_INSTALL_LIBDIR and instdir/include, respectively. The location can be changed by setting the CMake variable CMAKE_INSTALL_PREFIX. Although all installed libraries reside under libdir/CMAKE_INSTALL_LIBDIR, 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>
<thead>
<tr>
<th>Type</th>
<th>Libraries</th>
<th>Header files</th>
<th>Module files</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SHARED</strong></td>
<td>sundials/sundials_config.h</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sundials/sundials_fconfig.h</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sundials/sundials_types.h</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sundials/sundials_math.h</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sundials/sundials_nvector.h</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sundials/sundials_fnvector.h</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sundials/sundials_matrix.h</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sundials/sundials_linear_solver.h</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sundials/sundials_iterative.h</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sundials/sundials_direct.h</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sundials/sundials_dense.h</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sundials/sundials_band.h</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sundials/sundials_nonlinear_solver.h</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sundials/sundials_version.h</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sundials/sundials_mpi_types.h</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>NVECTOR_SERIAL</strong></td>
<td>libsundials_nvecserial.lib</td>
<td>nvector/nvecserial.h</td>
<td>fnvector_serial_mod.mod</td>
</tr>
<tr>
<td></td>
<td>libsundials_fnvecserial_mod.lib</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>libsundials_fnvecserial.a</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>NVECTOR_PARALLEL</strong></td>
<td>libsundials_nvecparallel.lib</td>
<td>nvector/nvecparallel.h</td>
<td>fnvector_parallel_mod.mod</td>
</tr>
<tr>
<td></td>
<td>libsundials_fnvecparallel_mod.lib</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>libsundials_fnvecparallel.a</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>NVECTOR_OPENMP</strong></td>
<td>libsundials_nvecopenmp.lib</td>
<td>nvector/nvecopenmp.h</td>
<td>fnvector_openmp_mod.mod</td>
</tr>
<tr>
<td></td>
<td>libsundials_fnvecopenmp_mod.lib</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>libsundials_fnvecopenmp.a</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>NVECTOR_OPENMPDEV</strong></td>
<td>libsundials_nvecopenmpdev.lib</td>
<td>nvector/nvecopenmpdev.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>NVECTOR_PTHREADS</strong></td>
<td>libsundials_nvecpthreads.lib</td>
<td>nvector/nvecpthreads.h</td>
<td>fnvector_pthreads_mod.mod</td>
</tr>
<tr>
<td></td>
<td>libsundials_fnvecpthreads_mod.lib</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>libsundials_fnvecpthreads.a</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>NVECTOR_PARHYP</strong></td>
<td>libsundials_nvecparhyp.lib</td>
<td>nvector/nvecparhyp.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*continued on next page*
### NVectors

<table>
<thead>
<tr>
<th>NVectors Type</th>
<th>Libraries</th>
<th>Header Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVECTOR_PETSC</td>
<td>libsundials_nvecpetsc.lib</td>
<td>nvector/nvector_petsc.h</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NVECTOR_CUDA</td>
<td>libsundials_nveccuda.lib</td>
<td>nvector/nvector_cuda.h nvector/nvector_mpicuda.h nvector/cuda/ThreadPartitioning.hpp nvector/cuda/Vector.hpp nvector/cuda/VectorKernels.cuh</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NVECTOR_RAJA</td>
<td>libsundials_nveccudaraja.lib</td>
<td>nvector/nvector_raja.h nvector/nvector_mpiraja.h nvector/raja/Vector.hpp</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NVECTOR_TRILINOS</td>
<td>libsundials_nvectrilinos.lib</td>
<td>nvector/nvector_trilinos.h nvector/trilinos/SundialsTpetraVectorInterface.hpp nvector/trilinos/SundialsTpetraVectorKernels.hpp</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUNMATRIX_BAND</td>
<td>libsundials_sunmatrixband.lib</td>
<td>nvector/trilinos/sunmatrixband.mod.lib nvector/trilinos/sunmatrixband.a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>summatrix/sunmatrix_band.h</td>
</tr>
<tr>
<td></td>
<td></td>
<td>summatrix/sunmatrix_band.mod</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUNMATRIX_DENSE</td>
<td>libsundials_sunmatrixdense.lib</td>
<td>nvector/trilinos/sunmatrixdense.mod.lib nvector/trilinos/sunmatrixdense.a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>summatrix/sunmatrix_dense.h</td>
</tr>
<tr>
<td></td>
<td></td>
<td>summatrix/sunmatrix_dense.mod</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUNMATRIX_SPARSE</td>
<td>libsundials_sunmatrixsparse.lib</td>
<td>nvector/trilinos/sunmatrixsparse.mod.lib nvector/trilinos/sunmatrixsparse.a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>summatrix/sunmatrix_sparse.h</td>
</tr>
<tr>
<td></td>
<td></td>
<td>summatrix/sunmatrix_sparse.mod</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUNLINSOL_BAND</td>
<td>libsundials_sunlinsolband.lib</td>
<td>nvector/trilinos/sunlinsolband.mod.lib nvector/trilinos/sunlinsolband.a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sunlinsol/sunlinsol_band.h</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sunlinsol/sunlinsol_band.mod</td>
</tr>
</tbody>
</table>
| SUNLINSOL_DENSE | Libraries           | libsundials_sunlinsoldense.lib  
|                 |                    | libsundials_fsunlinsoldense_mod.lib  
|                 |                    | libsundials_fsunlinsoldense.a  
|                   | Header files       | sunlinsol/sunlinsol_dense.h  
|                   | Module files       | fsunlinsol_dense_mod.mod  
| SUNLINSOL_KLU    | Libraries           | libsundials_sunlinsolklu.lib  
|                 |                    | libsundials_fsunlinsolklu_mod.lib  
|                 |                    | libsundials_fsunlinsolklu.a  
|                   | Header files       | sunlinsol/sunlinsol_klu.h  
|                   | Module files       | fsunlinsol_klu_mod.mod  
| SUNLINSOL_LAPACKBAND | Libraries   | libsundials_sunlinsollapackband.lib  
|                   |                    | libsundials_fsunlinsollapackband.a  
|                   | Header files       | sunlinsol/sunlinsol_lapackband.h  
|                   | SUNLINSOL_LAPACKDENSE Libraries | libsundials_sunlinsollapackdense.lib  
|                   |                    | libsundials_fsunlinsollapackdense.a  
|                   | Header files       | sunlinsol/sunlinsol_lapackdense.h  
|                   | SUNLINSOL_PCG      | Libraries           | libsundials_sunlinsolpcg.lib  
|                 |                    | libsundials_fsunlinsolpcg_mod.lib  
|                 |                    | libsundials_fsunlinsolpcg.a  
|                   | Header files       | sunlinsol/sunlinsol_pcg.h  
|                   | Module files       | fsunlinsol_pcg_mod.mod  
| SUNLINSOL_SPBCGS | Libraries           | libsundials_sunlinsolspbcgs.lib  
|                 |                    | libsundials_fsunlinsolspbcgs_mod.lib  
|                 |                    | libsundials_fsunlinsolspbcgs.a  
|                   | Header files       | sunlinsol/sunlinsol_spbcgs.h  
|                   | Module files       | fsunlinsol_spbcgs_mod.mod  
| SUNLINSOL_SPFGMR | Libraries           | libsundials_sunlinsolspfgmr.lib  
|                 |                    | libsundials_fsunlinsolspfgmr_mod.lib  
|                 |                    | libsundials_fsunlinsolspfgmr.a  
|                   | Header files       | sunlinsol/sunlinsol_spfgmr.h  
|                   | Module files       | fsunlinsol_spfgmr_mod.mod  
| SUNLINSOL_SPGMR  | Libraries           | libsundials_sunlinsolspgmr.lib  
|                 |                    | libsundials_fsunlinsolspgmr_mod.lib  
|                 |                    | libsundials_fsunlinsolspgmr.a  
|                   | Header files       | sunlinsol/sunlinsol_spgmr.h  
|                   | Module files       | fsunlinsol_spgmr_mod.mod  

*continued on next page*
### SUNLINSOL_SPTFQMR
- **Libraries:**
  - libsundials_sunlinsolsptfqmr.lib
  - libsundials_fsunlinsolsptfqmr_mod.lib
  - libsundials_fsunlinsolsptfqmr.a
- **Header files:**
  - sunlinsol/sunlinsol_sptfqmr.h
- **Module files:**
  - fsunlinsol_sptfqmr_mod.mod

### SUNLINSOL_SUPERLUMT
- **Libraries:**
  - libsundials_sunlinsolsuperlumt.lib
  - libsundials_fsunlinsolsuperlumt.a
- **Header files:**
  - sunlinsol/sunlinsol_superlumt.h
- **Module files:**
  - fsunlinsol_superlumt_mod.mod

### SUNNONLINSOL_NEWTON
- **Libraries:**
  - libsundials_sunnonlinsolnewton.lib
  - libsundials_fsunnonlinsolnewton_mod.lib
  - libsundials_fsunnonlinsolnewton.a
- **Header files:**
  - sunnonlinsol/sunnonlinsol_newton.h
- **Module files:**
  - fsunnonlinsol_newton_mod.mod

### SUNNONLINSOL_FIXEDPOINT
- **Libraries:**
  - libsundials_sunnonlinsolfixedpoint.lib
  - libsundials_fsunnonlinsolfixedpoint_mod.lib
  - libsundials_fsunnonlinsolfixedpoint.a
- **Header files:**
  - sunnonlinsol/sunnonlinsol_fixedpoint.h
- **Module files:**
  - fsunnonlinsol_fixedpoint_mod.mod

### CVODE
- **Libraries:**
  - libsundials_cvode.lib
  - libsundials_fcvode.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
- **Module files:**
  - fcvode_mod.mod

### CVODES
- **Libraries:**
  - libsundials_cvodes.lib
  - libsundials_fcvodes.a
- **Header files:**
  - cvodes/cvodes.h
  - cvodes/cvodes_impl.h
  - cvodes/cvodes_direct.h
  - cvodes/cvodes_ls.h
  - cvodes/cvodes_spils.h
  - cvodes/cvodes_bandpre.h

### ARKODE
- **Libraries:**
  - libsundials_arkode.lib
  - libsundials_farkode.a
- **Header files:**
  - arkode/arkode.h
  - arkode/arkode_impl.h
  - arkode/arkode_ls.h
  - arkode/arkode_bandpre.h

### IDA
- **Libraries:**
  - libsundials_ida.lib
  - libsundials_fida.a
- **Header files:**
  - ida/ida.h
  - ida/ida_impl.h
  - ida/ida_direct.h
  - ida/ida_ls.h
  - ida/ida_spils.h
  - ida/ida_bandpre.h

### IDAS
- **Libraries:**
  - libsundials_idas.lib
  - libsundials_fidasa.a
- **Header files:**
  - idas/idas.h
  - idas/idas_impl.h
  - idas/idas_direct.h
  - idas/idas_ls.h
  - idas/idas_spils.h
  - idas/idas_bandpre.h

---

**continued on next page**
<table>
<thead>
<tr>
<th></th>
<th>Header files</th>
<th>Libraries</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IDAS</strong></td>
<td>idas/idas.h</td>
<td>libsundials_kinsol.lib</td>
</tr>
<tr>
<td></td>
<td>idas/idas_impl.h</td>
<td>libsundials_fkinsol.a</td>
</tr>
<tr>
<td></td>
<td>idas/idas_direct.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>idas/idas_ls.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>idas/idas_spils.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bbdpre.h</td>
<td></td>
</tr>
<tr>
<td><strong>KINSOL</strong></td>
<td>kinsol/kinsol.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>kinsol/kinsol_impl.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>kinsol/kinsol_direct.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>kinsol/kinsol_ls.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>kinsol/kinsol_spils.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>kinsol/kinsol_bbdpre.h</td>
<td></td>
</tr>
</tbody>
</table>
# 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

<table>
<thead>
<tr>
<th>IDA main solver module</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDA_NORMAL</td>
</tr>
<tr>
<td>IDA_ONE_STEP</td>
</tr>
<tr>
<td>IDA_YA_YDP_INIT</td>
</tr>
<tr>
<td>IDA_Y_INIT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Iterative linear solver module</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREC_NONE</td>
</tr>
<tr>
<td>PREC_LEFT</td>
</tr>
<tr>
<td>MODIFIED_GS</td>
</tr>
<tr>
<td>CLASSICAL_GS</td>
</tr>
</tbody>
</table>

### B.2 IDA output constants

<table>
<thead>
<tr>
<th>IDA main solver module</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDA_SUCCESS</td>
</tr>
<tr>
<td>IDA_TSTOP_RETURN</td>
</tr>
<tr>
<td>IDA_ROOT_RETURN</td>
</tr>
<tr>
<td>IDA_WARNING</td>
</tr>
<tr>
<td>IDA_TOO_MUCH_WORK</td>
</tr>
<tr>
<td>IDA_TOO_MUCH_ACC</td>
</tr>
<tr>
<td>IDA_ERR_FAIL</td>
</tr>
<tr>
<td>IDA_Const</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>IDA_CONV_FAIL</td>
</tr>
<tr>
<td>IDA_INIT_FAIL</td>
</tr>
<tr>
<td>IDA_SETUP_FAIL</td>
</tr>
<tr>
<td>IDA_SOLVE_FAIL</td>
</tr>
<tr>
<td>IDA_RES_FAIL</td>
</tr>
<tr>
<td>IDA_REP_RES_FAIL</td>
</tr>
<tr>
<td>IDA_RT_FUNC_FAIL</td>
</tr>
<tr>
<td>IDA_CONSTR_FAIL</td>
</tr>
<tr>
<td>IDA_FIRST_RES_FAIL</td>
</tr>
<tr>
<td>IDA_LINSEARCH_FAIL</td>
</tr>
<tr>
<td>IDA_NO_RECOVERY</td>
</tr>
<tr>
<td>IDA_NLS_INIT_FAIL</td>
</tr>
<tr>
<td>IDA_NLS_SETUP_FAIL</td>
</tr>
<tr>
<td>IDA_MEM_NULL</td>
</tr>
<tr>
<td>IDA_MEM_FAIL</td>
</tr>
<tr>
<td>IDA_ILL_INPUT</td>
</tr>
<tr>
<td>IDA_NO_MALLOC</td>
</tr>
<tr>
<td>IDA_BAD_EWT</td>
</tr>
<tr>
<td>IDA_BAD_K</td>
</tr>
<tr>
<td>IDA_BAD_T</td>
</tr>
<tr>
<td>IDA_BAD_DKY</td>
</tr>
</tbody>
</table>

**IDALS linear solver interface**

<table>
<thead>
<tr>
<th>IDALS_Const</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDALS_SUCCESS</td>
<td>0</td>
<td>Successful function return.</td>
</tr>
<tr>
<td>IDALS_MEM_NULL</td>
<td>-1</td>
<td>The ida_mem argument was NULL.</td>
</tr>
<tr>
<td>IDALS_LMEM_NULL</td>
<td>-2</td>
<td>The IDALS linear solver has not been initialized.</td>
</tr>
<tr>
<td>IDALS_IILL_INPUT</td>
<td>-3</td>
<td>The IDALS solver is not compatible with the current NVECTOR module.</td>
</tr>
<tr>
<td>IDALS_MEM_FAIL</td>
<td>-4</td>
<td>A memory allocation request failed.</td>
</tr>
<tr>
<td>IDALS_PMEM_NULL</td>
<td>-5</td>
<td>The preconditioner module has not been initialized.</td>
</tr>
<tr>
<td>IDALS_JAC_FUNC_UNRECVR</td>
<td>-6</td>
<td>The Jacobian function failed in an unrecoverable manner.</td>
</tr>
<tr>
<td>IDALS_JAC_FUNC_RECVR</td>
<td>-7</td>
<td>The Jacobian function had a recoverable error.</td>
</tr>
<tr>
<td>IDALS_SUNMAT_FAIL</td>
<td>-8</td>
<td>An error occurred with the current SUNMATRIX module.</td>
</tr>
<tr>
<td>IDALS_SUNLS_FAIL</td>
<td>-9</td>
<td>An error occurred with the current SUNLINSOL module.</td>
</tr>
</tbody>
</table>
Bibliography


Index

BIG_REAL, 26, 103, 109
boolean, 26

CONSTR_VEC, 90

data types
Fortran, 79

error messages, 39
redirecting, 39
user-defined handler, 41, 67

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
FIDABBINIT, 93
FIDABBDOPT, 94
FIDABBDREINIT, 94
FIDABJAC, 85
FIDACOMMFN, 94
FIDADENSESETJAC, 85
FIDADJAC, 84
FIDADLSINIT, 84
FIDAET, 84
FIDAETSET, 84
FIDAFREE, 89
FIDAGETDKY, 88
FIDAGETERRWEIGHTS, 89
FIDAGETESTLOCALERR, 92
FIDAGLOCFN, 94
FIDAJTIMES, 86, 95
FIDAJTSETUP, 86, 95
FIDALSINIT, 84
FIDALSETJAC, 86
FIDALSSETPREC, 87
FIDAMALLOC, 84
FIDAMALLOC, 83
FIDAPSET, 87

FIDAPSO, 87
FIDAREINIT, 88
FIDARESFUN, 82
FIDASETINI, 89
FIDASETTRI, 89
FIDARETHIN, 89
FIDASOLVE, 88
FIDASPARSESETJAC, 86
FIDASPINSETJAC, 87
FIDASPINSETPREC, 88
FIDASPININIT, 84
FIDATOLREINIT, 89
fnvector_serial_mod, 115
FSUNBANDLINSOLINIT, 201
FSUNDESELINSOLINIT, 199
FSUNFIXEDPOINTINIT, 268
FSUNKLINIT, 210
FSUNKLUREINIT, 211
FSUNKLUSEORDERING, 212
FSUNLAPACKBANDINIT, 206
FSUNLAPACKDENSEINIT, 204
fsunlinsol_band_mod, 201
fsunlinsol_dense_mod, 199
fsunlinsol_klu_mod, 210
fsunlinsol_pcg_mod, 248
fsunlinsol_spbcgs_mod, 236
fsunlinsol_spfgmr_mod, 230
fsunlinsol_spfgmr_mod, 224
fsunlinsol_sptfqmr_mod, 242
FSUNMASSBANDLINSOLINIT, 202
FSUNMASSDENSELINSOLINIT, 199
FSUNMASSKLINIT, 211
FSUNMASSKLUREINIT, 211
FSUNMASSKLUSEORDERING, 212
FSUNMASSLAPACKBANDINIT, 207
FSUNMASSLAPACKDENSEINIT, 204
FSUNMASSPOGINIT, 248
FSUNMASSPCGSEMAXL, 249
FSUNMASSPCGSETPRECTYPE, 249
FSUNMASSPCGSEMAXL, 237
FSUNMASSPCGSEMAXL, 238
FSUNMASSPCGSEMAXL, 238
FSUNMASSPCGSEMAXL, 231
F SUN MSSSF GMRSETGSTYPE, 231
F SUN MSSSF GMRSETMAXRS, 232
F SUN MSSSF GMRSETPRECTYPE, 232
F SUN MSSSF GMRINIT, 224
F SUN MSSSF GMRSETGSTYPE, 225
F SUN MSSSF GMRSETMAXRS, 226
F SUN MSSSF GMRSETPRECTYPE, 225
F SUN MSSSF T QMRINIT, 242
F SUN MSSSF T QMRSETMAXL, 243
F SUN MSSSF T QMRSETPRECTYPE, 243
F SUN MSSS SPTF QMRINIT, 242
F SUN MSSS SPTF QMRSETMAXL, 243
F SUN MSSS SPTF QMRSETPRECTYPE, 243
F SUN SY M SUPER LUMTINIT, 219
F SUN SY M SUPER LUMTSETORDERING, 220
fsunmatrix_band_mod, 174
fsunmatrix_dense_mod, 169
fsunmatrix_sparse_mod, 182
F SUN NEW TONINIT, 264
fsunnonlinsol_fixedpoint_mod, 268
fsunnonlinsol_newton_mod, 264
FS UNPC GINI T, 248
FS UNPC GSET MAX L, 249
FS UNPC GSETPRECTYPE, 249
FS UNSPBC GSI NIT, 237
FS UNSPBCG SSET MAX L, 238
FS UNSPBC GSETPRECTYPE, 237
F SUNSPFGMRINIT, 230
F SUNSPFGMRSETGSTYPE, 231
F SUNSPFGMRSETMAXRS, 232
F SUNSPFGMRSETPRECTYPE, 232
F SUNSPGMRINIT, 224
F SUNSPGMRSETGSTYPE, 225
F SUNSPGMRSETMAXRS, 226
F SUNSPGMRSETPRECTYPE, 225
F SUNSP TQMRINIT, 242
F SUNSP TQMRSETMAXL, 243
F SUNSP TQMRSETPRECTYPE, 243
F SUNSUPERLUMTINIT, 219
F SUNSUPERLUMTSETORDERING, 219
half-bandwidths, 76
header files, 27, 75

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, 52
IDA_BAD_EWT, 37
IDA_BAD_K, 52
IDA_BAD_T, 52
IDA_CONSTR_FAIL, 37, 39
IDA_CONV_FAIL, 37, 39
IDA_ERR_FAIL, 39
IDA_FIRST_RES_FAIL, 37
IDA_ILI NPUT, 32, 33, 36–38, 42–45, 49–51, 60, 66
IDA_L INE SEAR CCH_FAIL, 37
IDA_LINIT_FAIL, 37, 39
IDA_LSETUP_FAIL, 37, 39
IDA_LSOLVE_FAIL, 37, 39
IDA_MEM_FAIL, 32, 43, 59
IDA_MEM_NULL, 32, 33, 36–38, 41–45, 49–52, 54–61, 66
IDA_NO_MALLOC, 33, 37, 66
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, 68
IDA_SUCCESS, 32, 33, 36–38, 41–45, 49–52, 60, 61, 65
IDA_TOO_MUCH_ACC, 39
IDA_TOO_MUCH_WORK, 39
IDA_TSTOP_RETURN, 38
IDA_WARNING, 67
IDA_Y_INIT, 36
IDA_YA_YDP_INIT, 36
IDABBDPre preconditioner
description, 73–74
optional output, 77–78
usage, 75–76
user-callable functions, 76–77
user-supplied functions, 74–75
IDABBDPrecGetNumGfnEvals, 78
IDABBDPrecGetWorkSpace, 78
IDABBDPrecInit, 76
IDABBDPrecReInit, 77
IDACalcIC, 36
IDACreate, 32
IDALsGetLastFlag, 64
IDALsGetNumJacEvals, 62
IDALsGetNumRhsEvals, 62
IDALsGetReturnFlagName, 65
IDALsGetWorkspace, 61
IDALsJacFn, 70
IDALsSetJacFn, 46
IDALsSetLinearSolver, 35
INDEX

IDAErrHandlerFn, 67
IDAEvtFn, 67
IDAFree, 31, 32
IDAGetActualInitStep, 57
IDAGetConsistentIC, 60
IDAGetCurrentOrder, 56
IDAGetCurrentStep, 57
IDAGetCurrentTime, 57
IDAGetDky, 51, 52
IDAGetErrWeights, 58
IDAGetEstLocalErrors, 58
IDAGetIntegratorStats, 58
IDAGetLastLinFlag, 64
IDAGetLastOrder, 56
IDAGetLastStep, 56
IDAGetLinReturnFlagName, 65
IDAGetLinWorkSpace, 61
IDAGetNumBacktrackOps, 60
IDAGetNumErrTestFails, 56
IDAGetNumGEvals, 61
IDAGetNumJacEvals, 61
IDAGetNumJtimesEvals, 64
IDAGetNumJTSetupEvals, 63
IDAGetNumLinConvFails, 62
IDAGetNumLinIters, 62
IDAGetNumLinResEvals, 62
IDAGetNumLinSolvSetups, 55
IDAGetNumNonlinSolvConvFails, 59
IDAGetNumNonlinSolvIters, 59
IDAGetNumPrecEvals, 63
IDAGetNumPrecSolves, 63
IDAGetNumResEvals, 55
IDAGetNumSteps, 55
IDAGetReturnFlagName, 59
IDAGetRootInfo, 60
IDAGetToIIScaleFactor, 57
IDAGetWorkSpace, 54
IDAInit, 32, 65
IDALS linear solver interface
  convergence test, 48
  Jacobian approximation used by, 45, 46
  memory requirements, 61
  optional input, 45–48
  optional output, 61–65
  preconditioner setup function, 48, 72
  preconditioner solve function, 48, 71
IDALS_IILL_INPUT, 35, 47, 48, 77
IDALS_LMEM_NULL, 46–48, 61–64, 77
IDALS_MEM_NULL, 35, 77
IDALS_MEM_NULL, 35, 46–48, 61–64
IDALS_PMEM_NULL, 77, 78
IDALS_SUCCESS, 35, 46–48, 61–64
IDALS_SUNLS_FAIL, 35, 46, 48
IDALSJacFn, 68
IDALSJacTimesSetupFn, 71
IDALSJacTimesVecFn, 70
IDALSPrecSetupFn, 72
IDALSPrecSolveFn, 71
IDAREInit, 65
IDAResFn, 32, 66
IDARootFn, 67
IDARootInit, 37
IDASetConstraints, 45
IDASetEpsLin, 48
IDASettErrFile, 41
IDASettErrHandlerFn, 41
IDASetId, 45
IDASetIncrementFactor, 47
IDASetInitStep, 42
IDASetJacFn, 46
IDASetJacTimes, 46
IDASetLinearSolver, 30, 35, 68, 165
IDASetLineSearchOffIC, 50
IDASetMaxBacksIC, 50
IDASetMaxConvFails, 44
IDASetMaxErrTestFails, 43
IDASetMaxNonlinIters, 43
IDASetMaxNumItersIC, 50
IDASetMaxNumJacSIC, 49
IDASetMaxNumSteps, 42
IDASetMaxNumStepsIC, 49
IDASetMaxOrd, 42
IDASetMaxStep, 42
IDASetNoInactiveRootWarn, 51
IDASetNonlinConvCoef, 44
IDASetNonlinConvCoefIC, 49
IDASetNonLinearSolver, 36
IDASetPreconditioner, 47, 48
IDASetRootDirection, 51
IDASetStepToleranceIC, 50
IDASetStopTime, 43
IDASetSuppressAlg, 44
IDASetUserData, 41
IDASolve, 30, 38
IDASpilsGetLastFlag, 64
IDASpilsGetNumConvFails, 63
IDASpilsGetNumJtimesEvals, 64
IDASpilsGetNumJTSetupEvals, 64
IDASpilsGetNumLinConvFails, 62
IDASpilsGetNumLinIters, 62
IDASpilsGetNumLinResEvals, 62
IDASpilsGetNumLinSolvSetups, 55
IDASpilsGetNumNonlinSolvConvFails, 59
IDASpilsGetNumNonlinSolvIters, 59
IDASpilsGetWorkspace, 61
IDASpilsJacTimesSetupFn, 71
IDASpilsJacTimesVecFn, 70
INDEX

IDASpilsPrecSetupFn, 73
IDASpilsPrecSolveFn, 72
IDASpilsSetEpsLin, 48
IDASpilsSetIncrementFactor, 47
IDASpilsSetJacTimes, 47
IDASpilsSetLinearSolver, 35
IDASpilsSetPreconditioner, 48
IDASStolerances, 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, 70
Jacobian-vector setup
  user-supplied, 71
sparse
  use in FIDA, 85
user-supplied, 45, 68–70
LS_OFF_IC, 90
MAX_CONVFAIL, 90
MAX_ERRFAIL, 90
MAX_NITERS, 90
MAX_NITERS_IC, 90
MAX_NJIE, 90
MAX_NSTEPS, 90
MAX_NSTEPS_IC, 90
MAX_ORD, 90
MAX_STEP, 90
maxord, 65
memory requirements
  IDA solver, 54
  IDABBDPRE preconditioner, 77
  IDALS linear solver interface, 61
N_VCloneVectorArray, 99
N_VCloneVectorArray_OpenMP, 121
N_VCloneVectorArray_OpenMPDEV, 148
N_VCloneVectorArray_ParHyp, 130
N_VCloneVectorArray_Petsc, 134
N_VCloneVectorArray_Pthreads, 126
N_VCloneVectorArray_Serial, 112
N_VCopyFromDevice_Cuda, 140
N_VCopyFromDevice_OpenMPDEV, 149
N_VCopyToDevice_Cuda, 140
N_VCopyToDevice_OpenMPDEV, 148
N_VCopyToDevice_Raja, 144
N_VEnableConstVectorArray_Cuda, 141
N_VEnableConstVectorArray_ManyVector, 156
N_VEnableConstVectorArray_OpenMP, 123
N_VEnableConstVectorArray_OpenMPDEV, 150
N_VEnableConstVectorArray_ParHyp, 132
N_VEnableConstVectorArray_Petsc, 135
N_VEnableConstVectorArray_Pthreads, 128
N_VEnableConstVectorArray_Raja, 145
N_VEnableDotProdMulti_Cuda, 141
N_VEnableDotProdMulti_ManyVector, 155
N_VEnableDotProdMulti_OpenMP, 123
N_VEnableDotProdMulti_OpenMPDEV, 149
N_VEnableDotProdMulti_ParHyp, 131
N_VEnableDotProdMulti_Petsc, 135
N_VEnableDotProdMulti_Pthreads, 127
N_VEnableDotProdMulti_Serial, 113
N_VEnableFusedOps_Cuda, 140
N_VEnableFusedOps_ManyVector, 155
N_VEnableFusedOps_OpenMP, 122
N_VEnableFusedOps_OpenMPDEV, 149
N_VEnableFusedOps_ParHyp, 131
N_VEnableFusedOps_Petsc, 134
N_VEnableFusedOps_Pthreads, 127
N_VEnableFusedOps_Raja, 145
N_VEnableFusedOps_Serial, 113
N_VEnableLinearCombination_Cuda, 140
N_VEnableLinearCombination_ManyVector, 155
optional input

generic linear solver interface, 45–48
initial condition calculation, 49–51
iterative linear solver, 47–48
matrix-based linear solver, 45–46
matrix-free linear solver, 46–47
rootfinding, 51
solver, 39–45

optional output

band-block-diagonal preconditioner, 77–78
generic linear solver interface, 61–65
initial condition calculation, 60
interpolated solution, 51
solver, 54–59
version, 52–54

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, 73
setup and solve phases, 20
user-supplied, 47–48, 71, 72

RCONST, 26
realtype, 26
reinitialization, 65
residual function, 66
Rootfinding, 17, 30, 37, 92
ROUT, 89, 91

SM_COLS_B, 171
SM_COLS_D, 166
SM_COLUMN_B, 69, 171
SM_COLUMN_D, 69, 167
SM_COLUMN_ELEMENT_B, 69, 171
SM_COLUMNS_B, 171
SM_COLUMNS_D, 166
SM_CONTENT_B, 178
SM_CONTENT_D, 166
SM_CONTENT_S, 178
SM_DATA_B, 171
SM_DATA_D, 166
SM_DATA_S, 178
SM_ELEMENT_B, 69, 171
SM_ELEMENT_D, 69, 167
SM_INDEXPTRS_S, 178
SM_INDEXVALS_S, 178
SM_LBAND_B, 171
SM_LDATA_B, 171
SM_LDATA_D, 166
SM_LDIM_B, 171
SM_NNZ_S, 70, 178
SM_NP_S, 178
SM_ROWS_B, 171
SM_ROWS_D, 166
SM_ROWS_S, 178
SM_SPARSETYPE_S, 178
SM_SUBAND_B, 171
SM_SUB_BAND_B, 171
SM_SUB_BAND_D, 171
SM_SUB_BAND_S, 171
SM_SUBREAL, 26
step size bounds, 42–43
STEP_TOL_IC, 90
STOP_TIME, 90
SUNBandMatrix, 29, 172
SUNBandMatrix_Cols, 172
SUNBandMatrix_Column, 174
SUNBandMatrix_Cols, 173
SUNBandMatrix_Columns, 173
SUNBandMatrix_Data, 174
SUNBandMatrix_LDim, 173
SUNBandMatrix_LowerBandwidth, 173
SUNBandMatrix_Print, 173
SUNBandMatrix_Rows, 173
SUNBandMatrix_StoredUpperBandwidth, 173
SUNBandMatrix_UpperBandwidth, 173
SUNBandMatrixStorage, 172
SUNDenseMatrix, 29, 167
SUNDenseMatrix_Cols, 168
SUNDenseMatrix_Column, 168
SUNDenseMatrix_Columns, 167
SUNDenseMatrix_Data, 168
SUNDenseMatrix_LData, 168
SUNDenseMatrix_Print, 167
SUNDenseMatrix_Rows, 167
sundials/sundials_linear solver.h, 185
sundials_nonlinear solver.h, 27
sundials_nvector.h, 27
sundials_types.h, 26, 27
SUNDIALSGetVersion, 52
SUNDIALSGetVersionNumber, 54
sunindextype, 26
SUNLinearSolver, 185, 192
SUNLinearSolver module, 185
SUNLINEAR_SOLVER_DIRECT, 187, 194
SUNLINEAR_SOLVER_ITERATIVE, 187, 195
SUNLINEAR_SOLVER_MATRIX_ITERATIVE, 187, 195
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_spbcgs.h, 28
sunlinsol/sunlinsol_spfmr.h, 27
sunlinsol/sunlinsol_spfmr.h, 27
sunlinsol/sunlinsol_spfmr.h, 27
sunlinsol/sunlinsol_superlumt.h, 27
SUNLinSol_Band, 35, 200
SUNLinSol_Dense, 35, 198
SUNLinSol_KLU, 35, 208
SUNLinSol_KLUReInit, 209
SUNLinSol_KLUSetOrdering, 211
SUNLinSol_LapackBand, 35, 205
SUNLinSol_LapackDense, 35, 203
SUNLinSol_PCG, 35, 246, 248
SUNLinSol_PCGSetMaxl, 247
SUNLinSol_PCGS, 35, 235, 237
SUNLinSol_SPBCGS, 35, 235, 237
SUNLinSol_SPBCGSSetMaxl, 236
SUNLinSol_SPBCGSSetPrecType, 236
SUNLinSol_SPFGMR, 35, 228, 231
SUNLinSol_SPFGMRS, 35, 228, 231
SUNLinSol_SPFGMRSSetMaxRestarts, 230
SUNLinSol_SPFGMRSSetPrecType, 229
SUNLinSol_SPGMR, 35, 221, 224
SUNLinSol_SPGMRS, 35, 221, 224
SUNLinSol_SPGMRSSetMaxRestarts, 223
SUNLinSol_SPGMRSSetPrecType, 223
SUNLinSol_SPTFQMR, 35, 240, 242
SUNLinSol_SPTFQMRSetMaxl, 241
SUNLinSol_SPTFQMRSetPrecType, 241
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUNLinSol_SPTFQMRSetPrecType</td>
<td>241</td>
</tr>
<tr>
<td>SUNLinSol_SuperLUDIST</td>
<td>214</td>
</tr>
<tr>
<td>SUNLinSol_SuperLUDIST_GetBerr</td>
<td>214</td>
</tr>
<tr>
<td>SUNLinSol_SuperLUDIST_GetGridinfo</td>
<td>215</td>
</tr>
<tr>
<td>SUNLinSol_SuperLUDIST_GetLUstruct</td>
<td>215</td>
</tr>
<tr>
<td>SUNLinSol_SuperLUDIST_GetScalePermstruct</td>
<td>215</td>
</tr>
<tr>
<td>SUNLinSol_SuperLUDIST_GetSOLVEstruct</td>
<td>216</td>
</tr>
<tr>
<td>SUNLinSol_SuperLUDIST_GetSuperLUDoptim</td>
<td>215</td>
</tr>
<tr>
<td>SUNLinSol_SuperLUUMT</td>
<td>35, 217</td>
</tr>
<tr>
<td>SUNLinSol_SuperLUUMT_SetOrdering</td>
<td>219</td>
</tr>
<tr>
<td>SUNLinSol_Free</td>
<td>31, 186, 188</td>
</tr>
<tr>
<td>SUNLinSolGetType</td>
<td>186, 187</td>
</tr>
<tr>
<td>SUNLinSolInitialize</td>
<td>186, 187</td>
</tr>
<tr>
<td>SUNLinSolLastFlag</td>
<td>190</td>
</tr>
<tr>
<td>SUNLinSolNumIters</td>
<td>189</td>
</tr>
<tr>
<td>SUNLinSolResNorm</td>
<td>189</td>
</tr>
<tr>
<td>SUNLinSolSetATimes</td>
<td>187, 188, 195</td>
</tr>
<tr>
<td>SUNLinSolSetPreconditioner</td>
<td>189</td>
</tr>
<tr>
<td>SUNLinSolSetScalingVectors</td>
<td>189</td>
</tr>
<tr>
<td>SUNLinSolSetup</td>
<td>186, 187, 195</td>
</tr>
<tr>
<td>SUNLinSolSolve</td>
<td>186, 188</td>
</tr>
<tr>
<td>SUNLinSolSpace</td>
<td>190</td>
</tr>
<tr>
<td>SUNMatDestroy</td>
<td>31</td>
</tr>
<tr>
<td>SUNMatrix</td>
<td>161</td>
</tr>
<tr>
<td>SUNMatrix module</td>
<td>161</td>
</tr>
<tr>
<td>SUNMatrix_SLUNRloc_OwnData</td>
<td>184</td>
</tr>
<tr>
<td>SUNMatrix_SLUNRloc_Print</td>
<td>183</td>
</tr>
<tr>
<td>SUNMatrix_SLUNRloc_ProcessGrid</td>
<td>183</td>
</tr>
<tr>
<td>SUNMatrix_SLUNRloc_SuperMatrix</td>
<td>183</td>
</tr>
<tr>
<td>SUNNonlinearSolver</td>
<td>27, 253</td>
</tr>
<tr>
<td>SUNNonlinearSolver module</td>
<td>253</td>
</tr>
<tr>
<td>SUNNONLINEARSOLVER_FIXEDPOINT</td>
<td>254</td>
</tr>
<tr>
<td>SUNNONLINEARSOLVER_ROOTFIND</td>
<td>254</td>
</tr>
<tr>
<td>SUNNonlinSol_FixedPoint</td>
<td>266, 269</td>
</tr>
<tr>
<td>SUNNonlinSol_FixedPointSens</td>
<td>267</td>
</tr>
<tr>
<td>SUNNonlinSol_Newton</td>
<td>263</td>
</tr>
<tr>
<td>SUNNonlinSol_NewtonSens</td>
<td>263</td>
</tr>
<tr>
<td>SUNNonlinSol_Free</td>
<td>31, 255</td>
</tr>
<tr>
<td>SUNNonlinSolGetCurIter</td>
<td>257</td>
</tr>
<tr>
<td>SUNNonlinSolGetNumConvFails</td>
<td>257</td>
</tr>
<tr>
<td>SUNNonlinSolGetNumIters</td>
<td>257</td>
</tr>
<tr>
<td>SUNNonlinSolGetSysFn_FixedPoint</td>
<td>267</td>
</tr>
<tr>
<td>SUNNonlinSolGetSysFn_Newton</td>
<td>264</td>
</tr>
<tr>
<td>SUNNonlinSolGetType</td>
<td>254</td>
</tr>
<tr>
<td>SUNNonlinSolInitialize</td>
<td>254</td>
</tr>
<tr>
<td>SUNNonlinSolLSsetup</td>
<td>255</td>
</tr>
<tr>
<td>SUNNonlinSolSetConvTestFn</td>
<td>256</td>
</tr>
<tr>
<td>SUNNonlinSolSetLSolveFn</td>
<td>256</td>
</tr>
<tr>
<td>SUNNonlinSolSetMaxIters</td>
<td>256</td>
</tr>
<tr>
<td>SUNNonlinSolSetSysFn</td>
<td>255</td>
</tr>
<tr>
<td>SUNNonlinSolSetup</td>
<td>254</td>
</tr>
<tr>
<td>SUNNonlinSolSolve</td>
<td>254</td>
</tr>
</tbody>
</table>

**SUPPRESS_ALG**, 90

- tolerances, 14, 33, 34, 67
- **UNIT_ROUNDOFF**, 26
- User main program
  - FIDA usage, 82
  - FIDABBD usage, 93
  - IDA usage, 28
  - IDABBDPRE usage, 75
- **user_data**, 41, 66–68, 74, 75
- weighted root-mean-square norm, 14