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

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

Daniel R. Reynolds
Department of Mathematics
Southern Methodist University

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 ix  
List of Figures xi  

1 Introduction 1  
1.1 Historical Background .................................................. 1  
1.2 Changes from previous versions ...................................... 2  
1.3 Reading this User Guide .............................................. 11  
1.4 SUNDIALS Release License ............................................. 12  
1.4.1 BSD 3-Clause License ................................................. 12  
1.4.2 Additional Notice .................................................... 13  
1.4.3 SUNDIALS Release Numbers ......................................... 13  

2 Mathematical Considerations 15  
2.1 IVP solution ............................................................... 15  
2.2 Preconditioning .......................................................... 19  
2.3 BDF stability limit detection ......................................... 20  
2.4 Rootfinding ............................................................... 21  

3 Code Organization 23  
3.1 SUNDIALS organization .................................................. 23  
3.2 CVODE organization ..................................................... 23  

4 Using CVODE for C Applications 27  
4.1 Access to library and header files .................................... 27  
4.2 Data Types ................................................................. 28  
4.2.1 Floating point types .................................................. 28  
4.2.2 Integer types used for vector and matrix indices ............... 28  
4.3 Header files ............................................................... 29  
4.4 A skeleton of the user’s main program ............................... 30  
4.5 User-callable functions ................................................. 33  
4.5.1 CVODE initialization and deallocation functions ................. 34  
4.5.2 CVODE tolerance specification functions .......................... 35  
4.5.3 Linear solver interface functions .................................. 37  
4.5.4 Nonlinear solver interface function ................................ 38  
4.5.5 Rootfinding initialization function ................................. 39  
4.5.6 CVODE solver function .............................................. 40  
4.5.7 Optional input functions ............................................. 41  
4.5.7.1 Main solver optional input functions ............................ 43  
4.5.7.2 Linear solver interface optional input functions .............. 48  
4.5.7.3 Rootfinding optional input functions ............................ 52  
4.5.8 Interpolated output function ........................................ 52  
4.5.9 Optional output functions ........................................... 53
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5.9.1 SUNDIALS version information</td>
<td>53</td>
</tr>
<tr>
<td>4.5.9.2 Main solver optional output functions</td>
<td>55</td>
</tr>
<tr>
<td>4.5.9.3 Rootfinding optional output functions</td>
<td>62</td>
</tr>
<tr>
<td>4.5.9.4 cvls linear solver interface optional output functions</td>
<td>62</td>
</tr>
<tr>
<td>4.5.9.5 Diagonal linear solver interface optional output functions</td>
<td>67</td>
</tr>
<tr>
<td>4.5.10 CVODE reinitialization function</td>
<td>68</td>
</tr>
<tr>
<td>4.6 User-supplied functions</td>
<td>69</td>
</tr>
<tr>
<td>4.6.1 ODE right-hand side</td>
<td>69</td>
</tr>
<tr>
<td>4.6.2 Error message handler function</td>
<td>70</td>
</tr>
<tr>
<td>4.6.3 Error weight function</td>
<td>70</td>
</tr>
<tr>
<td>4.6.4 Rootfinding function</td>
<td>71</td>
</tr>
<tr>
<td>4.6.5 Jacobian construction (matrix-based linear solvers)</td>
<td>71</td>
</tr>
<tr>
<td>4.6.6 Jacobian-vector product (matrix-free linear solvers)</td>
<td>73</td>
</tr>
<tr>
<td>4.6.7 Jacobian-vector product setup (matrix-free linear solvers)</td>
<td>74</td>
</tr>
<tr>
<td>4.6.8 Preconditioner solve (iterative linear solvers)</td>
<td>74</td>
</tr>
<tr>
<td>4.6.9 Preconditioner setup (iterative linear solvers)</td>
<td>75</td>
</tr>
<tr>
<td>4.7 Preconditioner modules</td>
<td>76</td>
</tr>
<tr>
<td>4.7.1 A serial banded preconditioner module</td>
<td>76</td>
</tr>
<tr>
<td>4.7.2 A parallel band-block-diagonal preconditioner module</td>
<td>79</td>
</tr>
<tr>
<td>5 Using CVODE for FORTRAN Applications</td>
<td>85</td>
</tr>
<tr>
<td>5.1 CVODE FORTRAN 2003 Interface Module</td>
<td>85</td>
</tr>
<tr>
<td>5.1.1 SUNDIALS FORTRAN 2003 Modules</td>
<td>85</td>
</tr>
<tr>
<td>5.1.2 Important note on portability</td>
<td>85</td>
</tr>
<tr>
<td>5.1.3 Data Types</td>
<td>85</td>
</tr>
<tr>
<td>5.2 FCVODE, an Interface Module for FORTRAN Applications</td>
<td>86</td>
</tr>
<tr>
<td>5.2.1 Important note on portability</td>
<td>86</td>
</tr>
<tr>
<td>5.2.2 Fortran Data Types</td>
<td>87</td>
</tr>
<tr>
<td>5.2.3 FCVODE routines</td>
<td>87</td>
</tr>
<tr>
<td>5.2.4 Usage of the FCVODE interface module</td>
<td>89</td>
</tr>
<tr>
<td>5.2.5 FCVODE optional input and output</td>
<td>97</td>
</tr>
<tr>
<td>5.2.6 Usage of the FCVROOT interface to rootfinding</td>
<td>98</td>
</tr>
<tr>
<td>5.2.7 Usage of the FCVBP interface to CVBANDPRE</td>
<td>100</td>
</tr>
<tr>
<td>5.2.8 Usage of the FCVBBD interface to CVBBDPRE</td>
<td>101</td>
</tr>
<tr>
<td>6 Description of the NVECTOR module</td>
<td>105</td>
</tr>
<tr>
<td>6.1 NVeCtor functions used by CVODE</td>
<td>117</td>
</tr>
<tr>
<td>6.2 The NVeCtor_SERIAL implementation</td>
<td>119</td>
</tr>
<tr>
<td>6.2.1 NVeCtor_SERIAL accessor macros</td>
<td>119</td>
</tr>
<tr>
<td>6.2.2 NVeCtor_SERIAL functions</td>
<td>120</td>
</tr>
<tr>
<td>6.2.3 NVeCtor_SERIAL Fortran interfaces</td>
<td>123</td>
</tr>
<tr>
<td>6.3 The NVeCtor_PARALLEL implementation</td>
<td>123</td>
</tr>
<tr>
<td>6.3.1 NVeCtor_PARALLEL accessor macros</td>
<td>123</td>
</tr>
<tr>
<td>6.3.2 NVeCtor_PARALLEL functions</td>
<td>124</td>
</tr>
<tr>
<td>6.3.3 NVeCtor_PARALLEL Fortran interfaces</td>
<td>127</td>
</tr>
<tr>
<td>6.4 The NVeCtor_OPENMP implementation</td>
<td>128</td>
</tr>
<tr>
<td>6.4.1 NVeCtor_OPENMP accessor macros</td>
<td>128</td>
</tr>
<tr>
<td>6.4.2 NVeCtor_OPENMP functions</td>
<td>129</td>
</tr>
<tr>
<td>6.4.3 NVeCtor_OPENMP Fortran interfaces</td>
<td>132</td>
</tr>
<tr>
<td>6.5 The NVeCtor_PTHREADS implementation</td>
<td>132</td>
</tr>
<tr>
<td>6.5.1 NVeCtor_PTHREADS accessor macros</td>
<td>133</td>
</tr>
<tr>
<td>6.5.2 NVeCtor_PTHREADS functions</td>
<td>134</td>
</tr>
<tr>
<td>6.5.3 NVeCtor_PTHREADS Fortran interfaces</td>
<td>137</td>
</tr>
<tr>
<td>6.6 The NVeCtor_PARHYP implementation</td>
<td>137</td>
</tr>
</tbody>
</table>
6.6.1 NVECTOR_PARHYP functions ............................................................... 138
6.7 The NVECTOR_PETSC implementation .......................................................... 141
  6.7.1 NVECTOR_PETSC functions ................................................................. 141
6.8 The NVECTOR_CUDA implementation .............................................................. 144
  6.8.1 NVECTOR_CUDA functions ................................................................. 145
6.9 The NVECTOR_RAJA implementation ............................................................ 150
  6.9.1 NVECTOR_RAJA functions ................................................................. 151
6.10 The NVECTOR_OPENMPDEV implementation .................................................. 154
  6.10.1 NVECTOR_OPENMPDEV accessor macros ................................................. 154
  6.10.2 NVECTOR_OPENMPDEV functions .......................................................... 155
6.11 The NVECTOR_TRILINOS implementation .................................................... 159
  6.11.1 NVECTOR_TRILINOS functions ........................................................... 159
6.12 The NVECTOR_MANYVECTOR implementation .............................................. 160
  6.12.1 NVECTOR_MANYVECTOR structure ....................................................... 161
  6.12.2 NVECTOR_MANYVECTOR functions ....................................................... 161
6.13 NVECTOR Examples .................................................................................... 164

7 Description of the SUNMatrix module ......................................................... 169
  7.1 The SUNMatrix API ................................................................................. 169
    7.1.1 SUNMatrix functions ....................................................................... 170
    7.1.2 SUNMatrix error codes ..................................................................... 171
  7.2 Compatibility of SUNMatrix modules ....................................................... 172
  7.3 Implementing a custom SUNMatrix .......................................................... 172
  7.4 SUNMatrix functions used by CVODE ..................................................... 173
  7.5 The SUNMatrix_Dense implementation .................................................... 173
    7.5.1 SUNMatrix_Dense accessor macros ..................................................... 174
    7.5.2 SUNMatrix_Dense functions .............................................................. 175
    7.5.3 SUNMatrix_Dense Fortran interfaces .................................................. 176
  7.6 The SUNMatrix_Band implementation ...................................................... 177
    7.6.1 SUNMatrix_Band accessor macros ..................................................... 179
    7.6.2 SUNMatrix_Band functions .............................................................. 180
    7.6.3 SUNMatrix_Band Fortran interfaces .................................................... 182
  7.7 The SUNMatrix_Sparse implementation ................................................... 183
    7.7.1 SUNMatrix_Sparse accessor macros .................................................... 186
    7.7.2 SUNMatrix_Sparse functions .............................................................. 187
    7.7.3 SUNMatrix_Sparse Fortran interfaces .................................................. 189
  7.8 The SUNMatrix_SLUNRloc implementation ............................................. 190
    7.8.1 SUNMatrix_SLUNRloc functions ........................................................ 191

8 Description of the SUNLinearSolver module ................................................. 193
  8.1 The SUNLinearSolver API ....................................................................... 194
    8.1.1 SUNLinearSolver core functions ....................................................... 194
    8.1.2 SUNLinearSolver set functions .......................................................... 196
    8.1.3 SUNLinearSolver get functions .......................................................... 197
    8.1.4 Functions provided by SUNDIALS packages ......................................... 198
    8.1.5 SUNLinearSolver return codes ........................................................... 199
    8.1.6 The generic SUNLinearSolver module .............................................. 200
  8.2 Compatibility of SUNLinearSolver modules ............................................. 201
  8.3 Implementing a custom SUNLinearSolver module ..................................... 202
    8.3.1 Intended use cases ............................................................................ 202
  8.4 CVODE SUNLinearSolver interface ......................................................... 203
    8.4.1 Lagged matrix information .............................................................. 204
    8.4.2 Iterative linear solver tolerance ....................................................... 205
  8.5 The SUNLinearSolver_Dense implementation ........................................... 206
# List of Tables

4.1 SUNDIALS linear solver interfaces and vector implementations that can be used for each. 33
4.2 Optional inputs for CVODE and CVLS 42
4.3 Optional outputs from CVODE, CVLS, and CVDIAG 54

5.1 Summary of FORTRAN 2003 interfaces for shared SUNDIALS modules. 86
5.2 Keys for setting FCVODE optional inputs 98
5.3 Description of the FCVODE optional output arrays IOUT and ROUT 99

6.1 Vector Identifications associated with vector kernels supplied with SUNDIALS. 108
6.2 Description of the NVECTOR operations 108
6.3 Description of the NVECTOR fused operations 111
6.4 Description of the NVECTOR vector array operations 112
6.5 Description of the NVECTOR local reduction operations 115
6.6 List of vector functions usage by CVODE code modules 118

7.1 Identifiers associated with matrix kernels supplied with SUNDIALS. 170
7.2 Description of the SUNMatrix operations 170
7.3 Description of the SUNMatrix error codes 172
7.4 SUNDIALS matrix interfaces and vector implementations that can be used for each. 172
7.5 List of matrix functions usage by CVODE code modules 173

8.1 Description of the SUNLinearSolver error codes 200
8.2 SUNDIALS matrix-based linear solvers and matrix implementations that can be used for each. 201
8.3 List of linear solver function usage in the CVLS interface 204

9.1 Description of the SUNNonlinearSolver return codes 267

A.1 SUNDIALS libraries and header files 295
List of Figures

3.1 High-level diagram of the SUNDIALS suite ........................................ 24
3.2 Organization of the SUNDIALS suite .................................................. 25
3.3 Overall structure diagram of the CVODE package ................................. 26

7.1 Diagram of the storage for a SUNMATRIX_BAND object ....................... 178
7.2 Diagram of the storage for a compressed-sparse-column matrix .............. 185

A.1 Initial ccmake configuration screen .................................................. 281
A.2 Changing the instdir ............................................................................. 282
Chapter 1

Introduction

CVODE is part of a software family called SUNDIALS: SUite of Nonlinear and Differential/Algebraic equation Solvers [27]. This suite consists of CVODE, ARKODE, KINSOL, and IDA, and variants of these with sensitivity analysis capabilities.

1.1 Historical Background

Fortran solvers for ODE initial value problems are widespread and heavily used. Two solvers that have been written at LLNL in the past are VODE [6] and VODPK [9]. VODE is a general purpose solver that includes methods for both stiff and nonstiff systems, and in the stiff case uses direct methods (full or banded) for the solution of the linear systems that arise at each implicit step. Externally, VODE is very similar to the well known solver LSODE [37]. VODPK is a variant of VODE that uses a preconditioned Krylov (iterative) method, namely GMRES, for the solution of the linear systems. VODPK is a powerful tool for large stiff systems 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]. The capabilities of both VODE and VODPK have been combined in the C-language package CVODE [14].

At present, CVODE 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) [40], FGMRES (Flexible Generalized Minimum RESidual) [39], Bi-CGSTab (Bi-Conjugate Gradient Stabilized) [41], TFQMR (Transpose-Free Quasi-Minimal Residual) [20], and PCG (Preconditioned Conjugate Gradient) [22] linear iterative methods. As Krylov methods, these require almost no 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 stiff ODE systems, the Krylov methods are preferable over direct linear solver methods, and are often the only feasible choice. Among the Krylov methods in SUNDIALS, we recommend GMRES as the best overall choice. However, users are encouraged to compare all options, especially if encountering convergence failures with GMRES. Bi-CGSTab and TFQMR have an advantage in storage requirements, in that the number of workspace vectors they require is fixed, while that number for GMRES depends on the desired Krylov subspace size. FGMRES has an advantage in that it is designed to support preconditioners that vary between iterations (e.g. iterative methods). PCG exhibits rapid convergence and minimal workspace vectors, but only works for symmetric linear systems.

In the process of translating the VODE and VODPK algorithms into C, the overall CVODE organization has been changed considerably. One key feature of the CVODE organization is that the linear system solvers comprise a layer of code modules that is separated from the integration algorithm, allowing for easy modification and expansion of the linear solver array. A second key feature is a separate module devoted to vector operations; this facilitated the extension to multiprocessor environments with minimal impacts on the rest of the solver, resulting in PVODE [12], the parallel variant of CVODE.
Around 2002, the functionality of CVODE and PVODE were combined into one single code, simply called CVODE. Development of this version of CVODE was concurrent with a redesign of the vector operations module across the SUNDIALS suite. The key feature of the NVECTOR module is that it is written in terms of abstract vector operations with the actual vector kernels attached by a particular implementation (such as serial or parallel) of NVECTOR. This allows writing the SUNDIALS solvers in a manner independent of the actual NVECTOR implementation (which can be user-supplied), as well as allowing more than one NVECTOR module linked into an executable file. SUNDIALS (and thus CVODE) is supplied with six different NVECTOR implementations: serial, MPI-parallel, and both OpenMP and Pthreads thread-parallel NVECTOR implementations, a Hypre parallel implementation, and a PETSc implementation.

There are several motivations for choosing the C language for CVODE. 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 CVODE 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.

1.2 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, \texttt{N_VGetCommunicator} and \texttt{N_VGetLength}, 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 \texttt{N_VDotProdLocal}, \texttt{N_VMaxNormLocal}, \texttt{N_VMinLocal}, \texttt{N_VL1NormLocal}, \texttt{N_VWSqrSumLocal}, \texttt{N_VWSqrSumMaskLocal}, \texttt{N_VInvTestLocal}, \texttt{N_VConstrMaskLocal}, and \texttt{N_VMinQuotientLocal}. 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, \texttt{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, \texttt{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 \texttt{EXAMPLES_ENABLE_RAJA} CMake option has been removed. The option \texttt{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 cvode_impl.h is no longer installed. This means users who are directly manipulating the CvodeMem structure will need to update their code to use CVODE’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 cvode library, libsundials_cvode.

Changes in v4.0.1

No changes were made in this release.

Changes in v4.0.0

CVODE’s previous direct and iterative linear solver interfaces, CVDLS and CVSPILS, have been merged into a single unified linear solver interface, CVLS, to support any valid SUNLINLSOL module. This includes the “DIRECT” and “ITERATIVE” types as well as the new “MATRIX_ITERATIVE” type. Details regarding how CVLS utilizes linear solvers of each type as well as discussion regarding intended use cases for user-supplied SUNLINLSOL implementations are included in Chapter 8. All CVODE example programs and the standalone linear solver examples have been updated to use the unified linear solver interface.

The unified interface for the new CVLS module is very similar to the previous CVDLS and CVSPILS 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 SUNLINLSOL 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_PCG, SUNLinSol_SPBCGS, SUNLinSol_SPGMR, 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 CVODE 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 SUNNONLINLSOL API. This API will ease the addition of new nonlinear solver options and allow for external or user-supplied nonlinear solvers. The SUNNONLINLSOL 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 SUNLINLSOL modules. Currently two SUNNONLINLSOL implementations are provided, SUNNONLINLSOL_NEWTON and SUNNONLINLSOL_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 SUNNONLINLSOL_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 CVODE example programs have been updated to use generic SUNNONLINLSOL modules.
With the introduction of SUNNONLINSOL modules, the input parameter \texttt{iter} to \texttt{CVodeCreate} has been removed along with the function \texttt{CVodeSetIterType} and the constants \texttt{CV\_NEWTON} and \texttt{CV\_FUNCTIONAL}. Similarly, the \texttt{ITMETH} parameter has been removed from the Fortran interface function \texttt{FCVMALLOC}. Instead of specifying the nonlinear iteration type when creating the \texttt{CVODE} memory structure, \texttt{CVODE} uses the SUNNONLINSOL\_NEWTON module implementation of a Newton iteration by default. For details on using a non-default or user-supplied nonlinear solver see Chapter 4. \texttt{CVODE} functions for setting the nonlinear solver options (e.g., \texttt{CVodeSetMaxNonlinIters}) or getting nonlinear solver statistics (e.g., \texttt{CVodeGetNumNonlinSolvIters}) remain unchanged and internally call generic SUNNONLINSOL functions as needed.

Three fused vector operations and seven vector array operations have been added to the \texttt{NVECTOR} API. These \textit{optional} operations are disabled by default and may be activated by calling vector specific routines after creating an \texttt{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 \texttt{N\_VLinearCombination}, \texttt{N\_VScaleAddMulti}, and \texttt{N\_VDotProdMulti} and the vector array operations are \texttt{N\_VLinearCombinationVectorArray}, \texttt{N\_VScaleVectorArray}, \texttt{N\_VConstVectorArray}, \texttt{N\_VWrmsNormVectorArray}, \texttt{N\_VWrmsNormMaskVectorArray}, \texttt{N\_VScaleAddMultiVectorArray}, and \texttt{N\_VLinearCombinationVectorArray}. If an \texttt{NVECTOR} implementation defines any of these operations as \texttt{NULL}, then standard \texttt{NVECTOR} operations will automatically be called as necessary to complete the computation.

Multiple updates to \texttt{NVECTOR\_CUDA} were made:

- Changed \texttt{N\_VGetLength\_Cuda} to return the global vector length instead of the local vector length.
- Added \texttt{N\_VGetLocalLength\_Cuda} to return the local vector length.
- Added \texttt{N\_VGetMPIComm\_Cuda} to return the MPI communicator used.
- Removed the accessor functions in the namespace suncudavec.
- Changed the \texttt{N\_VMake\_Cuda} function to take a host data pointer and a device data pointer instead of an \texttt{N\_VectorContent\_Cuda} object.
- Added the ability to set the \texttt{cudaStream\_t} used for execution of the \texttt{NVECTOR\_CUDA} kernels. See the function \texttt{N\_VSetCudaStreams\_Cuda}.
- Added \texttt{N\_VNewManaged\_Cuda}, \texttt{N\_VMakeManaged\_Cuda}, and \texttt{N\_VIsManagedMemory\_Cuda} functions to accommodate using managed memory with the \texttt{NVECTOR\_CUDA}.

Multiple changes to \texttt{NVECTOR\_RAJA} were made:

- Changed \texttt{N\_VGetLength\_Raja} to return the global vector length instead of the local vector length.
- Added \texttt{N\_VGetLocalLength\_Raja} to return the local vector length.
- Added \texttt{N\_VGetMPIComm\_Raja} to return the MPI communicator used.
- Removed the accessor functions in the namespace suncudavec.

A new \texttt{NVECTOR} implementation for leveraging OpenMP 4.5+ device offloading has been added, \texttt{NVECTOR\_OPENMPDEV}. See §6.10 for more details.

Two changes were made in the \texttt{CVODE/CVODES/ARKODE} initial step size algorithm:

1. Fixed an efficiency bug where an extra call to the right hand side function was made.
2. Changed the behavior of the algorithm if the max-iterations case is hit. Before the algorithm would exit with the step size calculated on the penultimate iteration. Now it will exit with the step size calculated on the final iteration.
A Fortran 2003 interface to cvode has been added along with Fortran 2003 interfaces to the following shared Sundials modules:

- SUNNONLINSOL_FIXEDPOINT and SUNNONLINSOL_NEWTON nonlinear solver modules
- SUNLINSOL_DENSE, SUNLINSOL_BAND, SUNLINSOL_KLU, SUNLINSOL_PCG, SUNLINSOL_SPBCGS, SUNLINSOL_SPFGMR, SUNLINSOL_SPGMR, and SUNLINSOL_SPTFQMR linear solver modules
- NVECTOR_SERIAL, NVECTOR_PTHREADS, and NVECTOR_OPENMP vector modules

**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**

Support for optional inequality constraints on individual components of the solution vector has been added to cvode and cvodes. See Chapter 2 and the description of CVodeSetConstraints in § 4.5.7.1 for more details. Use of CVodeSetConstraints requires the NVECTOR operations N_MinQuotient, N_VConstrMask, and N_VCompare that were not previously required by cvode and cvodes.

Fixed a problem with setting sunindextype which would occur with some compilers (e.g. arm-clang) 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_nvecudaraja.lib from libsundials_nvecraja.lib to better reflect that we only support CUDA as a backend for RAJA currently.

Several changes were made to the build system:

- CMake 3.1.3 is now the minimum required CMake version.
- Deprecate the behavior of the SUNDIALS_INDEX_TYPE CMake option and added the SUNDIALS_INDEX_SIZE CMake option to select the sunindextype integer size.
- The native CMake FindMPI module is now used to locate an MPI installation.
- If MPI is enabled and MPI compiler wrappers are not set, the build system will check if CMAKE_<language>_COMPILER can compile MPI programs before trying to locate and use an MPI installation.
- The previous options for setting MPI compiler wrappers and the executable for running MPI programs have been 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).

• Added the following examples from the usage notes page of the SUNDIALS website, and updated them to work with SUNDIALS 3.x:
  
  – `cvDisc_dns.c`, which demonstrates using CVODE with discontinuous solutions or RHS.
  – `cvRoberts_dns_negsol.c`, which illustrates the use of the RHS function return value to control unphysical negative concentrations.

• 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 minor bug in the cvSLdet routine, where a return was missing in the error check for three inconsistent roots.

• 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.
1.2 Changes from previous versions

- Fixed an indexing bug in the CUDA NVVECTOR implementation of NV_WrmsNormMask and revised the RAJA NVVECTOR implementation of NV_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 or SUNLINSOL module (e.g., iterative linear solvers or fixed-point iteration).

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 NVVECTOR print functions that write vector data to a specified file (e.g., NVPrintFile_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 ease in interfacing 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: dense, banded, LAPACK dense, LAPACK band, KLU, SuperLUMT, SPGMR, SPBCGS, SPTFQMR, SPFGMR, 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 to utilize 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 NVVECTOR 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 website, 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 `booleantype` 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.

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

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.

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 made in `CVodeFree` to call `lfree` unconditionally (if non-NULL).

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.

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

In `fcvode`, corrections were made to three Fortran interface functions. Missing Fortran interface routines were added so that users can supply the sparse Jacobian routine when using sparse direct solvers.

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 and for use of sparse direct solvers from Fortran.
Minor corrections and additions were made to the CVODE 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 CVODE 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 CVODE.

Otherwise, only relatively minor modifications were made to the CVODE solver:

In cvRootfind, 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 CVLapackBand, the line \( \text{smu} = \text{MIN}(N-1,mu+ml) \) was changed to \( \text{smu} = \text{mu} + \text{ml} \) to correct an illegal input error for \( \text{DGBTRF}/\text{DGBTRS} \).

In order to eliminate or minimize the differences between the sources for private functions in CVODE and CVODES, the names of 48 private functions were changed from \( \text{CV}** \) to \( \text{cv}** \), and a few other names were also changed.

Two minor bugs were fixed regarding the testing of input on the first call to CVode — one involving tstop and one involving the initialization of *tret.

In order to avoid possible name conflicts, the mathematical macro and function names \( \text{MIN}, \text{MAX}, \text{SQR}, \text{RAbs}, \text{RSqrt}, \text{RExp}, \text{RPowerI}, \text{and RPowerR} \) were changed to \( \text{SUNMIN}, \text{SUNMAX}, \text{SUNSQR}, \text{SUNRabs}, \text{SUNRsqrt}, \text{SUNRExp}, \text{SRpowerI}, \text{and SUNRpowerR} \), respectively. These names occur in both the solver and in various example programs.

The example program cvAdvDiff.diag.p was added to illustrate the use of CVDiag in parallel.

In the FCVODE optional input routines FCVSETIIIN and FCVSETRIN, the optional fourth argument key_length was removed, with hardcoded key string lengths passed to all strncmp tests.

In all CVODE examples, integer declarations were revised so that those which must match a C type \text{long int} are declared \text{INTEGER*8}, and a comment was added about the type match. All other integer declarations are just \text{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 \text{NVECTOR_OPENMP}, and one for Pthreads, denoted \text{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 \text{int} to type \text{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 \text{int} and \text{long int} arrays, respectively.

A large number of minor errors have been fixed. Among these are the following: In CVSetTqBDF, the logic was changed to avoid a divide by zero. After the solver memory is created, it is set to zero before being filled. 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. In the rootfinding functions CVRcheck1/CVRcheck2, when an exact zero is found, the array glo of g values at the left endpoint is adjusted, instead of shifting the t location tlo slightly. In the installation files, we modified the treatment of the macro SUNDIALS_USE_GENERIC_MATH, so that the parameter GENERIC_MATH_LIB is either defined (with no value) or not defined.
Changes in v2.6.0

Two new features were added in this release: (a) a new linear solver module, based on BLAS and LAPACK for both dense and banded matrices, and (b) an 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 existing 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; and (c) a general streamlining of the preconditioner modules distributed with the solver.

Changes in v2.5.0

The main changes in this release involve a rearrangement of the entire SUNDAILS source tree (see §3.1). At the user interface level, the main impact is in the mechanism of including SUNDAILS 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.

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

CVSPBCG and CVSPTFQMR modules have been added to interface with the Scaled Preconditioned Bi-CGstab (spbcgs) and Scaled Preconditioned Transpose-Free Quasi-Minimal Residual (spftfqmr) linear solver modules, respectively (for details see Chapter 4). Corresponding additions were made to the FORTRAN interface module fcvode. 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 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 (cvode_ 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 SUNDAILS solvers release user data right after its use. The build systems has been further improved to make it more robust.

Changes in v2.2.1

The changes in this minor SUNDAILS 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 SUNDAILS 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, \texttt{cvode} now provides a set of routines (with prefix \texttt{CVodeSet}) to change the default values for various quantities controlling the solver and a set of extraction routines (with prefix \texttt{CVodeGet}) 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 §4.5.7 and §4.5.9.

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.

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

Installation of \texttt{cvode} (and all of \texttt{sundials}) has been completely redesigned and is now based on configure scripts.

1.3 Reading this User Guide

This user guide is a combination of general usage instructions. Specific example programs are provided as a separate document. We expect that some readers will want to concentrate on the general instructions, while others will refer mostly to the examples, and the organization is intended to accommodate both styles.

There are different possible levels of usage of \texttt{cvode}. The most casual user, with a small IVP problem only, can get by with reading §2.1, then Chapter 4 through §4.5.6 only, and looking at examples in [29].

In a different direction, a more expert user with an IVP problem may want to (a) use a package preconditioner (§4.7), (b) supply his/her own Jacobian or preconditioner routines (§4.6), (c) do multiple runs of problems of the same size (§4.5.10), (d) supply a new \texttt{nvector} module (Chapter 6), (e) supply new \texttt{sunlinsol} and/or \texttt{sunmatrix} modules (Chapters 7 and 8), or even (f) supply new \texttt{sunnonlinsol} modules (Chapter 9).

The structure of this document is as follows:

- In Chapter 2, we give short descriptions of the numerical methods implemented by \texttt{cvode} for the solution of initial value problems for systems of ODEs, and continue with short descriptions of preconditioning (§2.2), stability limit detection (§2.3), and rootfinding (§2.4).

- The following chapter describes the structure of the \texttt{sundials} suite of solvers (§3.1) and the software organization of the \texttt{cvode} solver (§3.2).

- Chapter 4 is the main usage document for \texttt{cvode} for C applications. It includes a complete description of the user interface for the integration of ODE initial value problems.

- In Chapter 5, we describe the use of \texttt{cvode} with \texttt{fortran} applications.

- Chapter 6 gives a brief overview of the generic \texttt{nvector} module shared among the various components of \texttt{sundials}, and details on the \texttt{nvector} implementations provided with \texttt{sundials}.

- Chapter 7 gives a brief overview of the generic \texttt{sunmatrix} module shared among the various components of \texttt{sundials}, and details on the \texttt{sunmatrix} implementations provided with \texttt{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 \texttt{sunlinsol} module shared among the various components of \texttt{sundials}. This chapter contains details on the \texttt{sunlinsol} implementations provided with \texttt{sundials}. The chapter also contains details on the \texttt{sunlinsol} implementations provided with \texttt{sundials} that interface with external linear solver libraries.

- Chapter 9 describes the \texttt{sunnonlinsol} API and nonlinear solver implementations shared among the various components of \texttt{sundials}.
Finally, in the appendices, we provide detailed instructions for the installation of CVODE, within the structure of Sundials (Appendix A), as well as a list of all the constants used for input to and output from CVODE functions (Appendix B).

Finally, the reader should be aware of the following notational conventions in this user guide: program listings and identifiers (such as CVodeInit) within textual explanations appear in typewriter type style; fields in C structures (such as content) appear in italics; and packages or modules, such as CVLS, 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 CVODE and PVODE codes and their user guides by Scott D. Cohen [13] and George D. Byrne [11].

1.4 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.

1.4.1 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.
1.4.2 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.

1.4.3 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

cvode solves ODE initial value problems (IVPs) in real $N$-space, which we write in the abstract form

$$\dot{y} = f(t, y), \quad y(t_0) = y_0,$$  \hspace{1cm} (2.1)

where $y \in \mathbb{R}^N$. Here we use $\dot{y}$ to denote $dy/dt$. While we use $t$ to denote the independent variable, and usually this is time, it certainly need not be. cvode solves both stiff and nonstiff systems. Roughly speaking, stiffness is characterized by the presence of at least one rapidly damped mode, whose time constant is small compared to the time scale of the solution itself.

2.1 IVP solution

The methods used in cvode are variable-order, variable-step multistep methods, based on formulas of the form

$$\sum_{i=0}^{K_1} \alpha_{n,i} y_n^{n-i} + h_n \sum_{i=0}^{K_2} \beta_{n,i} \dot{y}_n^{n-i} = 0.$$  \hspace{1cm} (2.2)

Here the $y^n$ are computed approximations to $y(t_n)$, and $h_n = t_n - t_{n-1}$ is the step size. The user of cvode must choose appropriately one of two multistep methods. For nonstiff problems, cvode includes the Adams-Moulton formulas, characterized by $K_1 = 1$ and $K_2 = q$ above, where the order $q$ varies between 1 and 12. For stiff problems, cvode includes the Backward Differentiation Formulas (BDF) in so-called fixed-leading coefficient (FLC) form, given by $K_1 = q$ and $K_2 = 0$, with order $q$ varying between 1 and 5. The coefficients are uniquely determined by the method type, its order, the recent history of the step sizes, and the normalization $\alpha_{n,0} = -1$. See [10] and [31].

For either choice of formula, a nonlinear system must be solved (approximately) at each integration step. This nonlinear system can be formulated as either a rootfinding problem

$$F(y^n) \equiv y^n - h_n \beta_{n,0} f(t_n, y^n) - a_n = 0,$$  \hspace{1cm} (2.3)

or as a fixed-point problem

$$G(y^n) \equiv h_n \beta_{n,0} f(t_n, y^n) + a_n = y^n.$$  \hspace{1cm} (2.4)

where $a_n \equiv \sum_{i>0} (\alpha_{n,i} y^{n-i} + h_n \beta_{n,i} \dot{y}^{n-i})$. cvode provides several nonlinear solver choices as well as the option of using a user-defined nonlinear solver (see Chapter 9). By default cvode solves (2.3) with a Newton iteration which requires the solution of linear systems

$$M[y^{n(m+1)} - y^{n(m)}] = -F(y^{n(m)}),$$  \hspace{1cm} (2.5)

in which

$$M \approx I - \gamma J, \quad J = \partial f / \partial y, \quad \text{and} \quad \gamma = h_n \beta_{n,0}.$$  \hspace{1cm} (2.6)

The exact variation of the Newton iteration depends on the choice of linear solver and is discussed below and in §9.2. For nonstiff systems, a fixed-point iteration (previously referred to as a functional
iteration in this guide) solving (2.4) is also available. This involves evaluations of \( f \) only and can (optionally) use Anderson’s method [4, 42, 19, 36] to accelerate convergence (see §9.3 for more details). For any nonlinear solver, the initial guess for the iteration is a predicted value \( y^{(0)} \) computed explicitly from the available history data.

For nonlinear solvers that require the solution of the linear system (2.5) (e.g., the default Newton iteration), _CVODE_ provides several linear solver choices, including the option of a user-supplied linear solver module (see Chapter 8). The linear solver modules distributed with _SUNDIALS_ are organized in two families, a _direct_ family comprising direct linear solvers for dense, banded, or sparse matrices, and a _spils_ family comprising scaled preconditioned iterative (Krylov) linear solvers. The methods offered through these modules are as follows:

- dense direct solvers, using either an internal implementation or a BLAS/LAPACK implementation (serial or threaded vector modules only),
- band direct solvers, using either an internal implementation or a BLAS/LAPACK implementation (serial or threaded vector modules only),
- sparse direct solver interfaces, using either the KLU sparse solver library [15, 1], or the thread-enabled SuperLU_MT sparse solver library [33, 17, 3] (serial or threaded vector modules only) [Note that users will need to download and install the KLU or SUPERLUMT packages independent of _CVODE_],
- **SPGMR**, a scaled preconditioned GMRES (Generalized Minimal Residual method) solver,
- **SPFGMR**, a scaled preconditioned FGMRES (Flexible Generalized Minimal Residual method) solver,
- **SPBCGS**, a scaled preconditioned Bi-CGStab (Bi-Conjugate Gradient Stable method) solver,
- **SPTFQMR**, a scaled preconditioned TFQMR (Transpose-Free Quasi-Minimal Residual method) solver, or
- **PCG**, a scaled preconditioned CG (Conjugate Gradient method) solver.

For large stiff systems, where direct methods are often 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].

In addition, _CVODE_ also provides a linear solver module which only uses a diagonal approximation of the Jacobian matrix.

Note that the dense, band, and sparse direct linear solvers can only be used with the serial and threaded vector representations. The diagonal solver can be used with any vector representation.

In the process of controlling errors at various levels, _CVODE_ 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

\[
W_i = \frac{1}{|\text{rtol} \cdot |y_i| + |\text{atol}_i|}.
\]  

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 iteration matrix \( M \) is fixed throughout the nonlinear iterations. However, in the case that a matrix-free iterative linear solver is used, the default Newton iteration is an Inexact Newton iteration, in which \( M \) is applied in a matrix-free manner, with matrix-vector products \( \mathbf{Jv} \) obtained by either difference quotients or a user-supplied routine. With the default Newton iteration, the matrix \( M \) 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,
• more than 20 steps have been taken since the last update,
• the value $\bar{\gamma}$ of $\gamma$ at the last update satisfies $|\gamma / \bar{\gamma} - 1| > 0.3$,
• a non-fatal convergence failure just occurred, or
• an error test failure just occurred.

When forced by a convergence failure, an update of $M$ or $P$ may or may not involve a reevaluation of $J$ (in $M$) or of Jacobian data (in $P$), depending on whether Jacobian error was the likely cause of the failure. More generally, the decision is made to reevaluate $J$ (or instruct the user to reevaluate Jacobian data in $P$) when:

• starting the problem,
• more than 50 steps have been taken since the last evaluation,
• a convergence failure occurred with an outdated matrix, and the value $\bar{\gamma}$ of $\gamma$ at the last update satisfies $|\gamma / \bar{\gamma} - 1| < 0.2$, or
• a convergence failure occurred that forced a step size reduction.

The default stopping test for nonlinear solver iterations is related to the subsequent local error test, with the goal of keeping the nonlinear iteration errors from interfering with local error control. As described below, the final computed value $y^{n(m)}$ will have to satisfy a local error test $\|y^{n(m)} - y^{n(0)}\| \leq \epsilon$. Letting $y^n$ denote the exact solution of (2.3), we want to ensure that the iteration error $y^n - y^{n(m)}$ is small relative to $\epsilon$, specifically that it is less than $0.1\epsilon$. (The safety factor 0.1 can be changed by the user.) For this, we also estimate the linear convergence rate constant $R$ as follows. We initialize $R$ to 1, and reset $R = 1$ when $M$ or $P$ is updated. After computing a correction $\delta_m = y^{n(m)} - y^{n(m-1)}$, we update $R$ if $m > 1$ as

$$R \leftarrow \max\{0.3R, \|\delta_m\|/\|\delta_{m-1}\|\}.$$  

Now we use the estimate

$$\|y^n - y^{n(m)}\| \approx \|y^{n(m+1)} - y^{n(m)}\| \approx R||y^{n(m)} - y^{n(m-1)}|| = R\|\delta_m\|.$$  

Therefore the convergence (stopping) test is

$$R\|\delta_m\| < 0.1\epsilon.$$  

We allow at most 3 iterations (but this limit can be changed by the user). We also declare the iteration diverged if any $\|\delta_m\|/\|\delta_{m-1}\| > 2$ with $m > 1$. If convergence fails with $J$ or $P$ current, we are forced to reduce the step size, and we replace $h_n$ by $h_n/4$. The integration is halted after a preset number of convergence failures; the default value of this limit is 10, but this can be changed by the user.

When an iterative method is used to solve the linear system, its errors must also be controlled, and this also involves the local error test constant. The linear iteration error in the solution vector $\delta_m$ is approximated by the preconditioned residual vector. Thus to ensure (or attempt to ensure) that the linear iteration errors do not interfere with the nonlinear error and local integration error controls, we require that the norm of the preconditioned residual be less than $0.05 \cdot (0.1\epsilon)$.

When the Jacobian is stored using either dense or band SUNMATRIX objects, the Jacobian may be supplied by a user routine, or approximated by difference quotients, at the user’s option. In the latter case, we use the usual approximation

$$J_{ij} = [f_i(t, y + \sigma_j e_j) - f_i(t, y)]/\sigma_j.$$  

The increments $\sigma_j$ are given by

$$\sigma_j = \max \left\{ \sqrt{U} |y_j|, \sigma_0/W_j \right\}.$$
where $U$ is the unit roundoff, $\sigma_0$ is a dimensionless value, and $W_j$ is the error weight defined in (2.7). In the dense case, this scheme requires $N$ evaluations of $f$, one for each column of $J$. In the band case, the columns of $J$ are computed in groups, by the Curtis-Powell-Reid algorithm, with the number of $f$ evaluations equal to the bandwidth.

We note that with sparse and user-supplied sunmatrix objects, the Jacobian must be supplied by a user routine.

In the case of a Krylov method, preconditioning may be used on the left, on the right, or both, with user-supplied routines for the preconditioning setup and solve operations, and optionally also for the required matrix-vector products $Jv$. If a routine for $Jv$ is not supplied, these products are computed as

$$Jv = [f(t, y + \sigma v) - f(t, y)]/\sigma.$$  \hfill (2.8)

The increment $\sigma$ is $1/\|v\|$, so that $\sigma v$ has norm 1.

A critical part of cvode — making it an ODE “solver” rather than just an ODE method, is its control of local error. At every step, the local error is estimated and required to satisfy tolerance conditions, and the step is redone with reduced step size whenever that error test fails. As with any linear multistep method, the local truncation error LTE, at order $q$ and step size $h$, satisfies an asymptotic relation

$$\text{LTE} = Ch^{q+1}y^{q+1} + O(h^{q+2})$$

for some constant $C$, under mild assumptions on the step sizes. A similar relation holds for the error in the predictor $y^{n(0)}$. These are combined to get a relation

$$\text{LTE} = C'[y^n - y^{n(0)}] + O(h^{q+2}).$$

The local error test is simply $\|\text{LTE}\| \leq 1$. Using the above, it is performed on the predictor-corrector difference $\Delta_n \equiv y^{n(m)} - y^{n(0)}$ (with $y^{n(m)}$ the final iterate computed), and takes the form

$$\|\Delta_n\| \leq \epsilon \equiv 1/\|C'\|.$$  \hfill (2.9)

If this test passes, the step is considered successful. If it fails, the step is rejected and a new step size $h'$ is computed based on the asymptotic behavior of the local error, namely by the equation

$$(h'/h)^{q+1}\|\Delta_n\| = \epsilon/6.$$  \hfill (2.10)

Here $1/6$ is a safety factor. A new attempt at the step is made, and the error test repeated. If it fails three times, the order $q$ is reset to 1 (if $q > 1$), or the step is restarted from scratch (if $q = 1$). The ratio $h'/h$ is limited above to 0.2 after two error test failures, and limited below to 0.1 after three. After seven failures, cvode returns to the user with a give-up message.

In addition to adjusting the step size to meet the local error test, cvode periodically adjusts the order, with the goal of maximizing the step size. The integration starts out at order 1 and varies the order dynamically after that. The basic idea is to pick the order $q$ for which a polynomial of order $q$ best fits the discrete data involved in the multistep method. However, if either a convergence failure or an error test failure occurred on the step just completed, no change in step size or order is done. At the current order $q$, selecting a new step size is done exactly as when the error test fails, giving a tentative step size ratio

$$h'/h = (\epsilon/6\|\Delta_n\|)^{1/(q+1)} \equiv \eta_q.$$  \hfill (2.11)

We consider changing order only after taking $q + 1$ steps at order $q$, and then we consider only orders $q' = q - 1$ (if $q > 1$) or $q' = q + 1$ (if $q < 5$). The local truncation error at order $q'$ is estimated using the history data. Then a tentative step size ratio is computed on the basis that this error, LTE($q'$), behaves asymptotically as $h^{q'+1}$. With safety factors of 1/6 and 1/10 respectively, these ratios are:

$$h'/h = [1/6\|\text{LTE}(q - 1)\|]^{1/q} \equiv \eta_{q-1}$$

and

$$h'/h = [1/10\|\text{LTE}(q + 1)\|]^{1/(q+2)} \equiv \eta_{q+1}.$$
The new order and step size are then set according to
\[ \eta = \max\{\eta_{q-1}, \eta_q, \eta_{q+1}\}, \quad h' = \eta h, \]
with \( q' \) set to the index achieving the above maximum. However, if we find that \( \eta < 1.5 \), we do not bother with the change. Also, \( h'/h \) is always limited to 10, except on the first step, when it is limited to \( 10^4 \).

The various algorithmic features of CVODE described above, as inherited from VODE and VODEPK, are documented in [6, 9, 26]. They are also summarized in [27].

CVODE 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 Newton iteration and reduce the step size. Rather than cutting the step size by some arbitrary factor, CVODE 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).

Normally, CVODE takes steps until a user-defined output value \( t = t_{\text{out}} \) is overtaken, and then it 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 CVODE 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 the linear system (2.5) (e.g., the default Newton iteration), CVODE makes repeated use of a linear solver to solve linear systems of the form \( Mx = -r \), where \( x \) is a correction vector and \( r \) is a residual vector. 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, as \( (P^{-1}A)x = P^{-1}b \); on the right, as \( (AP^{-1})x = P^{-1}b \); or on both sides, as \( (PL^{-1}AP_R^{-1})x = PL^{-1}b \). The Krylov method is then applied to a system with the matrix \( P^{-1}A \), or \( AP^{-1} \), or \( PL^{-1}AP_R^{-1} \), instead of \( A \). In order to improve the convergence of the Krylov iteration, the preconditioner matrix \( P \), or the product \( PL \) \( PR \), in the last case, should in some sense approximate the system matrix \( A \). Yet at the same time, in order to be cost-effective, the matrix \( P \), or matrices \( PL \) and \( PR \), 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).

Most of the iterative linear solvers supplied with Sundials allow for preconditioning either side, or on both sides, although we know of no situation where preconditioning on both sides is clearly superior to preconditioning on one side only (with the product \( PL \) \( PR \)). Moreover, for a given preconditioner matrix, the merits of left vs. right preconditioning are unclear in general, and the user should experiment with both choices. Performance will differ because the inverse of the left preconditioner is included in the linear system residual whose norm is being tested in the Krylov algorithm. As a rule, however, if the preconditioner is the product of two matrices, we recommend that preconditioning be done either on the left only or the right only, rather than using one factor on each side.

Typical preconditioners used with CVODE are based on approximations to the system Jacobian, \( J = \partial f/\partial y \). Since the matrix involved is \( M = I - \gamma J \), any approximation \( \tilde{J} \) to \( J \) yields a matrix that is of potential use as a preconditioner, namely \( \tilde{P} = I - \gamma \tilde{J} \). 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 BDF stability limit detection

CVODE includes an algorithm, STALD (STAbility Limit Detection), which provides protection against potentially unstable behavior of the BDF multistep integration methods in certain situations, as described below.

When the BDF option is selected, CVODES uses Backward Differentiation Formula methods of orders 1 to 5. At order 1 or 2, the BDF method is A-stable, meaning that for any complex constant \( \lambda \) in the open left half-plane, the method is unconditionally stable (for any step size) for the standard scalar model problem \( \dot{y} = \lambda y \). For an ODE system, this means that, roughly speaking, as long as all modes in the system are stable, the method is also stable for any choice of step size, at least in the sense of a local linear stability analysis.

At orders 3 to 5, the BDF methods are not A-stable, although they are stiffly stable. In each case, in order for the method to be stable at step size \( h \) on the scalar model problem, the product \( h\lambda \) must lie within a region of absolute stability. That region excludes a portion of the left half-plane that is concentrated near the imaginary axis. The size of that region of instability grows as the order increases from 3 to 5. What this means is that, when running BDF at any of these orders, if an eigenvalue \( \lambda \) of the system lies close enough to the imaginary axis, the step sizes \( h \) for which the method is stable are limited (at least according to the linear stability theory) to a set that prevents \( h\lambda \) from leaving the stability region. The meaning of close enough depends on the order. At order 3, the unstable region is much narrower than at order 5, so the potential for unstable behavior grows with order.

System eigenvalues that are likely to run into this instability are ones that correspond to weakly damped oscillations. A pure undamped oscillation corresponds to an eigenvalue on the imaginary axis. Problems with modes of that kind call for different considerations, since the oscillation generally must be followed by the solver, and this requires step sizes \( (h \sim 1/\nu, \text{ where } \nu \text{ is the frequency}) \) that are stable for BDF anyway. But for a weakly damped oscillatory mode, the oscillation in the solution is eventually damped to the noise level, and at that time it is important that the solver not be restricted to step sizes on the order of \( 1/\nu \). It is in this situation that the new option may be of great value.

In terms of partial differential equations, the typical problems for which the stability limit detection option is appropriate are ODE systems resulting from semi-discretized PDEs (i.e., PDEs discretized in space) with advection and diffusion, but with advection dominating over diffusion. Diffusion alone produces pure decay modes, while advection tends to produce undamped oscillatory modes. A mix of the two with advection dominant will have weakly damped oscillatory modes.

The STALD algorithm attempts to detect, in a direct manner, the presence of a stability region boundary that is limiting the step sizes in the presence of a weakly damped oscillation [24]. The algorithm supplements (but differs greatly from) the existing algorithms in CVODES for choosing step size and order based on estimated local truncation errors. The STALD algorithm works directly with history data that is readily available in CVODE. If it concludes that the step size is in fact stability-limited, it dictates a reduction in the method order, regardless of the outcome of the error-based algorithm. The STALD algorithm has been tested in combination with the VODE solver on linear advection-dominated advection-diffusion problems [25], where it works well. The implementation in CVODE has been successfully tested on linear and nonlinear advection-diffusion problems, among others.

This stability limit detection option adds some computational overhead to the CVODES solution. (In timing tests, these overhead costs have ranged from 2% to 7% of the total, depending on the size and complexity of the problem, with lower relative costs for larger problems.) Therefore, it should be activated only when there is reasonable expectation of modes in the user’s system for which it is appropriate. In particular, if a CVODE solution with this option turned off appears to take an inordinately large number of steps at orders 3-5 for no apparent reason in terms of the solution time scale, then there is a good chance that step sizes are being limited by stability, and that turning on the option will improve the efficiency of the solution.
2.4 Rootfinding

The CVODE solver has been augmented to include a rootfinding feature. This means that, while integrating the Initial Value Problem (2.1), CVODE can also find the roots of a set of user-defined functions \( g_i(t, y) \) that depend both on \( t \) and on the solution vector \( y = y(t) \). 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 \( g_i(t, y(t)) \), 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 CVODE. 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 hone in on the root(s) with a modified secant method [23]. In addition, each time \( g \) is computed, CVODE 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 \( t \), CVODE 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, CVODE stops and reports an error. This way, each time CVODE 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, CVODE has an interval \((t_{hi}, t_{lo})\) 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 \( t_{n-1} \), 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 \( g_i \) at \( t_{hi} \) for zeros and for sign changes in \((t_{lo}, t_{hi})\). If no sign changes were 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

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

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 include the sign change. If there is none in \((t_{lo}, t_{mid})\) but some \( g_i(t_{mid}) = 0 \), then that root is reported. The loop continues until \(|t_{hi} - t_{lo}| < \tau \), and then the reported root location is \( t_{hi} \).

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

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

where \( \alpha \) is 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 \( \frac{dy}{dt} = f(t, y) \) 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 \( M\frac{dy}{dt} = f_E(t, y) + f_I(t, y) \) 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 CVODE organization

The CVODE package is written in ANSI C. 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 CVODE package is shown in Figure 3.3. The central integration module, implemented in the files cvode.h, cvode_impl.h, and cvode.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.

CVODE utilizes generic linear and nonlinear solver modules defined by the SUNLINSOL API (see Chapter 8) and SUNNONLINSOL API (see Chapter 9) respectively. As such, CVODE has no knowledge of the method being used to solve the linear and nonlinear systems that arise. For any given user problem, there exists a single nonlinear solver interface and, if necessary, one of the linear system solver interfaces is specified, and invoked as needed during the integration.

At present, the package includes two linear solver interfaces. The primary linear solver interface, CVLS, supports 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
Figure 3.1: High-level diagram of the SUNDIALS suite

information, or they may be matrix-free. Since CVODE can operate on any valid SUNLINSOL implementation, the set of linear solver modules available to CVODE will expand as new SUNLINSOL modules are developed.

Additionally, CVODE includes the diagonal linear solver interface, CVDIAG, that creates an internally generated diagonal approximation to the Jacobian.

For users employing dense or banded Jacobian matrices, CVODE includes algorithms for their approximation through difference quotients, although the user also has the option of supplying a routine to compute 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, CVODE includes an algorithm for the approximation by difference quotients of the product $Mv$. 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, 9], together with the example and demonstration programs included with CVODE, offer considerable assistance in building preconditioners.

CVODE’s linear solver interface consists of four primary phases, 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, and only as required to achieve convergence.

CVODE also provides two preconditioner modules, for use with any of the Krylov iterative linear solvers. The first one, CVBANDPRE, is intended to be used with NVECTOR_SERIAL, NVECTOR_OPENMP or NVECTOR_PTHREADS and provides a banded difference-quotient Jacobian-based preconditioner, with corresponding setup and solve routines. The second preconditioner module, CVBBDDPRE, works in conjunction with NVECTOR_PARALLEL and generates a preconditioner that is a block-diagonal...
3.2 CVODE organization

(a) Directory structure of the SUNDIALS source tree

(b) Directory structure of the SUNDIALS examples

Figure 3.2: Organization of the SUNDIALS suite
matrix with each block being a banded matrix.

All state information used by CVODE 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 CVODE package, and so, in this respect, it is reentrant. State information specific to the linear solver is saved in a separate structure, a pointer to which resides in the CVODE memory structure. The reentrancy of CVODE was motivated by the anticipated multicomputer extension, but is also essential in a uniprocessor setting where two or more problems are solved by intermixed calls to the package from within a single user program.
Chapter 4

Using CVODE for C Applications

This chapter is concerned with the use of CVODE for the solution of initial value problems (IVPs) in a C language setting. The following sections treat the header files and the layout of the user’s main program, and provide descriptions of the CVODE user-callable functions and user-supplied functions.

The sample programs described in the companion document [29] may also be helpful. Those codes may be used as templates (with the removal of some lines used in testing) and are included in the CVODE package.

Users with applications written in FORTRAN should see Chapter 5, which describes interfacing with CVODE from FORTRAN.

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 CVBANDPRE preconditioning module is only compatible with the NVECTOR_SERIAL, NVECTOR_OPENMP, and NVECTOR_PTHREADS vector implementations, and the preconditioner module CVBBDPRE 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.

CVODE 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 CVODE, 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 CVODE. The relevant library files are

- `libdir/libsundials_cvode.lib`
- `libdir/libsundials_nvec*.lib`

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

- `incdir/include/cvode`
- `incdir/include/sundials`
28 Using CVODE for C Applications

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

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

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

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

4.3 Header files

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

- `cvode/cvode.h`, the main header file for CVODE, which defines the several types and various constants, and includes function prototypes. This includes the header file for CVLS, `cvode/cvode_ls.h`.

Note that `cvode.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 `sunnonsol/sunnonsol_***.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 (2.5) (e.g., the default Newton iteration), then a linear solver module header file will be required. The header files corresponding to the SUNDIALS-provided linear solver modules available for use with CVODE 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_lapack.dense.h`, which is used with the LAPACK dense linear solver module, SUNLINSOL_LAPACKDENSE;
  - `sunlinsol/sunlinsol_lapack.banded.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`;

- `cvoe/cvoe_diag.h`, which is used with the CVDIAG diagonal linear solver module.

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 `cvDiurnal_kry.p` example (see [29]), 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 an ODE IVP. Most of the steps are independent of the `NVESOR`, `SUNMATRIX`, `SUNLINSOL`, and `SUNNONLINLSOL` implementations used. For the steps that are not, refer to Chapters 6, 7, 8, and 9 for the specific name of the function to be called or macro to be referenced.

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

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

2. **Set problem dimensions etc.**

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

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

3. **Set vector of initial values**

   To set the vector `y0` of initial values, use the appropriate functions defined by the particular `NVESOR` implementation.

   For native SUNDIALS vector implementations (except the CUDA and RAJA-based ones), use a call of the form `y0 = N_VMake_**(..., ydata)` if the `realtype` array `ydata` containing the initial values of `y` already exists. Otherwise, create a new vector by making a call of the form `y0 = N_VNew_**(...), and then set its elements by accessing the underlying data with a call of the form `ydata = N_VGetArrayPointer(y0)`. See §6.2-6.5 for details.

   For the hypre and PETSc vector wrappers, first create and initialize the underlying vector, and then create an `NVESOR` wrapper with a call of the form `y0 = N_VMake_**(yvec)`, where `yvec` is a hypre or PETSc vector. Note that calls like `N_VNew_**(...)` and `N_VGetArrayPointer(...)` are not available for these vector wrappers. See §6.6 and §6.7 for details.
4.4 A skeleton of the user’s main program

If using either the CUDA- or RAJA-based vector implementations use a call of the form \( y0 = NVMake_\text{(..., c)} \) where \( c \) is a pointer to a suncudavec or sunrajavec vector class if this class already exists. Otherwise, create a new vector by making a call of the form \( y0 = NVNew_\text{(..., ...) \) and then set its elements by accessing the underlying data where it is located with a call of the form \( NVGetDeviceArrayPointer_\text{ (...) or NVGetHostArrayPointer_\text{ (}) \}. Note that the vector class will allocate memory on both the host and device when instantiated. See §6.8-6.9 for details.

4. Create \textsc{cvode} object

Call \texttt{cvode\_mem = CVodeCreate(lmm)} to create the \textsc{cvode} memory block and to specify the linear multistep method. \texttt{CVodeCreate} returns a pointer to the \textsc{cvode} memory structure. See §4.5.1 for details.

5. Initialize \textsc{cvode} solver

Call \texttt{CVodeInit(...) \} to provide required problem specifications, allocate internal memory for \textsc{cvode}, and initialize \textsc{cvode}. \texttt{CVodeInit} returns a flag, the value of which indicates either success or an illegal argument value. See §4.5.1 for details.

6. Specify integration tolerances

Call \texttt{CVodeSS tolerances(...) \} or \texttt{CVodeSV tolerances(...) \} to specify either a scalar relative tolerance and scalar absolute tolerance, or a scalar relative tolerance and a vector of absolute tolerances, respectively. Alternatively, call \texttt{CVodeWF tolerances \} 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 solve 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 \textsc{sunmatrix} implementation.

For the \textsc{sundials}-supplied \textsc{sunmatrix} implementations, the matrix object may be created using a call of the form

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

or

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

or

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

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 \textsc{sunlinsol} implementation.

For any of the \textsc{sundials}-supplied \textsc{sunlinsol} implementations, the linear solver object may be created using a call of the form

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

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 CVLS linear solver interface by attaching the linear solver object (and matrix object, if applicable) with the call (for details see §4.5.3):

\begin{verbatim}
ier = CVodeSetLinearSolver(...);
\end{verbatim}

Alternately, if the CVODE-specific diagonal linear solver module, CVDIAG, is desired, initialize the linear solver module and attach it to CVODE with the call

\begin{verbatim}
ier = CVDiag(...);
\end{verbatim}

11. Set optional inputs

Call CVodeSet\* functions to change any optional inputs that control the behavior of CVODE from their default values. See §4.5.7.1 and §4.5.7 for details.

12. Create nonlinear solver object \textit{(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 \textit{(optional)}

If using a non-default nonlinear solver, then initialize the nonlinear solver interface by attaching the nonlinear solver object by calling \texttt{ier = CVodeSetNonlinearSolver(cvode_mem, NLS)}; (see §4.5.4 for details).

14. Set nonlinear solver optional inputs \textit{(optional)}

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

15. Specify rootfinding problem

Optionally, call CVodeRootInit to initialize a rootfinding problem to be solved during the integration of the ODE system. See §4.5.5, and see §4.5.7.3 for relevant optional input calls.

16. Advance solution in time

For each point at which output is desired, call \texttt{ier = CVode(cvode_mem, tout, yout, &tret, itask)}. Here \texttt{itask} specifies the return mode. The vector \texttt{yout} (which can be the same as the vector \texttt{y0} above) will contain \(y(t)\). See §4.5.6 for details.

17. Get optional outputs

Call CV\*Get\* functions to obtain optional output. See §4.5.9 for details.

18. Deallocate memory for solution vector

Upon completion of the integration, deallocate memory for the vector \(y\) (or \texttt{yout}) by calling the appropriate destructor function defined by the NVECTOR implementation:

\begin{verbatim}
N_VDestroy(y);
\end{verbatim}

19. Free solver memory
4.5 User-callable functions

This section describes the cvode functions that are called by the user to setup and then solve an IVP. Some of these are required. However, starting with §4.5.7, the functions listed involve optional inputs/outputs or restarting, and those paragraphs may be skipped for a casual use of cvode. In any case, refer to §4.4 for the correct order of these calls.

On an error, each user-callable function returns a negative value and sends an error message to the error handler routine, which prints the message on stderr by default. However, the user can set a file as error output or can provide his own error handler function (see §4.5.7.1).
4.5.1 CVODE 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 IVP solution is complete, as it frees the CVODE memory block created and allocated by the first two calls.

**CVodeCreate**

**Call**
```
cvode_mem = CVodeCreate(lmm);
```

**Description**
The function `CVodeCreate` instantiates a CVODE solver object and specifies the solution method.

**Arguments**
- `lmm` (int) specifies the linear multistep method and must be one of two possible values: `CV_ADAMS` or `CV_BDF`.
- The recommended choices for `lmm` are `CV_ADAMS` for nonstiff problems and `CV_BDF` for stiff problems. The default Newton iteration is recommended for stiff problems, and the fixed-point solver (previously referred to as the functional iteration in this guide) is recommended for nonstiff problems. For details on how to attach a different nonlinear solver module to CVODE see the description of `CvodeSetNonlinearSolver`.

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

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

**CVodeInit**

**Call**
```
flag = CVodeInit(cvode_mem, f, t0, y0);
```

**Description**
The function `CVodeInit` provides required problem and solution specifications, allocates internal memory, and initializes CVODE.

**Arguments**
- `cvode_mem` (void *) pointer to the CVODE memory block returned by `CVodeCreate`.
- `f` (CVRhsFn) is the C function which computes the right-hand side function `f` in the ODE. This function has the form `f(t, y, ydot, 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`.

**Return value**
The return value `flag` (of type `int`) will be one of the following:
- `CV_SUCCESS` The call to `CVodeInit` was successful.
- `CV_MEM_NULL` The CVODE memory block was not initialized through a previous call to `CVodeCreate`.
- `CV_MEM_FAIL` A memory allocation request has failed.
- `CV_ILL_INPUT` An input argument to `CVodeInit` has an illegal value.

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

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

**CVodeFree**

**Call**
```
CVodeFree(&cvode_mem);
```

**Description**
The function `CVodeFree` frees the memory allocated by a previous call to `CVodeCreate`.

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

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

**F2003 Name**
This function is callable as `FCVodeFree` when using the Fortran 2003 interface module.
4.5 User-callable functions

4.5.2 CVODE 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 CVodeInit.

CVodeSStolerances

Call

flag = CVodeSStolerances(cvode_mem, reltol, abstol);

Description
The function CVodeSStolerances specifies scalar relative and absolute tolerances.

Arguments
- cvode_mem (void *) pointer to the CVODE memory block returned by CVodeCreate.
- 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:
- CV_SUCCESS The call to CVodeSStolerances was successful.
- CV_MEM_NULL The CVODE memory block was not initialized through a previous call to CVodeCreate.
- CV_NO_MALLOC The allocation function CVodeInit has not been called.
- CV_I LL_INPUT One of the input tolerances was negative.

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

CVodeSVtolerances

Call

flag = CVodeSVtolerances(cvode_mem, reltol, abstol);

Description
The function CVodeSVtolerances specifies scalar relative tolerance and vector absolute tolerances.

Arguments
- cvode_mem (void *) pointer to the CVODE memory block returned by CVodeCreate.
- 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:
- CV_SUCCESS The call to CVodeSVtolerances was successful.
- CV_MEM_NULL The CVODE memory block was not initialized through a previous call to CVodeCreate.
- CV_NO_MALLOC The allocation function CVodeInit has not been called.
- CV_I LL_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.

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

CVodeWFtolerances

Call

flag = CVodeWFtolerances(cvode_mem, efun);

Description
The function CVodeWFtolerances 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. (2.7).

Arguments
- cvode_mem (void *) pointer to the CVODE memory block returned by CVodeCreate.
- efun (CVEwtFn) 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:

- CV_SUCCESS The call to CVodeWFtolerances was successful.
- CV_MEM_NULL The CVODE memory block was not initialized through a previous call to CVodeCreate.
- CV_NO_MALLOC The allocation function CVodeInit has not been called.

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

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

1. The scalar relative tolerance reltol is to be set to control relative errors. So reltol = $10^{-4}$ means that errors are controlled to .01%. We do not recommend using reltol larger than $10^{-3}$. On the other hand, reltol should not be so small that it is comparable to the unit roundoff of the machine arithmetic (generally around 1.0E-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 cvRoberts_dns in the CVODE package, and the discussion of it in the CVODE Examples document [29]. 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 some sort of accumulation of those per-step errors. A good rule of thumb is to reduce the tolerances by a factor of .01 from the actual desired limits on errors. So if you want .01% accuracy (globally), a good choice is reltol = $10^{-6}$. But in any case, it is a good idea to do a few experiments with the tolerances to see how the computed solution values vary as tolerances are reduced.

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

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

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

3. The user’s right-hand side routine f should never change a negative value in the solution vector y to a non-negative value, as a “solution” to this problem. This can cause instability. If the f 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 y vector) for the purposes of computing f(t, y).

4. Positivity and non-negativity constraints on components can be enforced by use of the recoverable error return feature in the user-supplied right-hand side function. However, because this option involves some extra overhead cost, it should only be exercised if the use of absolute tolerances to control the computed values is unsuccessful.
4.5.3 Linear solver interface functions

As previously explained, if the nonlinear solver requires the solution of linear systems of the form \((2.5)\) (e.g., the default Newton iteration), there are two CVODE linear solver interfaces currently available for this task: CVLS and CVDIAG.

The first corresponds to the main linear solver interface in CVODE, that supports all valid SUNLINSOL modules. Here, matrix-based SUNLINSOL modules utilize SUNMATRIX objects to store the approximate Jacobian matrix \(J = \partial f / \partial y\), the Newton matrix \(M = I - \gamma J\), and factorizations used throughout the solution process. Conversely, matrix-free SUNLINSOL modules instead use iterative methods to solve the Newton systems of equations, and only require the \textit{action} of the matrix on a vector, \(Mv\). With most of these methods, preconditioning can be done on the left only, the right only, on both the left and right, or not at all. The exceptions to this rule are SPFGMR that supports right preconditioning only and PCG that performs symmetric preconditioning. For the specification of a preconditioner, see the iterative linear solver sections in §4.5.7 and §4.6.

If preconditioning is done, user-supplied functions define linear operators corresponding to left and right preconditioner matrices \(P_1\) and \(P_2\) (either of which could be the identity matrix), such that the product \(P_1 P_2\) approximates the matrix \(M = I - \gamma J\) of \((2.6)\).

The CVDIAG linear solver interface supports a direct linear solver, that uses only a diagonal approximation to \(J\).

To specify a generic linear solver to CVODE, after the call to CVodeCreate but before any calls to CVode, the user’s program must create the appropriate SUNLinearSolver object and call the function CVodeSetLinearSolver, 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

\[
\text{SUNLinearSolver } LS = \text{SUNLinSol_*}(\ldots);
\]

The current list of such constructor routines includes SUNLinSol_Dense, SUNLinSol_Band, SUNLinSol_LapackDense, SUNLinSol_LapackBand, SUNLinSol_KLU, SUNLinSol_SuperLUMT, SUNLinSol_SPGMR, SUNLinSol_SPFGMR, 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 CVODE via a call to CVodeSetLinearSolver. The first argument passed to this function is the CVODE memory pointer returned by CVodeCreate; the second argument is the desired SUNLINSOL object to use for solving linear 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 CVLS linear solver interface, linking it to the main CVODE integrator, and allows the user to specify additional parameters and routines pertinent to their choice of linear solver.

To instead specify the CVODE-specific diagonal linear solver interface, the user’s program must call CVDiag, as documented below. The first argument passed to this function is the CVODE memory pointer returned by CVodeCreate.

**CVodeSetLinearSolver**

**Call**

\[
\text{flag} = \text{CVodeSetLinearSolver}(\text{cvode\_mem, LS, J});
\]

**Description**
The function CVodeSetLinearSolver attaches a generic SUNLINSOL object LS and corresponding template Jacobian SUNMATRIX object J (if applicable) to CVODE, initializing the CVLS linear solver interface.

**Arguments**
- cvode_mem (void *) pointer to the CVODE memory block.
- LS (SUNLinearSolver) SUNLINSOL object to use for solving linear systems of the form \((2.5)\).
J  (SUNMatrix) SUNMATRIX object for used as a template for the Jacobian (or NULL if not applicable).

Return value The return value flag (of type int) is one of:
- CVLS_SUCCESS: The CVLS initialization was successful.
- CVLS_MEM_NULL: The cvode_mem pointer is NULL.
- CVLS_ILL_INPUT: The CVLS interface is not compatible with the LS or J input objects or is incompatible with the current NV_VECTOR module.
- CVLS_SUNLS_FAIL: A call to the LS object failed.
- CVLS_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).

When using sparse linear solvers, it is typically much more efficient to supply J so that it includes the full sparsity pattern of the Newton system matrices \( M = I - \gamma J \), even if J itself has zeros in nonzero locations of I. The reasoning for this is that M is constructed in-place, on top of the user-specified values of J, so if the sparsity pattern in J is insufficient to store M then it will need to be resized internally by CVODE.

The previous routines CVDlsSetLinearSolver and CVSpilsSetLinearSolver 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.

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

4.5.4 Nonlinear solver interface function

By default CVODE uses the SUNNONLINSOL implementation of Newton’s method defined by the SUNNONLINSOL_NEwTON module (see §9.2). To specify a different nonlinear solver in CVODE, the user’s program must create a SUNNONLINSOL object by calling the appropriate constructor routine. The user must then attach the SUNNONLINSOL object by calling CVodeSetNonlinearSolver, as documented below.
When changing the nonlinear solver in CVODE, `CVodeSetNonlinearSolver` must be called after `CVodeInit`. If any calls to `CVode` have been made, then CVODE will need to be reinitialized by calling `CVodeReInit` to ensure that the nonlinear solver is initialized correctly before any subsequent calls to `CVode`.

The first argument passed to the routine `CVodeSetNonlinearSolver` is the CVODE memory pointer returned by `CVodeCreate` and the second argument is the SUNNONLINSOL object to use for solving the nonlinear system (2.3) or (2.4). A call to this function attaches the nonlinear solver to the main CVODE integrator.

```c
CVodeSetNonlinearSolver
```

**Call**

```c
flag = CVodeSetNonlinearSolver(cvode_mem, NLS);
```

**Description**

The function `CVodeSetNonlinearSolver` attaches a SUNNONLINSOL object (NLS) to CVODE.

**Arguments**

- `cvode_mem` *(void *)* pointer to the CVODE memory block.
- `NLS` *(SUNNonlinearSolver)* SUNNONLINSOL object to use for solving nonlinear systems (2.3) or (2.4).

**Return value**

The return value `flag` (of type `int`) is one of
- `CV_SUCCESS` The nonlinear solver was successfully attached.
- `CV_MEM_NULL` The `cvode_mem` pointer is NULL.
- `CV_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.

**F2003 Name**

This function is callable as `FCVodeSetNonlinearSolver` when using the Fortran 2003 interface module.

### 4.5.5 Rootfinding initialization function

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

```c
CVodeRootInit
```

**Call**

```c
flag = CVodeRootInit(cvode_mem, nrtfn, g);
```

**Description**

The function `CVodeRootInit` specifies that the roots of a set of functions $g_i(t, y)$ are to be found while the IVP is being solved.

**Arguments**

- `cvode_mem` *(void *)* pointer to the CVODE memory block returned by `CVodeCreate`.
- `nrtfn` *(int)* is the number of root functions $g_i$.
- `g` *(CVRootFn)* is the C function which defines the `nrtfn` functions $g_i(t, y)$ whose roots are sought. See §4.6.4 for details.

**Return value**

The return value `flag` (of type `int`) is one of
- `CV_SUCCESS` The call to `CVodeRootInit` was successful.
- `CV_MEM_NULL` The `cvode_mem` argument was NULL.
- `CV_MEM_FAIL` A memory allocation failed.
- `CV_Ill_INPUT` The function $g$ is NULL, but `nrtfn` > 0.

**Notes**

If a new IVP is to be solved with a call to `CVodeReInit`, where the new IVP has no rootfinding problem but the prior one did, then call `CVodeRootInit` with `nrtfn` = 0.

**F2003 Name**

This function is callable as `FCVodeRootInit` when using the Fortran 2003 interface module.
4.5.6 CVODE solver function

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

```
CVode
Call                      flag = CVode(cvode_mem, tout, yout, &tret, itask);
Description               The function CVode integrates the ODE over an interval in t.
Arguments                cvode_mem (void *) pointer to the CVODE memory block.
                          tout (realtype) the next time at which a computed solution is desired.
                          yout (N_Vector) the computed solution vector.
                          tret (realtype) the time reached by the solver (output).
                          itask (int) a flag indicating the job of the solver for the next user step. The
                          CV_NORMAL option causes the solver to take internal steps until it has reached
                          or just passed the user-specified tout parameter. The solver then interpo-
                          lates in order to return an approximate value of y(tout). The CV_ONE_STEP
                          option tells the solver to take just one internal step and then return the
                          solution at the point reached by that step.
Return value              CVode returns a vector yout and a corresponding independent variable value t = tret,
                          such that yout is the computed value of y(t).
                          In CV_NORMAL mode (with no errors), tret will be equal to tout and yout = y(tout).
                          The return value flag (of type int) will be one of the following:
                          CV_SUCCESS         CVode succeeded and no roots were found.
                          CV_TSTOP_RETURN   CVode succeeded by reaching the stopping point specified through
                          the optional input function CVodeSetStopTime (see §4.5.7.1).
                          CV_ROOT_RETURN    CVode succeeded and found one or more roots. In this case,
                          tret is the location of the root. If nrtfn > 1, call CVodeGetRootInfo
                          to see which gi were found to have a root.
                          CV_MEM_NULL       The cvode_mem argument was NULL.
                          CV_NO_MALLOC      The CVODE memory was not allocated by a call to CVodeInit.
                          CV_Ill_INPUT      One of the inputs to CVode 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 CVodeCreate) failed to set the linear solver-specific lsolve
                          field in cvode_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 error message for details.
                          CV_TOO_CLOSE      The initial time t0 and the output time tout are too close to
                          each other and the user did not specify an initial step size.
                          CV_TOO_MUCH_WORK  The solver took mxstep internal steps but still could not reach
                          tout. The default value for mxstep is MXSTEP_DEFAULT = 500.
                          CV_TOO_MUCH_ACC   The solver could not satisfy the accuracy demanded by the
                          user for some internal step.
                          CV_ERR_FAILURE    Either error test failures occurred too many times (MXNEF =
                          7) during one internal time step, or with |h| = h_min.
```
CV_CONV_FAILURE: Either convergence test failures occurred too many times ($\text{MXNCF} = 10$) during one internal time step, or with $|h| = h_{\text{min}}$.

CV_LINIT_FAIL: The linear solver interface’s initialization function failed.

CV_LSETUP_FAIL: The linear solver interface’s setup function failed in an unrecoverable manner.

CV_LSOLVE_FAIL: The linear solver interface’s solve function failed in an unrecoverable manner.

CV_CONSTR_FAIL: The inequality constraints were violated and the solver was unable to recover.

CV_RHSFUNC_FAIL: The right-hand side function failed in an unrecoverable manner.

CV_FIRST_RHSFUNC_FAIL: The right-hand side function had a recoverable error at the first call.

CV_REPTD_RHSFUNC_ERR: Convergence test failures occurred too many times due to repeated recoverable errors in the right-hand side function. This flag will also be returned if the right-hand side function had repeated recoverable errors during the estimation of an initial step size.

CV_UNREC_RHSFUNC_ERR: The right-hand function had a recoverable error, but no recovery was possible. This failure mode is rare, as it can occur only if the right-hand side function fails recoverably after an error test failed while at order one.

CV_RTFUNC_FAIL: The rootfinding function failed.

Notes: The vector $\text{yout}$ can occupy the same space as the vector $y_0$ of initial conditions that was passed to $\text{CVodeInit}$.

In the $\text{CV\_ONE\_STEP}$ mode, $\text{tout}$ is used only on the first call, and only to get the direction and a rough scale of the independent variable.

If a stop time is enabled (through a call to $\text{CVodeSetStopTime}$), then $\text{CVode}$ returns the solution at $t_{\text{stop}}$. Once the integrator returns at a stop time, any future testing for $t_{\text{stop}}$ is disabled (and can be reenabled only through a new call to $\text{CVodeSetStopTime}$).

All failure return values are negative and so the test $\text{flag} < 0$ will trap all $\text{CVode}$ failures.

On any error return in which one or more internal steps were taken by $\text{CVode}$, the returned values of $t_{\text{ret}}$ and $\text{yout}$ correspond to the farthest point reached in the integration. On all other error returns, $t_{\text{ret}}$ and $\text{yout}$ are left unchanged from the previous $\text{CVode}$ return.

F2003 Name: This function is callable as $\text{FCVode}$ when using the Fortran 2003 interface module.

### 4.5.7 Optional input functions

There are numerous optional input parameters that control the behavior of the CVODE solver. CVODE 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 CVODE which are then described in detail in the remainder of this section, beginning with those for the main CVODE solver and continuing with those for the linear solver interfaces. Note that the diagonal linear solver module has no optional inputs. For the most casual use of CVODE, the reader can skip to §4.6.

We note that, on an error return, all of the optional input functions send an error message to the error handler function. We also note that all error return values are negative, so the test $\text{flag} < 0$ will catch all errors.
Table 4.2: Optional inputs for CVODE and CVLS

<table>
<thead>
<tr>
<th>Optional input</th>
<th>Function name</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointer to an error file</td>
<td>CVodeSetErrFile</td>
<td>stderr</td>
</tr>
<tr>
<td>Error handler function</td>
<td>CVodeSetErrHandlerFn</td>
<td>internal fn.</td>
</tr>
<tr>
<td>User data</td>
<td>CVodeSetUserData</td>
<td>NULL</td>
</tr>
<tr>
<td>Maximum order for BDF method</td>
<td>CVodeSetMaxOrd</td>
<td>5</td>
</tr>
<tr>
<td>Maximum order for Adams method</td>
<td>CVodeSetMaxOrd</td>
<td>12</td>
</tr>
<tr>
<td>Maximum no. of internal steps before ( t_{\text{out}} )</td>
<td>CVodeSetMaxNumSteps</td>
<td>500</td>
</tr>
<tr>
<td>Maximum no. of warnings for ( t_n + h = t_n )</td>
<td>CVodeSetMaxHnilWarns</td>
<td>10</td>
</tr>
<tr>
<td>Flag to activate stability limit detection</td>
<td>CVodeSetStabLimDet</td>
<td>SUNFALSE</td>
</tr>
<tr>
<td>Initial step size</td>
<td>CVodeSetInitStep</td>
<td>estimated</td>
</tr>
<tr>
<td>Minimum absolute step size</td>
<td>CVodeSetMinStep</td>
<td>0.0</td>
</tr>
<tr>
<td>Maximum absolute step size</td>
<td>CVodeSetMaxStep</td>
<td>( \infty )</td>
</tr>
<tr>
<td>Value of ( t_{\text{stop}} )</td>
<td>CVodeSetStopTime</td>
<td>undefined</td>
</tr>
<tr>
<td>Maximum no. of error test failures</td>
<td>CVodeSetMaxErrTestFails</td>
<td>7</td>
</tr>
<tr>
<td>Maximum no. of nonlinear iterations</td>
<td>CVodeSetMaxNonlinIters</td>
<td>3</td>
</tr>
<tr>
<td>Maximum no. of convergence failures</td>
<td>CVodeSetMaxConvFails</td>
<td>10</td>
</tr>
<tr>
<td>Coefficient in the nonlinear convergence test</td>
<td>CVodeSetNonlinConvCoef</td>
<td>0.1</td>
</tr>
<tr>
<td>Inequality constraints on solution</td>
<td>CVodeSetConstraints</td>
<td>NULL</td>
</tr>
<tr>
<td>Direction of zero-crossing</td>
<td>CVodeSetRootDirection</td>
<td>both</td>
</tr>
<tr>
<td>Disable rootfinding warnings</td>
<td>CVodeSetNoInactiveRootWarn</td>
<td>none</td>
</tr>
</tbody>
</table>

CVLS linear solver interface

<table>
<thead>
<tr>
<th>Optional input</th>
<th>Function name</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacobian / preconditioner update frequency</td>
<td>CVodeSetMaxStepsBetweenJac</td>
<td>50</td>
</tr>
<tr>
<td>Jacobian function</td>
<td>CVodeSetJacFn</td>
<td>DQ</td>
</tr>
<tr>
<td>Jacobian-times-vector functions</td>
<td>CVodeSetJacTimes</td>
<td>NULL, DQ</td>
</tr>
<tr>
<td>Preconditioner functions</td>
<td>CVodeSetPreconditioner</td>
<td>NULL, NULL</td>
</tr>
<tr>
<td>Ratio between linear and nonlinear tolerances</td>
<td>CVodeSetEpsLin</td>
<td>0.05</td>
</tr>
</tbody>
</table>
4.5 User-callable functions

4.5.7.1 Main solver optional input functions

The calls listed here can be executed in any order. However, if either of the functions CVodeSetErrFile or CVodeSetErrHandlerFn is to be called, that call should be first, in order to take effect for any later error message.

`CVodeSetErrFile`

Call: \[ \text{flag} = \text{CVodeSetErrFile}(\text{cvode\_mem}, \text{errfp}); \]

Description: The function CVodeSetErrFile specifies a pointer to the file where all CVODE messages should be directed when the default CVODE error handler function is used.

Arguments:
- `cvode\_mem` (void *) pointer to the CVODE memory block.
- `errfp` (FILE *) pointer to output file.

Return value: The return value `flag` (of type int) is one of
- `CV\_SUCCESS` The optional value has been successfully set.
- `CV\_MEM\_NULL` The `cvode\_mem` pointer is NULL.

Notes:
- The default value for `errfp` is `stderr`.
- Passing a value of NULL disables all future error message output (except for the case in which the CVODE memory pointer is NULL). This use of CVodeSetErrFile is strongly discouraged.
- If CVodeSetErrFile is to be called, it should be called before any other optional input functions, in order to take effect for any later error message.

`CVodeSetErrHandlerFn`

Call: \[ \text{flag} = \text{CVodeSetErrHandlerFn}(\text{cvode\_mem}, \text{ehfun}, \text{eh\_data}); \]

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

Arguments:
- `cvode\_mem` (void *) pointer to the CVODE memory block.
- `ehfun` (CVErrHandlerFn) is the 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
- `CV\_SUCCESS` The function `ehfun` and data pointer `eh\_data` have been successfully set.
- `CV\_MEM\_NULL` The `cvode\_mem` pointer is NULL.

Notes:
- Error messages indicating that the CVODE solver memory is NULL will always be directed to `stderr`.
- F2003 Name: This function is callable as FCVodeSetErrHandlerFn when using the Fortran 2003 interface module.

`CVodeSetUserData`

Call: \[ \text{flag} = \text{CVodeSetUserData}(\text{cvode\_mem}, \text{user\_data}); \]

Description: The function CVodeSetUserData specifies the user data block `user\_data` and attaches it to the main CVODE memory block.

Arguments:
- `cvode\_mem` (void *) pointer to the CVODE memory block.
- `user\_data` (void *) pointer to the user data.

Return value: The return value `flag` (of type int) is one of
- `CV\_SUCCESS` The optional value has been successfully set.
- `CV\_MEM\_NULL` The `cvode\_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 CVodeSetUserData must be made before the call to specify the linear solver.

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

```c
CVodeSetMaxOrd
```

Call

```c
flag = CVodeSetMaxOrd(cvode_mem, maxord);
```

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

Arguments

- `cvode_mem` (void *) pointer to the CVODE 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

- CV_SUCCESS The optional value has been successfully set.
- CV_MEM_NULL The cvode_mem pointer is NULL.
- CV_ILL_INPUT The specified value maxord is ≤ 0, or larger than its previous value.

Notes The default value is ADAMS_Q_MAX = 12 for the Adams-Moulton method and BDF_Q_MAX = 5 for the BDF method. Since maxord affects the memory requirements for the internal cvode memory block, its value cannot be increased past its previous value.

An input value greater than the default will result in the default value.

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

```c
CVodeSetMaxNumSteps
```

Call

```c
flag = CVodeSetMaxNumSteps(cvode_mem, mxsteps);
```

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

Arguments

- `cvode_mem` (void *) pointer to the CVODE memory block.
- `mxsteps` (long int) maximum allowed number of steps.

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

- CV_SUCCESS The optional value has been successfully set.
- CV_MEM_NULL The cvode_mem pointer is NULL.

Notes Passing `mxsteps` = 0 results in CVODE using the default value (500).

Passing `mxsteps` < 0 disables the test (not recommended).

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

```c
CVodeSetMaxHnilWarns
```

Call

```c
flag = CVodeSetMaxHnilWarns(cvode_mem, mxhnil);
```

Description The function CVodeSetMaxHnilWarns specifies the maximum number of messages issued by the solver warning that \( t + h = t \) on the next internal step.

Arguments

- `cvode_mem` (void *) pointer to the CVODE memory block.
- `mxhnil` (int) maximum number of warning messages (> 0).

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

CV_SUCCESS  The optional value has been successfully set.
CV_MEM_NULL The cvode_mem pointer is NULL.

Notes  The default value is 10. A negative value for mxhnil indicates that no warning messages should be issued.

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

```c
CVodeSetStabLimDet
```

Call  

```c
flag = CVodeSetStabLimDet(cvode_mem, stldet);
```

Description  The function CVodeSetStabLimDet indicates if the BDF stability limit detection algorithm should be used. See §2.3 for further details.

Arguments  

cvode_mem (void *) pointer to the CVODE memory block.

stldet (booleantype) flag controlling stability limit detection (SUNTRUE = on; SUNFALSE = off).

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CV_SUCCESS</td>
<td>The optional value has been successfully set.</td>
</tr>
<tr>
<td>CV_MEM_NULL</td>
<td>The cvode_mem pointer is NULL.</td>
</tr>
<tr>
<td>CV_ILL_INPUT</td>
<td>The linear multistep method is not set to CV_BDF.</td>
</tr>
</tbody>
</table>

Notes  The default value is SUNFALSE. If stldet = SUNTRUE when BDF is used and the method order is greater than or equal to 3, then an internal function, CVsldet, is called to detect a possible stability limit. If such a limit is detected, then the order is reduced.

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

```c
CVodeSetInitStep
```

Call  

```c
flag = CVodeSetInitStep(cvode_mem, hin);
```

Description  The function CVodeSetInitStep specifies the initial step size.

Arguments  

cvode_mem (void *) pointer to the CVODE memory block.

hin (realtype) value of the initial step size to be attempted. Pass 0.0 to use the default value.

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CV_SUCCESS</td>
<td>The optional value has been successfully set.</td>
</tr>
<tr>
<td>CV_MEM_NULL</td>
<td>The cvode_mem pointer is NULL.</td>
</tr>
</tbody>
</table>

Notes  By default, CVODE estimates the initial step size to be the solution $h$ of the equation 

$$
\|0.5h^2\ddot{y}\|_{\text{WRMS}} = 1,
$$

where $\ddot{y}$ is an estimated second derivative of the solution at $t_0$.

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

```c
CVodeSetMinStep
```

Call  

```c
flag = CVodeSetMinStep(cvode_mem, hmin);
```

Description  The function CVodeSetMinStep specifies a lower bound on the magnitude of the step size.

Arguments  

cvode_mem (void *) pointer to the CVODE memory block.

hmin (realtype) minimum absolute value of the step size ($\geq 0.0$).

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CV_SUCCESS</td>
<td>The optional value has been successfully set.</td>
</tr>
<tr>
<td>CV_MEM_NULL</td>
<td>The cvode_mem pointer is NULL.</td>
</tr>
</tbody>
</table>

Notes  By default, CVODE estimates the initial step size to be the solution $h$ of the equation 

$$
\|0.5h^2\ddot{y}\|_{\text{WRMS}} = 1,
$$

where $\ddot{y}$ is an estimated second derivative of the solution at $t_0$. 

F2003 Name  This function is callable as FCVodeSetMinStep when using the Fortran 2003 interface module.
CV_SUCCESS  The optional value has been successfully set.
CV_MEM_NULL  The cvode_mem pointer is NULL.
CV_IILL_INPUT  Either hmin is nonpositive or it exceeds the maximum allowable step size.

Notes  The default value is 0.0.

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

```c
CVodeSetMaxStep
```

Call  `flag = CVodeSetMaxStep(cvode_mem, hmax);`

Description  The function `CVodeSetMaxStep` specifies an upper bound on the magnitude of the step size.

Arguments  `cvode_mem` (void *) pointer to the CVODE memory block.
`hmax` (realtype) maximum absolute value of the step size (\( \geq 0.0 \)).

Return value  The return value `flag` (of type `int`) is one of
- CV_SUCCESS  The optional value has been successfully set.
- CV_MEM_NULL  The cvode_mem pointer is NULL.
- CV_IILL_INPUT  Either hmax is nonpositive or it is smaller than the minimum allowable step size.

Notes  Pass `hmax` = 0.0 to obtain the default value \( \infty \).

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

```c
CVodeSetStopTime
```

Call  `flag = CVodeSetStopTime(cvode_mem, tstop);`

Description  The function `CVodeSetStopTime` specifies the value of the independent variable \( t \) past which the solution is not to proceed.

Arguments  `cvode_mem` (void *) pointer to the CVODE 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
- CV_SUCCESS  The optional value has been successfully set.
- CV_MEM_NULL  The cvode_mem pointer is NULL.
- CV_IILL_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 `CVodeSetStopTime`).

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

```c
CVodeSetMaxErrTestFails
```

Call  `flag = CVodeSetMaxErrTestFails(cvode_mem, maxnef);`

Description  The function `CVodeSetMaxErrTestFails` specifies the maximum number of error test failures permitted in attempting one step.

Arguments  `cvode_mem` (void *) pointer to the CVODE 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

- CV_SUCCESS The optional value has been successfully set.
- CV_MEM_NULL The cvode_mem pointer is NULL.

Notes The default value is 7.

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

---

**CVodeSetMaxNonlinIters**

Call `flag = CVodeSetMaxNonlinIters(cvode_mem, maxcor);`

Description The function `CVodeSetMaxNonlinIters` specifies the maximum number of nonlinear solver iterations permitted per step.

Arguments `cvode_mem` (void *) pointer to the CVODE memory block.
`maxcor` (int) maximum number of nonlinear solver iterations allowed per step (> 0).

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

- CV_SUCCESS The optional value has been successfully set.
- CV_MEM_NULL The cvode_mem pointer is NULL.
- CV_memFAIL The SUNNONLINSOL module is NULL.

Notes The default value is 3.

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

---

**CVodeSetMaxConvFails**

Call `flag = CVodeSetMaxConvFails(cvode_mem, maxncf);`

Description The function `CVodeSetMaxConvFails` specifies the maximum number of nonlinear solver convergence failures permitted during one step.

Arguments `cvode_mem` (void *) pointer to the CVODE memory block.
`maxncf` (int) maximum number of allowable nonlinear solver convergence failures per step (> 0).

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

- CV_SUCCESS The optional value has been successfully set.
- CV_MEM_NULL The cvode_mem pointer is NULL.

Notes The default value is 10.

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

---

**CVodeSetNonlinConvCoef**

Call `flag = CVodeSetNonlinConvCoef(cvode_mem, nlscoef);`

Description The function `CVodeSetNonlinConvCoef` specifies the safety factor used in the nonlinear convergence test (see §2.1).

Arguments `cvode_mem` (void *) pointer to the CVODE memory block.
`nlscoef` (realtype) coefficient in nonlinear convergence test (> 0.0).

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

- CV_SUCCESS The optional value has been successfully set.
- CV_MEM_NULL The cvode_mem pointer is NULL.
48 Using CVODE for C Applications

Notes

The default value is 0.1.

F2003 Name

This function is callable as FCVodeSetNonlinConvCoef when using the Fortran 2003 interface module.

**CVodeSetConstraints**

Call

```c
flag = CVodeSetConstraints(cvode_mem, constraints);
```

Description

The function `CVodeSetConstraints` specifies a vector defining inequality constraints for each component of the solution vector $y$.

Arguments

- `cvode_mem` (void *) pointer to the CVODE 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

- CV_SUCCESS The optional value has been successfully set.
- CV_MEM_NULL The `cvode_mem` pointer is NULL.
- CV_ILL_INPUT The constraints vector contains illegal values.

Notes

The presence of a non-NULL constraints vector that is not 0 in all components will cause constraint checking to be performed. However, a call with 0 in all components of `constraints` will result in an illegal input return. A NULL constraints vector will disable constraint checking.

F2003 Name

This function is callable as FCVodeSetConstraints when using the Fortran 2003 interface module.

4.5.7.2 Linear solver interface optional input functions

The mathematical explanation of the linear solver methods available to CVODE is provided in §2.1. We group the user-callable routines into four categories: general routines concerning the overall cvls 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.

As discussed in §2.1, CVODE strives to reuse matrix and preconditioner data for as many solves as possible to amortize the high costs of matrix construction and factorization. To that end, CVODE provides a user-callable routine to modify this behavior. To this end, we recall that the Newton system matrices are $M(t, y) = I - \gamma J(t, y)$, where the right-hand side function has Jacobian matrix $J(t, y) = \frac{\partial f(t, y)}{\partial y}$.

The matrix or preconditioner for $M$ can only be updated within a call to the linear solver ‘setup’ routine. In general, the frequency with which this setup routine is called may be controlled with the msbj argument to CVodeSetMaxStepsBetweenJac.

**CVodeSetMaxStepsBetweenJac**

Call

```c
retval = CVodeSetMaxStepsBetweenJac(cvode_mem, msbj);
```

Description

The function `CVodeSetMaxStepsBetweenJac` specifies the maximum number of time steps to wait before recomputation of the Jacobian or recommendation to update the preconditioner.
4.5 User-callable functions

Arguments  
- cvode_mem (void *) pointer to the CVODE memory block.
- msbj (long int) maximum number of time steps to wait before Jacobian/preconditioner reconstruction.

Return value  
The return value flag (of type int) is one of
- CVLS_SUCCESS The optional value has been successfully set.
- CVLS_MEM_NULL The cvode_mem pointer is NULL.
- CVLS_LMEM_NULL The cvls linear solver interface has not been initialized.

Notes  
- If msbj is less than 1, the default value of 50 will be used.
- This function must be called after the cvls linear solver interface has been initialized through a call to CVodeSetLinearSolver.

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

When using matrix-based linear solver modules, the CVLS solver interface needs a function to compute an approximation to the Jacobian matrix $J(t, y)$. This function must be of type CVLsJacFn. 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 CVLS solver. To specify a user-supplied Jacobian function jac, CVLS provides the function CVodeSetJacFn. The CVLS 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 CVodeSetUserData.

\[ CVodeSetJacFn \]

Call  
\[ flag = CVodeSetJacFn(cvode_mem, jac); \]

Description  
The function CVodeSetJacFn specifies the Jacobian approximation function to be used for a matrix-based solver within the CVLS interface.

Arguments  
- cvode_mem (void *) pointer to the CVODE memory block.
- jac (CVLsJacFn) user-defined Jacobian approximation function.

Return value  
The return value flag (of type int) is one of
- CVLS_SUCCESS The optional value has been successfully set.
- CVLS_MEM_NULL The cvode_mem pointer is NULL.
- CVLS_LMEM_NULL The CVLS linear solver interface has not been initialized.

Notes  
- This function must be called after the CVLS linear solver interface has been initialized through a call to CVodeSetLinearSolver.
- By default, CVLS 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 CVLsJacFn is described in §4.6.5.
- The previous routine CVDlsSetJacFn 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.

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

When using matrix-free linear solver modules, the CVLS 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 CVLS interface. A user-defined Jacobian-vector function must be of type CVLsJacTimesVecFn and can be specified through a call to CVodeSetJacTimes (see §4.6.6 for
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 \texttt{jtsetup} (see §4.6.7 for specification details).

The pointer \texttt{user\_data} received through \texttt{CVodeSetUserData} (or a pointer to NULL if \texttt{user\_data} was not specified) is passed to the Jacobian-times-vector setup and product functions, \texttt{jtsetup} and \texttt{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.

\begin{verbatim}
CVodeSetJacTimes
Call flag = CVodeSetJacTimes(cvode\_mem, jtsetup, jtimes);
Description The function CVSetJacTimes specifies the Jacobian-vector setup and product functions.
Arguments cvode\_mem (void *) pointer to the CVODE memory block.
jtsetup (CVLsJacTimesSetupFn) user-defined Jacobian-vector setup function. Pass NULL if no setup is necessary.
jtimes (CVLsJacTimesVecFn) user-defined Jacobian-vector product function.
Return value The return value flag (of type int) is one of
CVLS\_SUCCESS The optional value has been successfully set.
CVLS\_MEM\_NULL The cvode\_mem pointer is NULL.
CVLS\_LMEM\_NULL The CVLS linear solver has not been initialized.
CVLS\_SUNLINSOL\_FAIL An error occurred when setting up the system matrix-times-vector routines in the SUNLINSOL object used by the CVLS interface.
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 CVLS linear solver interface has been initialized through a call to CVodeSetLinearSolver.
The function type CVLsJacTimesSetupFn is described in §4.6.7.
The function type CVLsJacTimesVecFn is described in §4.6.6.
The previous routine CVSpilsSetJacTimes 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.
F2003 Name This function is callable as FCVodeSetJacTimes when using the Fortran 2003 interface module.
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, \texttt{psetup} and \texttt{psolve}, that are supplied to CVODE using the function \texttt{CVodeSetPreconditioner}. The \texttt{psetup} function supplied to this routine should handle evaluation and preprocessing of any Jacobian data needed by the user’s preconditioner solve function, \texttt{psolve}. The user data pointer received through \texttt{CVodeSetUserData} (or a pointer to NULL if user data was not specified) is passed to the \texttt{psetup} and \texttt{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 CVLS 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 \texttt{CVodeSetEpsLin} function.
4.5 User-callable functions

**CVodeSetPreconditioner**

Call

```c
flag = CVodeSetPreconditioner(cvode_mem, psetup, psolve);
```

Description
The function CVodeSetPreconditioner specifies the preconditioner setup and solve functions.

Arguments
- `cvode_mem` (void *) pointer to the CVODE memory block.
- `psetup` (CVLsPrecSetupFn) user-defined preconditioner setup function. Pass NULL if no setup is necessary.
- `psolve` (CVLsPrecSolveFn) user-defined preconditioner solve function.

Return value
The return value `flag` (of type int) is one of
- `CVLS_SUCCESS` The optional values have been successfully set.
- `CVLS_MEM_NULL` The `cvode_mem` pointer is NULL.
- `CVLS_LMEM_NULL` The CVLS linear solver has not been initialized.
- `CVLS_SUNLINS_FAIL` An error occurred when setting up preconditioning in the SUNLINSOL object used by the CVLS interface.

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

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

The function type CVLsPrecSolveFn is described in §4.6.8.

The function type CVLsPrecSetupFn is described in §4.6.9.

The previous routine CVSpilsSetPreconditioner 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.

**F2003 Name**
This function is callable as FCVodeSetPreconditioner when using the Fortran 2003 interface module.

**CVodeSetEpsLin**

Call

```c
flag = CVodeSetEpsLin(cvode_mem, eplifac);
```

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

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

Return value
The return value `flag` (of type int) is one of
- `CVLS_SUCCESS` The optional value has been successfully set.
- `CVLS_MEM_NULL` The `cvode_mem` pointer is NULL.
- `CVLS_LMEM_NULL` The CVLS linear solver has not been initialized.
- `CVLS_SUNLINS_FAIL` The factor `eplifac` is negative.

Notes
The default value is 0.05.

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

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

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

**F2003 Name**
This function is callable as FCVodeSetEpsLin when using the Fortran 2003 interface module.
4.5.7.3 Rootfinding optional input functions

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

**CVodeSetRootDirection**

Call

```c
flag = CVodeSetRootDirection(cvode_mem, rootdir);
```

Description

The function `CVodeSetRootDirection` specifies the direction of zero-crossings to be located and returned.

Arguments

- `cvode_mem` (void *) pointer to the CVODE memory block.
- `rootdir` (int *) state array of length `nrtfn`, the number of root functions $g_i$, as specified in the call to the function `CVodeRootInit`. A value of 0 for `rootdir[i]` indicates that crossing in either direction for $g_i$ should be reported. 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

- `CV_SUCCESS` The optional value has been successfully set.
- `CV_MEM_NULL` The `cvode_mem` pointer is NULL.
- `CV_IILL_INPUT` rootfinding has not been activated through a call to `CVodeRootInit`.

Notes

The default behavior is to monitor for both zero-crossing directions.

F2003 Name

This function is callable as `FCVodeSetRootDirection` when using the Fortran 2003 interface module.

**CVodeSetNoInactiveRootWarn**

Call

```c
flag = CVodeSetNoInactiveRootWarn(cvode_mem);
```

Description

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

Arguments

- `cvode_mem` (void *) pointer to the CVODE memory block.

Return value

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

- `CV_SUCCESS` The optional value has been successfully set.
- `CV_MEM_NULL` The `cvode_mem` pointer is NULL.

Notes

`CVODE` 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), `CVODE` will issue a warning which can be disabled with this optional input function.

F2003 Name

This function is callable as `FCVodeSetNoInactiveRootWarn` when using the Fortran 2003 interface module.

4.5.8 Interpolated output function

An optional function `CVodeGetDky` is available to obtain additional output values. This function should only be called after a successful return from `CVode` as it provides interpolated values either of $y$ or of its derivatives (up to the current order of the integration method) interpolated to any value of $t$ in the last internal step taken by `CVODE`.

The call to the `CVodeGetDky` function has the following form:
4.5 User-callable functions

**CVodeGetDky**

Call

```c
flag = CVodeGetDky(cvode_mem, t, k, dky);
```

Description

The function `CVodeGetDky` computes the $k$-th derivative of the function $y$ at time $t$, i.e. $d^{(k)}y/dt^{(k)}(t)$, where $t_n-h_u \leq t \leq t_n$, $t_n$ denotes the current internal time reached, and $h_u$ is the last internal step size successfully used by the solver. The user may request $k = 0, 1, \ldots, q_u$, where $q_u$ is the current order (optional output `qlast`).

Arguments

- `cvode_mem` (void *) pointer to the CVODE memory block.
- `t` (realtype) the value of the independent variable at which the derivative is to be evaluated.
- `k` (int) the derivative order requested.
- `dky` (N_Vector) vector containing the derivative. This vector must be allocated by the user.

Return value

- `CV_SUCCESS` `CVodeGetDky` succeeded.
- `CV_BAD_K` $k$ is not in the range $0, 1, \ldots, q_u$.
- `CV_BAD_T` $t$ is not in the interval $[t_n - h_u, t_n]$.
- `CV_BAD_DKY` The `dky` argument was NULL.
- `CV_MEM_NULL` The `cvode_mem` argument was NULL.

Notes

- It is only legal to call the function `CVodeGetDky` after a successful return from `CVode`.
- See `CVodeGetCurrentTime`, `CVodeGetLastOrder`, and `CVodeGetLastStep` in the next section for access to $t_n$, $q_u$, and $h_u$, respectively.

F2003 Name

This function is callable as `FCVodeGetDky` when using the Fortran 2003 interface module.

4.5.9 Optional output functions

CVODE provides an extensive set of functions that can be used to obtain solver performance information. Table 4.3 lists all optional output functions in CVODE, 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 CVODE solver is in doing its job. For example, the counters `nsteps` and `nfevals` 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.

4.5.9.1 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.
<table>
<thead>
<tr>
<th>Optional output</th>
<th>Function name</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CVODE main solver</strong></td>
<td></td>
</tr>
<tr>
<td>Size of CVODE real and integer workspaces</td>
<td>CVodeGetWorkSpace</td>
</tr>
<tr>
<td>Cumulative number of internal steps</td>
<td>CVodeGetNumSteps</td>
</tr>
<tr>
<td>No. of calls to r.h.s. function</td>
<td>CVodeGetNumRhsEvals</td>
</tr>
<tr>
<td>No. of calls to linear solver setup function</td>
<td>CVodeGetNumLinSolvSetups</td>
</tr>
<tr>
<td>No. of local error test failures that have occurred</td>
<td>CVodeGetNumErrTestFails</td>
</tr>
<tr>
<td>Order used during the last step</td>
<td>CVodeGetLastOrder</td>
</tr>
<tr>
<td>Order to be attempted on the next step</td>
<td>CVodeGetCurrentOrder</td>
</tr>
<tr>
<td>No. of order reductions due to stability limit detection</td>
<td>CVodeGetNumStabLimOrderReds</td>
</tr>
<tr>
<td>Actual initial step size used</td>
<td>CVodeGetActualInitStep</td>
</tr>
<tr>
<td>Step size used for the last step</td>
<td>CVodeGetLastStep</td>
</tr>
<tr>
<td>Step size to be attempted on the next step</td>
<td>CVodeGetCurrentStep</td>
</tr>
<tr>
<td>Current internal time reached by the solver</td>
<td>CVodeGetCurrentTime</td>
</tr>
<tr>
<td>Suggested factor for tolerance scaling</td>
<td>CVodeGetTolScaleFactor</td>
</tr>
<tr>
<td>Error weight vector for state variables</td>
<td>CVodeGetErrWeights</td>
</tr>
<tr>
<td>Estimated local error vector</td>
<td></td>
</tr>
<tr>
<td>No. of nonlinear solver iterations</td>
<td>CVodeGetNumNonlinSolvIters</td>
</tr>
<tr>
<td>No. of nonlinear convergence failures</td>
<td>CVodeGetNumNonlinSolvConvFails</td>
</tr>
<tr>
<td>All CVODE integrator statistics</td>
<td>CVodeGetIntegratorStats</td>
</tr>
<tr>
<td>CVODE nonlinear solver statistics</td>
<td>CVodeGetNonlinSolvStats</td>
</tr>
<tr>
<td>Array showing roots found</td>
<td>CvodeGetRootInfo</td>
</tr>
<tr>
<td>No. of calls to user root function</td>
<td>CVodeGetNumGEvals</td>
</tr>
<tr>
<td>Name of constant associated with a return flag</td>
<td>CVodeGetReturnFlagName</td>
</tr>
<tr>
<td><strong>CVLS linear solver interface</strong></td>
<td></td>
</tr>
<tr>
<td>Size of real and integer workspaces</td>
<td>CVodeGetLinWorkSpace</td>
</tr>
<tr>
<td>No. of Jacobian evaluations</td>
<td>CVodeGetNumJacEvals</td>
</tr>
<tr>
<td>No. of r.h.s. calls for finite diff. Jacobian[-vector] evals.</td>
<td>CVodeGetNumLinRhsEvals</td>
</tr>
<tr>
<td>No. of linear iterations</td>
<td>CVodeGetNumLinIters</td>
</tr>
<tr>
<td>No. of linear convergence failures</td>
<td>CVodeGetNumLinConvFails</td>
</tr>
<tr>
<td>No. of preconditioner evaluations</td>
<td>CVodeGetNumPrecEvals</td>
</tr>
<tr>
<td>No. of preconditioner solves</td>
<td>CVodeGetNumPrecSolves</td>
</tr>
<tr>
<td>No. of Jacobian-vector setup evaluations</td>
<td>CVodeGetNumJTSetupEvals</td>
</tr>
<tr>
<td>No. of Jacobian-vector product evaluations</td>
<td>CVodeGetNumJtimesEvals</td>
</tr>
<tr>
<td>Last return from a linear solver function</td>
<td>CVodeGetLastLinFlag</td>
</tr>
<tr>
<td>Name of constant associated with a return flag</td>
<td>CVodeGetLinReturnFlagName</td>
</tr>
<tr>
<td><strong>CVDIAG linear solver interface</strong></td>
<td></td>
</tr>
<tr>
<td>Size of CVDIAG real and integer workspaces</td>
<td>CVDiagGetWorkSpace</td>
</tr>
<tr>
<td>No. of r.h.s. calls for finite diff. Jacobian eva.</td>
<td>CVDiagGetNumRhsEvals</td>
</tr>
<tr>
<td>Last return from a CVDIAG function</td>
<td>CVDiagGetLastFlag</td>
</tr>
<tr>
<td>Name of constant associated with a return flag</td>
<td>CVDiagGetReturnFlagName</td>
</tr>
</tbody>
</table>
4.5 User-callable functions

Arguments  

version (char *) character array to hold the SUNDIALS version information.

len (int) allocated length of the version character array.

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.

4.5.9.2 Main solver optional output functions

CVODE 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 CVODE memory block (a suggested tolerance scaling factor, the error weight vector, and the vector of estimated local errors). Functions are also provided to extract statistics related to the performance of the CVODE nonlinear solver used. As a convenience, additional information extraction functions provide the optional outputs in groups. These optional output functions are described next.

CVodeGetWorkSpace

Call  

flag = CVodeGetWorkSpace(cvode_mem, &lenrw, &leniw);

Description  

The function CVodeGetWorkSpace returns the CVODE real and integer workspace sizes.

Arguments  

cvode_mem (void *) pointer to the CVODE memory block.

lenrw (long int) the number of realtype values in the CVODE workspace.

leniw (long int) the number of integer values in the CVODE workspace.

Return value  

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

CV_SUCCESS The optional output values have been successfully set.

CV_MEM_NULL The cvode_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 §4.4.5), the actual size of the real workspace, in realtype words, is given by the following:

- base value: lenrw = 96 + (maxord+5) * N_r + 3*nrtfn;
• using CVodeSVtolerances: lenrw = lenrw + N_r;
• with constraint checking (see CVodeSetConstraints): lenrw = lenrw + N_r;

where N_r is the number of real words in one N_Vector (≈ N).

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

• base value: leniw = 40 + (maxord+5) * N_i + nrtfn;
• using CVodeSVtolerances: leniw = leniw + N_i;
• with constraint checking: leniw = leniw + N_i;

where N_i is the number of integer words in one N_Vector (= 1 for NVECTOR_SERIAL and 2*npes nvector parallel and npes processors).

For the default value of maxord, no rootfinding, no constraints, and without using CVodeSVtolerances, these lengths are given roughly by:

• For the Adams method: lenrw = 96 + 17N and leniw = 57
• For the BDF method: lenrw = 96 + 10N and leniw = 50

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

```
CVodeGetNumSteps
```

Call
flag = CVodeGetNumSteps(cvode_mem, &nsteps);

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

Arguments
- cvode_mem (void *) pointer to the CVODE memory block.
- nsteps (long int) number of steps taken by CVODE.

Return value The return value flag (of type int) is one of
- CV_SUCCESS The optional output value has been successfully set.
- CV_MEM_NULL The cvode_mem pointer is NULL.

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

```
CVodeGetNumRhsEvals
```

Call
flag = CVodeGetNumRhsEvals(cvode_mem, &nfevals);

Description The function CVodeGetNumRhsEvals returns the number of calls to the user’s right-hand side function.

Arguments
- cvode_mem (void *) pointer to the CVODE memory block.
- nfevals (long int) number of calls to the user’s f function.

Return value The return value flag (of type int) is one of
- CV_SUCCESS The optional output value has been successfully set.
- CV_MEM_NULL The cvode_mem pointer is NULL.

Notes The nfevals value returned by CVodeGetNumRhsEvals does not account for calls made to f by a linear solver or preconditioner module.

F2003 Name This function is callable as FCVodeGetNumRhsEvals when using the Fortran 2003 interface module.
4.5 User-callable functions

**CVodeGetNumLinSolvSetups**

**Call**

```c
flag = CVodeGetNumLinSolvSetups(cvode_mem, &nlinsetups);
```

**Description**
The function `CVodeGetNumLinSolvSetups` returns the number of calls made to the linear solver’s setup function.

**Arguments**
- `cvode_mem` (void *) pointer to the CVODE memory block.
- `nlinsetups` (long int) number of calls made to the linear solver setup function.

**Return value**
The return value `flag` (of type `int`) is one of
- `CV_SUCCESS` The optional output value has been successfully set.
- `CV_MEM_NULL` The `cvode_mem` pointer is NULL.

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

**CVodeGetNumErrTestFails**

**Call**

```c
flag = CVodeGetNumErrTestFails(cvode_mem, &netfails);
```

**Description**
The function `CVodeGetNumErrTestFails` returns the number of local error test failures that have occurred.

**Arguments**
- `cvode_mem` (void *) pointer to the CVODE memory block.
- `netfails` (long int) number of error test failures.

**Return value**
The return value `flag` (of type `int`) is one of
- `CV_SUCCESS` The optional output value has been successfully set.
- `CV_MEM_NULL` The `cvode_mem` pointer is NULL.

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

**CVodeGetLastOrder**

**Call**

```c
flag = CVodeGetLastOrder(cvode_mem, &qlast);
```

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

**Arguments**
- `cvode_mem` (void *) pointer to the CVODE memory block.
- `qlast` (int) method order used on the last internal step.

**Return value**
The return value `flag` (of type `int`) is one of
- `CV_SUCCESS` The optional output value has been successfully set.
- `CV_MEM_NULL` The `cvode_mem` pointer is NULL.

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

**CVodeGetCurrentOrder**

**Call**

```c
flag = CVodeGetCurrentOrder(cvode_mem, &qcur);
```

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

**Arguments**
- `cvode_mem` (void *) pointer to the CVODE memory block.
- `qcur` (int) method order to be used on the next internal step.

**Return value**
The return value `flag` (of type `int`) is one of
- `CV_SUCCESS` The optional output value has been successfully set.
CV_MEM_NULL The cvode_mem pointer is NULL.

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

**CVodeGetLastStep**

Call `flag = CVodeGetLastStep(cvode_mem, &hlast);`

Description The function CVodeGetLastStep returns the integration step size taken on the last internal step.

Arguments `cvode_mem` (void *) pointer to the CVODE memory block.
`hlast` (realtype) step size taken on the last internal step.

Return value The return value flag (of type int) is one of
- `CV_SUCCESS` The optional output value has been successfully set.
- `CV_MEM_NULL` The cvode_mem pointer is NULL.

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

**CVodeGetCurrentStep**

Call `flag = CVodeGetCurrentStep(cvode_mem, &hcur);`

Description The function CVodeGetCurrentStep returns the integration step size to be attempted on the next internal step.

Arguments `cvode_mem` (void *) pointer to the CVODE 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
- `CV_SUCCESS` The optional output value has been successfully set.
- `CV_MEM_NULL` The cvode_mem pointer is NULL.

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

**CVodeGetActualInitStep**

Call `flag = CVodeGetActualInitStep(cvode_mem, &hinused);`

Description The function CVodeGetActualInitStep returns the value of the integration step size used on the first step.

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

Return value The return value flag (of type int) is one of
- `CV_SUCCESS` The optional output value has been successfully set.
- `CV_MEM_NULL` The cvode_mem pointer is NULL.

Notes Even if the value of the initial integration step size was specified by the user through a call to CVodeSetInitStep, this value might have been changed by CVODE to ensure that the step size is within the prescribed bounds ($h_{min} \leq h_{0} \leq h_{max}$), or to satisfy the local error test condition.

F2003 Name This function is callable as FCVodeGetActualInitStep when using the Fortran 2003 interface module.
4.5 User-callable functions

**CVodeGetCurrentTime**

Call `flag = CVodeGetCurrentTime(cvode_mem, &tcur);`

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

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

`tcur` (realtype) current internal time reached.

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

- `CV_SUCCESS` The optional output value has been successfully set.
- `CV_MEM_NULL` The `cvode_mem` pointer is NULL.

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

**CVodeGetNumStabLimOrderReds**

Call `flag = CVodeGetNumStabLimOrderReds(cvode_mem, &nslred);`

Description The function `CVodeGetNumStabLimOrderReds` returns the number of order reductions dictated by the BDF stability limit detection algorithm (see §2.3).

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

`nslred` (long int) number of order reductions due to stability limit detection.

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

- `CV_SUCCESS` The optional output value has been successfully set.
- `CV_MEM_NULL` The `cvode_mem` pointer is NULL.

Notes If the stability limit detection algorithm was not initialized (`CVodeSetStabLimDet` was not called), then `nslred = 0`.

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

**CVodeGetTolScaleFactor**

Call `flag = CVodeGetTolScaleFactor(cvode_mem, &tolsfac);`

Description The function `CVodeGetTolScaleFactor` 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 `cvode_mem` (void *) pointer to the CVODE memory block.

`tolsfac` (realtype) suggested scaling factor for user-supplied tolerances.

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

- `CV_SUCCESS` The optional output value has been successfully set.
- `CV_MEM_NULL` The `cvode_mem` pointer is NULL.

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

**CVodeGetErrWeights**

Call `flag = CVodeGetErrWeights(cvode_mem, eweight);`

Description The function `CVodeGetErrWeights` returns the solution error weights at the current time. These are the reciprocals of the $W_i$ given by (2.7).

Arguments `cvode_mem` (void *) pointer to the CVODE memory block.
eweight (N_Vector) solution error weights at the current time.

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

- CV_SUCCESS The optional output value has been successfully set.
- CV_MEM_NULL The cprob_mem pointer is NULL.

Notes The user must allocate memory for eweight.

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

CVodeGetEstLocalErrors

Call flag = CVodeGetEstLocalErrors(cprob_mem, ele);

Description The function CVodeGetEstLocalErrors returns the vector of estimated local errors.

Arguments
cprob_mem (void *) pointer to the CVODE memory block.
ele (N_Vector) estimated local errors.

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

- CV_SUCCESS The optional output value has been successfully set.
- CV_MEM_NULL The cprob_mem pointer is NULL.

Notes The user must allocate memory for ele.

The values returned in ele are valid only if CVode returned a non-negative value.

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

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

CVodeGetIntegratorStats

Call flag = CVodeGetIntegratorStats(cprob_mem, &nsteps, &nfevals,
&nlinsetups, &netfails, &qlast, &qcur,
&hinused, &hlast, &hcur, &tcur);

Description The function CVodeGetIntegratorStats returns the CVODE integrator statistics as a group.

Arguments
cprob_mem (void *) pointer to the CVODE memory block.
nsteps (long int) number of steps taken by CVODE.
nfevals (long int) number of calls to the user's f function.
nlinsetups (long int) number of calls made to the linear solver setup function.
netfails (long int) number of error test failures.
qlast (int) method order used on the last internal step.
qcur (int) method order to be used on the next internal step.
hinused (realttype) actual value of initial step size.
hlast (realttype) step size taken on the last internal step.
hcur (realttype) step size to be attempted on the next internal step.
tcur (realttype) current internal time reached.

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

CV_SUCCESS  the optional output values have been successfully set.
CV_MEM_NULL the cvode_mem pointer is NULL.

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

**CVodeGetNumNonlinSolvIters**

Call  flag = CVodeGetNumNonlinSolvIters(cvode_mem, &nniters);

Description The function CVodeGetNumNonlinSolvIters returns the number of nonlinear iterations performed.

Arguments  cvode_mem (void *) pointer to the CVODE memory block.
niters (long int) number of nonlinear iterations performed.

Return value The return value flag (of type int) is one of
- CV_SUCCESS  the optional output values have been successfully set.
- CV_MEM_NULL the cvode_mem pointer is NULL.
- CV_MEM_FAIL the SUNNONLINSOL module is NULL.

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

**CVodeGetNumNonlinSolvConvFails**

Call  flag = CVodeGetNumNonlinSolvConvFails(cvode_mem, &nncfails);

Description The function CVodeGetNumNonlinSolvConvFails returns the number of nonlinear convergence failures that have occurred.

Arguments  cvode_mem (void *) pointer to the CVODE memory block.
nncfails (long int) number of nonlinear convergence failures.

Return value The return value flag (of type int) is one of
- CV_SUCCESS  the optional output value has been successfully set.
- CV_MEM_NULL the cvode_mem pointer is NULL.

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

**CVodeGetNonlinSolvStats**

Call  flag = CVodeGetNonlinSolvStats(cvode_mem, &nniters, &nncfails);

Description The function CVodeGetNonlinSolvStats returns the CVODE nonlinear solver statistics as a group.

Arguments  cvode_mem (void *) pointer to the CVODE memory block.
niters (long int) number of nonlinear iterations performed.
nncfails (long int) number of nonlinear convergence failures.

Return value The return value flag (of type int) is one of
- CV_SUCCESS  the optional output value has been successfully set.
- CV_MEM_NULL the cvode_mem pointer is NULL.
- CV_MEM_FAIL the SUNNONLINSOL module is NULL.

F2003 Name This function is callable as FCVodeGetNonlinSolvStats when using the Fortran 2003 interface module.
CVodeGetReturnFlagName

Call
name = CVodeGetReturnFlagName(flag);

Description
The function CVodeGetReturnFlagName returns the name of the CVODE constant corresponding to flag.

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

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

4.5.9.3 Rootfinding optional output functions

There are two optional output functions associated with rootfinding.

CVodeGetRootInfo

Call
flag = CVodeGetRootInfo(cvode_mem, rootsfound);

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

Arguments
cvode_mem (void *) pointer to the CVODE memory block.
rootsfound (int *) array of length nrtfn with the indices of the user functions gi found to have a root. For i = 0,...,nrtfn−1, rootsfound[i]≠ 0 if gi has a root, and = 0 if not.

Return value
The return value flag (of type int) is one of:
CV_SUCCESS The optional output values have been successfully set.
CV_MEM_NULL The cvode_mem pointer is NULL.

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

The user must allocate memory for the vector rootsfound.

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

CVodeGetNumGEvals

Call
flag = CVodeGetNumGEvals(cvode_mem, &ngevals);

Description
The function CVodeGetNumGEvals returns the cumulative number of calls made to the user-supplied root function g.

Arguments
cvode_mem (void *) pointer to the CVODE memory block.
ungevals (long int) number of calls made to the user’s function g thus far.

Return value
The return value flag (of type int) is one of:
CV_SUCCESS The optional output value has been successfully set.
CV_MEM_NULL The cvode_mem pointer is NULL.

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

4.5.9.4 CVLS linear solver interface optional output functions

The following optional outputs are available from the CVLS modules: workspace requirements, number of calls to the Jacobian routine, number of calls to the right-hand side 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...
4.5 User-callable functions

to the Jacobian-vector setup and product routines, and last return value from a linear solver function.
Note that, where the name of an output would otherwise conflict with the name of an optional output
from the main solver, a suffix LS (for Linear Solver) has been added (e.g. lenrwLS).

**CVodeGetLinWorkSpace**

Call

```c
flag = CVodeGetLinWorkSpace(cvode_mem, &lenrwLS, &leniwLS);
```

Description

The function CVodeGetLinWorkSpace returns the sizes of the real and integer workspaces
used by the CVLS linear solver interface.

Arguments

- `cvode_mem` (void *) pointer to the CVODE memory block.
- `lenrwLS` (long int) the number of realtype values in the CVLS workspace.
- `leniwLS` (long int) the number of integer values in the CVLS workspace.

Return value

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

- `CVLS_SUCCESS` The optional output values have been successfully set.
- `CVLS_MEM_NULL` The `cvode_mem` pointer is NULL.
- `CVLS_LMEM_NULL` The CVLS linear solver has not been initialized.

Notes

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

The previous routines CVDlsGetWorkspace and CVSpilsGetWorkspace are now wrap-
pers 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.

F2003 Name

This function is callable as FCVodeGetLinWorkSpace when using the Fortran 2003 inter-
face module.

**CVodeGetNumJacEvals**

Call

```c
flag = CVodeGetNumJacEvals(cvode_mem, &njevals);
```

Description

The function CVodeGetNumJacEvals returns the number of calls made to the CVLS
Jacobian approximation function.

Arguments

- `cvode_mem` (void *) pointer to the CVODE memory block.
- `njevals` (long int) the number of calls to the Jacobian function.

Return value

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

- `CVLS_SUCCESS` The optional output value has been successfully set.
- `CVLS_MEM_NULL` The `cvode_mem` pointer is NULL.
- `CVLS_LMEM_NULL` The CVLS linear solver has not been initialized.

Notes

The previous routine CVDlsGetNumJacEvals is now a wrapper 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.

F2003 Name

This function is callable as FCVodeGetNumJacEvals when using the Fortran 2003 inter-
face module.

**CVodeGetNumLinRhsEvals**

Call

```c
flag = CVodeGetNumLinRhsEvals(cvode_mem, &nfevalsLS);
```

Description

The function CVodeGetNumLinRhsEvals returns the number of calls made to the user-
supplied right-hand side function due to the finite difference Jacobian approximation or
finite difference Jacobian-vector product approximation.
Arguments

- `cvode_mem` (void *) pointer to the CVODE memory block.
- `nfevalsLS` (long int) the number of calls made to the user-supplied right-hand side function.

Return value

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

- `CVLS_SUCCESS` The optional output value has been successfully set.
- `CVLS_MEM_NULL` The `cvode_mem` pointer is NULL.
- `CVLS_LMEM_NULL` The CVLS linear solver has not been initialized.

Notes

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

The previous routines `CVDlsGetNumRhsEvals` and `CVSpilsGetNumRhsEvals` 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.

F2003 Name

This function is callable as `FCVodeGetNumLinRhsEvals` when using the Fortran 2003 interface module.

---

**CVodeGetNumLinIters**

Call

```c
flag = CVodeGetNumLinIters(cvode_mem, &nliters);
```

Description

The function `CVodeGetNumLinIters` returns the cumulative number of linear iterations.

Arguments

- `cvode_mem` (void *) pointer to the CVODE memory block.
- `nliters` (long int) the current number of linear iterations.

Return value

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

- `CVLS_SUCCESS` The optional output value has been successfully set.
- `CVLS_MEM_NULL` The `cvode_mem` pointer is NULL.
- `CVLS_LMEM_NULL` The CVLS linear solver has not been initialized.

Notes

The previous routine `CVSpilsGetNumLinIters` 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.

F2003 Name

This function is callable as `FCVodeGetNumLinIters` when using the Fortran 2003 interface module.

---

**CVodeGetNumLinConvFails**

Call

```c
flag = CVodeGetNumLinConvFails(cvode_mem, &nlcfails);
```

Description

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

Arguments

- `cvode_mem` (void *) pointer to the CVODE memory block.
- `nlcfails` (long int) the current number of linear convergence failures.

Return value

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

- `CVLS_SUCCESS` The optional output value has been successfully set.
- `CVLS_MEM_NULL` The `cvode_mem` pointer is NULL.
- `CVLS_LMEM_NULL` The CVLS linear solver has not been initialized.

Notes

The previous routine `CVSpilsGetNumConvFails` 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.

F2003 Name

This function is callable as `FCVodeGetNumLinConvFails` when using the Fortran 2003 interface module.
4.5 User-callable functions

**CVodeGetNumPrecEvals**

Call

```c
flag = CVodeGetNumPrecEvals(cvode_mem, &npevals);
```

Description

The function `CVodeGetNumPrecEvals` returns the number of preconditioner evaluations, i.e., the number of calls made to `psetup` with `jok = SUNFALSE`.

Arguments

- `cvode_mem` (void *) pointer to the CVODE memory block.
- `npevals` (long int) the current number of calls to `psetup`.

Return value

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

- `CVLS_SUCCESS` The optional output value has been successfully set.
- `CVLS_MEM_NULL` The `cvode_mem` pointer is NULL.
- `CVLS_LMEM_NULL` The CVLS linear solver has not been initialized.

Notes

The previous routine `CVSpilsGetNumPrecEvals` 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.

F2003 Name

This function is callable as `FCVodeGetNumPrecEvals` when using the Fortran 2003 interface module.

**CVodeGetNumPrecSolves**

Call

```c
flag = CVodeGetNumPrecSolves(cvode_mem, &npsolves);
```

Description

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

Arguments

- `cvode_mem` (void *) pointer to the CVODE memory block.
- `npsolves` (long int) the current number of calls to `psolve`.

Return value

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

- `CVLS_SUCCESS` The optional output value has been successfully set.
- `CVLS_MEM_NULL` The `cvode_mem` pointer is NULL.
- `CVLS_LMEM_NULL` The CVLS linear solver has not been initialized.

Notes

The previous routine `CVSpilsGetNumPrecSolves` 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.

F2003 Name

This function is callable as `FCVodeGetNumPrecSolves` when using the Fortran 2003 interface module.

**CVodeGetNumJTSetupEvals**

Call

```c
flag = CVodeGetNumJTSetupEvals(cvode_mem, &njtsetup);
```

Description

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

Arguments

- `cvode_mem` (void *) pointer to the CVODE memory block.
- `njtsetup` (long int) the current number of calls to `jtsetup`.

Return value

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

- `CVLS_SUCCESS` The optional output value has been successfully set.
- `CVLS_MEM_NULL` The `cvode_mem` pointer is NULL.
- `CVLS_LMEM_NULL` The CVLS linear solver has not been initialized.

Notes

The previous routine `CVSpilsGetNumJTSetupEvals` 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.
F2003 Name  This function is callable as FCVodeGetNumJtimesEvals when using the Fortran 2003 interface module.

```c
CVodeGetNumJtimesEvals
```

Call  

```c
flag = CVodeGetNumJtimesEvals(cvode_mem, &njvevals);
```

Description  

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

Arguments  

- `cvode_mem` (void *) pointer to the CVODE memory block.
- `njvevals` (long int) the current number of calls to `jtimes`.

Return value  

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

- `CVLS_SUCCESS` The optional output value has been successfully set.
- `CVLS_MEM_NULL` The `cvode_mem` pointer is `NULL`.
- `CVLS_LMEM_NULL` The CVLS linear solver has not been initialized.

Notes  

The previous routine `CVSpilsGetNumJtimesEvals` 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.

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

```c
CVodeGetLastLinFlag
```

Call  

```c
flag = CVodeGetLastLinFlag(cvode_mem, &lsflag);
```

Description  

The function `CVodeGetLastLinFlag` returns the last return value from a CVLS routine.

Arguments  

- `cvode_mem` (void *) pointer to the CVODE memory block.
- `lsflag` (long int) the value of the last return flag from a CVLS function.

Return value  

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

- `CVLS_SUCCESS` The optional output value has been successfully set.
- `CVLS_MEM_NULL` The `cvode_mem` pointer is `NULL`.
- `CVLS_LMEM_NULL` The CVLS linear solver has not been initialized.

Notes  

If the CVLS setup function failed (i.e., `CVode` returned `CVLSSETUP_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 CVLS 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 CVLS solve function failed (i.e., `CVode` returned `CVLSOLVE_FAIL`), then `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 `Jv` function; `SUNLS_PGSOLVE_FAIL_UNREC`, indicating that the preconditioner solve function `psolve` failed unrecoverably; `SUNLS_CLS_FAIL`, indicating a failure in the Gram-Schmidt procedure (`spgmr` and `spfgmr` only); `SUNLS_QRSOLVE_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 `CVDIsGetLastFlag` and `CVSpilsGetLastFlag` 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

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

**CVodeGetLinReturnFlagName**

Call  

```c
name = CVodeGetLinReturnFlagName(lsflag);
```

Description  The function **CVodeGetLinReturnFlagName** returns the name of the CVLS constant corresponding to `lsflag`.

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

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

- If $1 \leq lsflag \leq N$ (LU factorization failed), this routine returns “NONE”.

Notes  

- The previous routines **CVDlsGetReturnFlagName** and **CVSpilsGetReturnFlagName** 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.

4.5.9.5 Diagonal linear solver interface optional output functions

The following optional outputs are available from the CVDIAG module: workspace requirements, number of calls to the right-hand side routine for finite-difference Jacobian approximation, and last return value from a CVDIAG 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 here (e.g. `lenrwLS`).

**CVDiagGetWorkSpace**

Call  

```c
flag = CVDiagGetWorkSpace(cvode_mem, &lenrwLS, &leniwLS);
```

Description  The function **CVDiagGetWorkSpace** returns the CVDIAG real and integer workspace sizes.

Arguments  

- `cvode_mem` (void *) pointer to the CVODE memory block.
- `lenrwLS` (long int) the number of realtype values in the CVDIAG workspace.
- `leniwLS` (long int) the number of integer values in the CVDIAG workspace.

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

- **CVDIAG_SUCCESS**  The optional output values have been successfully set.
- **CVDIAG_MEM_NULL**  The `cvode_mem` pointer is NULL.
- **CVDIAG_LMEM_NULL**  The CVDIAG linear solver has not been initialized.

Notes  

- In terms of the problem size $N$, the actual size of the real workspace is roughly $3N$ realtype words.

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

**CVDiagGetNumRhsEvals**

Call  

```c
flag = CVDiagGetNumRhsEvals(cvode_mem, &nfevalsLS);
```

Description  The function **CVDiagGetNumRhsEvals** returns the number of calls made to the user-supplied right-hand side function due to the finite difference Jacobian approximation.

Arguments  

- `cvode_mem` (void *) pointer to the CVODE memory block.
- `nfevalsLS` (long int) the number of calls made to the user-supplied right-hand side function.

Return value  The return value `flag` (of type `int`) is one of
CVDIAG_SUCCESS  The optional output value has been successfully set.
CVDIAG_MEM_NULL  The cvode_mem pointer is NULL.
CVDIAG_LMEM_NULL  The cvdiag linear solver has not been initialized.

Notes  The number of diagonal approximate Jacobians formed is equal to the number of calls made to the linear solver setup function (see CVodeGetNumLinSolvSetups).

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

CVDiagGetLastFlag

Call  flag = CVDiagGetLastFlag(cvode_mem, &lsflag);
Description  The function CVDiagGetLastFlag returns the last return value from a cvdiag routine.
Arguments  cvode_mem (void *) pointer to the cvode memory block.
lsflag (long int) the value of the last return flag from a cvdiag function.
Return value  The return value flag (of type int) is one of
CVDIAG_SUCCESS  The optional output value has been successfully set.
CVDIAG_MEM_NULL  The cvode_mem pointer is NULL.
CVDIAG_LMEM_NULL  The cvdiag linear solver has not been initialized.

Notes  If the cvdiag setup function failed (CVode returned CV_LSETUP_FAIL), the value of lsflag is equal to CVDIAG_INV_FAIL, indicating that a diagonal element with value zero was encountered. The same value is also returned if the cvdiag solve function failed (CVode returned CV_LSOLVE_FAIL).

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

CVDiagGetReturnFlagName

Call  name = CVDiagGetReturnFlagName(lsflag);
Description  The function CVDiagGetReturnFlagName returns the name of the cvdiag constant corresponding to lsflag.
Arguments  The only argument, of type long int, is a return flag from a cvdiag function.
Return value  The return value is a string containing the name of the corresponding constant.

4.5.10 CVODE reinitialization function

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

The use of CVodeReInit requires that the maximum method order, denoted by maxord, be no larger for the new problem than for the previous problem. This condition is automatically fulfilled if the multistep method parameter lmm is unchanged (or changed from CV_ADAMS to CV_BDF) and the default value for maxord is specified.

If there are changes to the linear solver specifications, make the appropriate calls to either the linear solver objects themselves, or to the cvls interface routines, as described in §4.5.3. Otherwise, all solver inputs set previously remain in effect.
One important use of the CVodeReInit function is in the treating of jump discontinuities in the RHS 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 ODE model, using a call to CVodeReInit. 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 RHS 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 RHS 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.

### CVodeReInit

**Call**

```c
flag = CVodeReInit(cvode_mem, t0, y0);
```

**Description**
The function CVodeReInit provides required problem specifications and reinitializes CVODE.

**Arguments**
- `cvode_mem` (*void *) pointer to the CVODE memory block.
- `t0` (*realtype*) is the initial value of `t`.
- `y0` (*N_Vector*) is the initial value of `y`.

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

- **CV_SUCCESS**
  - The call to CVodeReInit was successful.
- **CV_MEM_NULL**
  - The CVODE memory block was not initialized through a previous call to CVodeCreate.
- **CV_NO_MALLOC**
  - Memory space for the CVODE memory block was not allocated through a previous call to CVodeInit.
- **CV_ILL_INPUT**
  - An input argument to CVodeReInit has an illegal value.

**Notes**
If an error occurred, CVodeReInit also sends an error message to the error handler function.

**F2003 Name**
This function is callable as FCVodeReInit when using the Fortran 2003 interface module.

## 4.6 User-supplied functions

The user-supplied functions consist of one function defining the ODE, (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 iterative algorithms.

### 4.6.1 ODE right-hand side

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

### CVRhsFn

**Definition**

```c
typedef int (*CVRhsFn)(realtype t, N_Vector y, N_Vector ydot, void *user_data);
```

**Purpose**
This function computes the ODE right-hand side for a given value of the independent variable `t` and state vector `y`.

**Arguments**
- `t` is the current value of the independent variable.
- `y` is the current value of the dependent variable vector, `y(t)`. 
ydot  is the output vector \( f(t, y) \).
user_data  is the \texttt{user_data} pointer passed to \texttt{CVodeSetUserData}.

Return value  A \texttt{CVRhsFn} should return 0 if successful, a positive value if a recoverable error occurred (in which case \texttt{CVODE} will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and \texttt{CV_RHSFUNC_FAIL} is returned).

Notes  Allocation of memory for \texttt{ydot} is handled within \texttt{CVODE}.

A recoverable failure error return from the \texttt{CVRhsFn} 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, \texttt{CVODE} 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 right-hand side 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.)

There are two other situations in which recovery is not possible even if the right-hand side function returns a recoverable error flag. One is when this occurs at the very first call to the \texttt{CVRhsFn} (in which case \texttt{CVODE} returns \texttt{CV_FIRST_RHSFUNC_ERR}). The other is when a recoverable error is reported by \texttt{CVRhsFn} after an error test failure, while the linear multistep method order is equal to 1 (in which case \texttt{CVODE} returns \texttt{CV_UNREC_RHSFUNC_ERR}).

### 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 \texttt{errfp} (see \texttt{CVodeSetErrFile}), the user may provide a function of type \texttt{CVErrHandlerFn} to process any such messages. The function type \texttt{CVErrHandlerFn} is defined as follows:

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

**Definition**

- \texttt{error_code}  is the error code.
- \texttt{module}  is the name of the \texttt{CVODE} module reporting the error.
- \texttt{function}  is the name of the function in which the error occurred.
- \texttt{msg}  is the error message.
- \texttt{eh_data}  is a pointer to user data, the same as the \texttt{eh_data} parameter passed to \texttt{CVodeSetErrHandlerFn}.

**Purpose**

This function processes error and warning messages from \texttt{CVODE} and its sub-modules.

**Arguments**

- \texttt{error_code}  is the error code.
- \texttt{module}  is the name of the \texttt{CVODE} module reporting the error.
- \texttt{function}  is the name of the function in which the error occurred.
- \texttt{msg}  is the error message.
- \texttt{eh_data}  is a pointer to user data, the same as the \texttt{eh_data} parameter passed to \texttt{CVodeSetErrHandlerFn}.

**Return value**

A \texttt{CVErrHandlerFn} function has no return value.

**Notes**

- \texttt{error_code}  is negative for errors and positive (\texttt{CV_WARNING}) for warnings. If a function that returns a pointer to memory encounters an error, it sets \texttt{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 \texttt{CVEwtFn} to compute a vector \( ewt \) containing the weights in the WRMS norm \( \| v \|_{\text{WRMS}} = \sqrt{(1/N) \sum_{i=1}^{N} (W_i \cdot v_i)^2} \). These weights will be used in place of those defined by Eq. (2.7). The function type \texttt{CVEwtFn} is defined as follows:
4.6 User-supplied functions

CVEwtFn

Definition

typedef int (*CVEwtFn)(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 CVodeSetUserData.

Return value

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

Notes

Allocation of memory for ewt is handled within cvode.

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 ODE system, the user must supply a C function of type CVRootFn, defined as follows:

CVRootFn

Definition

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

Purpose

This function implements a vector-valued function \( g(t, y) \) such that the roots of the \( nrtfn \) components \( g_i(t, y) \) are sought.

Arguments

\( t \) is the current value of the independent variable.

\( y \) is the current value of the dependent variable vector, \( y(t) \).

\( gout \) is the output array, of length \( nrtfn \), with components \( g_i(t, y) \).

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

Return value

A CVRootFn should return 0 if successful or a non-zero value if an error occurred (in which case the integration is halted and CVode returns CV_RTFUNC_FAIL).

Notes

Allocation of memory for gout is automatically handled within cvode.

4.6.5 Jacobian construction (matrix-based linear solvers)

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

CVLsJacFn

Definition

typedef int (*CVLsJacFn)(realtype t, N_Vector y, N_Vector fy, SUNMatrix Jac, void *user_data,
N_Vector tmp1, N_Vector tmp2, N_Vector tmp3);

Purpose

This function computes the Jacobian matrix \( J = \partial f / \partial y \) (or an approximation to it).

Arguments

\( t \) is the current value of the independent variable.

\( y \) is the current value of the dependent variable vector, namely the predicted value of \( y(t) \).

\( fy \) is the current value of the vector \( f(t,y) \).

\( Jac \) is the output Jacobian matrix (of type SUNMatrix).
user_data is a pointer to user data, the same as the user_data parameter passed to CVodeSetUserData.

tmp1

tmp2

tmp3 are pointers to memory allocated for variables of type N_Vector which can be used by a CVLSJacFn function as temporary storage or work space.

Return value A CVLSJacFn should return 0 if successful, a positive value if a recoverable error occurred (in which case cvode will attempt to correct, while CVLS sets last_flag to CVLS_JACFUNC_RECFNR), or a negative value if it failed unrecoverably (in which case the integration is halted, CVode returns CV_LSETUP_FAIL and CVLS sets last_flag to CVLS_JACFUNC_UNRECVR).

Notes Information regarding the structure of the specific SUNMATRIX structure (e.g. number of rows, upper/lower bandwidth, sparsity type) may be obtained through using the implementation-specific SUNMATRIX interface functions (see Chapter 7 for details).

Prior to calling the user-supplied Jacobian function, the Jacobian matrix $J(t, y)$ is zeroed out, so only nonzero elements need to be loaded into Jac.

If the user’s CVLSJacFn function uses difference quotient approximations, then it may need to access quantities not in the argument list. These include the current step size, the error weights, etc. To obtain these, the user will need to add a pointer to cv_mem to user_data and then use the CVodeGet* functions described in §4.5.9.2. The unit roundoff can be accessed as UNIT_ROUNDOFF defined in sundials_types.h.

dense:

A user-supplied dense Jacobian function must load the $N \times N$ dense matrix Jac with an approximation to the Jacobian matrix $J(t, y)$ at the point $(t, y)$. The accessor macros SM_ELEMENT_D and SM_COLUMN_D allow the user to read and write dense matrix elements without making explicit references to the underlying representation of the SUNMATRIX_DENSE type. SM_ELEMENT_D(J, i, j) references the $(i, j)$-th element of the dense matrix Jac (with $i, j = 0 \ldots N - 1$). This macro is meant for small problems for which efficiency of access is not a major concern. Thus, in terms of the indices $m$ and $n$ ranging from 1 to $N$, the Jacobian element $J_{m,n}$ can be set using the statement SM_ELEMENT_D(J, m-1, n-1) = $J_{m,n}$. Alternatively, SM_COLUMN_D(J, j) returns a pointer to the first element of the $j$-th column of Jac (with $j = 0 \ldots N - 1$), and the elements of the $j$-th column can then be accessed using ordinary array indexing. Consequently, $J_{m,n}$ can be loaded using the statements col_n = SM_COLUMN_D(J, n-1); col_n[m-1] = $J_{m,n}$. For large problems, it is more efficient to use SM_COLUMN_D than to use SM_ELEMENT_D. Note that both of these macros number rows and columns starting from 0. The SUNMATRIX_DENSE type and accessor macros are documented in §7.5.

banded:

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

\[ \text{SM\_COLUMN\_B}(J, n-1) \text{; SM\_COLUMN\_ELEMENT\_B}(\text{col}_n, m-1, n-1) = J_{m,n}. \]

The elements of the \( j \)-th column can also be accessed via ordinary array indexing, but this approach requires knowledge of the underlying storage for a band matrix of type \text{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 \text{SUNMATRIX\_BAND} type and accessor macros are documented in \S 7.6.

**sparse:**

A user-supplied sparse Jacobian function must load the \( N \) by \( N \) compressed-sparse-column or compressed-sparse-row matrix \( \text{Jac} \) with an approximation to the Jacobian matrix \( J(t,y) \) at the point \((t,y)\). Storage for \( \text{Jac} \) already exists on entry to this function, although the user should ensure that sufficient space is allocated in \( \text{Jac} \) to hold the nonzero values to be set; if the existing space is insufficient the user may reallocate the data and index arrays as needed. The amount of allocated space in a \text{SUNMATRIX\_SPARSE} object may be accessed using the macro \text{SM\_NNZ\_S} or the routine \text{SUNSparseMatrix\_NNZ}.

The \text{SUNMATRIX\_SPARSE} type and accessor macros are documented in \S 7.7.

The previous function type \text{CVDlsJacFn} is identical to \text{CVLsJacFn}, 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 \text{NULL}-valued \text{SUNMATRIX} was supplied to \text{CVodeSetLinearSolver}), the user may provide a function of type \text{CVLsJacTimesVecFn} 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.

**CVLsJacTimesVecFn**

**Definition**

\[
\text{typedef int (*CVLsJacTimesVecFn)(N\_Vector v, N\_Vector Jv,}
\text{ realtype t, N\_Vector y, N\_Vector fy,}
\text{ void *user\_data, N\_Vector tmp);}\]

**Purpose**

This function computes the product \( Jv = (\partial f/\partial y)v \) (or an approximation to it).

**Arguments**

- \( v \) is the vector by which the Jacobian must be multiplied.
- \( Jv \) is the output vector computed.
- \( t \) is the current value of the independent variable.
- \( y \) is the current value of the dependent variable vector.
- \( fy \) is the current value of the vector \( f(t,y) \).
- \( \text{user\_data} \) is a pointer to user data, the same as the \text{user\_data} parameter passed to \text{CVodeSetUserData}.
- \( \text{tmp} \) is a pointer to memory allocated for a variable of type \text{N\_Vector} which can be used for work space.

**Return value**

The value returned by the Jacobian-vector product function should be 0 if successful. Any other return value will result in an unrecoverable error of the generic Krylov solver, in which case the integration is halted.

**Notes**

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

If the user’s \text{CVLsJacTimesVecFn} function uses difference quotient approximations, it may need to access quantities not in the argument list. These include the current step size, the error weights, etc. To obtain these, the user will need to add a pointer to
The previous function type CVSpilsJacTimesVecFn is identical to CVLsJacTimesVecFn, 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 routine requires that any Jacobian-related data be preprocessed or evaluated, then this needs to be done in a user-supplied function of type CVLsJacTimesSetupFn, defined as follows:

```c
typedef int (*CVLsJacTimesSetupFn)(realtype t, N_Vector y, N_Vector fy, void *user_data);
```

**Definition**

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

**Arguments**

- `t` is the current value of the independent variable.
- `y` is the current value of the dependent variable vector.
- `fy` is the current value of the vector \( f(t,y) \).
- `user_data` is a pointer to user data, the same as the `user_data` parameter passed to `CVodeSetUserData`.

**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 CVRhsFn user function with the same \((t,y)\) arguments. Thus, the setup function can use any auxiliary data that is computed and saved during the evaluation of the ODE right-hand side.

If the user’s `CVLsJacTimesSetupFn` function uses difference quotient approximations, it may need to access quantities not in the argument list. These include the current step size, the error weights, etc. To obtain these, the user will need to add a pointer to `cv_mem` to `user_data` and then use the `CVodeGet*` functions described in §4.5.9.2. The unit roundoff can be accessed as `UNIT_ROUNDOFF` defined in `sundials_types.h`.

The previous function type CVSpilsJacTimesSetupFn is identical to CVLsJacTimesSetupFn, 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 \( P \) may be either a left or right preconditioner matrix. Here \( P \) should approximate (at least crudely) the matrix \( M = I - \gamma J \), where \( J = \partial f / \partial y \). If preconditioning is done on both sides, the product of the two preconditioner matrices should approximate \( M \). This function must be of type CVLsPrecSolveFn, defined as follows:
4.6 User-supplied functions

**CVLsPrecSolveFn**

**Definition**

typedef int (*CVLsPrecSolveFn)(realtype t, N_Vector y, N_Vector fy,
N_Vector r, N_Vector z, realtype gamma,
realtype delta, int lr, void *user_data);

**Purpose**
This function solves the preconditioned system \( Pz = r \).

**Arguments**
- \( t \) is the current value of the independent variable.
- \( y \) is the current value of the dependent variable vector.
- \( fy \) is the current value of the vector \( f(t, y) \).
- \( r \) is the right-hand side vector of the linear system.
- \( z \) is the computed output vector.
- \( \gamma \) is the scalar \( \gamma \) appearing in the matrix given by \( M = I - \gamma J \).
- \( \delta \) 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 the weighted \( l_2 \) norm, i.e., \( \sqrt{\sum_i (Res_i \cdot ewt_i)^2} < \delta \). To obtain the \( N \)-vector \( ewt \), call \text{CVodeGetErrWeights} (see §4.5.9.2).
- \( lr \) is an input flag indicating whether the preconditioner solve function is to use the left preconditioner (\( lr = 1 \)) or the right preconditioner (\( lr = 2 \));
- \( \text{user_data} \) is a pointer to user data, the same as the \( \text{user_data} \) parameter passed to the function \text{CVodeSetUserData}.

**Return value**
The value 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), or negative for an unrecoverable error (in which case the integration is halted).

**Notes**
The previous function type \text{CVSpilsPrecSolveFn} is identical to \text{CVLsPrecSolveFn}, 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 preprocessed or evaluated, then this needs to be done in a user-supplied function of type \text{CVLsPrecSetupFn}, defined as follows:

**CVLsPrecSetupFn**

**Definition**

typedef int (*CVLsPrecSetupFn)(realtype t, N_Vector y, N_Vector fy,
booleantype jok, booleantype *jcurPtr,
realtype gamma, void *user_data);

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

**Arguments**
- \( t \) is the current value of the independent variable.
- \( y \) is the current value of the dependent variable vector, namely the predicted value of \( y(t) \).
- \( fy \) is the current value of the vector \( f(t, y) \).
- \( jok \) is an input flag indicating whether the Jacobian-related data needs to be updated. The \( jok \) argument provides for the reuse of Jacobian data in the preconditioner solve function. \( jok = \text{SUNFALSE} \) means that the Jacobian-related data must be recomputed from scratch. \( jok = \text{SUNTRUE} \) means that the Jacobian data, if saved from the previous call to this function, can be reused (with the current value of \( \gamma \)). A call with \( jok = \text{SUNTRUE} \) can only occur after a call with \( jok = \text{SUNFALSE} \).
jcurPtr is a pointer to a flag which should be set to SUNTRUE if Jacobian data was recomputed, or set to SUNFALSE if Jacobian data was not recomputed, but saved data was still reused.

gamma is the scalar γ appearing in the matrix \( M = I - \gamma J \).

user_data is a pointer to user data, the same as the user_data parameter passed to the function CVodeSetUserData.

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), or 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 of the resulting approximation to \( M = I - \gamma J \).

Each call to the preconditioner setup function is preceded by a call to the CVRhsFn user function with the same \((t, y)\) arguments. Thus, the preconditioner setup function can use any auxiliary data that is computed and saved during the evaluation of the ODE right-hand side.

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 CVLsPrecSetupFn function uses difference quotient approximations, it may need to access quantities not in the call list. These include the current step size, the error weights, etc. To obtain these, the user will need to add a pointer to cv_mem to user_data and then use the CVodeGet* functions described in §4.5.9.2. The unit roundoff can be accessed as UNIT_ROUNDOFF defined in sundials_types.h.

The previous function type CVSpilsPrecSetupFn is identical to CVLsPrecSetupFn, 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 Preconditioner modules

The efficiency of Krylov iterative methods for the solution of linear systems can be greatly enhanced through preconditioning. For problems in which the user cannot define a more effective, problem-specific preconditioner, CVODE provides a banded preconditioner in the module CVBANDPRE and a band-block-diagonal preconditioner module CVBBDPRE.

4.7.1 A serial banded preconditioner module

This preconditioner provides a band matrix preconditioner for use with iterative SUNLINSOL modules through the CVLS linear solver interface, in a serial setting. It uses difference quotients of the ODE right-hand side function \( f \) to generate a band matrix of bandwidth \( m_l + m_u + 1 \), where the number of super-diagonals \( m_u \), the upper half-bandwidth \( m_l \), the lower half-bandwidth are specified by the user, and uses this to form a preconditioner for use with the Krylov linear solver. Although this matrix is intended to approximate the Jacobian \( \partial f / \partial y \), it may be a very crude approximation. The true Jacobian need not be banded, or its true bandwidth may be larger than \( m_l + m_u + 1 \), as long as the banded approximation generated here is sufficiently accurate to speed convergence as a preconditioner.

In order to use the CVBANDPRE module, the user need not define any additional functions. Aside from the header files required for the integration of the ODE problem (see §4.3), to use the CVBANDPRE module, the main program must include the header file cvode_bandpre.h which declares the needed function prototypes. The following is a summary of the usage of this module. Steps that are unchanged from the skeleton program presented in §4.4 are grayed out.
1. Initialize multi-threaded environment, if appropriate

2. Set problem dimensions etc.

3. Set vector of initial values

4. Create CVODE object

5. Initialize CVODE solver

6. Specify integration tolerances

7. Create linear solver object

   When creating the iterative linear solver object, specify the type of preconditioning (PREC_LEFT or PREC_RIGHT) to use.

8. Set linear solver optional inputs

9. Attach linear solver module

10. Initialize the CVBANDPRE preconditioner module

    Specify the upper and lower half-bandwidths (mu and ml, respectively) and call

    \[
    \text{flag} = \text{CVBandPrecInit}(\text{cvoode}_\text{mem}, N, \mu, \text{ml});
    \]

    to allocate memory and initialize the internal preconditioner data.

11. Set optional inputs

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

12. Create nonlinear solver object

13. Attach nonlinear solver module

14. Set nonlinear solver optional inputs

15. Specify rootfinding problem

16. Advance solution in time

17. Get optional outputs

   Additional optional outputs associated with CVBANDPRE are available by way of two routines described below, CVBandPrecGetWorkSpace and CVBandPrecGetNumRhsEvals.

18. Deallocate memory for solution vector

19. Free solver memory

20. Free nonlinear solver memory

21. Free linear solver memory

The CVBANDPRE preconditioner module is initialized and attached by calling the following function:

\texttt{CVBandPrecInit}

Call \[
\text{flag} = \text{CVBandPrecInit}(\text{cvoode}_\text{mem}, N, \mu, \text{ml});
\]

Description The function CVBandPrecInit initializes the CVBANDPRE preconditioner and allocates required (internal) memory for it.
Arguments

-\texttt{cvode}\_mem (void *) pointer to the \texttt{CVODE} memory block.
-\texttt{N} (sunindextype) problem dimension.
-\texttt{mu} (sunindextype) upper half-bandwidth of the Jacobian approximation.
-\texttt{ml} (sunindextype) lower half-bandwidth of the Jacobian approximation.

Return value

The return value \texttt{flag} (of type \texttt{int}) is one of:

- \texttt{CVLS\_SUCCESS} The call to \texttt{CVBandPrecInit} was successful.
- \texttt{CVLS\_MEM\_NULL} The \texttt{cvode}\_mem pointer was \texttt{NULL}.
- \texttt{CVLS\_MEM\_FAIL} A memory allocation request has failed.
- \texttt{CVLS\_LMEM\_NULL} A CVLS linear solver memory was not attached.
- \texttt{CVLS\_ILL\_INPUT} The supplied vector implementation was not compatible with block band preconditioner.

Notes

The banded approximate Jacobian will have nonzero elements only in locations \((i, j)\) with \(-mL \leq j - i \leq mL\).

F2003 Name

This function is callble as \texttt{FCVBandPrecInit} when using the Fortran 2003 interface module.

The following three optional output functions are available for use with the \texttt{CVBANDPRE} module:

### \texttt{CVBandPrecGetWorkSpace}

Call

```c
flag = CVBandPrecGetWorkSpace(cvode_mem, &lenrwBP, &leniwBP);
```

Description

The function \texttt{CVBandPrecGetWorkSpace} returns the sizes of the \texttt{CVBANDPRE} real and integer workspaces.

Arguments

- \texttt{cvode}\_mem (void *) pointer to the \texttt{CVODE} memory block.
- \texttt{lenrwBP} (long int) the number of \texttt{realtype} values in the \texttt{CVBANDPRE} workspace.
- \texttt{leniwBP} (long int) the number of integer values in the \texttt{CVBANDPRE} workspace.

Return value

The return value \texttt{flag} (of type \texttt{int}) is one of:

- \texttt{CVLS\_SUCCESS} The optional output values have been successfully set.
- \texttt{CVLS\_PMEM\_NULL} The \texttt{CVBANDPRE} preconditioner has not been initialized.

Notes

The workspace requirements reported by this routine correspond only to memory allocated within the \texttt{CVBANDPRE} module (the banded matrix approximation, banded SUNLINSOL object, and temporary vectors).

The workspaces referred to here exist in addition to those given by the corresponding function \texttt{CVodeGetLinWorkSpace}.

### \texttt{CVBandPrecGetNumRhsEvals}

Call

```c
flag = CVBandPrecGetNumRhsEvals(cvode_mem, &nfevalsBP);
```

Description

The function \texttt{CVBandPrecGetNumRhsEvals} returns the number of calls made to the user-supplied right-hand side function for the finite difference banded Jacobian approximation used within the preconditioner setup function.

Arguments

- \texttt{cvode}\_mem (void *) pointer to the \texttt{CVODE} memory block.
- \texttt{nfevalsBP} (long int) the number of calls to the user right-hand side function.

Return value

The return value \texttt{flag} (of type \texttt{int}) is one of:

- \texttt{CVLS\_SUCCESS} The optional output value has been successfully set.
- \texttt{CVLS\_PMEM\_NULL} The \texttt{CVBANDPRE} preconditioner has not been initialized.

Notes

The counter \texttt{nfevalsBP} is distinct from the counter \texttt{nfevalsLS} returned by the corresponding function \texttt{CVodeGetNumLinRhsEvals} and \texttt{nfevals} returned by \texttt{CVodeGetNumRhsEvals}. The total number of right-hand side function evaluations is the sum of all three of these counters.
4.7 Preconditioner modules

This function is callable as \texttt{FCVBandPrecGetNumRhsEvals} when using the Fortran 2003 interface module.

4.7.2 A parallel band-block-diagonal preconditioner module

A principal reason for using a parallel ODE solver such as \texttt{CVODE} 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 (2.5) 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 [30] and is included in a software module within the \texttt{CVODE} package. This module works with the parallel vector module \texttt{nvector} and is usable with any of the Krylov iterative linear solvers through the \texttt{CVLS} interface. It 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 \texttt{CVBBDPRE}.

One way to envision these preconditioners is to think of the domain of the computational PDE problem as being subdivided into \( M \) non-overlapping subdomains. Each of these subdomains is then assigned to one of the \( M \) processes to be used to solve the ODE system. The basic idea is to isolate the preconditioning so that it is local to each process, and also to use a (possibly cheaper) approximate right-hand side function. This requires the definition of a new function \( g(t, y) \) which approximates the function \( f(t, y) \) in the definition of the ODE system (2.1). However, the user may set \( g = f \). Corresponding to the domain decomposition, there is a decomposition of the solution vector \( y \) into \( M \) disjoint blocks \( y_m \), and a decomposition of \( g \) into blocks \( g_m \). The block \( g_m \) depends both on \( y_m \) and on components of blocks \( y_m' \) associated with neighboring subdomains (so-called ghost-cell data). Let \( \tilde{y}_m \) denote \( y_m \) augmented with those other components on which \( g_m \) depends. Then we have

\[
g(t, y) = \begin{bmatrix} g_1(t, \tilde{y}_1), g_2(t, \tilde{y}_2), \ldots, g_M(t, \tilde{y}_M) \end{bmatrix}^T
\]

and each of the blocks \( g_m(t, \tilde{y}_m) \) is uncoupled from the others.

The preconditioner associated with this decomposition has the form

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

where

\[
P_m \approx I - \gamma J_m
\]

and \( J_m \) is a difference quotient approximation to \( \partial g_m / \partial y_m \). This matrix is taken to be banded, with upper and lower half-bandwidths \( \text{mudq} \) and \( \text{mldq} \) 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 Jacobian of the local block of \( g \), if smaller values provide a more efficient preconditioner. The solution of the complete linear system

\[
P x = b
\]

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 treatments of the blocks \( P_m \). For example, incomplete LU factorization or an iterative method could be used instead of banded LU factorization.
The cvbbdpre module calls two user-provided functions to construct $P$: a required function $\text{gloc}$ (of type CVLocalFn) which approximates the right-hand side function $g(t, y) \approx f(t, y)$ and which is computed locally, and an optional function $\text{cfn}$ (of type CVCommFn) which performs all interprocess communication necessary to evaluate the approximate right-hand side $g$. These are in addition to the user-supplied right-hand side function $f$. Both functions take as input the same pointer user_data that is passed by the user to CVodeSetUserData and that was passed to the user’s function $f$. The user is responsible for providing space (presumably within user_data) for components of $y$ that are communicated between processes by $\text{cfn}$, and that are then used by $\text{gloc}$, which should not do any communication.

**CVLocalFn**

**Definition**

typedef int (*CVLocalFn)(sunindextype Nlocal, realtype t, N_Vector y, N_Vector glocal, void *user_data);

**Purpose**

This $\text{gloc}$ function computes $g(t, y)$. It loads the vector $\text{glocal}$ as a function of $t$ and $y$.

**Arguments**

- $Nlocal$ is the local vector length.
- $t$ is the value of the independent variable.
- $y$ is the dependent variable.
- $\text{glocal}$ is the output vector.
- $\text{user_data}$ is a pointer to user data, the same as the $\text{user_data}$ parameter passed to CVodeSetUserData.

**Return value**

A CVLocalFn should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODE will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and CVode returns CV_LSETUP_FAIL).

**Notes**

This function must assume that all interprocess communication of data needed to calculate $\text{glocal}$ has already been done, and that this data is accessible within user_data. The case where $g$ is mathematically identical to $f$ is allowed.

**CVCommFn**

**Definition**

typedef int (*CVCommFn)(sunindextype Nlocal, realtype t, N_Vector y, void *user_data);

**Purpose**

This $\text{cfn}$ function performs all interprocess communication necessary for the execution of the $\text{gloc}$ function above, using the input vector $y$.

**Arguments**

- $Nlocal$ is the local vector length.
- $t$ is the value of the independent variable.
- $y$ is the dependent variable.
- $\text{user_data}$ is a pointer to user data, the same as the $\text{user_data}$ parameter passed to CVodeSetUserData.

**Return value**

A CVCommFn should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODE will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and CVode returns CV_LSETUP_FAIL).

**Notes**

The $\text{cfn}$ function is expected to save communicated data in space defined within the data structure user_data. Each call to the $\text{cfn}$ function is preceded by a call to the right-hand side function $f$ with the same $(t, y)$ arguments. Thus, $\text{cfn}$ can omit any communication done by $f$ if relevant to the evaluation of $\text{glocal}$. If all necessary communication was done in $f$, then $\text{cfn} = \text{NULL}$ can be passed in the call to CV.BBDPreInit (see below).
4.7 Preconditioner modules

Besides the header files required for the integration of the ODE problem (see §4.3), to use the cvbbdpre module, the main program must include the header file cvode_bbdpre.h which declares the needed function prototypes.

The following is a summary of the proper usage of this module. Steps that are unchanged from the skeleton program presented in §4.4 are grayed out.

1. Initialize MPI environment
2. Set problem dimensions etc.
3. Set vector of initial values
4. Create cvode object
5. Initialize cvode solver
6. Specify integration tolerances
7. Create linear solver object
   When creating the iterative linear solver object, specify the type of preconditioning (PREC_LEFT or PREC_RIGHT) to use.
8. Set linear solver optional inputs
9. Attach linear solver module
10. Initialize the cvbbdpre preconditioner module
    Specify the upper and lower half-bandwidths mudq and mldq, and mukeep and mlkeep, and call
    flag = CVBBDPrecInit(cvode_mem, local_N, mudq, mldq, mukeep, mlkeep, dqrely, gloc, cfn);
    to allocate memory and initialize the internal preconditioner data. The last two arguments of CVBBDPrecInit are the two user-supplied functions described above.

11. Set optional inputs
    Note that the user should not overwrite the preconditioner setup function or solve function through calls to the CVodeSetPreconditioner optional input function.
12. Create nonlinear solver object
13. Attach nonlinear solver module
14. Set nonlinear solver optional inputs
15. Specify rootfinding problem
16. Advance solution in time
17. Get optional outputs
    Additional optional outputs associated with cvbbdpre are available by way of two routines described below, CVBBDPrecGetWorkSpace and CVBBDPrecGetNumGfnEvals.
18. Deallocate memory for solution vector
19. Free solver memory
20. Free nonlinear solver memory
21. **Free linear solver memory**

22. **Finalize MPI**

The user-callable functions that initialize (step 10 above) or re-initialize the CVBBDPRE preconditioner module are described next.

### CVBBDPrecInit

**Call**

```c
flag = CVBBDPrecInit(cvode_mem, local_N, mudq, mldq, mukeep, mlkeep, dqrely, gloc, cfn);
```

**Description**
The function CVBBDPrecInit initializes and allocates (internal) memory for the CVBB- DPRE preconditioner.

**Arguments**
- `cvode_mem` *(void *)* pointer to the CVODE memory block.
- `local_N` *(sunindextype)* local vector length.
- `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.
- `dqrely` *(realtype)* the relative increment in components of \( y \) used in the difference quotient approximations. The default is \( dqrely = \sqrt{\text{unit roundoff}} \), which can be specified by passing \( dqrely = 0.0 \).
- `gloc` *(CVLocalFn)* the C function which computes the approximation \( g(t, y) \approx f(t, y) \).
- `cfn` *(CVCommFn)* the optional C function which performs all interprocess communication required for the computation of \( g(t, y) \).

**Return value**
The return value `flag` (of type `int`) is one of
- `CVLS_SUCCESS` The call to CVBBDPrecInit was successful.
- `CVLS_MEM_NULL` The `cvode_mem` pointer was NULL.
- `CVLS_MEM_FAIL` A memory allocation request has failed.
- `CVLS_LMEM_NULL` A CVLS linear solver was not attached.
- `CVLS_ILL_INPUT` The supplied vector implementation was not compatible with block band preconditioner.

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

The half-bandwidths `mudq` and `mldq` need not be the true half-bandwidths of the Jacobian of the local block of \( g \) when smaller values may provide a greater efficiency.

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

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

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

The CVBBDPRE module also provides a reinitialization function to allow solving a sequence of problems of the same size, with the same linear solver choice, provided there is no change in `local_N`, `mukeep`, or `mlkeep`. After solving one problem, and after calling `CVodeReInit` to re-initialize CVODE
for a subsequent problem, a call to CVBBDPrecReInit can be made to change any of the following: the half-bandwidths \texttt{mudq} and \texttt{mldq} used in the difference-quotient Jacobian approximations, the relative increment \texttt{dqrely}, or one of the user-supplied functions \texttt{gloc} and \texttt{cfn}. If there is a change in any of the linear solver inputs, an additional call to the “Set” routines provided by the \texttt{SUNLINSOL} module, and/or one or more of the corresponding \texttt{CVLS} “set” functions, must also be made (in the proper order).

\textbf{CVBBDPrecReInit}

Call
\begin{verbatim}
flag = CVBBDPrecReInit(cvode_mem, mudq, mldq, dqrely);
\end{verbatim}

Description
The function CVBBDPrecReInit re-initializes the CVBBDPRE preconditioner.

Arguments
\begin{verbatim}
cvode_mem (void *) pointer to the CVODE memory block.
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.
dqrely (realtype) the relative increment in components of \( y \) used in the difference quotient approximations. The default is \( dqrely = \sqrt{\text{unit roundoff}} \), which can be specified by passing \( dqrely = 0.0 \).
\end{verbatim}

Return value
The return value \texttt{flag} (of type \texttt{int}) is one of
\begin{itemize}
  \item \texttt{CVLS\_SUCCESS} The call to CVBBDPrecReInit was successful.
  \item \texttt{CVLS\_MEM\_NULL} The \texttt{cvode_mem} pointer was NULL.
  \item \texttt{CVLS\_LMEM\_NULL} A \texttt{CVLS} linear solver memory was not attached.
  \item \texttt{CVLS\_PMEM\_NULL} The function CVBBDPrecInit was not previously called.
\end{itemize}

Notes
If one of the half-bandwidths \texttt{mudq} or \texttt{mldq} is negative or exceeds the value \texttt{local.N-1}, it is replaced by 0 or \texttt{local.N-1} accordingly.

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

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

\textbf{CVBBDPrecGetWorkSpace}

Call
\begin{verbatim}
flag = CVBBDPrecGetWorkSpace(cvode_mem, &lenrwBBDP, &leniwBBDP);
\end{verbatim}

Description
The function CVBBDPrecGetWorkSpace returns the local CVBBDPRE real and integer workspace sizes.

Arguments
\begin{verbatim}
cvode_mem (void *) pointer to the CVODE memory block.
lenrwBBDP (long int) local number of \texttt{realtype} values in the CVBBDPRE workspace.
leniwBBDP (long int) local number of integer values in the CVBBDPRE workspace.
\end{verbatim}

Return value
The return value \texttt{flag} (of type \texttt{int}) is one of
\begin{itemize}
  \item \texttt{CVLS\_SUCCESS} The optional output value has been successfully set.
  \item \texttt{CVLS\_MEM\_NULL} The \texttt{cvode_mem} pointer was NULL.
  \item \texttt{CVLS\_PMEM\_NULL} The CVBBDPRE preconditioner has not been initialized.
\end{itemize}

Notes
The workspace requirements reported by this routine correspond only to memory allocated within the CVBBDPRE 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 \texttt{CVodeGetLinWorkSpace}.

F2003 Name
This function is callable as FCVBBDPrecGetWorkSpace when using the Fortran 2003 interface module.
The function \texttt{CVBBDPrecGetNumGfnEvals} returns the number of calls made to the user-supplied \texttt{gloc} function due to the finite difference approximation of the Jacobian blocks used within the preconditioner setup function.

**Arguments**
- \texttt{cvode_mem} (void *) pointer to the CVODE memory block.
- \texttt{ngevalsBBDP} (long int) the number of calls made to the user-supplied \texttt{gloc} function.

**Return value**
The return value \texttt{flag} (of type \texttt{int}) is one of
- \texttt{CVLS_SUCCESS} The optional output value has been successfully set.
- \texttt{CVLS_MEM_NULL} The \texttt{cvode_mem} pointer was NULL.
- \texttt{CVLS_PMEM_NULL} The CVBBDPRE preconditioner has not been initialized.

**F2003 Name**
This function is callable as \texttt{FCVBBDPrecGetNumGfnEvals} when using the Fortran 2003 interface module.

In addition to the \texttt{ngevalsBBDP} \texttt{gloc} evaluations, the costs associated with CVBBDPRE also include \texttt{nlinsetups} LU factorizations, \texttt{nlinsetups} calls to \texttt{cfn}, \texttt{np solves} banded backsolve calls, and \texttt{nfevalsLS} right-hand side function evaluations, where \texttt{nlinsetups} is an optional CVODE output and \texttt{np solves} and \texttt{nfevalsLS} are linear solver optional outputs (see §4.5.9).
Chapter 5

Using CVODE for FORTRAN Applications

A FORTRAN 2003 module (fcvode_mod) as well as a FORTRAN 77 style interface (fcvode) are provided to support the use of cvode, for the solution of ODE systems \( \frac{dy}{dt} = f(t, y) \), in a mixed FORTRAN/C setting. While cvode is written in C, it is assumed here that the user’s calling program and user-supplied problem-defining routines are written in FORTRAN.

5.1 CVODE FORTRAN 2003 Interface Module

The fcvode_mod FORTRAN module defines interfaces to most cvode C functions using the intrinsic iso_c_binding module which provides a standardized mechanism for interoperating with C. All interfaced functions are named after the corresponding C function, but with a leading ‘F’. For example, the cvode function CVodeCreate is interfaced as FCVodeCreate. Thus, the steps to use cvode and the function calls in FORTRAN 2003 are identical (ignoring language differences) to those in C. The C functions with FORTRAN 2003 interfaces indicate this in their description in Chapter 4.

The FORTRAN 2003 cvode interface module can be accessed by the use statement, i.e. use fcvode_mod, and linking to the library libsundials_fcvode_mod.lib in addition to libsundials_cvode.lib.

5.1.1 SUNDIALS FORTRAN 2003 Modules

Many sundials modules provide FORTRAN 2003 interfaces as well. A module can be accessed with the use statement, e.g. use fnvector_openmp_mod, and linking to the FORTRAN 2003 library in addition to the C library, e.g. libsundials_fnvectoropenmp_mod.lib and libsundials_nvectoropenmp.lib.

A summary of shared sundials modules with FORTRAN 2003 interfaces is given in Table 5.1. For more information on FORTRAN 2003 interface modules to NVECTOR, SUNMATRIX, SUNLINSOL, and SUNNONLINSOL implementations see Chapters 6, 7, 8, and 9 respectively. For details on where the FORTRAN 2003 module (.mod) files and libraries are installed see Appendix A.

5.1.2 Important note on portability

Upon compilation of sundials, FORTRAN module (.mod) files are generated for each FORTRAN 2003 interface. These files are highly compiler specific, and thus it is almost always necessary to compile a consuming application with the same compiler used to generate the modules.

5.1.3 Data Types

Currently, the FORTRAN 2003 interfaces are only compatible with double precision and 64-bit indices. The interfaces use types from the iso_c_binding module that correspond to the C type. Specifically:
Table 5.1: Summary of Fortran 2003 interfaces for shared Sundials modules.

<table>
<thead>
<tr>
<th>Module</th>
<th>Fortran 2003 Module Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVECTOR_SERIAL</td>
<td>fnvector_serial_mod</td>
</tr>
<tr>
<td>NVECTOR_PARALLEL</td>
<td>Not interfaced</td>
</tr>
<tr>
<td>NVECTOR_OPENMP</td>
<td>fnvector_openmp_mod</td>
</tr>
<tr>
<td>NVECTOR_PTHREADS</td>
<td>fnvector_pthreads_mod</td>
</tr>
<tr>
<td>NVECTOR_PARHYP</td>
<td>Not interfaced</td>
</tr>
<tr>
<td>NVECTOR_PETSC</td>
<td>Not interfaced</td>
</tr>
<tr>
<td>NVECTOR_CUDA</td>
<td>Not interfaced</td>
</tr>
<tr>
<td>NVECTOR_RAJA</td>
<td>Not interfaced</td>
</tr>
<tr>
<td>SUNMATRIX_BAND</td>
<td>fsunmatrix_band_mod</td>
</tr>
<tr>
<td>SUNMATRIX_DENSE</td>
<td>fsunmatrix_dense_mod</td>
</tr>
<tr>
<td>SUNMATRIX_SPARSE</td>
<td>fsunmatrix_sparse_mod</td>
</tr>
<tr>
<td>SUNLINSOL_BAND</td>
<td>fsunlinsol_band_mod</td>
</tr>
<tr>
<td>SUNLINSOL_DENSE</td>
<td>fsunlinsol_dense_mod</td>
</tr>
<tr>
<td>SUNLINSOL_LPACKBAND</td>
<td>Not interfaced</td>
</tr>
<tr>
<td>SUNLINSOL_LPACKDENSE</td>
<td>Not interfaced</td>
</tr>
<tr>
<td>SUNLINSOL_KLU</td>
<td>fsunlinsol_klu_mod</td>
</tr>
<tr>
<td>SUNLINSOL_SUPERLUMT</td>
<td>Not interfaced</td>
</tr>
<tr>
<td>SUNLINSOL_SPGMR</td>
<td>fsunlinsol_spgmr_mod</td>
</tr>
<tr>
<td>SUNLINSOL_SPFGMR</td>
<td>fsunlinsol_spfmr_mod</td>
</tr>
<tr>
<td>SUNLINSOL_SPBCGS</td>
<td>fsunlinsol_spbcgs_mod</td>
</tr>
<tr>
<td>SUNLINSOL_SPTFQMR</td>
<td>fsunlinsol_sptfqmr_mod</td>
</tr>
<tr>
<td>SUNLINSOL_PCG</td>
<td>fsunlinsol_pcg_mod</td>
</tr>
<tr>
<td>SUNNONLIN_SOL_NEWTON</td>
<td>fsunnonlinsol_newton_mod</td>
</tr>
<tr>
<td>SUNNONLIN_SOL_FIXED</td>
<td>fsunnonlinsol_fixedpoint_mod</td>
</tr>
</tbody>
</table>

- double is equivalent to real(c_double)
- int32_t and int are equivalent to integer(c_int)
- int64_t and long are equivalent to integer(c_long)
- Sundials types, e.g. N_Vector, are equivalent to type(c_ptr)

5.2 FCVODE, an Interface Module for FORTRAN Applications

The FCVODE interface module is a package of C functions which support the use of the CVODE. This package provides the necessary interface to CVODE for all supplied serial and parallel NVECTOR implementations.

5.2.1 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).

### 5.2.2 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

### 5.2.3 FCVODE routines

The user-callable functions, with the corresponding CVODE 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.
– 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.
– 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.
– 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.
– FSUNKLUINIT (defined by SUNLINSOL_KLU) interfaces to SUNLinSol_KLU.
– FSUNKLUREINIT (defined by SUNLINSOL_KLU) interfaces to SUNLinSol_KLUReinit.

• Interface to the main CVODE module

– FCVMALLOC interfaces to CVodeCreate, CVodeSetUserData, and CVodeInit, as well as one of CVodeSStolerances or CVodeSVtolerances.
– FCVREINIT interfaces to CVodeReInit.
– FCVSETIIN and FCVSETRIN interface to CVodeSet* functions.
– FCVEWTSET interfaces to CVodeWt tolerances.
– FCVODE interfaces to CVode, CVodeGet* functions, and to the optional output functions for the selected linear solver module.
– FCVDKY interfaces to the interpolated output function CVodeGetDky.
– FCVGETERRWEIGHTS interfaces to CVodeGetErrWeights.
– FCVGETESTLOCALERR interfaces to CVodeGetEstLocalErrors.
– FCVFREE interfaces to CVodeFree.

• Interface to the linear solver interfaces

– FCVLSINIT interfaces to CVodeSetLinearSolver.
– FCVLSSETEPSLIN interfaces to CVodeSetEpsLin.
– FCVLSSETJAC interfaces to CVodeSetJacTimes.
– FCVLSSETPREC interfaces to CVodeSetPreconditioner.
– FCVDENSESETJAC interfaces to CVodeSetJacFn.
– FCVBANDSETJAC interfaces to CVodeSetJacFn.
– FCVSPARSESETJAC interfaces to CVodeSetJacFn.
– FCVDIAG interfaces to CVDiag.

• Interface to the nonlinear solver interface

– FCVNLNSINIT interfaces to CVSetNonlinearSolver.

The user-supplied functions, each listed with the corresponding internal interface function which calls it (and its type within CVODE), are as follows:
In contrast to the case of direct use of cvode, and of most FORTRAN ODE solvers, the names of all user-supplied routines here are fixed, in order to maximize portability for the resulting mixed-language program.

### 5.2.4 Usage of the FCVODE interface module

The usage of FCVODE 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 cvode 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 FCVODE for rootfinding and with preconditioner modules is described in later subsections.

1. **Right-hand side specification**

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

   ```fortran
   SUBROUTINE FCVFUN(T, Y, YDOT, IPAR, RPAR, IER)
   DIMENSION Y(*), YDOT(*), IPAR(*), RPAR(*)
   
   It must set the YDOT array to \( f(t, y) \), the right-hand side of the ODE system, as function of \( T = t \) and the array \( Y = y \). The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FCVMALLOC. IER is an error return flag which should be set to 0 if successful, a positive value if a recoverable error occurred (in which case cvode will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted).
   ```

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**

   If using a nonlinear solver that requires a linear solver (e.g., the default Newton iteration) and the linear solver is a direct linear solver, then 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 nonlinear solver that requires a linear solver (e.g., the default Newton iteration) and one of the SUNLINSOL linear solver modules supplied with SUNDIALS, the user must make a call of the form

\[
\begin{align*}
&\text{CALL FSUNBANDLINSOLINIT}(\ldots) \\
&\text{CALL FSUNDENSELINSOLINIT}(\ldots) \\
&\text{CALL FSUNKLUINIT}(\ldots) \\
&\text{CALL FSUNLAPACKBANDINIT}(\ldots) \\
&\text{CALL FSUNLAPACKDENSEINIT}(\ldots) \\
&\text{CALL FSUNPCGINIT}(\ldots) \\
&\text{CALL FSUNSPBCGSINIT}(\ldots) \\
&\text{CALL FSUNSPFGMRINIT}(\ldots) \\
&\text{CALL FSUNSPGMRINIT}(\ldots) \\
&\text{CALL FSUNSPTFQMRINIT}(\ldots) \\
&\text{CALL FSUNSUPERLUMTINIT}(\ldots)
\end{align*}
\]

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 has been initialized, its solver parameters may be modified using a call to the functions

\[
\begin{align*}
&\text{CALL FSUNKLUSETORDERING}(\ldots) \\
&\text{CALL FSUNSUPERLUMTSETORDERING}(\ldots) \\
&\text{CALL FSUNPCGSETPRECTYPE}(\ldots) \\
&\text{CALL FSUNSUPERLUMTSETPRECTYPE}(\ldots) \\
&\text{CALL FSUNSPBCGSSETMAXL}(\ldots) \\
&\text{CALL FSUNSPBCGSSETPRECTYPE}(\ldots) \\
&\text{CALL FSUNSPFGMRSETGSTYPE}(\ldots) \\
&\text{CALL FSUNSPFGMRSETPRECTYPE}(\ldots) \\
&\text{CALL FSUNSPGMRSETGSTYPE}(\ldots) \\
&\text{CALL FSUNSPGMRSETPRECTYPE}(\ldots) \\
&\text{CALL FSUNSPTFQMRSETPRECTYPE}(\ldots) \\
&\text{CALL FSUNSPTFQMRSETMAXL}(\ldots)
\end{align*}
\]

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

5. SUNNONLINSOL module initialization

By default CVODE uses the SUNNONLINSOL implementation of Newton’s method defined by the SUNNONLINSOL_NEWTON module (see §9.2). To specify a non-default nonlinear solver in CVODE, the user’s program must create a SUNNONLINSOL object by calling the appropriate FORTRAN interface function to the constructor routine (see Chapter 9). For example, to create the SUNNONLINSOL_FIXEDPOINT solver call the function

\[
\text{CALL FSUNFIXEDPOINTINIT}(\ldots)
\]

in which the call sequence is described in §9.3.

6. Problem specification

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

Call

CALL FCVMALLOC(T0, Y0, METH, 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 cvode.

Arguments
T0 is the initial value of t.
Y0 is an array of initial conditions.
METH specifies the basic integration method: 1 for Adams (nonstiff) or 2 for BDF (stiff).
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 FCVEWTSET and provide the function FCVEWT.
RTOL is the relative tolerance (scalar).
ATOL is the absolute tolerance (scalar or array).
IOUT is an integer array of length 21 for integer optional outputs.
ROUT is a real array of length 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 IPAR and RPAR inside a user-provided routine will be propagated to all subsequent calls to such routines.
The optional outputs associated with the main cvode integrator are listed in Table 5.3.

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

SUBROUTINE FCVEWT (Y, EWT, IPAR, RPAR, IER)
DIMENSION Y(*), EWT(*), IPAR(*), RPAR(*)

It must set the positive components of the error weight vector EWT for the calculation of the WRMS norm of Y. On return, set IER = 0 if FCVEWT 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 FCVMALLOC.

If the FCVEWT routine is provided, then, following the call to FCVMALLOC, the user must make the call:

CALL FCVEWTSET (FLAG, IER)

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

7. Set optional inputs
Call FCVINSETII and/or FCVINSETRIN to set desired optional inputs, if any. See §5.2.5 for details.

8. Linear solver interface specification
To attach the linear solver (and optionally the matrix) objects initialized in steps 3 and 4 above, the user of fcvode must initialize the cvls linear solver interface. To attach any sunlinsol object (and optional sunmatrix object) to cvode, then following calls to initialize the sunlinsol (and sunmatrix) object(s) in steps 3 and 4 above, the user must make the call:

\[
\text{CALL FCVLSINIT (IER)}
\]

IER is an error return flag set on 0 on success, −1 if a memory failure occurred, or −2 for an illegal input.

The previous routines fcvdlsinit and fcvspilsinit 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.

**CVLS with dense Jacobian matrix**

As an option when using the cvls interface with the sunlinsol dense or sunlinsol lapackdense linear solvers, the user may supply a routine that computes a dense approximation of the system Jacobian \( J = \partial f/\partial y \). If supplied, it must have the following form:

\[
\text{SUBROUTINE FCVDJAC (NEQ, T, Y, FY, DJAC, H, IPAR, RPAR,}
\]

\[
&\quad \text{WK1, WK2, WK3, IER)}
\]

\[
\text{DIMENSION Y(*), FY(*), DJAC(NEQ,*), IPAR(*), RPAR(*),}
\]

\[
&\quad \text{WK1(*), WK2(*), WK3(*)}
\]

Typically this routine will use only NEQ, T, Y, and DJAC. It must compute the Jacobian and store it columnwise in DJAC. The input arguments T, Y, and FY contain the current values of \( t, y \), and \( f(t,y) \), respectively. The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FCVMALLOC. The vectors WK1, WK2, and WK3 of length NEQ are provided as work space for use in FCVDJAC. IER is an error return flag which should be set to 0 if successful, a positive value if a recoverable error occurred (in which case cvode will attempt to correct), or a negative value if FCVDJAC failed unrecoverably (in which case the integration is halted). 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 FCVDJAC uses difference quotient approximations, it may need to use the error weight array EWT and current stepsize \( H \) in the calculation of suitable increments. The array EWT can be obtained by calling FCVGETERRWEIGHTS using one of the work arrays as temporary storage for EWT. 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 either RPAR or a common block.

If the FCVDJAC routine is provided, then, following the call to FCVLSINIT, the user must make the call:

\[
\text{CALL FCVDENSESETJAC (FLAG, IER)}
\]

with FLAG ≠ 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.

**CVLS with band Jacobian matrix**

As an option when using the cvls 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 \( J = \partial f/\partial y \). If supplied, it must have the following form:

\[
\text{SUBROUTINE FCVBJAC(NEQ, MU, ML, MDIM, T, Y, FY, BJAC, H, IPAR, RPAR,}
\]

\[
&\quad \text{WK1, WK2, WK3, IER)}
\]

\[
\text{DIMENSION Y(*), FY(*), BJAC(MDIM,*), IPAR(*), RPAR(*),}
\]

\[
&\quad \text{WK1(*), WK2(*), WK3(*)}
\]
Typically this routine will use only NEQ, MU, ML, T, Y, and BJAC. It must load the MDIM by N array BJAC with the Jacobian matrix at the current \((t, y)\) 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 input arguments \(T, Y,\) and \(FY\) contain the current values of \(t, y,\) and \(f(t, y),\) respectively. The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FCVMALLOC. The vectors WK1, WK2, and WK3 of length NEQ are provided as work space for use in FCVBJAC. IER is an error return flag which should be set to 0 if successful, a positive value if a recoverable error occurred (in which case CVODE will attempt to correct), or a negative value if FCVBJAC failed unrecoverably (in which case the integration is halted). 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 FCVBJAC uses difference quotient approximations, it may need to use the error weight array EWT and current stepsize \(H\) in the calculation of suitable increments. The array EWT can be obtained by calling FCCVGETERRWEIGHTS using one of the work arrays as temporary storage for EWT. 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 either RPAR or a common block.

If the FCVBJAC routine is provided, then, following the call to FCVLSINIT, the user must make the call:

\[
\text{CALL FCVBJAC}(\text{FLAG, IER})
\]

with \(\text{FLAG} \neq 0\) 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.

**CVLS with sparse Jacobian matrix**

When using the CVLS interface with the SUNLINSOL_KLU or SUNLINSOL_SUPERLUMT linear solvers, the user must supply the FCVSPJAC routine that computes a compressed-sparse-column or compressed-sparse-row approximation of the system Jacobian \(J = \partial f/\partial y\). If supplied, it must have the following form:

\[
\text{SUBROUTINE FCVSPJAC}(T, Y, FY, N, NNZ, JDATA, JINDEXVALS, JINDEXPTRS, H, IPAR, RPAR, WK1, WK2, WK3, IER)
\]

It must load the \(N\) by \(N\) compressed sparse column [or compressed sparse row] matrix with storage for NNZ nonzeros, stored in the arrays JDATA, JINDEXVALS and JINDEXPTRS, with the Jacobian matrix at the current \((t, y)\) in CSC [or CSR] form (see sunmatrix_sparse.h for more information). The arguments are T, the current time; Y, an array containing state variables; FY, an array containing state derivatives; H, the number of matrix rows/columns in the Jacobian; NNZ, allocated length of nonzero storage; JDATA, nonzero values in the Jacobian (of length NNZ); JINDEXVALS, row [or column] indices for each nonzero in Jacobian (of length NNZ); JINDEXPTRS, pointers to each Jacobian column [or row] in the two preceding arrays (of length \(N+1\)); H, the current step size; IPAR, an array containing integer user data that was passed to FCVMALLOC; RPAR, an array containing real user data that was passed to FCVMALLOC; 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 FCVSPJAC routine has been provided, then following the call to FCVLSINIT, the following call must be made

\[
\text{CALL FCVSPARSESETJAC(IER)}
\]

The int return flag IER is an error return flag which is 0 for success or non-zero for an error.
CVLS with Jacobian-vector product

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

```
SUBROUTINE FCVJTIMES (V, FJV, T, Y, FY, H, IPAR, RPAR, WORK, IER)
   DIMENSION V(*), FJV(*), Y(*), FY(*), IPAR(*), RPAR(*), WORK(*)
```

Typically this routine will use only $T$, $Y$, and $FJV$. It must compute the product vector $Jv$, where the vector $v$ is stored in $V$, and store the product in $FJV$. The input arguments $T$, $Y$, and $FY$ contain the current values of $t$, $y$, and $f(t,y)$, respectively. On return, set $IER = 0$ if FCVJTIMES 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 FCVMALLOC. The vector $WORK$, of length commensurate with the input $Y0$ to FCVMALLOC, is provided as work space for use in FCVJTIMES.

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

```
SUBROUTINE FCVJTSETUP (T, Y, FY, H, IPAR, RPAR, IER)
   DIMENSION Y(*), FY(*), IPAR(*), RPAR(*)
```

Typically this routine will use only $T$ and $Y$. It should compute any necessary data for subsequent calls to FCVJTIMES. On return, set $IER = 0$ if FCVJSETUP 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 FCVMALLOC.

To indicate that the FCVJTIMES and FCVJTSETUP routines have been provided, then following the call to FCVLSINIT, the following call must be made

```
CALL FCVLSSETJAC (FLAG, IER)
```

with $FLAG \neq 0$ to specify use of the user-supplied Jacobian-times-vector setup and product routines. The argument $IER$ is an error return flag which is 0 for success or non-zero if an error occurred.

The previous routine FCVSPILSETJAC 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 FCVLSETJAC, the routine FCVJTSETUP must be provided, even if it is not needed, and it must return $IER=0$.

Notes

(a) If the user’s FCVJTIMES routine uses difference quotient approximations, it may need to use the error weight array $EWT$, the current stepsize $H$, and/or the unit roundoff, in the calculation of suitable increments.

(b) If needed in FCVJTIMES or FCVJTSETUP, the error weight array $EWT$ can be obtained by calling FCVGETERRWEIGHTS using a user-allocated array as temporary storage for $EWT$.

(c) If needed in FCVJTIMES or FCVJTSETUP, the unit roundoff can be obtained as the optional output ROUT(6) (available after the call to FCVMALLOC) and can be passed using either the $RPAR$ user data array, a common block or a module.

CVLS with preconditioning
5.2 FCVODE, an Interface Module for FORTRAN Applications

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

```fortran
SUBROUTINE FCVPSOL(T, Y, FY, R, Z, GAMMA, DELTA, LR, IPAR, RPAR, IER)
DIMENSION Y(*), FY(*), R(*), Z(*), IPAR(*), RPAR(*)
```

It must solve the preconditioner linear system \( Pz = r \), where \( r = R \) is input, and store the solution \( z \) in \( Z \). Here \( P \) is the left preconditioner if \( LR=1 \) and the right preconditioner if \( LR=2 \). The preconditioner (or the product of the left and right preconditioners if both are nontrivial) should be an approximation to the matrix \( I - \gamma J \), where \( I \) is the identity matrix, \( J \) is the system Jacobian, and \( \gamma = GAMMA \). The input arguments \( T, Y, \) and \( FY \) contain the current values of \( t, y, \) and \( f(t,y) \), respectively. On return, set \( IER = 0 \) if FCVPSOL 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 FCVMALLOC.

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

```fortran
SUBROUTINE FCVPSET(T, Y, FY, JOK, JCUR, GAMMA, H, IPAR, RPAR, IER)
DIMENSION Y(*), FY(*), EWT(*), IPAR(*), RPAR(*)
```

It must perform any evaluation of Jacobian-related data and preprocessing needed for the solution of the preconditioner linear systems by FCVPSOL. The input argument \( JOK \) allows for Jacobian data to be saved and reused: If \( JOK = 0 \), this data should be recomputed from scratch. If \( JOK = 1 \), a saved copy of it may be reused, and the preconditioner constructed from it. The input arguments \( T, Y, \) and \( FY \) contain the current values of \( t, y, \) and \( f(t,y) \), respectively. On return, set \( JCUR = 1 \) if Jacobian data was computed, and set \( JCUR = 0 \) otherwise. Also on return, set \( IER = 0 \) if FCVPSET 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 FCVMALLOC.

To indicate that the FCVPSET and FCVPSOL routines are supplied, then the user must call

```fortran
CALL FCVLSSETPREC(FLAG, IER)
```

with \( FLAG \neq 0 \). The return flag \( IER \) is 0 if successful, or negative if a memory error occurred. In addition, the user program must include preconditioner routines FCVPSOL and FCVPSET.

The previous routine FCVSPILSETPREC 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 FCVLSSETPREC, the routine FCVPSET must be provided, even if it is not needed, and it must return \( IER=0 \).

Notes

(a) If the user's FCVPSET routine uses difference quotient approximations, it may need to use the error weight array \( EWT \), the current stepsize \( H \), and/or the unit roundoff, in the calculation of suitable increments. Also, if FCVPSOL uses an iterative method in its solution, the residual vector \( \rho = r - Pz \) of the system should be made less than \( \text{DELT}A \) in weighted \( \ell_2 \) norm, i.e.
\[
\sqrt{\sum (\rho_i * EWT[i])^2} < \text{DELT}A.
\]

(b) If needed in FCVPSOL or FCVPSET, the error weight array \( EWT \) can be obtained by calling FCVGETERRWEIGHTS using a user-allocated array as temporary storage for \( EWT \).
(c) If needed in FCVP SOL or FCVPSET, the unit roundoff can be obtained as the optional output ROUT(6) (available after the call to FCVMALLOC) and can be passed using either the RPAR user data array, a common block or a module.

**CV DIAG** diagonal linear solver interface

CVODE is also packaged with a CVODE-specific diagonal approximate Jacobian and linear solver interface. This choice is appropriate when the Jacobian can be well-approximated by a diagonal matrix. The user must make the call:

```fortran
CALL FCVDIAG(IER)
```

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

There are no additional user-supplied routines for the CV DIAG interface.

Optional outputs specific to the CV DIAG case are listed in Table 5.3.

9. **Nonlinear solver interface specification**

If a non-default SUNNONLINSOL object was created in step 5, the user must attach it to CVODE with the call:

```fortran
CALL FCVNLSINIT(IER)
```

IER is an error return flag set on 0 on success or −1 if an error occurred.

Once attached, the user may specify non-default inputs for the SUNNONLINSOL object (e.g. the maximum number of nonlinear iterations) by calling appropriate FORTRAN interface routines (see Chapter 9).

10. **Problem solution**

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

```fortran
CALL FCVODE(TOUT, T, Y, ITASK, IER)
```

The arguments are as follows. TOUT specifies the next value of t at which a solution is desired (input). T is the value of t reached by the solver on output. Y is an array containing the computed solution on output. ITASK is a task indicator and should be set to 1 for normal mode (overshoot TOUT and interpolate), or to 2 for one-step mode (return after each internal step taken). IER is a completion flag and will be set to a positive value upon successful return or to a negative value if an error occurred. These values correspond to the CVode returns (see §4.5.6 and §B.2). The current values of the optional outputs are available in IO UT and ROUT (see Table 5.3).

11. **Additional solution output**

After a successful return from FCVODE, the routine FCVDKY may be used to obtain a derivative of the solution, of order up to the current method order, at any t within the last step taken. For this, make the following call:

```fortran
CALL FCVDKY(T, K, DKY, IER)
```

where T is the value of t at which solution derivative is desired, and K is the derivative order (0 ≤ K ≤ QU). On return, DKY is an array containing the computed K-th derivative of y. The value T must lie between TCUR − HU and TCUR. The return flag IER is set to 0 upon successful return or to a negative value to indicate an illegal input.

12. **Problem reinitialization**

To re-initialize the CVODE solver for the solution of a new problem of the same size as one already solved, make the following call:
CALL FCVREINIT(T0, Y0, IATOL, RTOL, ATOL, IER)

The arguments have the same names and meanings as those of FCVMALLOC. FCVREINIT performs
the same initializations as FCVMALLOC, but does no memory allocation, using instead the existing
internal memory created by the previous FCVMALLOC call. The call to specify the linear system
solution method may or may not be needed.

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 CVLS interface, as shown
above. If only linear solver parameters are being modified, then these calls may be made without
re-attaching to the CVLS interface.

13. Memory deallocation

   To free the internal memory created by the call to FCVMALLOC, FCVLSINIT, FNVINIT* and FSUN***MATINIT,
   make the call

   CALL FCVFREE

5.2.5 FCVODE optional input and output

In order to keep the number of user-callable FCVODE interface routines to a minimum, optional inputs
to the CVODE solver are passed through only three routines: FCVSETIIN for integer optional inputs,
FCVSETRIN for real optional inputs, and FCVSETVIN for real vector (array) optional inputs. These
functions should be called as follows:

   CALL FCVSETIIN(KEY, IVAL, IER)
   CALL FCVSETRIN(KEY, RVAL, IER)
   CALL FCVSETVIN(KEY, VVAL, IER)

   where KEY is a quoted string indicating which optional input is set (see Table 5.2), IVAL is the integer
   input value to be used, RVAL is the real input value to be used, VVAL is the real input array to be
   used, and IER is an integer return flag which is set to 0 on success and a negative value if a failure
   occurred. The integer IVAL should be declared in a manner consistent with C type long int.

   When using FCVSETVIN 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
   CVodeSetConstraints (§4.5.7.1) for details.

   The optional outputs from the CVODE 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 FCVMALLOC. Table 5.3 lists the entries in these two arrays and specifies the optional variable as well
as the CVODE function which is actually called to extract the optional output.

   For more details on the optional inputs and outputs, see §4.5.7 and §4.5.9.

   In addition to the optional inputs communicated through FCVSET* calls and the optional outputs
   extracted from IOUT and ROUT, the following user-callable routines are available:

   To obtain the error weight array EWT, containing the multiplicative error weights used the WRMS
   norms, make the following call:

   CALL FCVGETEWEIGHTS (EWT, IER)

   This computes the EWT array normally defined by Eq. (2.7). 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.

   To obtain the estimated local errors, following a successful call to FCVSOLVE, make the following
call:

   CALL FCVGETESTLOCALERR (ELE, IER)
Table 5.2: Keys for setting fcVODE optional inputs

<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 (BDF), 12 (Adams)</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>7</td>
</tr>
<tr>
<td>MAX_NITERS</td>
<td>Maximum no. of nonlinear iterations</td>
<td>3</td>
</tr>
<tr>
<td>MAX_CONVFAIL</td>
<td>Maximum no. of convergence failures</td>
<td>10</td>
</tr>
<tr>
<td>HNIL_WARNINGS</td>
<td>Maximum no. of warnings for $t_n + h = t_n$</td>
<td>10</td>
</tr>
<tr>
<td>STAB_LIM</td>
<td>Flag to activate stability limit detection</td>
<td>0</td>
</tr>
</tbody>
</table>

Real optional inputs (FCVSETRIN)

<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>MIN_STEP</td>
<td>Minimum absolute step size</td>
<td>0.0</td>
</tr>
<tr>
<td>STOP_TIME</td>
<td>Value of $t_{\text{stop}}$</td>
<td>undefined</td>
</tr>
<tr>
<td>NLCONV_COEF</td>
<td>Coefficient in the nonlinear convergence test</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Real vector optional inputs (FCVSETVIN)

<table>
<thead>
<tr>
<th>Key</th>
<th>Optional Input</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSTR_VEC</td>
<td>Inequality constraints on solution</td>
<td>undefined</td>
</tr>
</tbody>
</table>

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

5.2.6 Usage of the FCVROOT interface to rootfinding

The FCVROOT interface package allows programs written in FORTRAN to use the rootfinding feature of the CVODE solver module. The user-callable functions in FCVROOT, with the corresponding CVODE functions, are as follows:

- FCVROOTINIT interfaces to CVodeRootInit.
- FCVROOTINFO interfaces to CVodeGetRootInfo.
- FCVROOTFREE interfaces to CVodeRootFree.

Note that at this time, FCVROOT does not provide support to specify the direction of zero-crossing that is to be monitored. Instead, all roots are considered. However, the actual direction of zero-crossing is reported (through the sign of the non-zero elements in the array INFO returned by FCVROTINFO).

In order to use the rootfinding feature of CVODE, the following call must be made, after calling FCVMALLOC but prior to calling FCVODE, to allocate and initialize memory for the FCVROOT module:

```fortran
CALL FCVROOTINIT (NRTFN, IER)
```

The arguments are as follows: NRTFN is the number of root functions. IER is a return completion flag; its values are 0 for success, −1 if the CVODE memory was NULL, and −11 if a memory allocation failed.

To specify the functions whose roots are to be found, the user must define the following routine:

```fortran
SUBROUTINE FCVROOTFN (T, Y, G, IPAR, RPAR, IER)
DIMENSION Y(*), G(*), IPAR(*), RPAR(*)
```
Table 5.3: Description of the FCVODE optional output arrays IOUT and ROUT

### Integer output array IOUT

<table>
<thead>
<tr>
<th>Index</th>
<th>Optional output</th>
<th>CVODE function</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>CVODE main solver</td>
</tr>
<tr>
<td>1</td>
<td>LENRW</td>
<td>CVodeGetWorkSpace</td>
</tr>
<tr>
<td>2</td>
<td>LENIW</td>
<td>CVodeGetWorkSpace</td>
</tr>
<tr>
<td>3</td>
<td>NST</td>
<td>CVodeGetNumSteps</td>
</tr>
<tr>
<td>4</td>
<td>NFE</td>
<td>CVodeGetNumRhsEvals</td>
</tr>
<tr>
<td>5</td>
<td>NETF</td>
<td>CVodeGetNumErrTestFails</td>
</tr>
<tr>
<td>6</td>
<td>NCFN</td>
<td>CVodeGetNumNonlinSolvConvFails</td>
</tr>
<tr>
<td>7</td>
<td>NNI</td>
<td>CVodeGetNumNonlinSolvIters</td>
</tr>
<tr>
<td>8</td>
<td>NSETUPS</td>
<td>CVodeGetNumLinSolvSetups</td>
</tr>
<tr>
<td>9</td>
<td>QU</td>
<td>CVodeGetLastOrder</td>
</tr>
<tr>
<td>10</td>
<td>QCUR</td>
<td>CVodeGetCurrentOrder</td>
</tr>
<tr>
<td>11</td>
<td>NOR</td>
<td>CVodeGetNumStabLimOrderReds</td>
</tr>
<tr>
<td>12</td>
<td>NGE</td>
<td>CVodeGetNumGEvals</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CVLS linear solver interface</td>
</tr>
<tr>
<td>13</td>
<td>LENRWLS</td>
<td>CVodeGetLinWorkSpace</td>
</tr>
<tr>
<td>14</td>
<td>LENIWLS</td>
<td>CVodeGetLinWorkSpace</td>
</tr>
<tr>
<td>15</td>
<td>LS_FLAG</td>
<td>CVodeGetLastLinFlag</td>
</tr>
<tr>
<td>16</td>
<td>NFELS</td>
<td>CVodeGetNumLinRhsEvals</td>
</tr>
<tr>
<td>17</td>
<td>NJE</td>
<td>CVodeGetNumJacEvals</td>
</tr>
<tr>
<td>18</td>
<td>NJTS</td>
<td>CVodeGetNumJTSolveEvals</td>
</tr>
<tr>
<td>19</td>
<td>NJTV</td>
<td>CVodeGetNumJtimesEvals</td>
</tr>
<tr>
<td>20</td>
<td>NPE</td>
<td>CVodeGetNumPrecEvals</td>
</tr>
<tr>
<td>21</td>
<td>NPS</td>
<td>CVodeGetNumPrecSolves</td>
</tr>
<tr>
<td>22</td>
<td>NLI</td>
<td>CVodeGetNumLinIters</td>
</tr>
<tr>
<td>23</td>
<td>NCFL</td>
<td>CVodeGetNumLinConvFails</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CVDIAG linear solver interface</td>
</tr>
<tr>
<td>13</td>
<td>LENRWLS</td>
<td>CVDiagGetWorkSpace</td>
</tr>
<tr>
<td>14</td>
<td>LENIWLS</td>
<td>CVDiagGetWorkSpace</td>
</tr>
<tr>
<td>15</td>
<td>LS_FLAG</td>
<td>CVDiagGetLastLinFlag</td>
</tr>
<tr>
<td>16</td>
<td>NFELS</td>
<td>CVDiagGetNumRhsEvals</td>
</tr>
</tbody>
</table>

### Real output array ROUT

<table>
<thead>
<tr>
<th>Index</th>
<th>Optional output</th>
<th>CVODE function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>H0U</td>
<td>CVodeGetActualInitStep</td>
</tr>
<tr>
<td>2</td>
<td>HU</td>
<td>CVodeGetLastStep</td>
</tr>
<tr>
<td>3</td>
<td>HCUR</td>
<td>CVodeGetCurrentStep</td>
</tr>
<tr>
<td>4</td>
<td>TCUR</td>
<td>CVodeGetCurrentTime</td>
</tr>
<tr>
<td>5</td>
<td>TOLSF</td>
<td>CVodeGetTolScaleFactor</td>
</tr>
<tr>
<td>6</td>
<td>UROUND</td>
<td>unit roundoff</td>
</tr>
</tbody>
</table>
It must set the $G$ array, of length $NRTFN$, with components $g_i(t, y)$, as a function of $T = t$ and the array $Y = y$. The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FCVMLLOC. Set IER on 0 if successful, or on a non-zero value if an error occurred.

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

\[
\text{CALL FCVROOTINFO} \ (NRTFN, \ INFO, \ IER)
\]

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

The total number of calls made to the root function FCVROOTFN, denoted $NGE$, can be obtained from IOUT(12). If the FCVODE/CVODE memory block is reinitialized to solve a different problem via a call to FCVREINIT, then the counter $NGE$ is reset to zero.

To free the memory resources allocated by a prior call to FCVROOTINIT, make the following call:

\[
\text{CALL FCVROOTFREE}
\]

### 5.2.7 Usage of the FCVB P interface to CVBANDPRE

The FCVB P interface sub-module is a package of C functions which, as part of the FCVODE interface module, support the use of the CVODE solver with the serial NVECTOR_SERIAL module or multi-threaded NVECTOR_OPENMP or NVECTOR_PTHREADS, and the combination of the CVBANDPRE preconditioner module (see §4.7.1) with the CVLS interface and any of the Krylov iterative linear solvers.

The two user-callable functions in this package, with the corresponding CVODE function around which they wrap, are:

- FCVBPINIT interfaces to CVBandPrecInit.
- FCVBPOPT interfaces to CVBANDPRE optional output functions.

As with the rest of the FCVODE routines, the names of the user-supplied routines are mapped to actual values through a series of definitions in the header file fcvbp.h.

The following is a summary of the usage of this module. Steps that are unchanged from the main program described in §5.2.4 are grayed-out.

1. Right-hand side specification
2. NVECTOR module initialization
3. SUNLINSOL module initialization
   - Initialize one of the iterative SUNLINSOL modules, by calling one of FSUNPCGINIT, FSUNSPBCGSINIT, FSUNSPFGMRINIT, FSUNSPGMRINIT or FSUNSPTFQMRINIT.
4. SUNNONLINSOL module initialization
5. Problem specification
6. Set optional inputs
7. Linear solver interface specification
   - First, initialize the CVLS linear solver interface by calling FCVLSINIT.
   - Then, to initialize the CVBANDPRE preconditioner, make the following call:
CALL FCVBPINIT(NEQ, MU, ML, IER)

The arguments are as follows. NEQ is the problem size. MU and ML are the upper and lower half-bandwidths of the band matrix that is retained as an approximation of the Jacobian. IER is a return completion flag. A value of 0 indicates success, while a value of −1 indicates that a memory failure occurred.

Optionally, to specify that cvls should use the supplied FCVJTIMES and FCVJTSETUP, make the call

CALL FCVLSSETJAC(FLAG, IER)

with FLAG ≠ 0 (see step 8 in §5.2.4 for details).

8. Nonlinear solver interface specification

9. Problem solution

10. Additional solution output

11. cvbandpre Optional outputs

Optional outputs specific to the cvls solver interface are listed in Table 5.3. To obtain the optional outputs associated with the cvbandpre module, make the following call:

CALL FCVBPOPT(LENRWBP, LENIWBP, NFEBP)

The arguments should be consistent with C type long int. Their returned values are as follows: LENRWBP is the length of real preconditioner work space, in realtype words. LENIWBP is the length of integer preconditioner work space, in integer words. NFEBP is the number of \( f(t, y) \) evaluations (calls to FCVFUN) for difference-quotient banded Jacobian approximations.

12. Memory deallocation

(The memory allocated for the FCVBP module is deallocated automatically by FCVFREE.)

### 5.2.8 Usage of the FCVBBD interface to CVBBDPRE

The FCVBBD interface sub-module is a package of C functions which, as part of the FCVODE interface module, support the use of the cvode solver with the parallel nvector_parallel module, and the combination of the CVBBDPRE preconditioner module (see §4.7.2) with any of the Krylov iterative linear solvers.

The user-callable functions in this package, with the corresponding CVODE and CVBBDPRE functions, are as follows:

- **FCVBBDINIT** interfaces to CVBBDPrecInit.
- **FCVBBDREINIT** interfaces to CVBBDPrecReInit.
- **FCVBBDOP** interfaces to CVBBDPRE optional output functions.

In addition to the Fortran right-hand side function FCVFUN, the user-supplied functions used by this package, are listed below, each with the corresponding interface function which calls it (and its type within CVBBDPRE or CVODE):

<table>
<thead>
<tr>
<th>FCVBBD routine (Fortran, user-supplied)</th>
<th>CVODE function (C, interface)</th>
<th>CVODE type of interface function</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCVLOCFN</td>
<td>FCVgloc</td>
<td>CVLocalFn</td>
</tr>
<tr>
<td>FCVCOMMF</td>
<td>FCVcfn</td>
<td>CVCommFn</td>
</tr>
<tr>
<td>FCVJTIMES</td>
<td>FCVJtimes</td>
<td>CVLSJacTimesVecFn</td>
</tr>
<tr>
<td>FCVJTSETUP</td>
<td>FCVJSetup</td>
<td>CVLSJacTimesSetupFn</td>
</tr>
</tbody>
</table>
As with the rest of the FCVODE 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 §5.2.3, the names of the user-supplied routines are mapped to actual values through a series of definitions in the header file fcvbbd.h.

The following is a summary of the usage of this module. Steps that are unchanged from the main program described in §5.2.4 are grayed-out.

1. Right-hand side specification
2. NVECTOR module initialization
3. SUNLINSOL module initialization
   - Initialize one of the iterative SUNLINSOL modules, by calling one of FSUNPCGINIT, FSUNSPBCGSINIT, FSUNSPFGMRINIT, FSUNSPGMRINIT or FSUNSPTFQMRINIT.
4. SUNNONLINSOL module initialization
5. Problem specification
6. Set optional inputs
7. Linear solver interface specification
   - First, initialize the cvls iterative linear solver interface by calling FCVLSINIT.
   - Then, to initialize the cvbbdpre preconditioner, make the following call:

   ```fortran
   CALL FCVBBDINIT(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.

   Optionally, to specify that SPGMR, SPBCGS, or SPTFQMR should use the supplied FCVJTIMES, make the call

   ```fortran
   CALL FCVLSSETJAC(FLAG, IER)
   ```

   with FLAG ≠ 0 (see step 8 in §5.2.4 for details).
8. Nonlinear solver interface specification
9. Problem solution
10. Additional solution output
11. cvbbdpre Optional outputs
   - Optional outputs specific to the cvls solver interface are listed in Table 5.3. To obtain the optional outputs associated with the cvbbdpre module, make the following call:

   ```fortran
   CALL FCVBBDOPT(LENRWBBD, LENIWBBD, NGEBBD)
   ```
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. These sizes are local to the current processor.
- `NGEBBD` is the number of \(g(t,y)\) evaluations (calls to `FCVLOCFN`) 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 `cvbbdpre` preconditioner, then the `cvode` package can be re-initialized for the second and subsequent problems by calling `FCVREINIT`, following which a call to `FCVBBDINIT` may or may not be needed. If the input arguments are the same, no `FCVBBDINIT` call is needed. If there is a change in input arguments other than `MU` or `ML`, then the user program should make the call

```
CALL FCVBBDREINIT(NLOCAL, MUDQ, MLDQ, DQRELY, IER)
```

This reinitializes the `cvbbdpre` preconditioner, but without reallocating its memory. The arguments of the `FCVBBDREINIT` routine have the same names and meanings as those of `FCVBBDINIT`. If the value of `MU` or `ML` is being changed, then a call to `FCVBBDINIT` must be made. Finally, if there is a change in any of the linear solver inputs, then a call to one of `FSUNPCGINIT`, `FSUNSPBCGSINIT`, `FSUNSPFGMRINIT`, `FSUNSPGMRINIT` or `FSUNSPTFQMRINIT`, followed by a call to `FCVLSINIT` must also be made; in this case the linear solver memory is reallocated.

### 13. Memory deallocation

(The memory allocated for the `FCVBBBD` module is deallocated automatically by `FCVFREE`.)

### 14. User-supplied routines

The following two routines must be supplied for use with the `CVBBDPRE` module:

```
SUBROUTINE FCVLOCFN (NLOC, T, YLOC, GLOC, IPAR, RPAR, IER)
DIMENSION YLOC(*), GLOC(*), IPAR(*), RPAR(*)
```

This routine is to evaluate the function \(g(t,y)\) approximating \(f\) (possibly identical to \(f\)), in terms of \(T = t\), and the array `YLOC` (of length `NLOC`), which is the sub-vector of `y` local to this processor. The resulting (local) sub-vector is to be stored in the array `GLOC`. The arrays `IPAR` (of integers) and `RPAR` (of reals) contain user data and are the same as those passed to `FCVMALLOC`. `IER` is an error return flag which should be set to 0 if successful, a positive value if a recoverable error occurred (in which case `CVODE` will attempt to correct), or a negative value if `FCVLOCFN` failed unrecoverably (in which case the integration is halted).

```
SUBROUTINE FCVCOMMFN (NLOC, T, YLOC, IPAR, RPAR, IER)
DIMENSION YLOC(*), IPAR(*), RPAR(*)
```

This routine is to perform the inter-processor communication necessary for the `FCVLOCFN` routine. Each call to `FCVCOMMFN` is preceded by a call to the right-hand side routine `FCVFUN` with the same arguments `T` and `YLOC`. The arrays `IPAR` (of integers) and `RPAR` (of reals) contain user data and are the same as those passed to `FCVMALLOC`. `IER` is an error return flag (currently not used; set `IER=0`). Thus `FCVCOMMFN` can omit any communications done by `FCVFUN` if relevant to the evaluation of `GLOC`. `IER` is an error return flag which should be set to 0 if successful, a positive value if a recoverable error occurred (in which case `CVODE` will attempt to correct), or a negative value if `FCVCOMMFN` failed unrecoverably (in which case the integration is halted).

The subroutine `FCVCOMMFN` must be supplied even if it is not needed and must return `IER=0`.

Optionally, the user can supply routines `FCVJT_TIMES` and `FCVJ_SETUP` for the evaluation of Jacobian-vector products, as described above in step 8 in §5.2.4.
Chapter 6

Description of the NVVECTOR 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 NVVECTOR implementation. Users can provide their own specific implementation of the NVVECTOR 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

```c
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

```c
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 (*nvconst)(realtype, N_Vector);
    void (*nvprod)(N_Vector, N_Vector, N_Vector);
    void (*nvdiv)(N_Vector, N_Vector, N_Vector);
    void (*nvscale)(realtype, N_Vector, N_Vector);
    void (*nvabs)(N_Vector, N_Vector);
    void (*nvinv)(N_Vector, N_Vector);
    void (*nvaddconst)(N_Vector, realtype, N_Vector);
    realtype (*nvdotprod)(N_Vector, N_Vector);
};
```
realtype (*nvmaxnorm)(N_Vector);
realtype (*nvwrmsnorm)(N_Vector, N_Vector);
realtype (*nvwrmsnormmask)(N_Vector, N_Vector, N_Vector);
realtype (*nvmin)(N_Vector);
realtype (*nvwl2norm)(N_Vector, N_Vector);
realtype (*nvl1norm)(N_Vector);
void (*nvcompare)(realtype, N_Vector, N_Vector);
booleantype (*nvinvtest)(N_Vector, N_Vector);
booleantype (*nvconstrmask)(N_Vector, N_Vector, N_Vector);
realtype (*nvminquotient)(N_Vector, N_Vector);
int (*nvlinearcombination)(int, realtype*, N_Vector*, N_Vector);
int (*nvscaleaddmulti)(int, realtype*, N_Vector, N_Vector*);
int (*nvdotprodmulti)(int, N_Vector, N_Vector*, realtype*);
int (*nvlinearcombinationvectorarray)(int, realtype, N_Vector*, realtype,
                                       N_Vector*, N_Vector*);
int (*nvlinearcombinationvectorarray)(int, int, realtype*, N_Vector*,
                                      N_Vector**, N_Vector**);
int (*nvlinearcombinationvectorarray)(int, int, realtype*, N_Vector**,
                                      N_Vector*);
realtype (*nvdotprodlocal)(N_Vector, N_Vector);
realtype (*nvmaxnormlocal)(N_Vector);
realtype (*nvminlocal)(N_Vector);
realtype (*nvlinormlocal)(N_Vector);
booleantype (*nvinvtestlocal)(N_Vector, N_Vector);
booleantype (*nvconstrmasklocal)(N_Vector, N_Vector, N_Vector);
realtype (*nvminquotientlocal)(N_Vector, N_Vector);
realtype (*nvwsqrsumlocal)(N_Vector, N_Vector);
realtype (*nvwsqrsummasklocal)(N_Vector, 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 NVECTOR objects are combined together within a NVECTOR_MANYVECTOR object (see Section 6.12). If a particular NVECTOR implementation defines a local reduction operation as NULL, the NVECTOR_MANYVECTOR module will automatically call standard vector reduction operations as necessary to complete the desired operation. All SUNDIALS-provided NVECTOR implementations include these local reduction operations, which may be used as templates for user-defined NVECTOR implementations.

Finally, note that the generic NVECTOR module defines the functions N_VCloneVectorArray and N_VCloneVectorArrayEmpty. Both functions create (by cloning) an array of count variables of type N_Vector, each of the same type as an existing N_Vector. Their prototypes are

\[
\text{N_Vector *N_VCloneVectorArray(int count, N_Vector w);} \\
\text{N_Vector *N_VCloneVectorArrayEmpty(int count, N_Vector w);} \\
\]

and their definitions are based on the implementation-specific N_VClone and N_VCloneEmpty operations, respectively.

An array of variables of type N_Vector can be destroyed by calling N_VDestroyVectorArray, whose prototype is

\[
\text{void N_VDestroyVectorArray(N_Vector *vs, int count);} \\
\]

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

A particular implementation of the NVECTOR module must:

- Specify the content field of N_Vector.
- Define and implement the vector operations. Note that the names of these routines should be unique to that implementation in order to permit using more than one NVECTOR module (each with different N_Vector internal data representations) in the same code.
- Define and implement user-callable constructor and destructor routines to create and free an N_Vector with the new content field and with ops pointing to the new vector operations.
- Optionally, define and implement additional user-callable routines acting on the newly defined N_Vector (e.g., a routine to print the content for debugging purposes).
- Optionally, provide accessor macros as needed for that particular implementation to be used to access different parts in the content field of the newly defined N_Vector.

Each NVECTOR implementation included in SUNDIALS has a unique identifier specified in enumeration and shown in Table 6.1. It is recommended that a user-supplied NVECTOR implementation use the SUNDIALS_NVEC_CUSTOM identifier.
Table 6.1: 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_PARHYP</td>
<td>hypre ParHyp parallel vector</td>
<td>4</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_OPENMPDEV</td>
<td>OpenMP parallel vector with device offloading</td>
<td>8</td>
</tr>
<tr>
<td>SUNDIALS_NVEC_TRILINOS</td>
<td>Trilinos Tpetra 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>id = N_VGetVectorID(w); Returns the vector type identifier for the vector w. It is used to determine the vector implementation type (e.g. serial, parallel,...) from the abstract N_Vector interface. Returned values are given in Table 6.1.</td>
</tr>
<tr>
<td>N_VClone</td>
<td>v = N_VClone(w); Creates a new N_Vector of the same type as an existing vector w and sets the ops field. It does not copy the vector, but rather allocates storage for the new vector.</td>
</tr>
<tr>
<td>N_VCloneEmpty</td>
<td>v = N_VCloneEmpty(w); Creates a new N_Vector of the same type as an existing vector w and sets the ops field. It does not allocate storage for data.</td>
</tr>
<tr>
<td>N_VDestroy</td>
<td>N_Destroy(v); Destroys the N_Vector v and frees memory allocated for its internal data.</td>
</tr>
<tr>
<td>N_VSpace</td>
<td>N_Space(nvSpec, &amp;lrw, &amp;liw); Returns storage requirements for one N_Vector. lrw contains the number of realtype words and liw 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>Returns a pointer to a <code>realtype</code> array from the <code>N_Vector</code> v. Note that this assumes that the internal data in <code>N_Vector</code> is a contiguous array of <code>realtype</code>. This routine is only used in the solver-specific interfaces to the dense and banded (serial) linear solvers, the sparse linear solvers (serial and threaded), and in the interfaces to the banded (serial) and band-block-diagonal (parallel) preconditioner modules provided with SUNDIALS.</td>
</tr>
<tr>
<td>N_VSetArrayPointer</td>
<td>Overwrites the data in an <code>N_Vector</code> with a given array of <code>realtype</code>. Note that this assumes that the internal data in <code>N_Vector</code> is a contiguous array of <code>realtype</code>. This routine is only used in the interfaces to the dense (serial) linear solver, hence need not exist in a user-supplied <code>NVECTOR</code> module for a parallel environment.</td>
</tr>
<tr>
<td>N_VGetCommunicator</td>
<td>Returns a pointer to the MPI <code>Comm</code> object associated with the vector (if applicable). For MPI-unaware vector implementations, this should return <code>NULL</code>.</td>
</tr>
<tr>
<td>N_VGetLength</td>
<td>Returns the global length (number of ‘active’ entries) in the <code>NVECTOR</code> 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>Performs the operation ( z = ax + by ), where a and b are <code>realtype</code> scalars and x and y are of type <code>N_Vector</code>: ( z_i = ax_i + by_i ), ( i = 0, \ldots, n - 1 ).</td>
</tr>
<tr>
<td>N_VConst</td>
<td>Sets all components of the <code>N_Vector z</code> to <code>realtype</code> c: ( z_i = c, i = 0, \ldots, n - 1 ).</td>
</tr>
<tr>
<td>N_VProd</td>
<td>Sets the <code>N_Vector z</code> to be the component-wise product of the <code>N_Vector</code> 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>
<tr>
<td>N_VDiv</td>
<td>Sets the <code>N_Vector z</code> to be the component-wise ratio of the <code>N_Vector</code> inputs x and y: ( z_i = x_i / y_i ), ( i = 0, \ldots, n - 1 ).</td>
</tr>
</tbody>
</table>

*continued on next page*
Description of the NVECTOR module

<table>
<thead>
<tr>
<th>Name</th>
<th>Usage and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N_VScale</td>
<td><code>N_VScale(c, x, z);</code> Scales the N_Vector x by the realtype scalar c and returns the result in z: <code>z_i = cx_i</code>, <code>i = 0, ..., n - 1</code>.</td>
</tr>
<tr>
<td>N_VAbs</td>
<td><code>N_VAbs(x, z);</code> 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><code>N_VInv(x, z);</code> Sets the components of the N_Vector z to be the inverses of the components of the N_Vector x: <code>z_i = 1.0 / x_i</code>, <code>i = 0, ..., n - 1</code>. 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><code>N_VAddConst(x, b, z);</code> Adds the realtype scalar b to all components of x and returns the result in the N_Vector z: <code>z_i = x_i + b</code>, <code>i = 0, ..., n - 1</code>.</td>
</tr>
<tr>
<td>N_VDotProd</td>
<td><code>d = N_VDotProd(x, y);</code> Returns the value of the ordinary dot product of x and y: <code>d = \sum_{i=0}^{n-1} x_i y_i</code>.</td>
</tr>
<tr>
<td>N_VMaxNorm</td>
<td><code>m = N_VMaxNorm(x);</code> Returns the maximum norm of the N_Vector x: `m = \max_i</td>
</tr>
<tr>
<td>N_VRmsNorm</td>
<td><code>m = N_VRmsNorm(x, w)</code> Returns the weighted root-mean-square norm of the N_Vector x with realtype weight vector w: <code>m = \sqrt{\sum_{i=0}^{n-1} (x_i w_i)^2} / n</code>.</td>
</tr>
<tr>
<td>N_VRmsNormMask</td>
<td><code>m = N_VRmsNormMask(x, w, id)</code> 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: <code>m = \sqrt{\sum_{i=0}^{n-1} (x_i w_i H(id_i))^2} / n</code>, where <code>H(\alpha) = \begin{cases} 1 &amp; \alpha &gt; 0 \\ 0 &amp; \alpha \leq 0 \end{cases}</code>.</td>
</tr>
<tr>
<td>N_VMin</td>
<td><code>m = N_VMin(x);</code> Returns the smallest element of the N_Vector x: <code>m = \min_i x_i</code>.</td>
</tr>
<tr>
<td>N_VL2Norm</td>
<td><code>m = N_VL2Norm(x, w);</code> Returns the weighted Euclidean $\ell_2$ norm of the N_Vector x with realtype weight vector w: <code>m = \sqrt{\sum_{i=0}^{n-1} (x_i w_i)^2}</code>.</td>
</tr>
</tbody>
</table>

continued on next page
### Usage and Description

<table>
<thead>
<tr>
<th>Name</th>
<th>Usage and Description</th>
</tr>
</thead>
</table>
| N_VL1Norm       | \( m = N_{VL1Norm}(x); \)  
|                 | Returns the \( \ell_1 \) norm of the N_Vector \( x \): \( m = \sum_{i=0}^{n-1} |x_i| \).                                                                 |
| N_VCompare      | \( N_{VCompare}(c, x, z); \)  
|                 | Comparisons the components of the N_Vector \( x \) to the realtype scalar \( c \) and returns an N_Vector \( z \) such that: \( z_i = 1.0 \) if \( |x_i| \geq c \) and \( z_i = 0.0 \) otherwise. |
| N_VInvTest      | \( t = N_{VInvTest}(x, z); \)  
|                 | 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 SUNTRUE if all components of \( x \) are nonzero (successful inversion) and returns SUNFALSE otherwise. |
| N_VConstrMask   | \( t = N_{VConstrMask}(c, x, m); \)  
|                 | Performs the following constraint tests: \( x_i > 0 \) if \( c_i = 2 \), \( x_i \geq 0 \) if \( c_i = 1 \), \( x_i \leq 0 \) if \( c_i = -1 \), \( x_i < 0 \) if \( c_i = -2 \). There is no constraint on \( x_i \) if \( c_i = 0 \). This routine returns a boolean assigned to SUNFALSE if any 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_VMinQuotient  | \( \text{minq} = N_{VMinQuotient}(\text{num}, \text{denom}); \)  
|                 | This routine returns the minimum of the quotients obtained by term-wise dividing \( \text{num} \) by \( \text{denom} \). 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 6.3: Description of the NVectors fused operations
### Name | Usage and Description
--- | ---
N_VLinearCombination | \( \text{ier} = \text{N_VLinearCombination}(n_v, c, X, Z); \)
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 \text{realtype*}), \( X \) is an array of \( n_v \) vectors (type \text{N_Vector*}), and \( Z \) is the output vector (type \text{N_Vector}). If the output vector \( Z \) is one of the vectors in \( X \), then it \textit{must} be the first vector in the vector array. The operation returns 0 for success and a non-zero value otherwise.

N_VScaleAddMulti | \( \text{ier} = \text{N_VScaleAddMulti}(n_v, c, x, Y, Z); \)
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 \text{realtype*}), \( x \) is the vector (type \text{N_Vector}) to be scaled and added to each vector in the vector array of \( n_v \) vectors \( Y \) (type \text{N_Vector*}), and \( Z \) (type \text{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 | \( \text{ier} = \text{N_VDotProdMulti}(n_v, x, Y, d); \)
This routine computes the dot product of a vector with \( n_v \) other vectors:
\[
d_j = \sum_{i=0}^{n_v-1} x_i y_{j,i}, \quad j = 0, \ldots, n_v - 1,
\]
where \( d \) (type \text{realtype*}) is an array of \( n_v \) scalars containing the dot products of the vector \( x \) (type \text{N_Vector}) with each of the \( n_v \) vectors in the vector array \( Y \) (type \text{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);</td>
</tr>
<tr>
<td></td>
<td>This routine computes the linear sum of two vector arrays containing (nv) 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, nv - 1,]</td>
</tr>
<tr>
<td></td>
<td>where (a) and (b) are \texttt{realtype} scalars and (X, Y,) and (Z) are arrays of (nv) vectors (type \texttt{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);</td>
</tr>
<tr>
<td></td>
<td>This routine scales each vector of (n) elements in a vector array of (nv) vectors by a potentially different constant:</td>
</tr>
<tr>
<td></td>
<td>[z_{j,i} = c_j x_{j,i}, \quad i = 0, \ldots, n - 1 \quad j = 0, \ldots, nv - 1,]</td>
</tr>
<tr>
<td></td>
<td>where (c) is an array of (nv) scalars (type \texttt{realtype*}) and (X) and (Z) are arrays of (nv) vectors (type \texttt{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);</td>
</tr>
<tr>
<td></td>
<td>This routine sets each element in a vector of (n) elements in a vector array of (nv) 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, nv - 1,]</td>
</tr>
<tr>
<td></td>
<td>where (c) is a \texttt{realtype} scalar and (X) is an array of (nv) vectors (type \texttt{N_Vector*}). The operation returns 0 for success and a non-zero value otherwise.</td>
</tr>
</tbody>
</table>
### Name: N_VWrmsNormVectorArray

- **Usage and Description:**
  
  ```
  ier = N_VWrmsNormVectorArray(nv, X, W, m);
  ```
  
  This routine computes the weighted root mean square norm of \( n_v \) 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 `realtype*) contains the \( n_v \) norms of the vectors in the vector array \( X \) (type `N_Vector*`) with corresponding weight vectors \( W \) (type `N_Vector*`). The operation returns 0 for success and a non-zero value otherwise.

### Name: N_VWrmsNormMaskVectorArray

- **Usage and Description:**
  
  ```
  ier = N_VWrmsNormMaskVectorArray(nv, X, W, id, m);
  ```
  
  This routine computes the masked weighted root mean square norm of \( n_v \) vectors with \( n \) elements:
  
  \[
  m_j = \left( \frac{1}{n} \sum_{i=0}^{n-1} (x_{j,i}w_{j,i}H(id_i))^2 \right)^{1/2}, \quad j = 0, \ldots, n_v - 1,
  \]
  
  \( H(id_i) = 1 \) for \( id_i > 0 \) and is zero otherwise, \( m \) (type `realtype*) contains the \( n_v \) norms of the vectors in the vector array \( X \) (type `N_Vector*`) with corresponding weight vectors \( W \) (type `N_Vector*`) and mask vector \( id \) (type `N_Vector`). The operation returns 0 for success and a non-zero value otherwise.

### Name: N_VScaleAddMultiVectorArray

- **Usage and Description:**
  
  ```
  ier = N_VScaleAddMultiVectorArray(nv, ns, c, X, YY, ZZ);
  ```
  
  This routine scales and adds a vector in a vector array of \( n_v \) vectors to the corresponding vector in \( n_s \) 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 `realtype`), \( X \) is a vector array of \( n_v \) vectors (type `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 `N_Vector**`) and stored in the output array of vector arrays \( ZZ \) (type `N_Vector**`). The operation returns 0 for success and a non-zero value otherwise.
N_Vector Linear Combination Vector Array

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 \texttt{realtype*}), \( XX \) (type \texttt{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 \texttt{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>
<thead>
<tr>
<th>Name</th>
<th>Usage and Description</th>
</tr>
</thead>
</table>
| N_VLinearCombinationVectorArray | \( \text{ier} = \text{N_VLinearCombinationVectorArray}(\text{nv}, \text{ns}, \text{c}, \text{XX}, \text{Z}); \) \newline This routine computes the linear combination of \( n_s \) vector arrays containing \( n_v \) vectors with \( n \) elements: \newline \[
    z_{j,i} = \sum_{k=0}^{n_s-1} c_k x_{k,j,i}, \quad i = 0, \ldots, n - 1 \quad j = 0, \ldots, n_v - 1,
\] \newline where \( c \) is an array of \( n_s \) scalars (type \texttt{realtype*}), \( XX \) (type \texttt{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 \texttt{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>
</table>
| N_VDotProdLocal             | \( d = \text{N_VDotProdLocal}(x, y); \) \newline This routine computes the MPI task-local portion of the ordinary dot product of \( x \) and \( y \): \newline \[
    d = \sum_{i=0}^{n_{local}-1} x_i y_i,
\] \newline where \( n_{local} \) corresponds to the number of components in the vector on this MPI task (or \( n_{local} = n \) for MPI-unaware applications). |
| N_VMaxNormLocal             | \( m = \text{N_VMaxNormLocal}(x); \) \newline This routine computes the MPI task-local portion of the maximum norm of the \( \text{N_Vector} \) \( x \): \newline \[
    m = \max_{0\leq i<n_{local}} |x_i|,
\] \newline where \( n_{local} \) corresponds to the number of components in the vector on this MPI task (or \( n_{local} = n \) for MPI-unaware applications). |

continued on next page
### Description of the NVENCTOR module

<table>
<thead>
<tr>
<th>Name</th>
<th>Usage and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>N_WMinLocal</strong></td>
<td>m = N_WMinLocal(x); 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><strong>N_VL1NormLocal</strong></td>
<td>n = N_VL1NormLocal(x); 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><strong>N_VWSqrSumLocal</strong></td>
<td>s = N_VWSqrSumLocal(x,w); This routine computes the MPI task-local portion of the weighted squared sum of the N_Vector x with weight vector w: $s = \sum_{i=0}^{n_{local}-1} (x_i w_i)^2$, 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><strong>N_VWSqrSumMaskLocal</strong></td>
<td>s = N_VWSqrSumMaskLocal(x,w,id); 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: $m = \sum_{i=0}^{n_{local}-1} (x_i w_i H(id_i))^2$, 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 CVODE

In Table 6.6 below, we list the vector functions in the `nvector` module used within the `cvode` package. The table also shows, for each function, which of the code modules uses the function. The `cvode` column shows function usage within the main integrator module, while the remaining columns show function usage within each of the `cvode` linear solver interfaces, the `cvbandpre` and `cvbbdpre` preconditioner modules, and the `fcvode` module. Here `cvls` stands for the generic linear solver interface in `cvode`, and `cvdiag` stands for the diagonal linear solver interface in `cvode`.

At this point, we should emphasize that the `cvode` user does not need to know anything about the usage of vector functions by the `cvode` code modules in order to use `cvode`. 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 `cvode` modules for user feedback.

### Table 6.6: NVECTOR functions used by CVODE

<table>
<thead>
<tr>
<th>Name</th>
<th>Usage and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N_VInvTestLocal</td>
<td>( t = \text{N.VInvTestLocal}(x); )</td>
</tr>
<tr>
<td></td>
<td>Sets the MPI task-local components of the N_Vector ( z ) to be the inverses of the</td>
</tr>
<tr>
<td></td>
<td>components of the N_Vector ( x ), with prior testing for zero values: ( z_i = \frac{1}{x_i}, \ i = 0, \ldots, n_{\text{local}} - 1, )</td>
</tr>
<tr>
<td></td>
<td>where ( n_{\text{local}} ) corresponds to the number of components in the vector on</td>
</tr>
<tr>
<td></td>
<td>this MPI task (or ( n_{\text{local}} = n ) for MPI-unaware applications). This</td>
</tr>
<tr>
<td></td>
<td>routine returns a boolean assigned to \text{SUNTRUE} if all task-local components of ( x</td>
</tr>
<tr>
<td></td>
<td>are nonzero (successful inversion) and returns \text{SUNFALSE} otherwise.</td>
</tr>
<tr>
<td>N_VConstrMaskLocal</td>
<td>( t = \text{N.VConstrMaskLocal}(c,x,m); )</td>
</tr>
<tr>
<td></td>
<td>Performs the following constraint tests:</td>
</tr>
<tr>
<td></td>
<td>( x_i &gt; 0 ) if ( c_i = 2, )</td>
</tr>
<tr>
<td></td>
<td>( x_i \geq 0 ) if ( c_i = 1, )</td>
</tr>
<tr>
<td></td>
<td>( x_i \leq 0 ) if ( c_i = -1, )</td>
</tr>
<tr>
<td></td>
<td>( x_i &lt; 0 ) if ( c_i = -2, ) and ( c_i = 0, )</td>
</tr>
<tr>
<td></td>
<td>for all MPI task-local components of the vectors. This routine returns a boolean assigned to</td>
</tr>
<tr>
<td></td>
<td>\text{SUNFALSE} if any task-local element failed the constraint test and assigned to \text{SUNTRUE}</td>
</tr>
<tr>
<td></td>
<td>if all passed. It also sets a mask vector ( m ), with elements equal to 1.0 where the</td>
</tr>
<tr>
<td></td>
<td>constraint test failed, and 0.0 where the test passed. This routine is used only for</td>
</tr>
<tr>
<td></td>
<td>constraint checking.</td>
</tr>
<tr>
<td>N_VMinQuotientLocal</td>
<td>( \text{minq} = \text{N.VMinQuotientLocal}(\text{num,denom}); )</td>
</tr>
</tbody>
</table>
|                             | This routine returns the minimum of the quotients obtained by term-wise dividing \( \text{num}_i \) by \( \text{denom}_i \), 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 \text{BIG}_\text{REAL} (defined in the header file sundials_types.h) is returned.
### Description of the NVECTOR module

#### Table 6.6: List of vector functions usage by CVODE code modules

<table>
<thead>
<tr>
<th>Function</th>
<th>CVODE</th>
<th>CVLS</th>
<th>CVDIAG</th>
<th>CVBANDPRE</th>
<th>CVBBDDPRE</th>
<th>FCVODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>N_VGetVectorID</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VClone</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VCloneEmpty</td>
<td>1</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VDestroy</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VSpace</td>
<td>✓</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VGetArrayPointer</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VSetArrayPointer</td>
<td>1</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VLinearSum</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VProd</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VConst</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VDiv</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VScale</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VAbs</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VInv</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VAddConst</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VDotProd</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VMaxNorm</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VWrmsNorm</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VMin</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VMinQuotient</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VConstrMask</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VCompare</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VInvTest</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VLinearCombination</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VScaleAddMulti</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VDotProdMulti</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VScaleVectorArray</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3. The optional function `N_VDotProdMulti` is only used in the `SUNNONLSOL_FIXEDPOINT` module, or 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 CVODE could omit these operations.

Each SUNLINSOL object may require additional NVECTOR routines not listed in the table above. Please see the relevant descriptions of these modules in Sections 8.5-8.16 for additional detail on their NVECTOR requirements.

The vector functions listed in Table 6.2 that are not used by CVODE are: `N_VWL2Norm`, `N_VL1Norm`, `N_VWrmsNormMask`, `N_VGetCommunicator`, and `N_VGetLength`. Therefore, a user-supplied NVECTOR module for CVODE could omit these functions. The functions `N_MinQuotient`, `N_VConstrMask`, and `N_VCompare` are only used when constraint checking is enabled and may be omitted if this feature is not used.
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;
    booleantype own_data;
    realtype *data;
};
```

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

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`.

  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:

**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.
6.2 The NVECTOR_SERIAL implementation

**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.

**N_VPrintFile_Serial**
Prototype: void N_VPrintFile_Serial(N_Vector v, FILE *outfile);
Description: This function prints the content of a serial vector to 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 N_VNew_Serial, 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_Serial will have the default settings for the NVECTOR_SERIAL module.

**N_VEnableFusedOps_Serial**
Prototype: int N_VEnableFusedOps_Serial(N_Vector v, boolantype tf);
Description: This function enables (SUNTRUE) or disables (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 ops structure are NULL.

**N_VEnableLinearCombination_Serial**
Prototype: int N_VEnableLinearCombination_Serial(N_Vector v, boolantype tf);
Description: This function enables (SUNTRUE) or disables (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 ops structure are NULL.

**N_VEnableScaleAddMulti_Serial**
Prototype: int N_VEnableScaleAddMulti_Serial(N_Vector v, boolantype tf);
Description: This function enables (SUNTRUE) or disables (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 ops structure are NULL.

**N_VEnableDotProdMulti_Serial**
Prototype: int N_VEnableDotProdMulti_Serial(N_Vector v, boolantype tf);
Description: This function enables (SUNTRUE) or disables (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 ops structure are NULL.

**N_VEnableLinearSumVectorArray_Serial**
Prototype: int N_VEnableLinearSumVectorArray_Serial(N_Vector v, boolantype tf);
Description: This function enables (SUNTRUE) or disables (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 ops structure are NULL.
Description of the NVECTOR module

**N_VEnableScaleVectorArray_Serial**

Prototype int N_VEnableScaleVectorArray_Serial(N_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (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 ops structure are NULL.

**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_VDestroyVectorArrayEmpty_Serial** will not attempt to free the pointer data for any N_Vector with own_data set to SUNFALSE. In such a case, it is the user’s responsibility to deallocate the data pointer.
To maximize efficiency, vector operations in the `NVECTOR_SERIAL` implementation that have more than one `N_Vector` argument do not check for consistent internal representation of these vectors. It is the user’s responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the same internal representations.

### 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 `NVNew_Serial` is interfaced as `FNVNew_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 `libsundials_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 `libsundials_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`.

```c
struct _N_VectorContent_Parallel {
    sunindextype local_length;
    sunindextype global_length;
    booleantype own_data;
    realtype *data;
    MPI_Comm comm;
};
```

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

#### 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_llen = NV_LOCLENGTH_P(v) sets v_llen to be the length of the local part of v. The call NV_LENGTH_P(v) = llen_v sets the local length of v to be llen_v.

The assignment v_glen = NV_GLOBLENGTH_P(v) sets v_glen to be the global length of the vector v. The call NV_GLOBLENGTH_P(v) = glen_v sets the global length of v to be glen_v.

Implementation:
#define NV_OWN_DATA_P(v) ( NV_CONTENT_P(v)->own_data )
#define NV_DATA_P(v) ( NV_CONTENT_P(v)->data )
#define NV_LOCLENGTH_P(v) ( NV_CONTENT_P(v)->local_length )
#define NV_GLOBLENGTH_P(v) ( NV_CONTENT_P(v)->global_length )

• NV_COMM_P

This macro provides access to the MPI communicator used by the nvector_parallel vectors.

Implementation:
#define NV_COMM_P(v) ( NV_CONTENT_P(v)->comm )

• NV_Ith_P

This macro gives access to the individual components of the local data array of an N_Vector.

The assignment r = NV_Ith_P(v,i) sets r to be the value of the i-th component of the local part of v. The assignment NV_Ith_P(v,i) = r sets the value of the i-th component of the local part of v to be r.

Here i ranges from 0 to n − 1, where n is the local length.

Implementation:
#define NV_Ith_P(v,i) ( NV_DATA_P(v)[i] )

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.
6.3 The NV_VECTOR_PARALLEL implementation

**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.

**N_VM Make_Parallel**

Prototype: `N_Vector N_VMMake_Parallel(MPI_Comm comm, sunindextype local_length, sunindextype global_length, realtype *v_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_data itself.

**N_VCloneVectorArray_Parallel**

Prototype: `N_Vector *N_VCloneVectorArray_Parallel(int count, N_Vector w);`

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

**N_VCloneVectorArrayEmpty_Parallel**

Prototype: `N_Vector *N_VCloneVectorArrayEmpty_Parallel(int count, N_Vector w);`

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

**N_VDestroyVectorArray_Parallel**

Prototype: `void N_VDestroyVectorArray_Parallel(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_Parallel or with N_VCloneVectorArrayEmpty_Parallel.

**N_VGetLocalLength_Parallel**

Prototype: `sunindextype N_VGetLocalLength_Parallel(N_Vector v);`

Description: This function returns the local vector length.

**N_VPrint_Parallel**

Prototype: `void N_VPrint_Parallel(N_Vector v);`

Description: This function prints the local content of a parallel vector to stdout.

**N_VPrintFile_Parallel**

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

Description: This function prints the local content of a parallel vector to outfile.

By default all fused and vector array operations are disabled in the NV_VECTOR_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 N_VNew_Parallel, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using N_VClone 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 N_VNew_Parallel will have the default settings for the NV_VECTOR_PARALLEL module.
Description of the NVECTOR module

**N_EnableFusedOps_Parallel**
Prototype: \texttt{int N_EnableFusedOps\_Parallel(N\_Vector v, booleantype tf);}  
Description: This function enables (\texttt{SUNTRUE}) or disables (\texttt{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 \texttt{ops} structure are \texttt{NULL}.

**N_EnableLinearCombination_Parallel**
Prototype: \texttt{int N_EnableLinearCombination\_Parallel(N\_Vector v, booleantype tf);}  
Description: This function enables (\texttt{SUNTRUE}) or disables (\texttt{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 \texttt{ops} structure are \texttt{NULL}.

**N_EnableScaleAddMulti_Parallel**
Prototype: \texttt{int N_EnableScaleAddMulti\_Parallel(N\_Vector v, booleantype tf);}  
Description: This function enables (\texttt{SUNTRUE}) or disables (\texttt{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 \texttt{ops} structure are \texttt{NULL}.

**N_EnableDotProdMulti_Parallel**
Prototype: \texttt{int N_EnableDotProdMulti\_Parallel(N\_Vector v, booleantype tf);}  
Description: This function enables (\texttt{SUNTRUE}) or disables (\texttt{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 \texttt{ops} structure are \texttt{NULL}.

**N_EnableLinearSumVectorArray_Parallel**
Prototype: \texttt{int N_EnableLinearSumVectorArray\_Parallel(N\_Vector v, booleantype tf);}  
Description: This function enables (\texttt{SUNTRUE}) or disables (\texttt{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 \texttt{ops} structure are \texttt{NULL}.

**N_EnableScaleVectorArray_Parallel**
Prototype: \texttt{int N_EnableScaleVectorArray\_Parallel(N\_Vector v, booleantype tf);}  
Description: This function enables (\texttt{SUNTRUE}) or disables (\texttt{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 \texttt{ops} structure are \texttt{NULL}.

**N_EnableConstVectorArray_Parallel**
Prototype: \texttt{int N_EnableConstVectorArray\_Parallel(N\_Vector v, booleantype tf);}  
Description: This function enables (\texttt{SUNTRUE}) or disables (\texttt{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 \texttt{ops} structure are \texttt{NULL}.
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:

**N_VNew_OpenMP**

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

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

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

**N_VNewEmpty_OpenMP**

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

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

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

**N_VMake_OpenMP**

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

Description: 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.

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

**N_VCloneVectorArray_OpenMP**

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

Description: This function creates (by cloning) an array of count OpenMP vectors.
Description of the NVECTOR module

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, boolean_type 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, boolean_type 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, boolean_type 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 NV ECTOR module

Prototype

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

Description

This function enables (\texttt{SUNTRUE}) or disables (\texttt{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 \texttt{ops} structure are NULL.

Notes

- When looping over the components of an \texttt{N_Vector} \texttt{v}, it is more efficient to first obtain the component array via \texttt{v.data = NV_DATA_OMP(v)} and then access \texttt{v.data[i]} within the loop than it is to use \texttt{NV_Ith_OMP(v,i)} within the loop.

- \texttt{N_VNewEmpty_OpenMP}, \texttt{N_VMake_OpenMP}, and \texttt{N_VCloneVectorArrayEmpty_OpenMP} set the field \texttt{own.data = SUNFALSE}. \texttt{N_VDestroy_OpenMP} and \texttt{N_VDestroyVectorArray_OpenMP} will not attempt to free the pointer \texttt{data} for any \texttt{N_Vector} with \texttt{own.data} set to \texttt{SUNFALSE}. In such a case, it is the user's responsibility to deallocate the \texttt{data} pointer.

- To maximize efficiency, vector operations in the \texttt{NVECTOR_OPENMP} implementation that have more than one \texttt{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 \texttt{N_Vector} arguments that were all created with the same internal representations.

6.4.3 \texttt{NVECTOR_OPENMP} Fortran interfaces

The \texttt{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 \texttt{nvector_openmp.mod} Fortran module defines interfaces to most \texttt{NVECTOR_OPENMP} C functions using the intrinsic \texttt{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 \texttt{N_VNew_OpenMP} is interfaced as \texttt{FN_VNew_OpenMP}.

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

FORTRAN 77 interface functions

For solvers that include a Fortran 77 interface module, the \texttt{NVECTOR_OPENMP} module also includes a FORTRAN-callable function \texttt{FNWINITOMP(code, NEQ, NUMTHREADS, IER)}, to initialize this module. Here \texttt{code} is an input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); \texttt{NEQ} is the problem size (declared so as to match C type \texttt{long int}); \texttt{NUMTHREADS} is the number of threads; and \texttt{IER} is an error return flag equal 0 for success and -1 for failure.

6.5 \texttt{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 \texttt{NVECTOR} using OpenMP, called \texttt{NVECTOR_OPENMP}, and an implementation using Pthreads, called \texttt{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 lib sundials_nvec pthreads.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.

- **NV_CONTENT_PT**
  This routine gives access to the contents of the Pthreads vector N_Vector.
  The assignment `v_cont = NV_CONTENT_PT(v)` sets `v_cont` to be a pointer to the Pthreads N_Vector content structure.
  Implementation:
  ```c
  #define NV_CONTENT_PT(v) ( (N_VectorContent_Pthreads)(v->content) )
  ```

- **NV_OWN_DATA_PT, NV_DATA_PT, NV_LENGTH_PT, NV_NUM_THREADS_PT**
  These macros give individual access to the parts of the content of a Pthreads N_Vector.
  The assignment `v_data = NV_DATA_PT(v)` sets `v_data` to be a pointer to the first component of the data for the N_Vector `v`. The assignment `NV_DATA_PT(v) = v_data` sets the component array of `v` to be `v_data` by storing the pointer `v_data`.
  The assignment `v_len = NV_LENGTH_PT(v)` sets `v_len` to be the length of `v`. On the other hand, the call `NV_LENGTH_PT(v) = len_v` sets the length of `v` to be `len_v`.
  The assignment `v_num_threads = NV_NUM_THREADS_PT(v)` sets `v_num_threads` to be the number of threads from `v`. On the other hand, the call `NV_NUM_THREADS_PT(v) = num_threads_v` sets the number of threads for `v` to be `num_threads_v`.
  Implementation:
  ```c
  #define NV_OWN_DATA_PT(v) ( NV_CONTENT_PT(v)->own_data )
  #define NV_DATA_PT(v) ( NV_CONTENT_PT(v)->data )
  #define NV_LENGTH_PT(v) ( NV_CONTENT_PT(v)->length )
  #define NV_NUM_THREADS_PT(v) ( NV_CONTENT_PT(v)->num_threads )
  ```

- **NV_Ith_PT**
  This macro gives access to the individual components of the data array of an N_Vector.
  The assignment `r = NV_Ith_PT(v,i)` sets `r` to be the value of the `i`-th component of `v`. The assignment `NV_Ith_PT(v,i) = r` sets the value of the `i`-th component of `v` to be `r`.
  Here `i` ranges from 0 to `n - 1` for a vector of length `n`.
  Implementation:
  ```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:

<table>
<thead>
<tr>
<th>Function</th>
<th>Prototype</th>
<th>Description</th>
<th>F2003 Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>N_VNew_Pthreads</td>
<td>N_Vector N_VNew_Pthreads(sunindextype vec_length, int num_threads)</td>
<td>This function creates and allocates memory for a Pthreads N_Vector. Arguments are the vector length and number of threads.</td>
<td>FN_VNew_Pthreads when using the Fortran 2003 interface module.</td>
</tr>
<tr>
<td>N_VNewEmpty_Pthreads</td>
<td>N_Vector N_VNewEmpty_Pthreads(sunindextype vec_length, int num_threads)</td>
<td>This function creates a new Pthreads N_Vector with an empty (NULL) data array.</td>
<td>FN_VNewEmpty_Pthreads when using the Fortran 2003 interface module.</td>
</tr>
<tr>
<td>N_VMake_Pthreads</td>
<td>N_Vector N_VMake_Pthreads(sunindextype vec_length, realtype *v_data, int num_threads);</td>
<td>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.</td>
<td>FN_VMake_Pthreads when using the Fortran 2003 interface module.</td>
</tr>
<tr>
<td>N_VCloneVectorArray_Pthreads</td>
<td>N_Vector *N_VCloneVectorArray_Pthreads(int count, N_Vector w)</td>
<td>This function creates (by cloning) an array of count Pthreads vectors.</td>
<td></td>
</tr>
<tr>
<td>N_VCloneVectorArrayEmpty_Pthreads</td>
<td>N_Vector *N_VCloneVectorArrayEmpty_Pthreads(int count, N_Vector w)</td>
<td>This function creates (by cloning) an array of count Pthreads vectors, each with an empty (NULL) data array.</td>
<td></td>
</tr>
<tr>
<td>N_VDestroyVectorArray_Pthreads</td>
<td>void N_VDestroyVectorArray_Pthreads(N_Vector *vs, int count)</td>
<td>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.</td>
<td></td>
</tr>
</tbody>
</table>
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 enable/disable 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`. 
**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 NVECTOR_PTHREADS implementation that have more than one N_Vector argument do not check for consistent internal representation of these vectors. It is the user’s responsibility to ensure that such routines are called with N_Vector arguments that were all created with the same internal representations.

6.5.3 NVECTOR_PTHREADS Fortran interfaces

The NVECTOR_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 nvector_pthreads_mod FORTRAN module defines interfaces to most NVECTOR_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 N_VNew_Pthreads is interfaced as FN_VNew_Pthreads.

The FORTRAN 2003 NVECTOR_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 NVECTOR_PTHREADS module also includes a FORTRAN-callable function FNVINITPTS(code, NEQ, NUMTHREADS, IER), to initialize this module. Here code is an input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); NEQ is the problem size (declared so as to match C type long int); NUMTHREADS is the number of threads; and IER is an error return flag equal 0 for success and -1 for failure.

6.6 The NVECTOR_PARHYP implementation

The NVECTOR_PARHYP implementation of the NVECTOR 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.

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

Unlike native SUNDIALS vector types, NVECTOR_PARHYP does not provide macros to access its member variables. Note that NVECTOR_PARHYP requires SUNDIALS to be built with MPI support.
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 N_VSetArrayPointer and 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 cvAdvDiff_non_ph.c example program for CVODE [29] and the ark_diurnal_kry_ph.c example program for ARKODE [38].

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

[N_VNewEmpty_ParHyp]
Prototype: N_Vector N_VNewEmpty_ParHyp(MPI_Comm comm, sunindextype local_length, sunindextype global_length)
Description: This function creates a new parhyp N_Vector with the pointer to the hypre vector set to NULL.

[N_VMake_ParHyp]
Prototype: N_Vector N_VMake_ParHyp(HYPRE_ParVector x)
Description: This function creates an N_Vector wrapper around an existing hypre parallel vector. It does not allocate memory for x itself.

[N_VGetVector_ParHyp]
Prototype: HYPRE_ParVector N_VGetVector_ParHyp(N_Vector v)
Description: This function returns the underlying hypre vector.

[N_VCloneVectorArray_ParHyp]
Prototype: N_Vector *N_VCloneVectorArray_ParHyp(int count, N_Vector w)
Description: This function creates (by cloning) an array of count parallel vectors.

[N_VCloneVectorArrayEmpty_ParHyp]
Prototype: N_Vector *N_VCloneVectorArrayEmpty_ParHyp(int count, N_Vector w)
Description: This function creates (by cloning) an array of count parallel vectors, each with an empty (NULL) data array.

[N_VDestroyVectorArray_ParHyp]
Prototype: void N_VDestroyVectorArray_ParHyp(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_ParHyp or with N_VCloneVectorArrayEmpty_ParHyp.

[N_VPrint_ParHyp]
Prototype: void N_VPrint_ParHyp(N_Vector v)
Description: This function prints the local content of a parhyp vector to stdout.
6.6 The NVECTOR_PARHYP implementation

**N_VPrintFile_ParHyp**
Prototype: `void N_VPrintFile_ParHyp(N_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 NVECTOR_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 `N_VMake_ParHyp`, 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_VMake_ParHyp` will have the default settings for the NVECTOR_PARHYP module.

**N_VEnableFusedOps_ParHyp**
Prototype: `int N_VEnableFusedOps_ParHyp(N_Vector v, booleantype tf)`
Description: This function enables (`SUNTRUE`) or disables (`SUNFALSE`) 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_VEnableLinearCombination_ParHyp**
Prototype: `int N_VEnableLinearCombination_ParHyp(N_Vector v, booleantype tf)`
Description: This function enables (`SUNTRUE`) or disables (`SUNFALSE`) 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_VEnableScaleAddMulti_ParHyp**
Prototype: `int N_VEnableScaleAddMulti_ParHyp(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 parhyp vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are NULL.

**N_VEnableDotProdMulti_ParHyp**
Prototype: `int N_VEnableDotProdMulti_ParHyp(N_Vector v, booleantype tf)`
Description: This function enables (`SUNTRUE`) or disables (`SUNFALSE`) 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_VEnableLinearSumVectorArray_ParHyp**
Prototype: `int N_VEnableLinearSumVectorArray_ParHyp(N_Vector v, booleantype tf)`
Description: This function enables (`SUNTRUE`) or disables (`SUNFALSE`) 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_VEnableScaleVectorArray_ParHyp**
Prototype: `int N_VEnableScaleVectorArray_ParHyp(N_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 \texttt{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.
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.

```c
struct _N_VectorContent_Petsc {
    sunindextype local_length;
    sunindextype global_length;
    booleantype own_data;
    Vec *pvec;
    MPI_Comm comm;
};
```

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

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

#### 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 [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 _Petsc (e.g. N_Destroy_Petsc). The module NVECTOR_PETSC provides the following additional user-callable routines:

**N_VNewEmpty_Petsc**

**Prototype**

```c
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**

```c
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**

```c
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_VMake_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_VMake_Petsc` will have the default settings for the NVECTOR_PETSC module.

**N_VEnableFusedOps_Petsc**
Prototype: `int N_VEnableFusedOps_Petsc(N_Vector v, booleantype 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, booleantype 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 NVVECTOR module

N_VEnableScaleAddMultiVectorArray_Petsc
Prototype 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 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 NVVECTOR_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 NVVECTOR_CUDA implementation

The NVVECTOR_CUDA module is an experimental NVVECTOR 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:

template <class T, class I>
class Vector {
    I size_;  
    I mem_size_; 
    I global_size_; 
    T* h_vec_; 
    T* d_vec_; 
    ThreadPartitioning<T, I>* partStream_; 
    ThreadPartitioning<T, I>* partReduce_; 
    bool ownPartitioning_; 
    bool ownData_; 
    bool managed_mem_; 
    SUNMPI_Comm comm_; 
    ...
};
6.8 The NVECTOR_CUDA implementation

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 suncudavec::Vector class will change frequently in future Sundials releases. The code is structured so that it can tolerate significant changes in the suncudavec::Vector class without requiring changes to the user API.

When instantiated with NVNew_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 NVMake_CUDA constructor. To use CUDA managed memory, the constructors NVNewManaged_CUDA and NVMakeManaged_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 is 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.
```
**N_VIsManagedMemory_Cuda**

Prototype: `boolanteype *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 [29].

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 NVVECTOR_CUDA implementation

**N_VNewEmpty_Cuda**

Prototype: `N_Vector N_VNewEmpty_Cuda()`  
Description: This function creates a new NVVECTOR 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 NVVECTOR_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 `nvvector_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 NVVECTOR_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 `nvvector_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 NVVECTOR_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 `nvvector_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 NVVECTOR_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 `nvvector_mpicuda.h` and the library to link to is `libsundials_nvecmpicuda.lib`.

The module NVVECTOR_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 NVVECTOR module

Description  This function sets the CUDA stream that all vector kernels will be launched on. By default an NVVECTOR_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 NVVECTOR_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 NVVECTOR_CUDA module.

N_VEnableFusedOps_Cuda
Prototype int N_VEnableFusedOps_Cuda(N_Vector v, boolean type 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, boolean type 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.
### 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 `sunraja::vec` manages the vector data layout:

```cpp
template <class T, class I>
class Vector {
    I size_;  // vector size (length)
    I mem_size_;  // size of the vector data memory block
    I global_size_;  // global vector size (length)
    T* h_vec_;  // pointers to vector data on the host
    T* d_vec_;  // pointers to vector data on the device
    SUNMPI_Comm comm_;  // MPI communicator
    ...
};
```

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 `sunraja::Vector` class will change frequently in future SUNDIALS releases. The
code is structured so that it can tolerate significant changes in the sunrajevec::Vector class without requiring changes to the user API.

The nvector_raja module can be utilized for single-node parallelism or in a distributed context with MPI. The header file to include when using this module for single-node parallelism is nvector_raja.h. The header file to include when using this module in the distributed case is nvector_mpiraja.h. The installed module libraries to link to are lib sundials_nvecraja.lib in the single-node case, or lib sundials_nvecmpicudaraja.lib in the distributed case. Only one of these libraries may be linked to when creating an executable or library. SUNDIALS must be built with MPI support if the distributed library is desired.

6.9.1 NVECTOR_RAJA functions

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

**N_VGetLocalLength_Raja**

Prototype: sunindextype N_VGetLocalLength_Raja(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_mpiraja.h and the library to link to is libsundials_nvecmpicudaraja.lib.

**N_VGetHostArrayPointer_Raja**

Prototype: realtype *N_VGetHostArrayPointer_Raja(N_Vector v)

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

**N_VGetDeviceArrayPointer_Raja**

Prototype: realtype *N_VGetDeviceArrayPointer_Raja(N_Vector v)

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

**N_VGetMPIComm_Raja**

Prototype: MPI_Comm N_VGetMPIComm_Raja(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_mpiraja.h and the library to link to is libsundials_nvecmpicudaraja.lib.

The nvector_raja module defines the implementations of all vector operations listed in Tables 6.2, 6.3, 6.4, and 6.5, except for N_VDotProdMulti, N_VWrmsNormVectorArray, and N_VWrmsNormMaskVectorArray as support for arrays of reduction vectors is not yet supported in RAJA. These function will be added to the NVECTOR_RAJA implementation in the future. Additionally the vector operations N_VGetArrayPointer and N_VSetArrayPointer are not implemented by the RAJA vector. As such, this vector cannot be used with the SUNDIALS Fortran interfaces, nor with the SUNDIALS direct solvers and preconditioners. The nvector_raja module provides separate functions to access data on the host and on the device. It also provides methods for copying data from the host to the device and vice versa. Usage examples of NVECTOR_RAJA are provided in some example programs for CVODE [29].

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

N_VNew_Raja

Single-node usage

Prototype  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 nvector_raja.h and the library to link to is libsundials_nveccudaraja.lib.

Distributed-memory parallel usage

Prototype  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 nvector_mpiraja.h and the library to link to is libsundials_nvecmpicudaraja.lib.

N_VNewEmpty_Raja

Prototype  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  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  realtype *N_VCopyToDevice_Raja(N_Vector v)

Description This function copies host vector data to the device.

N_VCopyFromDevice_Raja

Prototype  realtype *N_VCopyFromDevice_Raja(N_Vector v)

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

N_VPrint_Raja

Prototype  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

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.

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.

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.

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.

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.

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.

Prototype: `int N_VEnableConstVectorArray_Raja(N_Vector v, booleantype tf)`
Description of the NVECTOR module

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.

N_VEnableScaleAddMultiVectorArray_Raja
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.

N_VEnableLinearCombinationVectorArray_Raja
Prototype
int N_VEnableLinearCombinationVectorArray_Raja(N_Vector v, booleantype tf)

Description
This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination 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.

Notes

• When there is a need to access components of an N_Vector_Raja, v, it is recommended to use functions N_VGetDeviceArrayPointer_Raja or N_VGetHostArrayPointer_Raja.

• To maximize efficiency, vector operations in the NVECTOR_RAJA implementation that have more than one N_Vector argument do not check for consistent internal representations of these vectors. It is the user’s responsibility to ensure that such routines are called with N_Vector arguments that were all created with the same internal representations.

6.10 The NVECTOR_OPENMPDEV implementation

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

The NVECTOR_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.

```
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 NVECTOR_OPENMPDEV accessor macros

The following macros are provided to access the content of an NVECTOR_OPENMPDEV vector.
6.10 The NVECTOR_OPENMPDEV implementation

- **NVCONTENT_OMPDEV**
  This routine gives access to the contents of the NVECTOR_OPENMPDEV vector N.Vector.
  The assignment \( v_{\text{cont}} = \text{NVCONTENT}_\text{OMPDEV}(v) \) sets \( v_{\text{cont}} \) to be a pointer to the NVECTOR_OPENMPDEV N.Vector content structure.
  Implementation:
  ```c
  #define NVCONTENT_OMPDEV(v) ( (N_VectorContent_OpenMPDEV)(v->content) )
  ```

- **NVOWNDATA_OMPDEV, NVDATAHOST_OMPDEV, NVDATADEV_OMPDEV, NVLENGTH_OMPDEV**
  These macros give individual access to the parts of the content of an NVECTOR_OPENMPDEV N_Vector.
  The assignment \( v_{\text{data}} = \text{NVDATAHOST}_\text{OMPDEV}(v) \) sets \( v_{\text{data}} \) to be a pointer to the first component of the data on the host for the N_Vector \( v \). The assignment \( \text{NVDATAHOST}_\text{OMPDEV}(v) = v_{\text{data}} \) sets the host component array of \( v \) to be \( v_{\text{data}} \) by storing the pointer \( v_{\text{data}} \).
  The assignment \( v_{\text{devdata}} = \text{NVDATADEV}_\text{OMPDEV}(v) \) sets \( v_{\text{devdata}} \) to be a pointer to the first component of the data on the device for the N_Vector \( v \). The assignment \( \text{NVDATADEV}_\text{OMPDEV}(v) = v_{\text{devdata}} \) sets the device component array of \( v \) to be \( v_{\text{devdata}} \) by storing the pointer \( v_{\text{devdata}} \).
  The assignment \( v_{\text{len}} = \text{NVLENGTH}_\text{OMPDEV}(v) \) sets \( v_{\text{len}} \) to be the length of \( v \). On the other hand, the call \( \text{NVLENGTH}_\text{OMPDEV}(v) = \text{len}_v \) sets the length of \( v \) to be \( \text{len}_v \).
  Implementation:
  ```c
  #define NVOWNDATA_OMPDEV(v) ( NVCONTENT_OMPDEV(v)->own_data )
  #define NVDATAHOST_OMPDEV(v) ( NVCONTENT_OMPDEV(v)->host_data )
  #define NVDATADEV_OMPDEV(v) ( NVCONTENT_OMPDEV(v)->dev_data )
  #define NVLENGTH_OMPDEV(v) ( NVCONTENT_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 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. 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. N_VDestroy_OpenMPDEV). The module NVECTOR_OPENMPDEV provides the following additional user-callable routines:

**N_VNew_OpenMPDEV**

Prototype \( \text{N_Vector N_VNew_OpenMPDEV(sunindextype vec_length)} \)

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

**N_VNewEmpty_OpenMPDEV**

Prototype \( \text{N_Vector N_VNewEmpty_OpenMPDEV(sunindextype vec_length)} \)

Description This function creates a new NVECTOR_OPENMPDEV N_Vector with an empty (NULL) host and device data arrays.
Description of the NVECTOR module

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.
6.10 The NV\textunderscore VECTOR\_OPENMPDEV implementation

**N\_VCopyFromDevice\_OpenMPDEV**

Prototype: `void N\_VCopyFromDevice\_OpenMPDEV(N\_Vector v)`

Description: This function copies the content of an NV\textunderscore VECTOR\_OPENMPDEV vector’s device data array to the host data array.

By default all fused and vector array operations are disabled in the NV\textunderscore VECTOR\_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 NV\textunderscore VECTOR\_OPENMPDEV module.

**N\_VEnableFusedOps\_OpenMPDEV**

Prototype: `int N\_VEnableFusedOps\_OpenMPDEV(N\_Vector v, booleantype tf)`

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

**N\_VEnableLinearCombination\_OpenMPDEV**

Prototype: `int N\_VEnableLinearCombination\_OpenMPDEV(N\_Vector v, booleantype tf)`

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

**N\_VEnableScaleAddMulti\_OpenMPDEV**

Prototype: `int N\_VEnableScaleAddMulti\_OpenMPDEV(N\_Vector v, booleantype tf)`

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

**N\_VEnableDotProdMulti\_OpenMPDEV**

Prototype: `int N\_VEnableDotProdMulti\_OpenMPDEV(N\_Vector v, booleantype tf)`

Description: This function enables (\texttt{SUNTRUE}) or disables (\texttt{SUNFALSE}) the multiple dot products fused operation in the NV\textunderscore VECTOR\_OPENMPDEV vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are \texttt{NULL}.

**N\_VEnableLinearSumVectorArray\_OpenMPDEV**

Prototype: `int N\_VEnableLinearSumVectorArray\_OpenMPDEV(N\_Vector v, booleantype tf)`

Description: This function enables (\texttt{SUNTRUE}) or disables (\texttt{SUNFALSE}) the linear sum operation for vector arrays in the NV\textunderscore VECTOR\_OPENMPDEV vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are \texttt{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.
• When accessing individual components of an N_Vector v on the host remember to first copy the array back from the device with N_VCopyFromDevice_OpenMPDEV(v) to ensure the array is up to date.

• N_VNewEmpty_OpenMPDEV, N_VMake_OpenMPDEV, and N_VCloneVectorArrayEmpty_OpenMPDEV set the field own_data = SUNFALSE. N_VDestroy_OpenMPDEV and N_VDestroyVectorArray_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 libsundials_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 N_VGetArrayPointer and N_VSetArrayPointer. 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 [28].

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. N_VDestroy_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:

• N_VGetVector_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 NVECTOR_TRILINOS wrapper around a user-provided Tpetra vector. This is a standalone function defined in the global namespace.

  N_Vector N_VMake_Trilinos(Teuchos::RCP<vector_type> v);

**Notes**

- The template parameter `vector_type` should be set as:
  
  ```
  typedef Sundials::TpetraVectorInterface::vector_type vector_type
  ```
  
  This will ensure that data types used in Tpetra vector match those in SUNDIALS.

- When there is a need to access components of an 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 N_VDestroy_Trilinos and N_VDestroyVectorArray_Trilinos only delete the 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 NVECTOR_MANYVECTOR implementation

The NVECTOR_MANYVECTOR implementation of the NVECTOR module provided with 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 NVECTOR modules for each component, that are then combined together to form the NVECTOR_MANYVECTOR. We envision three generic use cases for this implementation:

A. **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 NVECTOR_PARALLEL, another single-node component for GPU accelerators based on NVECTOR_CUDA, and another threaded single-node component based on NVECTOR_OPENMP.

B. **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 intercommunicator to connect these distinct process sets together.

C. **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 NVECTOR_MANYVECTOR for the overall "solution".

We note that the above use cases are not mutually exclusive, and the NVECTOR_MANYVECTOR implementation should support arbitrary combinations of these cases.

The NVECTOR_MANYVECTOR implementation is designed to work with any NVECTOR subvectors that implement the minimum 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, NVECTOR_MANYVECTOR sets no limit on the number of subvectors that may be attached (aside from the limitations of using sunindextype 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 subvec-
tors 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.

```
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:

```
N_VNew_ManyVector
```

Prototype

```
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{N\_VECTOR\_MANYVECTOR} module. We note that these routines \textit{do not} call the corresponding routines on subvectors, so those should be set up as desired \textit{before} attaching them to the ManyVector in \texttt{N\_VNew\_ManyVector} or \texttt{N\_VMake\_ManyVector}.

**\texttt{N\_VEnableFusedOps\_ManyVector}**

Prototype:  
\texttt{int N\_VEnableFusedOps\_ManyVector(N\_Vector v, booleantype 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}.

**\texttt{N\_VEnableLinearCombination\_ManyVector}**

Prototype:  
\texttt{int N\_VEnableLinearCombination\_ManyVector(N\_Vector v, booleantype 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}.

**\texttt{N\_VEnableScaleAddMulti\_ManyVector}**

Prototype:  
\texttt{int N\_VEnableScaleAddMulti\_ManyVector(N\_Vector v, booleantype 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}.

**\texttt{N\_VEnableDotProdMulti\_ManyVector}**

Prototype:  
\texttt{int N\_VEnableDotProdMulti\_ManyVector(N\_Vector v, booleantype 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}.

**\texttt{N\_VEnableLinearSumVectorArray\_ManyVector}**

Prototype:  
\texttt{int N\_VEnableLinearSumVectorArray\_ManyVector(N\_Vector v, booleantype 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}.

**\texttt{N\_VEnableScaleVectorArray\_ManyVector}**

Prototype:  
\texttt{int N\_VEnableScaleVectorArray\_ManyVector(N\_Vector v, booleantype 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}.
**N_VEnableConstVectorArray_ManyVector**

Prototype: `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: `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: `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_VWrsNorm**: Create vector of known values, find and validate the weighted root mean square.

- **Test.N_VWrsNormMask**: 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 \( x = a \times x \)
  Case 1b: Test \( z = a \times x \)
  Case 2a: Test \( x = a \times x + b \times y \)
  Case 2b: Test \( z = a \times x + b \times y \)
  Case 3a: Test \( x = x + a \times y + b \times z \)
  Case 3b: Test \( x = a \times x + b \times y + c \times z \)
  Case 3c: Test \( w = a \times x + b \times y + c \times z \)
• Test_N_VScaleAddMulti Case 1a: \( y = a \times x + y \)
  Case 1b: \( z = a \times x + y \)
  Case 2a: \( Y[i] = c[i] \times x + Y[i], \ i = 1,2,3 \)
  Case 2b: \( Z[i] = c[i] \times x + Y[i], \ i = 1,2,3 \)
• Test_N_VDotProdMulti Case 1: Calculate the dot product of two vectors
  Case 2: Calculate the dot product of one vector with three other vectors in a vector array.
• Test_N_VLinearSumVectorArray Case 1: \( z = a \times x + b \times y \)
  Case 2a: \( Z[i] = a \times X[i] + b \times Y[i] \)
  Case 2b: \( X[i] = a \times X[i] + b \times Y[i] \)
  Case 2c: \( Y[i] = a \times X[i] + b \times Y[i] \)
• Test_N_VScaleVectorArray Case 1a: \( y = c \times y \)
  Case 1b: \( z = c \times y \)
  Case 2a: \( Y[i] = c[i] \times Y[i] \)
  Case 2b: \( Z[i] = c[i] \times Y[i] \)
  Case 1a: \( z = c \)
  Case 1b: \( Z[i] = c \)
• Test_N_VWrmsNormVectorArray Case 1a: Create a vector of known values, find and validate the weighted root mean square norm.
  Case 1b: Create a vector array of three vectors of known values, find and validate the weighted root mean square norm of each.
6.13 NVECTOR Examples

- **Test_N_VWrmsNormMaskVectorArray** Case 1a: Create a vector of know values, find and validate the weighted root mean square norm using all elements except one.

- **Test_N_VWrmsNormMaskVectorArray** Case 1b: Create a vector array of three vectors of know values, find and validate the weighted root mean square norm of each using all elements except one.

- **Test_N_VScaleAddMultiVectorArray** Case 1a: $y = a \times + y$

- **Test_N_VScaleAddMultiVectorArray** Case 1b: $z = a \times + y$

- **Test_N_VScaleAddMultiVectorArray** Case 2a: $Y[j][0] = a[j] \times[0] + Y[j][0]$

- **Test_N_VScaleAddMultiVectorArray** Case 2b: $Z[j][0] = a[j] \times[0] + Y[j][0]$

- **Test_N_VScaleAddMultiVectorArray** Case 3a: $Y[0][i] = a[0] \times[i] + Y[0][i]$

- **Test_N_VScaleAddMultiVectorArray** Case 3b: $Z[0][i] = a[0] \times[i] + Y[0][i]$

- **Test_N_VScaleAddMultiVectorArray** Case 4a: $Y[j][i] = a[j] \times[i] + Y[j][i]$

- **Test_N_VScaleAddMultiVectorArray** Case 4b: $Z[j][i] = a[j] \times[i] + Y[j][i]$

- **Test_N_VLinearCombinationVectorArray** Case 1a: $x = a \times$

- **Test_N_VLinearCombinationVectorArray** Case 1b: $z = a \times$

- **Test_N_VLinearCombinationVectorArray** Case 2a: $x = a \times + b \times$

- **Test_N_VLinearCombinationVectorArray** Case 2b: $z = a \times + b \times$

- **Test_N_VLinearCombinationVectorArray** Case 3a: $x = a \times + b \times + c \times$

- **Test_N_VLinearCombinationVectorArray** Case 3b: $w = a \times + b \times + c \times$

- **Test_N_VLinearCombinationVectorArray** Case 4a: $X[0][i] = c[0] \times[0][i]$

- **Test_N_VLinearCombinationVectorArray** Case 4b: $Z[i] = c[0] \times[0][i]$

- **Test_N_VLinearCombinationVectorArray** Case 5a: $X[0][i] = c[0] \times[0][i] + c[1] \times[1][i]$

- **Test_N_VLinearCombinationVectorArray** Case 5b: $Z[i] = c[0] \times[0][i] + c[1] \times[1][i]$

- **Test_N_VLinearCombinationVectorArray** Case 6a: $X[0][i] = X[0][i] + c[1] \times[1][i] + c[2] \times[2][i]$

- **Test_N_VLinearCombinationVectorArray** Case 6b: $X[0][i] = c[0] \times[0][i] + c[1] \times[1][i] + c[2] \times[2][i]$

- **Test_N_VLinearCombinationVectorArray** Case 6c: $Z[i] = c[0] \times[0][i] + c[1] \times[1][i] + c[2] \times[2][i]$

- **Test_N_VDotProdLocal**: Calculate MPI task-local portion of the dot product of two vectors.

- **Test_N_VMaxNormLocal**: Create vector with known values, find and validate the MPI task-local portion of the max norm.

- **Test_N_VMinLocal**: Create vector, find and validate the MPI task-local min.

- **Test_N_VL1NormLocal**: Create vector, find and validate the MPI task-local portion of the L1 norm.

- **Test_N_VWSqrSumLocal**: Create vector of known values, find and validate the MPI task-local portion of the weighted squared sum of two vectors.
• **Test_NVWSqrSumMaskLocal**: 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 SUNMATRIX_CUSTOM 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>

...continued on next page...
### SUNMatDestroy
SUNMatDestroy(A);
Destroy the SUNMatrix A and frees memory allocated for its internal data.

### SUNMatSpace
ier = SUNMatSpace(A, &lrw, &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.

### SUNMatZero
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.

### SUNMatCopy
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.

### SUNMatScaleAdd
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.

### SUNMatScaleAddI
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.

### SUNMatMatvecSetup
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.

### SUNMatMatvec
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.

### 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.
Table 7.3: 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 NVVECTOR types provided with SUNDIALS. This is primarily due to the need for compatibility within the SUNMatMatvec routine; however, compatibility between SUNMATRIX and NVVECTOR 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 7.4: SUNDIALS matrix interfaces and vector implementations that can be used for each.

<table>
<thead>
<tr>
<th>Matrix Interface</th>
<th>Serial</th>
<th>Parallel (MPI)</th>
<th>OpenMP</th>
<th>pThreads</th>
<th>hypre Vec.</th>
<th>PETSc Vec.</th>
<th>CUDA</th>
<th>RAJA</th>
<th>User Suppl.</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>Sparse</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>SLUNRloc</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>User supplied</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</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 CVODE

In Table 7.5, we list the matrix functions in the SUNMATRIX module used within the CVODE package. The table also shows, for each function, which of the code modules uses the function. The main CVODE integrator does not call any SUNMATRIX functions directly, so the table columns are specific to the CVLS interface and the CVBANDPRE and CVBBDPRE preconditioner modules. We further note that the CVLS interface only utilizes these routines when supplied with a matrix-based linear solver, i.e., the SUNMATRIX object passed to CVodeSetLinearSolver was not NULL.

At this point, we should emphasize that the CVODE user does not need to know anything about the usage of matrix functions by the CVODE code modules in order to use CVODE. The information is presented as an implementation detail for the interested reader.

<table>
<thead>
<tr>
<th>Function</th>
<th>CVLS</th>
<th>CVBANDPRE</th>
<th>CVBBDPRE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUNMatGetID</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUNMatClone</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUNMatDestroy</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SUNMatZero</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SUNMatCopy</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SUNMatScaleAddI</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SUNMatSpace</td>
<td>†</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 CVODE are: SUNMatScaleAdd and SUNMatMatvec. Therefore a user-supplied SUNMATRIX module for CVODE could omit these functions.

We note that the CVBANDPRE and CVBBDPRE preconditioner modules are hard-coded to use the SUNDIALS-supplied band SUNMATRIX type, so the most useful information above for user-supplied SUNMATRIX implementations is the column relating the CVLS requirements.

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 \(0 \leq i < M\) and \(0 \leq j < N\)) may be accessed via \(\text{data}[j*M+i]\).

ldata - length of the data array (= \(M*N\)).

cols - array of pointers. \(\text{cols}[j]\) points to the first element of the \(j\)-th column of the matrix in the array \(\text{data}\). The \((i,j)\)-th element of a dense SUNMatrix \(A\) (with \(0 \leq i < M\) and \(0 \leq j < N\)) may be accessed via \(\text{cols}[j][i]\).

The header file to include when using this module is \texttt{sunmatrix/sunmatrix_dense.h}. The SUNMatrix.dense module is accessible from all SUNDIALS solvers without linking to the \texttt{libsundials} 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 \texttt{SM} in the names denotes that these macros are for \texttt{SUNMatrix} implementations, and the suffix \texttt{D} denotes that these are specific to the \textit{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:

\[
\text{#define SM\_CONTENT\_D}(A) \quad (\text{SUNMatrixContent\_Dense}(A->\text{content}) )
\]

**• 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:

\[
\text{#define SM\_ROWS\_D}(A) \quad (\text{SM\_CONTENT\_D}(A)->M )
\]

\[
\text{#define SM\_COLUMNS\_D}(A) \quad (\text{SM\_CONTENT\_D}(A)->N )
\]

\[
\text{#define SM\_LDATA\_D}(A) \quad (\text{SM\_CONTENT\_D}(A)->\text{ldata} )
\]

**• SM\_DATA\_D and SM\_COLS\_D**

These macros give access to the \texttt{data} and \texttt{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:

\[
\text{#define SM\_DATA\_D}(A) \quad (\text{SM\_CONTENT\_D}(A)->\text{data} )
\]

\[
\text{#define SM\_COLS\_D}(A) \quad (\text{SM\_CONTENT\_D}(A)->\text{cols} )
\]
7.5 The SUNMatrix_Dense implementation

- **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 \text{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**

\[ \text{SUNMatrix SUNDenseMatrix(sunindextype M, sunindextype N)} \]

**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**

\[ \text{void SUNDenseMatrix_Print(SUNMatrix A, FILE* outfile)} \]

**Description**

This function prints the content of a dense SUNMatrix to the output stream specified by \( \text{outfile} \). Note: stdout or stderr may be used as arguments for \( \text{outfile} \) to print directly to standard output or standard error, respectively.

#### SUNDenseMatrix_Rows

**Prototype**

\[ \text{sunindextype SUNDenseMatrix_Rows(SUNMatrix A)} \]

**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**

\[ \text{sunindextype SUNDenseMatrix_Columns(SUNMatrix A)} \]

**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.
Description of the SUNMatrix 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.

**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 \( A\_data = \text{SM\_DATA\_D}(A) \) or \( A\_data = \text{SUNDenseMatrix\_Data}(A) \) and then access \( A\_data[i] \) within the loop.
  - First obtain the array of column pointers via \( A\_cols = \text{SM\_COLS\_D}(A) \) or \( A\_cols = \text{SUNDenseMatrix\_Cols}(A) \), and then access \( A\_cols[j][i] \) within the loop.
  - Within a loop over the columns, access the column pointer via \( A\_colj = \text{SUNDenseMatrix\_Column}(A,j) \) and then to access the entries within that column using \( A\_colj[i] \) within the loop.

All three of these are more efficient than using \( \text{SM\_ELEMENT\_D}(A,i,j) \) 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 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 `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 ≤ mu < N`
- `ml` - lower half-bandwidth, `0 ≤ ml < N`
- `s_mu` - storage upper bandwidth, `mu ≤ 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 \(\min(N-1, \mu + \mu + ml)\) because of partial pivoting. The `s_mu` field holds the upper half-bandwidth allocated for `A`.
- `ldim` - leading dimension (`ldim ≥ s_mu + ml + 1`)
Description of the SUNMatrix module

Figure 7.1: Diagram of the storage for the SUNMATRIX_BAND module. Here $A$ is an $N \times N$ band matrix with upper and lower half-bandwidths $\mu$ and $ml$, respectively. The rows and columns of $A$ are numbered from 0 to $N-1$ and the $(i,j)$-th element of $A$ is denoted $A(i,j)$. The greyed out areas of the underlying component storage are used by the associated SUNLINSOL_BAND linear solver.

**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 $smu-\mu$ to $smu+ml$ to access the uppermost element within the band in the $j$-th column. Indices from 0 to $smu-\mu-1$ give access to extra storage elements required by the LU decomposition function. Finally, **cols[j][i-j+smu]** 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 The SUNMatrix_Band implementation

7.6.1 SUNMatrix_Band accessor macros

The following macros are provided to access the content of a SUNMATRIX_BAND matrix. The prefix SM_ in the names denotes that these macros are for SUNMatrix implementations, and the suffix _B denotes that these are specific to the banded version.

- **SM_CONTENT_B**
  
  This routine gives access to the contents of the banded SUNMatrix.
  
  The assignment \( A._{\text{cont}} = \text{SM\_CONTENT\_B}(A) \) sets \( A._{\text{cont}} \) to be a pointer to the banded SUNMatrix content structure.
  
  Implementation:
  
  ```c
  #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 \( A._{\text{rows}} = \text{SM\_ROWS\_B}(A) \) sets \( A._{\text{rows}} \) to be the number of rows in the matrix \( A \). Similarly, the assignment \( \text{SM\_COLUMNS\_B}(A) = A._{\text{cols}} \) sets the number of columns in \( A \) to equal \( A._{\text{cols}} \).
  
  Implementation:
  
  ```c
  #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 \( A._{\text{data}} = \text{SM\_DATA\_B}(A) \) sets \( A._{\text{data}} \) to be a pointer to the first component of the data array for the banded SUNMatrix \( A \). The assignment \( \text{SM\_DATA\_B}(A) = A._{\text{data}} \) sets the data array of \( A \) to be \( A._{\text{data}} \) by storing the pointer \( A._{\text{data}} \).
  
  Similarly, the assignment \( A._{\text{cols}} = \text{SM\_COLS\_B}(A) \) sets \( A._{\text{cols}} \) to be a pointer to the array of column pointers for the banded SUNMatrix \( A \). The assignment \( \text{SM\_COLS\_B}(A) = A._{\text{cols}} \) sets the column pointer array of \( A \) to be \( A._{\text{cols}} \) by storing the pointer \( A._{\text{cols}} \).
  
  Implementation:
  
  ```c
  #define SM_DATA_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 \( \text{SM\_ELEMENT\_B}(A,i,j) = a_{i,j} \) and \( a_{i,j} = \text{SM\_ELEMENT\_B}(A,i,j) \) reference the \((i,j)-th\) element of the \( N \times N \) band matrix \( A \), where \( 0 \leq i,j \leq N - 1 \). The location \((i,j)\) should further satisfy \( j - \mu \leq i \leq j + ml \).
  
  The assignment \( \text{col}_j = \text{SM\_COLUMN\_B}(A,j) \) sets \( \text{col}_j \) to be a pointer to the diagonal element of the \( j \)-th column of the \( N \times N \) band matrix \( A \), \( 0 \leq j \leq N - 1 \). The type of the expression
SM_COLUMN_B(A,j) is realtype *. The pointer returned by the call SM_COLUMN_B(A,j) can be treated as an array which is indexed from \(-\mu\) to \(m\).

The assignments SM_COLUMN_ELEMENT_B(col_j,i,j) = a_{ij} and 
a_{ij} = SM_COLUMN_ELEMENT_B(col_j,i,j) reference the \((i,j)\)-th entry of the band matrix \(A\) when used in conjunction with SM_COLUMN_B to reference the \(j\)-th column through col_j. The index \((i,j)\) should satisfy \(j-\mu \leq i \leq j+m\).

Implementation:

\[
\begin{align*}
\text{#define } \text{SM_COLUMN_B}(A,j) & \quad \left( ((\text{SM_CONTENT_B}(A)->cols)[j])+\text{SM_SUBAND_B}(A) \right) \\
\text{#define } \text{SM_COLUMN_ELEMENT_B}(\text{col}_j,i,j) & \quad (\text{col}_j[(i)-(j)]) \\
\text{#define } \text{SM_ELEMENT_B}(A,i,j) & \quad ( (\text{SM_CONTENT_B}(A)->cols)[j][(i)-(j)+\text{SM_SUBAND_B}(A)] )
\end{align*}
\]

### 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**

\[
\text{SUNMatrix } \text{SUNBandMatrix}(\text{sunindextype } N, \text{sunindextype } \mu, \text{sunindextype } m)
\]

**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 \(m\). The stored upper bandwidth is set to \(\mu+m\) 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**

\[
\text{SUNMatrix } \text{SUNBandMatrixStorage}(\text{sunindextype } N, \text{sunindextype } \mu, \text{sunindextype } m, \text{sunindextype } \text{smu})
\]

**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 \(m\), and the stored upper bandwidth, \(\text{smu}\). When creating a band SUNMatrix, this value should be

\begin{itemize}
  \item at least \(\min(N-1,\mu+m)\) if the matrix will be used by the SUNLINSOL_BAND module;
  \item exactly equal to \(\mu+m\) if the matrix will be used by the SUNLINSOL_LAPACKBAND module;
  \item at least \(\mu\) if used in some other manner.
\end{itemize}

**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.
7.6 The SUNMatrix_Band implementation

**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.

**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 $m_l$.

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 $A_{\text{data}} = \text{SM\_DATA\_B}(A)$ or $A_{\text{data}} = \text{SUNBandMatrix\_Data}(A)$ and then access $A_{\text{data}}[i]$ within the loop.
  - First obtain the array of column pointers via $A_{\text{cols}} = \text{SM\_COLS\_B}(A)$ or $A_{\text{cols}} = \text{SUNBandMatrix\_Cols}(A)$, and then access $A_{\text{cols}}[j][i]$ within the loop.
  - Within a loop over the columns, access the column pointer via $A_{\text{col}\_j} = \text{SUNBandMatrix\_Column}(A, j)$ and then to access the entries within that column using $\text{SM\_COLUMN\_ELEMENT\_B}(A_{\text{col}\_j}, i, j)$.

All three of these are more efficient than using $\text{SM\_ELEMENT\_B}(A, i, j)$ 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 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 \(\text{indexptr}[3]=7\), then the first nonzero entry in the fourth column of the matrix is located in data[7], and is located in row indexvals[7] of the matrix. The last entry contains the total number of nonzero values in the matrix and hence points one past the end of the active data in the data and indexvals arrays. For CSR matrices, each entry provides the index of the first row entry into the data and indexvals arrays.

The following pointers are added to the SlsMat type for user convenience, to provide a more intuitive interface to the CSC and CSR sparse matrix data structures. They are set automatically when creating a sparse SUNMATRIX, based on the sparse matrix storage type.

rowvals - pointer to indexvals when sparsetype is CSC_MAT, otherwise set to NULL.
colptrs - pointer to indexptrs when sparsetype is CSC_MAT, otherwise set to NULL.
colvals - pointer to indexvals when sparsetype is CSR_MAT, otherwise set to NULL.
rowptrs - pointer to indexptrs when sparsetype is CSR_MAT, otherwise set to NULL.

For example, the 5 \(\times\) 4 CSC matrix
\[
\begin{bmatrix}
0 & 3 & 1 & 0 \\
3 & 0 & 0 & 2 \\
0 & 7 & 0 & 0 \\
1 & 0 & 0 & 9 \\
0 & 0 & 0 & 5
\end{bmatrix}
\]
could be stored in this structure as either

\[
M = 5; \\
N = 4; \\
NNZ = 8; \\
NP = N; \\
data = \{3.0, 1.0, 3.0, 7.0, 1.0, 2.0, 9.0, 5.0\}; \\
sparsetype = \text{CSC\_MAT}; \\
\text{indexvals} = \{1, 3, 0, 2, 0, 1, 3, 4\}; \\
\text{indexptrs} = \{0, 2, 4, 5, 8\};
\]

or

\[
M = 5; \\
N = 4; \\
NNZ = 10; \\
NP = N; \\
data = \{3.0, 1.0, 3.0, 7.0, 1.0, 2.0, 9.0, 5.0, *, *\}; \\
sparsetype = \text{CSC\_MAT}; \\
\text{indexvals} = \{1, 3, 0, 2, 0, 1, 3, 4, *, *\}; \\
\text{indexptrs} = \{0, 2, 4, 5, 8\};
\]

where the first has no unused space, and the second has additional storage (the entries marked with * may contain any values). Note in both cases that the final value in indexptrs is 8, indicating the total number of nonzero entries in the matrix.

Similarly, in CSR format, the same matrix could be stored as

\[
M = 5; \\
N = 4; \\
NNZ = 8; \\
NP = N; \\
data = \{3.0, 1.0, 3.0, 2.0, 7.0, 1.0, 9.0, 5.0\}; \\
sparsetype = \text{CSR\_MAT}; \\
\text{indexvals} = \{1, 2, 0, 3, 1, 0, 3, 3\}; \\
\text{indexptrs} = \{0, 2, 4, 5, 7, 8\};
\]
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 \( \text{NNZ} \) nonzero entries (the allocated length of both \text{data} and \text{indexvals}). The entries in \text{indexvals} may assume values from 0 to \( M - 1 \), corresponding to the row index (zero-based) of each nonzero value. The entries in \text{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 \text{indexptrs} array contains \( N + 1 \) entries; the first \( N \) denote the starting index of each column within the \text{indexvals} and \text{data} arrays, while the final entry points one past the final nonzero entry. Here, although \( \text{NNZ} \) values are allocated, only \( \text{nz} \) are actually filled in; the greyed-out portions of \text{data} and \text{indexvals} indicate extra allocated space.
The header file to include when using this module is `sunmatrix/sunmatrix_sparse.h`. The `sunmatrix` 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.

  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 `A_rows = SM_ROWS_S(A)` sets `A_rows` to be the number of rows in the matrix `A`. Similarly, the assignment `SM_COLUMNS_S(A) = A_cols` sets the number of columns in `A` to equal `A_cols`.

  Implementation:

  ```
  #define SM_ROWS_S(A) ( SM_CONTENT_S(A)->M )
  #define SM_COLUMNS_S(A) ( SM_CONTENT_S(A)->N )
  #define SM_NNZ_S(A) ( SM_CONTENT_S(A)->NNZ )
  #define SM_NP_S(A) ( SM_CONTENT_S(A)->NP )
  #define SM_SPARSETYPE_S(A) ( SM_CONTENT_S(A)->sparsetype )
  ```

- **SM_DATA_S, SM_INDEXVALS_S, and SM_INDEXPTRS_S**

  These macros give access to the data and index arrays for the matrix entries.

  The assignment `A_data = SM_DATA_S(A)` sets `A_data` to be a pointer to the first component of the data array for the sparse SUNMatrix `A`. The assignment `SM_DATA_S(A) = A_data` sets the data array of `A` to be `A_data` by storing the pointer `A_data`.

  Similarly, the assignment `A_indexvals = SM_INDEXVALS_S(A)` sets `A_indexvals` to be a pointer to the array of index values (i.e. row indices for a CSC matrix, or column indices for a CSR matrix) for the sparse SUNMatrix `A`. The assignment `A_indexptrs = SM_INDEXPTRS_S(A)` sets `A_indexptrs` to be a pointer to the array of index pointers (i.e. the starting indices in the data/indexvals arrays for each row or column in CSR or CSC formats, respectively).

  Implementation:

  ```
  #define SM_DATA_S(A) ( SM_CONTENT_S(A)->data )
  #define SM_INDEXVALS_S(A) ( SM_CONTENT_S(A)->indexvals )
  #define SM_INDEXPTRS_S(A) ( SM_CONTENT_S(A)->indexptrs )
  ```
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:

**SUNSparseMatrix**

Prototype: SUNMatrix SUNSparseMatrix(sunindextype M, sunindextype N, sunindextype NNZ, int sparsetype)

Description: This function creates and allocates memory for a sparse SUNMatrix. Its arguments are the number of rows and columns of the matrix, M and N, the maximum number of nonzeros to be stored in the matrix, NNZ, and a flag sparsetype indicating whether to use CSR or CSC format (valid arguments are CSR_MAT or CSC_MAT).

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

**SUNSparseFromDenseMatrix**

Prototype: 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: 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.
Description of the SUNMatrix module

**SUNSparseMatrix**

**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.

**SUNSparseMatrix_Print**

**Prototype**

```c
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**

```c
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**

```c
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**

```c
sunindextype SUNSparseMatrix_NNZ(SUNMatrix A)
```

**Description**

This function returns the number of entries allocated for nonzero storage for the sparse matrix SUNMatrix.

**F2003 Name**

This function is callable as `FSUNSparseMatrix_NNZ` when using the Fortran 2003 interface module.
7.7 The SUNMatrix_Sparse implementation

**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.

**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` 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` 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 the 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, 21, 34, 35]. 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:

```c
struct _SUNMatrixContent_SLUNRloc {
    boolantype 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\textunderscore 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:

**SUNMatrix\_SLUNRloc**

Call: \[ A = \text{SUNMatrix\_SLUNRloc}(\text{Asuper}, \text{grid}); \]

Description: The function SUNMatrix\_SLUNRloc creates and allocates memory for a SUNMATRIX\_SLUNRLOC object.

Arguments: \n\text{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 \n\text{grid} (gridinfo\_t\*) the initialized SuperLU\_DIST 2D process grid structure

Return value: a SUNMatrix object if Asuper is compatible else NULL

Notes

**SUNMatrix\_SLUNRloc\_Print**

Call: \[ \text{SUNMatrix\_SLUNRloc\_Print}(A, fp); \]

Description: The function SUNMatrix\_SLUNRloc\_Print prints the underlying SuperMatrix content.

Arguments: \n\text{A} (SUNMatrix) the matrix to print \n\text{fp} (FILE) the file pointer used for printing

Return value: void

Notes

**SUNMatrix\_SLUNRloc\_SuperMatrix**

Call: \[ \text{Asuper = SUNMatrix\_SLUNRloc\_SuperMatrix}(A); \]

Description: The function SUNMatrix\_SLUNRloc\_SuperMatrix provides access to the underlying SuperLU\_DIST SuperMatrix of A.

Arguments: \n\text{A} (SUNMatrix) the matrix to access

Return value: SuperMatrix*

Notes

**SUNMatrix\_SLUNRloc\_ProcessGrid**

Call: \[ \text{grid = SUNMatrix\_SLUNRloc\_ProcessGrid}(A); \]

Description: The function SUNMatrix\_SLUNRloc\_ProcessGrid provides access to the SuperLU\_DIST gridinfo\_t structure associated with A.

Arguments: \n\text{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 booleantype

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_SLUNRloc** - performs \( A = cA + B \), but \( A \) and \( B \) must have the same sparsity pattern
- **SUNMatScaleAddI_SLUNRloc** - performs \( A = cA + I \), but the diagonal of \( A \) must be present
- **SUNMatMatvecSetup_SLUNRloc** - 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_SLUNRloc**

⚠️ The `SUNMATRIX_SLUNRLOC` 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 NVVECTOR 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 NVVECTOR 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

\[ \tilde{A}\tilde{x} = \tilde{b} \]  

(8.1)

where

\[ \tilde{A} = S_1P_1^{-1}AP_2^{-1}S_2^{-1}, \]

\[ \tilde{b} = S_1P_1^{-1}b, \]

\[ \tilde{x} = S_2P_2x, \]  

(8.2)

and where

- \( P_1 \) is the left preconditioner,
- \( P_2 \) is the right preconditioner,
- \( S_1 \) is a diagonal matrix of scale factors for \( P_1^{-1}b \),
- \( S_2 \) is a diagonal matrix of scale factors for \( P_2x \).
The scaling matrices are chosen so that $S_1 P_1^{-1} b$ and $S_2 P_2 x$ 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 $P x$, 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

$$\| \tilde{b} - \tilde{A} \tilde{x} \|_2 < \text{tol}.$$ 

When provided an iterative SUNLINSOL implementation that does not support the scaling matrices $S_1$ and $S_2$, SUNDIALS’ packages will adjust the value of tol accordingly (see §8.4.2 for more details). In this case, they instead request that iterative linear solvers stop based on the criteria

$$\| P_1^{-1} b - P_1^{-1} A x \|_2 < \text{tol}.$$ 

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_linearsolver.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 $A x = b$ (SUNLinSolSolve). The remaining routine for destruction of the linear solver object (SUNLinSolFree) is optional.
8.1 The SUNLinearSolver API

**SUNLinSolGetType**

Call  
\[ \text{type} = \text{SUNLinSolGetType}(\text{LS}); \]

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 inexactly (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 inexactly (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  
\[ \text{retval} = \text{SUNLinSolInitialize}(\text{LS}); \]

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  
\[ \text{retval} = \text{SUNLinSolSetup}(\text{LS}, \text{A}); \]

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

```c
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

```c
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

```c
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.
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: \[
\text{rvec} = \text{SUNLinSolResid}(\text{LS});
\]

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: \text{LS} (SUNLinearSolver) a SUNLINSOL object.

Return value: N_Vector containing the final residual vector.

Notes: Since \text{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: \[
\text{lflag} = \text{SUNLinSolLastFlag}(\text{LS});
\]

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: \text{LS} (SUNLinearSolver) a SUNLINSOL object.

Return value: long int containing the most recent error flag.

**SUNLinSolSpace**

Call: \[
\text{retval} = \text{SUNLinSolSpace}(\text{LS}, \&\text{lrw}, \&\text{liw});
\]

Description: The optional function SUNLinSolSpace should return the storage requirements for the linear solver \text{LS}.

Arguments: \text{LS} (SUNLinearSolver) a SUNLINSOL object.

\text{lrw} (long int*) the number of realtype words stored by the linear solver.
\text{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: \[
\text{typedef int} (*\text{ATimesFn})(\text{void} *\text{A_data}, \text{N_Vector v, N_Vector z});
\]

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  
\( A_{\text{data}} \) is a pointer to client data, the same as that supplied to \texttt{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.

\textbf{PSetupFn}

Definition  
\texttt{typedef int (*PSetupFn)(void *P\_data)}

Purpose  
These functions set up any requisite problem data in preparation for calls to the corresponding \texttt{PSolveFn}.

Arguments  
\( P_{\text{data}} \) is a pointer to client data, the same pointer as that supplied to the routine \texttt{SUNLinSolSetPreconditioner}.

Return value  
This routine should return 0 if successful and a non-zero value if unsuccessful.

\textbf{PSolveFn}

Definition  
\texttt{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 \( P_{\text{data}} \) is a pointer to any information about \( P \) which the function needs in order to do its job (set up by the corresponding \texttt{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

\[
\|Pz - r\|_{\text{wrms}} < tol
\]

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 \texttt{PSolveFn}.

Arguments  
\( P_{\text{data}} \) is a pointer to client data, the same pointer as that supplied to the routine \texttt{SUNLinSolSetPreconditioner}.  
\( r \) is the right-hand side vector for the preconditioner system.  
\( z \) is the solution vector for the preconditioner system.  
\( \text{tol} \) is the desired tolerance for an iterative preconditioner.  
\( \text{lr} \) 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 \texttt{SUNLINSOL} modules by each \texttt{SUNDIALS} package, and functions within the \texttt{SUNDIALS}-provided \texttt{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 \texttt{SUNLinearSolver} error codes

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{SUNLS_SUCCESS}</td>
<td>0</td>
<td>successful call or converged solve</td>
</tr>
<tr>
<td>\texttt{SUNLS_MEM_NULL}</td>
<td>-1</td>
<td>the memory argument to the function is \texttt{NULL}</td>
</tr>
<tr>
<td>\texttt{SUNLS_ILL_INPUT}</td>
<td>-2</td>
<td>an illegal input has been provided to the function</td>
</tr>
<tr>
<td>\texttt{SUNLS_MEM_FAIL}</td>
<td>-3</td>
<td>failed memory access or allocation</td>
</tr>
<tr>
<td>\texttt{SUNLS_ATIMES_FAIL_UNREC}</td>
<td>-4</td>
<td>an unrecoverable failure occurred in the \texttt{ATimes} routine</td>
</tr>
<tr>
<td>\texttt{SUNLS_PSET_FAIL_UNREC}</td>
<td>-5</td>
<td>an unrecoverable failure occurred in the \texttt{Pset} routine</td>
</tr>
<tr>
<td>\texttt{SUNLS_PSOLVE_FAIL_UNREC}</td>
<td>-6</td>
<td>an unrecoverable failure occurred in the \texttt{Psolve} routine</td>
</tr>
<tr>
<td>\texttt{SUNLS_PACKAGE_FAIL_UNREC}</td>
<td>-7</td>
<td>an unrecoverable failure occurred in an external linear solver package</td>
</tr>
<tr>
<td>\texttt{SUNLS_GS_FAIL}</td>
<td>-8</td>
<td>a failure occurred during Gram-Schmidt orthogonalization</td>
</tr>
<tr>
<td>\texttt{SUNLS_QRSOL_FAIL}</td>
<td>-9</td>
<td>a singular ( R ) matrix was encountered in a QR factorization</td>
</tr>
<tr>
<td>\texttt{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>\texttt{SUNLS_CONV_FAIL}</td>
<td>2</td>
<td>an iterative solver did not converge (and the residual was not reduced)</td>
</tr>
<tr>
<td>\texttt{SUNLS_ATIMES_FAIL_REC}</td>
<td>3</td>
<td>a recoverable failure occurred in the \texttt{ATimes} routine</td>
</tr>
<tr>
<td>\texttt{SUNLS_PSET_FAIL_REC}</td>
<td>4</td>
<td>a recoverable failure occurred in the \texttt{Pset} routine</td>
</tr>
<tr>
<td>\texttt{SUNLS_PSOLVE_FAIL_REC}</td>
<td>5</td>
<td>a recoverable failure occurred in the \texttt{Psolve} routine</td>
</tr>
<tr>
<td>\texttt{SUNLS_PACKAGE_FAIL_REC}</td>
<td>6</td>
<td>a recoverable failure occurred in an external linear solver package</td>
</tr>
<tr>
<td>\texttt{SUNLS_QRFAC_FAIL}</td>
<td>7</td>
<td>a singular matrix was encountered during a QR factorization</td>
</tr>
<tr>
<td>\texttt{SUNLS_LUFAC_FAIL}</td>
<td>8</td>
<td>a singular matrix was encountered during a LU factorization</td>
</tr>
</tbody>
</table>

\subsection{8.1.6 The generic SUNLinearSolver module}

\texttt{SUNDIALS} packages interact with specific SUNLINSOL implementations through the generic SUNLINSOL module on which all other SUNLINSOL implementations are built. The \texttt{SUNLinearSolver} type is a pointer to a structure containing an implementation-dependent content field, and an ops field. The type \texttt{SUNLinearSolver} is defined as

\begin{verbatim}
typedef struct _generic_SUNLinearSolver *SUNLinearSolver;

struct _generic_SUNLinearSolver {    
    void *content;
    struct _generic_SUNLinearSolver_Ops *ops;
};
\end{verbatim}

where the \texttt{._generic_SUNLinearSolver\_Ops} structure is a list of pointers to the various actual linear solver operations provided by a specific implementation. The \texttt{._generic_SUNLinearSolver\_Ops} structure is defined as

\begin{verbatim}
struct _generic_SUNLinearSolver_Ops {    
    SUNLinearSolver_Type (*gettype)(SUNLinearSolver);
};
\end{verbatim}
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 8.2: SUNDIALS matrix-based linear solvers and matrix implementations that can be used for each.

<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_SPFGMR, SUNLINSOL_SPBCGS, SUNLINSOL_SPTQMR, or SUNLINSOL_PCGL) 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 §4.5.7.2) 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 CVODE SUNLinearSolver interface

Table 8.3 below lists the SUNLINSOL module linear solver functions used within the CVLS interface. As with the SUNMATRIX module, we emphasize that the CVODE user does not need to know detailed usage of linear solver functions by the CVODE code modules in order to use CVODE. 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. SUNLinSolNumIters is only used to accumulate overall iterative linear solver statistics. If it is not implemented by the SUNLINSOL module, then CVLS will consider all solves as requiring zero iterations.

2. Although CVLS does not call SUNLinSolLastFlag directly, this routine is available for users to query linear solver issues directly.

3. Although CVLS 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 CVLS 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>&quot;SUNLinSolNumIters&quot;</td>
<td></td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td>&quot;SUNLinSolLastFlag&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;SUNLinSolFree&quot;</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 CVLS 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

\[ M\bar{x} = b \leftrightarrow (I - \gamma J)\bar{x} = b \]

it is likely that the value \( \bar{\gamma} \) used to construct \( M \) differs from the current value of \( \gamma \) in the linear multistep method, since \( M \) is updated infrequently. Therefore, after calling the SUNLINSOL-provided SUNLinSolSolve routine, we test whether \( \gamma/\bar{\gamma} \neq 1 \), and if this is the case we scale the solution \( \bar{x} \) to obtain the desired linear system solution \( x \) via

\[ x = \frac{2}{1 + \gamma/\bar{\gamma}} \bar{x}. \]
For values of $\gamma/\bar{\gamma}$ that are “close” to 1, this rescaling approximately solves the original linear system, as discussed below. We first note that the equation (8.3) is equivalent to

$$\bar{x} = \frac{1}{2} \left(1 + \frac{\gamma}{\bar{\gamma}}\right) x.$$ 

Adding the two equations $(I - \gamma J)x = b$ and $(I - \bar{\gamma} J)\bar{x} = b$, and inserting the above relationship, we have

$$2b = (I - \gamma J)x + (I - \bar{\gamma} J)\bar{x} = x - \gamma Jx + \bar{x} - J(\bar{\gamma} \bar{x})$$

$$= \frac{3}{2} (I - \gamma J)x + \frac{1}{2} \left(\frac{\gamma}{\bar{\gamma}} I - \bar{\gamma} J\right)x$$

When $\gamma/\bar{\gamma} \approx 1$, this latter term is approximately equal to $\frac{1}{2} b$.

### 8.4.2 Iterative linear solver tolerance

If the `sunlinsol` object self-identifies as having type `SUNLINEARSOLVER_ITERATIVE` or `SUNLINEARSOLVER_MATRIX_ITERATIVE` then `cvls` 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 `cvls` 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 `cvode` 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)):

$$\left\| \tilde{b} - \tilde{A} \tilde{x} \right\|_2 < \text{tol}$$

$$\iff \left\| SP_1^{-1} \tilde{b} - SP_1^{-1} \tilde{A} \tilde{x} \right\|_2 < \text{tol}$$

$$\iff \sum_{i=0}^{n-1} \left[ W_i \left( P_1^{-1} (b - Ax)_i \right) \right]^2 < \text{tol}^2$$

$$\iff W_{\text{mean}}^2 \sum_{i=0}^{n-1} \left[ (P_1^{-1} (b - Ax)_i) \right]^2 < \text{tol}^2$$

$$\iff \sum_{i=0}^{n-1} \left[ (P_1^{-1} (b - Ax)_i) \right]^2 < \left( \frac{\text{tol}}{W_{\text{mean}}} \right)^2$$

$$\iff \left\| P_1^{-1} (b - Ax) \right\|_2 < \frac{\text{tol}}{W_{\text{mean}}}$$

Therefore the tolerance scaling factor

$$W_{\text{mean}} = \|W\|_2 / \sqrt{n}$$

is computed and the scaled tolerance `delta` = `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 SUNLINSOL_DENSE 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_DENSE module, include the header file sunlinsol/sunlinsol_dense.h. We note that the SUNLINSOL_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.

```c
SUNLinSol_Dense
Call
LS = SUNLinSol_Dense(y, A);
Description
The function SUNLinSol_Dense creates and allocates memory for a dense SUNLinearSolver object.
Arguments
y (N_Vector) a template for cloning vectors needed within the solver
A (SUNMatrix) a SUNMATRIX_DENSE matrix template for cloning matrices needed within the solver
Return value
This returns a SUNLinearSolver object. If either A or y are incompatible then this routine will return NULL.
Notes
This routine will perform consistency checks to ensure that it is called with consistent NVECTOR and SUNMATRIX implementations. These are currently limited to the SUNMATRIX_DENSE matrix type and the NVECTOR_SERIAL, NVECTOR_OPENMP, and NVECTOR_PTHREADS vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.
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.
8.5 The SUNLinearSolver_Dense implementation

- **SUNLinSolSetup_Dense** – this performs the $LU$ factorization.
- **SUNLinSolSolve_Dense** – this uses the $LU$ factors and pivots array to perform the solve.
- **SUNLinSolLastFlag_Dense**
- **SUNLinSolSpace_Dense** – this only returns information for the storage within the solver object, i.e. storage for $N$, last_flag, and pivots.
- **SUNLinSolFree_Dense**

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.

**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.

```fortran
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.
208 Description of the SUNLinearSolver module

**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:

```c
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 SUNLIN- SOL_BAND 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_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 \( mu \) and lower bandwidth \( ml \), then the upper triangular factor \( U \) can have upper bandwidth as big as \( smu = \text{MIN}(N-1, mu+ml) \). The lower triangular factor \( L \) has lower bandwidth \( ml \).
8.6 The SUNLinearSolver_Band implementation

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**

```c
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 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 `SUNBandLinearSolver` with identical input and output arguments is also provided.

**F2003 Name**
This function is callable as `FSUNLinSol_Band` 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 NVVECTOR 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 NVVECTOR 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.

```
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 consis-
tent NV ECTOR and SUNMATRIX implementations. These are currently limited to 
the SUNMATRIX_DENSE matrix type and the NV ECTOR_SERIAL, 
NV ECTOR_OPENMP, 
and NV ECTOR_PTHREADS vector types. As additional compatible matrix and vec-
tor 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 per-
formed at solver creation.
- SUNLinSolSetup_LapackDense – this calls either DGETRF or SGETRF to perform the LU factor-
ization.
- 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  FSUNLAPACKDENSEINIT(code, ier)

Description  The function FSUNLAPACKDENSEINIT can be called for Fortran programs to create a 
LAPACK-based 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 NV ECTOR 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 ob-
ject.

**FSUNMASSLAPACKDENSEINIT**

Call  FSUNMASSLAPACKDENSEINIT(ier)

Description  The function FSUNMASSLAPACKDENSEINIT can be called for Fortran programs to create a 
LAPACK-based, dense SUNLinearSolver object for mass matrix linear systems.

Arguments  None
8.8 The SUNLinearSolver_LapackBand implementation

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.

8.7.4 SUNLinearSolver_LapackDense content

The SUNLINSOL_LAPACKDENSE module defines the `content` field of a SUNLinearSolver as the following structure:

```c
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.

8.8 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 `libsundials_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 factor-
ization. More precisely, if $A$ is a band matrix with upper bandwidth $\mu$ and lower bandwidth $ml$, then the upper triangular factor $U$ can have upper bandwidth as big as $smu = \min(N-1, \mu+ml)$. The lower triangular factor $L$ has lower bandwidth $ml$.

### 8.8.2 SUNLinearSolver_LapackBand functions

The SUNLINSOL_LAPACKBAND module provides the following user-callable constructor for creating a SUNLinearSolver object.

```c
SUNLinSol_LapackBand
```

**Call**

```c
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.
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.

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:

```
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/sunlinsolklu.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, 15]. In order to use the `sunlinsol_klu` interface to KLU, it is assumed that KLU has been installed on the system prior to installation of Sundials, and that Sundials has been configured appropriately to link with KLU (see Appendix A for details). Additionally, this wrapper only supports double-precision calculations, and therefore cannot be compiled if Sundials is configured to have `realtypes` 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 `sunindex` 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.

```
[SUNLinSol_KLU]
Call
   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
```
8.9 The SUNLinearSolver_KLU implementation

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.

Deprecated 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
- 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 SUNLINSOL_KLU module also defines the following additional user-callable functions.

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUNLinSol_KLUReInit</td>
<td>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).</td>
<td>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 nonzeroes in the matrix, reinit_type (int) flag governing the level of reinitialization. The allowed values are:</td>
</tr>
</tbody>
</table>
• **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_IILL_INPUT` (A does not have type `SUNMATRIX_SPARSE` or `reinit_type` is invalid), `SUNLS_MEM_FAIL` (reallocation of the sparse matrix failed) or `SUNLS_SUCCESS`.

**Notes**

This routine will perform consistency checks to ensure that it is called with consistent `nvector` and `sunmatrix` implementations. These are currently limited to the `SUNMATRIX_SPARSE` matrix type (using either CSR or CSC storage formats) and the `nvector_serial`, `nvector_openmp`, and `nvector_pthreads` vector types. As additional compatible matrix and vector implementations are added to Sundials, these will be included within this compatibility check.

This routine assumes no other changes to solver use are necessary.

**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_KLUSetOrdering`

**Call**

```c
retval = SUNLinSol_KLUSetOrdering(LS, ordering);
```

**Description**

This function sets the ordering used by KLU for reducing fill in the linear solve.

**Arguments**

- `LS` *(SUNLinearSolver)* the `SUNINSOL_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_IILL_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.
8.9 The SUNLinearSolver_KLU implementation

**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.

**FSUNKLINIT**

Call

```
FSUNKLINIT(code, ier)
```

Description The function `FSUNKLINIT` can be called for Fortran programs to create a `SUNLINSOL_KLU` object.

Arguments

- `code` ([int*]) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).

Return value `ier` is a return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes This routine must be called after both the `NVECTOR` and `SUNMATRIX` objects have been initialized.

Additionally, when using ARKODE with a non-identity mass matrix, the `SUNLINSOL_KLU` module includes a Fortran-callable function for creating a `SUNLinearSolver` mass matrix solver object.

**FSUNMASSKLUINIT**

Call

```
FSUNMASSKLUINIT(ier)
```

Description The function `FSUNMASSKLUINIT` can be called for Fortran programs to create a KLU-based `SUNLinearSolver` object for mass matrix linear systems.

Arguments None

Return value `ier` is a `int` return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes This routine must be called after both the `NVECTOR` and `SUNMATRIX` mass-matrix objects have been initialized.

The `SUNLinSol_KLUReInit` and `SUNLinSol_KLUSetOrdering` routines also support FORTRAN interfaces for the system and mass matrix solvers:

**FSUNKLUREINIT**

Call

```
FSUNKLUREINIT(code, nnz, reinit_type, ier)
```

Description The function `FSUNKLUREINIT` can be called for Fortran programs to re-initialize a `SUNLINSOL_KLU` object.

Arguments

- `code` ([int*]) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).
- `nnz` ([sunindextype*]) the new number of nonzeros in the matrix
Description of the SUNLinearSolver module

reinit_type (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 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 nnz given in the sparse matrix provided to the original constructor routine (or the previous SUNLinSol_KLUReInit call).

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_KLUReInit for complete further documentation of this routine.

FSUNMASSKLUREINIT

Call  FSUNMASSKLUREINIT(nnz, reinit_type, ier)
Description The function 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 FSUNKLUREINIT 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_KLUReInit for complete further documentation of this routine.

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.

FSUNMASSKLUESTORDERING

Call  FSUNMASSKLUESTORDERING(ier)
Description The function FSUNMASSKLUESTORDERING 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.
8.10 The SUNLinearSolver_SuperLUDIST implementation

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_SLUNRLOC 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_sunlinsol superludist.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, 21, 34, 35]. 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_SLUNRLOC matrix template for cloning matrices needed within the solver
grid (gridinfo_t*)
lu (LUstruct_t*)
```
8.10 The SUNLinearSolver_SuperLUDIST implementation

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_PARHYPER, 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 SUNLIN-SOL_SUPERLUDIST solver. Specifically the member Fact, is modified in the setup and solve routines.

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.
Arguments    LS (SUNLinearSolver) the SUNLIN-SOL_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 SUNLIN-SOL_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 SUNLIN-SOL_SUPERLUDIST object
Return value  LUstruct_t*
Notes
224  Description of the SUNLinearSolver module

```
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

```
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

```
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:

```
struct _SUNLinearSolverContent_SuperLUDIST {
  boolentype first_factorize;
  long int last_flag;
  realtype berr;
  gridinfo_t *grid;
  LUstruct_t *lu;
  superlu_dist_options_t *options;
```
8.11 The SUNLinearSolver_SuperLUMT implementation

```c
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 SUNLinearSolver module 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 NVECTOR implementations (NVECTOR_SERIAL, NVECTOR_OPENMP, or NVECTOR_PTHREADS). While these are compatible, it is not recommended to use a threaded vector module with SUNINSOL_SUNLINEAR_SOLVER unless it is the NVECTOR_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 SUNINSOL_SUNLINEAR_SOLVER module is a SUNINSOL wrapper for the SUPERLUMT sparse matrix factorization and solver library written by X. Sherry Li [3, 33, 17]. The package performs matrix factorization using threads to enhance efficiency in shared memory parallel environments. It should be noted that threads are only used in the factorization step. In order to use the SUNINSOL_SUNLINEAR_SOLVER interface to SUPERLUMT, it is assumed that SUPERLUMT has been installed on the system prior to installation of SUNDIALS, and that SUNDIALS has been configured appropriately to link with SUPERLUMT (see Appendix A for details). Additionally, this wrapper only supports single- and double-precision calculations, and therefore cannot be compiled if SUNDIALS is configured to have **realtype** set to `extended` (see Section 4.2). Moreover, since the SUPERLUMT library may be installed to support either 32-bit or 64-bit integers, it is assumed that the SUPERLUMT library is installed using the same integer precision as the SUNDIALS **sunindextype** option.

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 \( A^T \cdot A \), minimal degree ordering on \( A^T + A \), or natural ordering). Of these ordering choices, the default value in the SUNINSOL_SUNLINEAR_SOLVER 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.
8.11 The SUNLinearSolver_SuperLUMT implementation

- **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.

```c
SUNLinSol_SuperLUMTSetOrdering
```

**Call**

```c
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.

```f90
FSUNSUPERLUMTINIT
```

**Call**

```f90
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 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.

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**
- `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_SuperLUMTSetOrdering for complete further documentation of this routine.

**FSUNMASSSUPERLUMTSETORDERING**

Call: `FSUNMASSSUPERLUMTSETORDERING(ordering, ier)`

**Description**
The function `FSUNMASSSUPERLUMTSETORDERING` can be called for Fortran programs to update the ordering algorithm in a SUNLINSOL_SUPERLUMT object for mass matrix linear systems.

**Arguments**
- `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**
- `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_SuperLUMTSetOrdering for complete further documentation of this routine.
8.11.4 SUNLinearSolver_SuperLUMT content

The SUNLINSOL_SUPERLUMT module defines the `content` field of a SUNLinearSolver as the following structure:

```c
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;
};
```

These entries of the `content` field contain the following information:
- `last_flag` - last error return flag from internal function evaluations,
- `first_factorize` - flag indicating whether the factorization has ever been performed,
- `A, AC, L, U, B` - SuperMatrix pointers used in solve,
- `Gstat` - GStat_t object used in solve,
- `perm_r, perm_c` - permutation arrays used in solve,
- `N` - size of the linear system,
- `num_threads` - number of OpenMP/Pthreads threads to use,
- `diag_pivot_thresh` - threshold on diagonal pivoting,
- `ordering` - flag for which reordering algorithm to use,
- `options` - pointer to SUPERLUMT options structure.

8.12 The SUNLinearSolver_SPGMR implementation

This section describes the SUNLINSOL implementation of the SPGMR (Scaled, Preconditioned, Generalized Minimum Residual [40]) iterative linear solver. The SUNLINSOL_SPGMR module is designed to be compatible with any NVector implementation that supports a minimal subset of operations (NVClone, NVDotProd, NVScale, NVLinearSum, NVProd, NVConst, NVDiv, and NVDestroy). When using Classical Gram-Schmidt, the optional function NVDotProdMulti 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 NVector that is input, and default solver parameters are set.
- User-facing “set” routines may be called to modify default solver parameters.
- Additional “set” routines are called by the SUNDIALS solver that interfaces with SUNLINSOL_SPGMR to supply the ATimes, PSetup, and Psolve function pointers and s1 and s2 scaling vectors.
In the “initialize” call, the remaining solver data is allocated (\(V\), \(\text{Hes}\), \(\text{givens}\), and \(yg\)).

In the “setup” call, any non-NULL \(\text{PSetup}\) function is called. Typically, this is provided by the \text{SUNDIALS}\ solver itself, that translates between the generic \(\text{PSetup}\) 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 \text{SUNLINSOL\_SPGMR}\ module provides the following user-callable constructor for creating a \text{SUNLinearSolver} object.

```c
SUNLinSol\_SPGMR
Call
LS = SUNLinSol\_SPGMR(y, pretype, maxl);
```

**Description**
The function \text{SUNLinSol\_SPGMR} creates and allocates memory for a \text{SPGMR}\ \text{SUNLinearSolver} object.

**Arguments**
- \(y\) (\text{N\_Vector}) a template for cloning vectors needed within the solver
- \(pretype\) (int) flag indicating the desired type of preconditioning, allowed values are:
  - \text{PREC\_NONE} (0)
  - \text{PREC\_LEFT} (1)
  - \text{PREC\_RIGHT} (2)
  - \text{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 \(\leq 0\) will result in the default value (5).

**Return value**
This returns a \text{SUNLinearSolver} object. If either \(y\) is incompatible then this routine will return \text{NULL}.

**Notes**
This routine will perform consistency checks to ensure that it is called with a consistent \text{NVECTOR} implementation (i.e. that it supplies the requisite vector operations). If \(y\) is incompatible, then this routine will return \text{NULL}.

We note that some \text{SUNDIALS}\ solvers are designed to only work with left preconditioning (\text{IDA} and \text{IDAS}) and others with only right preconditioning (\text{KINSOL}). While it is possible to configure a \text{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 \text{SUNSPGMR} with identical input and output arguments is also provided.

**F2003 Name**
This function is callable as \text{FSUNLinSol\_SPGMR} when using the Fortran 2003 interface module.

The \text{SUNLINSOL\_SPGMR}\ module defines implementations of all “iterative” linear solver operations listed in Sections 8.1.1 – 8.1.3:

- \text{SUNLinSolGetType\_SPGMR}
- \text{SUNLinSolInitialize\_SPGMR}
- \text{SUNLinSolSetATimes\_SPGMR}
- \text{SUNLinSolSetPreconditioner\_SPGMR}
8.12 The SUNLinearSolver_SPGMR implementation

- 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.

**SUNLinSol_SPGMRSetPrecType**

Call: `retval = SUNLinSol_SPGMRSetPrecType(LS, 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.

**SUNLinSol_SPGMRSetGSType**

Call: `retval = SUNLinSol_SPGMRSetGSType(LS, 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 pretype), 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.
Description of the SUNLinearSolver module

```c
SUNLinSol_SPGMRSetMaxRestarts
Call retval = SUNLinSol_SPGMRSetMaxRestarts(LS, maxrs);
Description The function SUNLinSol_SPGMRSetMaxRestarts sets the number of GMRES restarts to allow in the SUNLINSOL_SPGMR object.
Arguments LS (SUNLinearSolver) the SUNLINSOL_SPGMR object to update
maxrs (int) integer indicating number of restarts to allow. A negative input will result in the default of 0.
Return value This routine will return with one of the error codes SUNLS_MEM_NULL (S is NULL) or SUNLS_SUCCESS.
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.

```c
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 NVVECTOR object has been initialized.
Allowable values for pretype and maxl are the same as for the C function SUNLinSol_SPGMR.
```
Additionally, when using ARKODE with a non-identity mass matrix, the SUNLINSOL_SPGMR module includes a Fortran-callable function for creating a SUNLinearSolver mass matrix solver object.

```
FSUNMASSSPGMRINIT
Call
FSUNMASSSPGMRINIT(pretype, maxl, ier)
Description
The function FSUNMASSSPGMRINIT can be called for Fortran programs to create a
SUNLINSOL_SPGMR object for mass matrix linear systems.
Arguments
pretype (int*) flag indicating desired preconditioning type
maxl (int*) flag indicating Krylov subspace size
Return value
ier is a int return completion flag equal to 0 for a success return and -1 otherwise.
See printed message for details in case of failure.
Notes
This routine must be called after the NVECTOR object has been initialized.
Allowable values for pretype and maxl are the same as for the C function
SUNLinSol_SPGMR.
The SUNLinSol_SPGMRSetPrecType, SUNLinSol_SPGMRSetGSType and
SUNLinSol_SPGMRSetMaxRestarts routines also support Fortran interfaces for the system and mass
matrix solvers.
```

```
FSUNSPGMRSETGSTYPE
Call
FSUNSPGMRSETGSTYPE(code, gstype, ier)
Description
The function FSUNSPGMRSETGSTYPE can be called for Fortran programs to change the
Gram-Schmidt orthogonalisation algorithm.
Arguments
code (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for
KINSOL, and 4 for ARKODE).
gstype (int*) flag indicating the desired orthogonalization algorithm.
Return value
ier is a int return completion flag equal to 0 for a success return and -1 otherwise.
See printed message for details in case of failure.
Notes
See SUNLinSol_SPGMRSetGSType for complete further documentation of this routine.
```

```
FSUNMASSSPGMRSETGSTYPE
Call
FSUNMASSSPGMRSETGSTYPE(gstype, ier)
Description
The function FSUNMASSSPGMRSETGSTYPE can be called for Fortran programs to change
the Gram-Schmidt orthogonalisation algorithm for mass matrix linear systems.
Arguments
The arguments are identical to FSUNSPGMRSETGSTYPE above, except that code is not
needed since mass matrix linear systems only arise in ARKODE.
Return value
ier is a int return completion flag equal to 0 for a success return and -1 otherwise.
See printed message for details in case of failure.
Notes
See SUNLinSol_SPGMRSetGSType for complete further documentation of this routine.
```

```
FSUNSPGMRSETPRECTYPE
Call
FSUNSPGMRSETPRECTYPE(code, pretype, ier)
Description
The function FSUNSPGMRSETPRECTYPE can be called for Fortran programs to change the
type of preconditioning to use.
Arguments
code (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3
for KINSOL, and 4 for ARKODE).
pretype (int*) flag indicating the type of preconditioning to use.
```
Return value \textbf{ier} is an \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\_SPGMRSetPrecType for complete further documentation of this routine.

\textbf{FSUNMASSSPGMRSETPRECTYPE}

Call \texttt{FSUNMASSSPGMRSETPRECTYPE} (pretype, ier)

Description The function \texttt{FSUNMASSSPGMRSETPRECTYPE} can be called for Fortran programs to change the type of preconditioning for mass matrix linear systems.

Arguments The arguments are identical to \texttt{FSUNSPGMRSETPRECTYPE} above, except that \texttt{code} is not needed since mass matrix linear systems only arise in ARKODE.

Return value \textbf{ier} is an \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\_SPGMRSetPrecType for complete further documentation of this routine.

\textbf{FSUNSPGMRSETMAXRS}

Call \texttt{FSUNSPGMRSETMAXRS} (code, maxrs, ier)

Description The function \texttt{FSUNSPGMRSETMAXRS} can be called for Fortran programs to change the maximum number of restarts allowed for SPGMR.

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{maxrs} (\texttt{int}*) maximum allowed number of restarts.

Return value \textbf{ier} is an \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\_SPGMRSetMaxRestarts for complete further documentation of this routine.

\textbf{FSUNMASSSPGMRSETMAXRS}

Call \texttt{FSUNMASSSPGMRSETMAXRS} (maxrs, ier)

Description The function \texttt{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 \texttt{FSUNSPGMRSETMAXRS} above, except that \texttt{code} is not needed since mass matrix linear systems only arise in ARKODE.

Return value \textbf{ier} is an \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\_SPGMRSetMaxRestarts for complete further documentation of this routine.

\subsection{SUNLinearSolver\_SPGMR content}

The SUNLINSOL\_SPGMR module defines the \textit{content} field of a SUNLinearSolver as the following structure:

\begin{verbatim}
struct _SUNLinearSolverContent_SPGMR {
  int maxl;
  int pretype;
  int gstype;
  int max_restarts;
  int numiters;
}
\end{verbatim}
These entries of the `content` field contain the following information:

- `maxl` - number of GMRES basis vectors to use (default is 5),
- `pretype` - flag for type of preconditioning to employ (default is none),
- `gstype` - flag for type of Gram-Schmidt orthogonalization (default is modified Gram-Schmidt),
- `max restarts` - number of GMRES restarts to allow (default is 0),
- `numiters` - number of iterations from the most-recent solve,
- `resnorm` - final linear residual norm from the most-recent solve,
- `last_flag` - last error return flag from an internal function,
- `ATimes` - function pointer to perform $Av$ product,
- `ATData` - pointer to structure for `ATimes`,
- `Psetup` - function pointer to preconditioner setup routine,
- `Psolve` - function pointer to preconditioner solve routine,
- `PData` - pointer to structure for `Psetup` and `Psolve`,
- `s1, s2` - vector pointers for supplied scaling matrices (default is NULL),
- `V` - the array of Krylov basis vectors $v_1, \ldots, v_{maxl+1}$, stored in $V[0], \ldots, V[maxl]$. 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 $F_0, F_1, \ldots, F_j$, where

$$F_i = \begin{bmatrix} 1 & \cdots & 1 & c_i & -s_i \\ s_i & c_i & 1 & \cdots & 1 \end{bmatrix},$$

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`,
236 Description of the SUNLinearSolver module

xcor       - a vector which holds the scaled, preconditioned correction to the initial guess,
yg         - a length \( (\text{maxl}+1) \) array of \text{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 [39]) iterative linear solver. The SUNLINSOL\_SPFGMR module is designed to be compatible with any NVECTOR implementation that supports a minimal subset of operations \( (N\_VClone, N\_VDotProd, N\_VScale, N\_VLinearSum, N\_VProd, N\_VConst, N\_VDiv, \text{and} N\_VDestroy) \). When using Classical Gram-Schmidt, the optional function \( N\_VDotProdMulti \) may be supplied for increased efficiency. Unlike the other Krylov iterative linear solvers supplied with SUNDIALS, 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 \texttt{sunlinsol/sunlinsol\_spfgmr.h}.

We note that the SUNLINSOL\_SPFGMR module is accessible from SUNDIALS packages without separately linking to the \texttt{libsundials/sunlinsol\_spfgmr} 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 \( \text{ATimes}, \text{PSetup}, \text{and Psolve} \) function pointers and \( s1 \) and \( s2 \) scaling vectors.
- In the “initialize” call, the remaining solver data is allocated \( (V, Hes, \text{givens}, \text{and} yg) \)
- In the “setup” call, any non-NULL \( \text{PSetup} \) function is called. Typically, this is provided by the SUNDIALS solver itself, that translates between the generic \( \text{PSetup} \) 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       \((\text{N\_Vector})\) a template for cloning vectors needed within the solver
             pretype (\text{int}) flag indicating the desired type of preconditioning, allowed values are:
             - PREC\_NONE \((0)\)
             - PREC\_LEFT \((1)\)
```

...
8.13 The SUNLinearSolver_SPFGMR implementation

- 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 \( \leq 0 \) will result in the default value (5).

Return value This returns a SUNLinearSolver object. If either \( y \) is incompatible then this routine will return NULL.

Notes This routine will perform consistency checks to ensure that it is called with a consistent NVECTOR implementation (i.e. that it supplies the requisite vector operations). If \( y \) is incompatible, then this routine will return NULL.

We note that some SUNDIALS solvers are designed to only work with left preconditioning (IDA and IDAS) and others with only right preconditioning (KINSOL). While it is possible to configure a SUNLINSOL_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:

- 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 SUNSPFGMRSesetPrecType with identical input and output arguments is also provided.

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

SUNLinSol_SPFGMRSesetGSType

Call  
\[
\text{retval} = \text{SUNLinSol_SPFGMRSesetGSType}(\text{LS}, \text{gstype});
\]

Description The function SUNLinSol_SPFGMRSesetPrecType sets the type of Gram-Schmidt orthogonalization to use in the SUNLINSOL_SPFGMR object.

Arguments
- \text{LS} (SUNLinearSolver) the SUNLINSOL_SPFGMR object to update
- \text{gstype} (int) flag indicating the desired orthogonalization algorithm; allowed values are:
  - \text{MODIFIED_GS} (1)
  - \text{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 SUNSPFGMRSesetGSType with identical input and output arguments is also provided.

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

SUNLinSol_SPFGMRSesetMaxRestarts

Call  
\[
\text{retval} = \text{SUNLinSol_SPFGMRSesetMaxRestarts}(\text{LS}, \text{maxrs});
\]

Description The function SUNLinSol_SPFGMRSesetMaxRestarts sets the number of GMRES restarts to allow in the SUNLINSOL_SPFGMR object.

Arguments
- \text{LS} (SUNLinearSolver) the SUNLINSOL_SPFGMR object to update
- \text{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 SUNSPFGMRSesetMaxRestarts with identical input and output arguments is also provided.

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

8.13.3 SUNLinearSolver_SPFGMR Fortran interfaces

The SUNLINSOL_SPFGMR module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.
8.13 The SUNLinearSolver SPF GMR implementation

FORTRAN 2003 interface module

The fsunlinsol_spfgmr_mod FORTRAN module defines interfaces to all SUNLINSOL_SPFGMR 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_SPFGMR is interfaced as FSUNLinSol_SPFGMR.

The FORTRAN 2003 SUNLINSOL_SPFGMR interface module can be accessed with the use statement, i.e. use fsunlinsol_spfgmr_mod, and linking to the library libsundials_fsunlinsolspfgmr_mod.lib in addition to the C library. For details on where the library and module file fsunlinsol_spfgmr_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_fsunlinsolspfgmr_mod library.

FORTRAN 77 interface functions

For solvers that include a FORTRAN 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 NVVECTOR 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 NVVECTOR 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.
**Description of the SUNLinearSolver module**

**FSUNSPFGMRSETGSTYPE**

Call

\[ \text{FSUNSPFGMRSETGSTYPE}(\text{code, gstype, ier}) \]

Description

The function \text{FSUNSPFGMRSETGSTYPE} can be called for Fortran programs to change the Gram-Schmidt orthogonalization algorithm.

Arguments

- \text{code} (int*) is an integer input specifying the solver id (1 for \text{cvode}, 2 for \text{ida}, 3 for \text{kinsol}, and 4 for \text{arkode}).
- \text{gstype} (int*) flag indicating the desired orthogonalization algorithm.

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

See SUNLinSol\_SPFGMRSetGSType for complete further documentation of this routine.

**FSUNMASSSPFGMRSETGSTYPE**

Call

\[ \text{FSUNMASSSPFGMRSETGSTYPE}(\text{gstype, ier}) \]

Description

The function \text{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 \text{FSUNSPFGMRSETGSTYPE} above, except that \text{code} is not needed since mass matrix linear systems only arise in \text{arkode}.

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

See SUNLinSol\_SPFGMRSetGSType for complete further documentation of this routine.

**FSUNSPFGMRSETPRECTYPE**

Call

\[ \text{FSUNSPFGMRSETPRECTYPE}(\text{code, pretype, ier}) \]

Description

The function \text{FSUNSPFGMRSETPRECTYPE} can be called for Fortran programs to change the type of preconditioning to use.

Arguments

- \text{code} (int*) is an integer input specifying the solver id (1 for \text{cvode}, 2 for \text{ida}, 3 for \text{kinsol}, and 4 for \text{arkode}).
- \text{pretype} (int*) flag indicating the type of preconditioning to use.

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

See SUNLinSol\_SPFGMRSetPrecType for complete further documentation of this routine.

**FSUNMASSSPFGMRSETPRECTYPE**

Call

\[ \text{FSUNMASSSPFGMRSETPRECTYPE}(\text{pretype, ier}) \]

Description

The function \text{FSUNMASSSPFGMRSETPRECTYPE} can be called for Fortran programs to change the type of preconditioning for mass matrix linear systems.

Arguments

The arguments are identical to \text{FSUNSPFGMRSETPRECTYPE} above, except that \text{code} is not needed since mass matrix linear systems only arise in \text{arkode}.

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

See SUNLinSol\_SPFGMRSetPrecType for complete further documentation of this routine.
8.13 The SUNLinearSolver_SPFGMR implementation

**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 SPFGMR 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 $Hes[i][j]$.,
- **givens** - a length $2*maxl$ array which represents the Givens rotation matrices that arise in the FGMRES algorithm. These matrices are $F_0, F_1, \ldots, F_j$, where

\[
F_i = \begin{bmatrix}
1 & & & & \\
& 1 & & & \\
& & c_i & -s_i & \\
& & s_i & c_i & \\
& & & & 1 \\
& & & & & & & 1
\end{bmatrix},
\]

- **xicor** - 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.

### 8.14 The SUNLinearSolver_SPBCGS implementation

This section describes the SUNLINSOL implementation of the SPBCGS (Scaled, Preconditioned, Bi-Conjugate Gradient, Stabilized [41]) iterative linear solver. The SUNLINSOL_SPBCGS module is designed to be compatible with any nvector implementation that supports a minimal subset of operations ($NVClone$, $NVDotProd$, $NVScale$, $NVLinearSum$, $NVPdot$, $NVDiv$, and $NVDestroy$). 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 `sunlinsol/sunlinsol_spbcgs.h`. We note that the SUNLINSOL_SPBCGS module is accessible from SUNDIALS packages *without* separately linking to the `libsundials_sunlinsolspbcs` 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 ATimes, PSetup, and Psolve function pointers and s1 and s2 scaling vectors.
- In the “initialize” call, the solver parameters are checked for validity.
- In the “setup” call, any non-NULL PSetup function is called. Typically, this is provided by the SUNDIALS solver itself, that translates between the generic PSetup function and the solver-specific routine (solver-supplied or user-supplied).
- In the “solve” call the spbcgs iteration is performed. This will include scaling and preconditioning if those options have been supplied.

### 8.14.2 SUNLinearSolver_SPBCGS functions

The SUNLINSOL_SPBCGS module provides the following user-callable constructor for creating a SUNLinearSolver object.

```c
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.
Description of the SUNLinearSolver module

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 SUNLINSOL_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
- 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.

**SUNLinSol_SPBCGSSetPrecType**

Call

```c
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.
8.14 The SUNLinearSolver_SPBCGS implementation

![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 a int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes: This routine must be called after the NVECTOR object has been initialized. Allowable values for `pretype` and `maxl` are the same as for the C function SUNLinSol_SPBCGS.

The SUNLinSol_SPBCGSSetPrecType and SUNLinSol_SPBCGSSetMaxl routines also support Fortran interfaces for the system and mass matrix solvers.

**FSUNSPBCGSSETPRECTYPE**

Call: ```FSUNSPBCGSSETPRECTYPE(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.

**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: `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.

**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 \( ier \) 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 \texttt{SUNLinSol\_SPBCGSSetMaxl} for complete further documentation of this routine.

\texttt{FSUNMASSSPBCGSSETMAXL}

Call \texttt{FSUNMASSSPBCGSSETMAXL(maxl, ier)}

Description The function \texttt{FSUNMASSSPBCGSSETMAXL} can be called for Fortran programs to change the type of preconditioning for mass matrix linear systems.

Arguments The arguments are identical to \texttt{FSUNSPBCGSSETMAXL} above, except that \texttt{code} is not needed since mass matrix linear systems only arise in ARKODE.

Return value \( ier \) 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 \texttt{SUNLinSol\_SPBCGSSetMaxl} for complete further documentation of this routine.

8.14.4 SUNLinearSolver\_SPBCGS content

The \texttt{sunlinsol\_spbcgs} module defines the \texttt{content} field of a \texttt{SUNLinearSolver} as the following structure:

\texttt{struct _SUNLinearSolverContent\_SPBCGS { } }

\begin{verbatim}
  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;
};
\end{verbatim}

These entries of the \texttt{content} field contain the following information:

- \texttt{maxl} - number of \texttt{SPBCGS} iterations to allow (default is 5),
- \texttt{pretype} - flag for type of preconditioning to employ (default is none),
- \texttt{numiters} - number of iterations from the most-recent solve,
- \texttt{resnorm} - final linear residual norm from the most-recent solve,
- \texttt{last_flag} - last error return flag from an internal function,
- \texttt{ATimes} - function pointer to perform \( Av \) product,
- \texttt{ATData} - pointer to structure for \texttt{ATimes},
- \texttt{Psetup} - function pointer to preconditioner setup routine,
Psolve - function pointer to preconditioner solve routine,
PData - pointer to structure for Psetup and Psolve,
s1, s2 - vector pointers for supplied scaling matrices (default is NULL),
r - a NVECTOR which holds the current scaled, preconditioned linear system residual,
r_star - a NVECTOR which holds the initial scaled, preconditioned linear system residual,
p, q, u, Ap, vtemp - NVECTORS used for workspace by the SPBCGS algorithm.

8.15 The SUNLinearSolver_SPTFQMR implementation

This section describes the SUNLINSOL implementation of the SPTFQMR (Scaled, Preconditioned, Transpose-Free Quasi-Minimum Residual [20]) iterative linear solver. The SUNLINSOL_SPTFQMR module is designed to be compatible with any NVECTOR implementation that supports a minimal subset of operations ($N_{VClone}$, $N_{VDotProd}$, $N_{VScale}$, $N_{VLinearSum}$, $N_{VProd}$, $N_{VConst}$, $N_{VDiv}$, 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 NVECTOR solver data is allocated, with vectors cloned from a template NVECTOR that is input, and default solver parameters are set.
- User-facing “set” routines may be called to modify default solver parameters.
- Additional “set” routines are called by the SUNDIALS solver that interfaces with SUNLINSOL_SPTFQMR to supply the $ATimes$, PSetup, and Psolve function pointers and s1 and s2 scaling vectors.
- In the “initialize” call, the solver parameters are checked for validity.
- In the “setup” call, any non-NULL PSetup function is called. Typically, this is provided by the SUNDIALS solver itself, that translates between the generic PSetup function and the solver-specific routine (solver-supplied or user-supplied).
- In the “solve” call the TFQMR iteration is performed. This will include scaling and preconditioning if those options have been supplied.

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:
8.15 The SUNLinearSolver_SPTFQMR implementation

- 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 NVVECTOR 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
- 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.
Description of the SUNLinearSolver module

SUNLinSol_SPTFQMRSetPrecType
Call
    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
    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 \leq 0 will result in the default value (5)

Return value
This routine will return with one of the error codes SUNLS_MEM_NULL (S is NULL) or SUNLS_SUCCESS.

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_SPFGMR 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_sptfqmr_mod Fortran module defines interfaces to all SUNLINSOL_SPFGMR 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_SPFGMR interface module can be accessed with the use statement, i.e. use fsunlinsol_sptfqmr_mod, and linking to the library libsundials_fsunlinsolsptfqmr_mod.lib in addition to the C library. For details on where the library and module file fsunlinsol_sptfqmr_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_fsunlinsolsptfqmr_mod library.

FORTRAN 77 interface functions

For solvers that include a Fortran 77 interface module, the SUNLINSOL_SPTFQMR module also includes a Fortran-callable function for creating a SUNLinearSolver object.
8.15 The SUNLinearSolver_SPTFQMR implementation

**FSUNSPTFQMRINIT**

Call

FSUNSPTFQMRINIT(code, pretype, maxl, ier)

Description

The function FSUNSPTFQMRINIT can be called for Fortran programs to create a SUNLIN-SOL_SPTFQMR object.

Arguments

- **code** (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).
- **pretype** (int*) flag indicating desired preconditioning type
- **maxl** (int*) flag indicating number of iterations to allow

Return value

- **ier** is a return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes

- This routine must be called after the NVECTOR object has been initialized.
- Allowable values for pretype and maxl are the same as for the C function SUNLinSol_SPTFQMR.

Additionally, when using ARKODE with a non-identity mass matrix, the SUNLINSOL_SPTFQMR module includes a Fortran-callable function for creating a SUNLinearSolver mass matrix solver object.

**FSUNMASSSPTFQMRINIT**

Call

FSUNMASSSPTFQMRINIT(pretype, maxl, ier)

Description

The function FSUNMASSSPTFQMRINIT can be called for Fortran programs to create a SUNLINSOL_SPTFQMR object for mass matrix linear systems.

Arguments

- **pretype** (int*) flag indicating desired preconditioning type
- **maxl** (int*) flag indicating number of iterations to allow

Return value

- **ier** is 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_SPTFQMR.

The SUNLinSol_SPTFQMRSetPrecType and SUNLinSol_SPTFQMRSetMaxl routines also support Fortran interfaces for the system and mass matrix solvers.

**FSUNSPTFQMRSETPRECTYPE**

Call

FSUNSPTFQMRSETPRECTYPE(code, pretype, ier)

Description

The function FSUNSPTFQMRSETPRECTYPE can be called for Fortran programs to change the type of preconditioning to use.

Arguments

- **code** (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).
- **pretype** (int*) flag 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_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 a `int` return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes See `SUNLinSol_SPTFQMRSetPrecType` for complete further documentation of this routine.

### `FSUNSPTFQMRSETMAXL`

Call `FSUNSPTFQMRSETMAXL(code, maxl, ier)`

Description The function `FSUNSPTFQMRSETMAXL` can be called for Fortran programs to change the maximum number of iterations to allow.

Arguments `code` (`int*`) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).

`maxl` (`int*`) the number of iterations to allow.

Return value `ier` is a `int` return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes See `SUNLinSol_SPTFQMRSetMaxl` for complete further documentation of this routine.

### `FSUNMASSSPTFQMRSETMAXL`

Call `FSUNMASSSPTFQMRSETMAXL(maxl, ier)`

Description The function `FSUNMASSSPTFQMRSETMAXL` can be called for Fortran programs to change the type of preconditioning for mass matrix linear systems.

Arguments The arguments are identical to `FSUNSPTFQMRSETMAXL` above, except that `code` is not needed since mass matrix linear systems only arise in ARKODE.

Return value `ier` is a `int` return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes See `SUNLinSol_SPTFQMRSetMaxl` for complete further documentation of this routine.

### 8.15.4 SUNLinearSolver_SPTFQMR content

The `sulinsol_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;
};
```
8.16 The SUNLinearSolver_PCG implementation

```c
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 [22]) 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_PCG module, include the header file `sunlinsol/sunlinsol_pcg.h`. We note that the SUNLINSOL_PCG 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

$$\tilde{A}\tilde{x} = \tilde{b}$$  \hspace{1cm} (8.4)

where

$$\tilde{A} = SP^{-1}AP^{-1}S,$$

$$\tilde{b} = SP^{-1}b,$$

$$\tilde{x} = S^{-1}P x.$$  \hspace{1cm} (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 $\|v\|_S = \sqrt{v^T S S v}$, 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 a 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 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
```
The SUNLinearSolver_PC_preconditioning algorithm is designed to only support symmetric preconditioning, then any
one 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 \leq 0 will result in
the default value (5).

Return value This returns a SUNLinearSolver object. If either \( y \) is incompatible then this
routine will return NULL.

Notes This routine will perform consistency checks to ensure that it is called with a consis-
tent 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 precondition-
ing (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 idential input and
output arguments is also provided.

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

The SUNLINSOL_PC module defines implementations of all “iterative” linear solver operations listed
in Sections 8.1.1 – 8.1.3:

- SUNLinSolGetType_PC
- SUNLinSolInitialize_PC
- SUNLinSolSetATimes_PC
- SUNLinSolSetPreconditioner_PC
- SUNLinSolSetScalingVectors_PC – since PCG only supports symmetric scaling, the second
  nvector argument to this function is ignored
- SUNLinSolSetup_PC
- SUNLinSolSolve_PC
- SUNLinSolNumIters_PC
- SUNLinSolResNorm_PC
- SUNLinSolResid_PC
- SUNLinSolLastFlag_PC
- SUNLinSolSpace_PC
- SUNLinSolFree_PC

All of the listed operations are callable via the FORTRAN 2003 interface module by prepending an ‘F’
to the function name.

The SUNLINSOL_PC module also defines the following additional user-callable functions.
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.

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.
The SUNLinearSolver<sub>PCG</sub> implementation

### FSUNPCGINIT

**Call**

FSUNPCGINIT(code, pretype, maxl, ier)

**Description**
The function FSUNPCGINIT can be called for Fortran programs to create a SUNLIN-<sub>SO</sub>l<sub>PCG</sub> 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<sub>PCG</sub>.
- Additionally, when using ARKODE with a non-identity mass matrix, the SUNLinS<sub>O</sub>l<sub>PCG</sub> module includes a Fortran-callable function for creating a SUNLinearSolver mass matrix solver object.

### FSUNMASSPCGINIT

**Call**

FSUNMASSPCGINIT(pretype, maxl, ier)

**Description**
The function FSUNMASSPCGINIT can be called for Fortran programs to create a SUNLIN-<sub>SO</sub>l<sub>PCG</sub> 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<sub>PCG</sub>.
- The SUNLinSol<sub>PCG</sub>SetPrecType and SUNLinSol<sub>PCG</sub>SetMaxI routines also support Fortran interfaces for the system and mass matrix solvers.

### FSUNPCGSETPRECTYPE

**Call**

FSUNPCGSETPRECTYPE(code, pretype, ier)

**Description**
The function FSUNPCGSETPRECTYPE can be called for Fortran programs to change the type of preconditioning to use.

**Arguments**
- **code** (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).
- **pretype** (int*) flag 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<sub>PCG</sub>SetPrecType for complete further documentation of this routine.

### FSUNMASSPCGSETPRECTYPE

**Call**

FSUNMASSPCGSETPRECTYPE(pretype, ier)

**Description**
The function FSUNMASSPCGSETPRECTYPE can be called for Fortran programs to change the type of preconditioning for mass matrix linear systems.

**Arguments**
The arguments are identical to FSUNPCGSETPRECTYPE above, except that code is not needed since mass matrix linear systems only arise in ARKODE.
Return value \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 \texttt{SUNLinSol\_PCGSetPrecType} for complete further documentation of this routine.

\textbf{FSUNPCGSETMAXL}

Call \texttt{FSUNPCGSETMAXL(code, maxl, ier)}

Description The function \texttt{FSUNPCGSETMAXL} can be called for Fortran programs to change the maximum number of iterations to allow.

Arguments \texttt{code (int*)} is an integer input specifying the solver id (1 for \texttt{cvode}, 2 for \texttt{ida}, 3 for \texttt{kinsol}, and 4 for \texttt{arkode}).

\texttt{maxl (int*)} the number of iterations to allow.

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 \texttt{SUNLinSol\_PCGSetMaxl} for complete further documentation of this routine.

\textbf{FSUNMASSPCGSETMAXL}

Call \texttt{FSUNMASSPCGSETMAXL(maxl, ier)}

Description The function \texttt{FSUNMASSPCGSETMAXL} can be called for Fortran programs to change the type of preconditioning for mass matrix linear systems.

Arguments The arguments are identical to \texttt{FSUNPCGSETMAXL} above, except that \texttt{code} is not needed since mass matrix linear systems only arise in \texttt{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 \texttt{SUNLinSol\_PCGSetMaxl} for complete further documentation of this routine.

### 8.16.4 SUNLinearSolver\_PCG content

The \texttt{SUNLINSOL\_PCG} module defines the \texttt{content} field of a \texttt{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 \texttt{content} field contain the following information:

- \texttt{maxl} - number of PCG iterations to allow (default is 5),
- \texttt{pretype} - flag for use of preconditioning (default is none),
numiters - number of iterations from the most-recent solve,
resnorm - final linear residual norm from the most-recent solve,
last_flag - last error return flag from an internal function,
ATimes - function pointer to perform $Av$ product,
ATData - pointer to structure for ATimes,
Psetup - function pointer to preconditioner setup routine,
Psolve - function pointer to preconditioner solve routine,
PData - pointer to structure for Psetup and Psolve,
s - vector pointer for supplied scaling matrix (default is NULL),
r - a NVECTOR which holds the preconditioned linear system residual,
p, z, Ap - NVECTORS used for workspace by the PCG algorithm.

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 (SUNNonlinSolGetType) 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 type = SUNNonlinSolGetType(NLS);

Description The required function SUNNonlinSolGetType returns nonlinear solver type.

Arguments NLS (SUNNonlinearSolver) a SUNNONLINSOL object.

Return value The return value type (of type int) will be one of the following:
- SUNNONLINEARSOLVER_ROOTFIND 0, the SUNNONLINSOL module solves $F(y) = 0$.
- SUNNONLINEARSOLVER_FIXEDPOINT 1, the SUNNONLINSOL module solves $G(y) = y$.

**SUNNonlinSolInitialize**

Call retval = SUNNonlinSolInitialize(NLS);

Description The optional function SUNNonlinSolInitialize performs nonlinear solver initialization and may perform any necessary memory allocations.

Arguments NLS (SUNNonlinearSolver) a SUNNONLINSOL object.

Return value The return value retval (of type int) is zero for a successful call and a negative value for a failure.

Notes It is assumed all solver-specific options have been set prior to calling SUNNonlinSolInitialize. SUNNONLINSOL implementations that do not require initialization may set this operation to NULL.

**SUNNonlinSolSetup**

Call retval = SUNNonlinSolSetup(NLS, y, mem);

Description The optional function SUNNonlinSolSetup performs any solver setup needed for a nonlinear solve.

Arguments NLS (SUNNonlinearSolver) a SUNNONLINSOL object.
- y (N_Vector) the initial iteration passed to the nonlinear solver.
- mem (void *) the SUNDIALS integrator memory structure.

Return value The return value retval (of type int) is zero for a successful call and a negative value for a failure.

Notes SUNDIALS integrators call SUNNonlinSolSetup before each step attempt. SUNNONLINSOL implementations that do not require setup may set this operation to NULL.

**SUNNonlinSolSolve**

Call retval = SUNNonlinSolSolve(NLS, y0, y, w, tol, callLSetup, mem);

Description The required function SUNNonlinSolSolve solves the nonlinear system $F(y) = 0$ or $G(y) = y$.

Arguments NLS (SUNNonlinearSolver) a SUNNONLINSOL object.
- y0 (N_Vector) the initial iterate for the nonlinear solve. This must remain unchanged throughout the solution process.
- y (N_Vector) the solution to the nonlinear system.
- w (N_Vector) the solution error weight vector used for computing weighted error norms.
- tol (realtype) the requested solution tolerance in the weighted root-mean-squared norm.
- callLSetup (booleantype) a flag indicating that the integrator recommends for the linear solver setup function to be called.
mem (void *) the SUNDIALS integrator memory structure.

Return value The return value retval (of type int) is zero for a successful solve, a positive value for a recoverable error, and a negative value for an unrecoverable error.

**SUNNonlinSolFree**

Call retval = SUNNonlinSolFree(NLS);

Description The *optional* function SUNNonlinSolFree frees any memory allocated by the nonlinear solver.

Arguments NLS (SUNNonlinearSolver) a SUNNONSOL object.

Return value The return value retval (of type int) should be zero for a successful call, and a negative value for a failure. SUNNONSOL 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 SUNNONSOL 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 SUNNONSOL 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). SUNNONSOL 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.
The optional function \texttt{SUNNonlinSolSetLSolveFn} is called by SUNDIALS integrators to provide the nonlinear solver with access to its linear solver solve function.

**Arguments**
- \texttt{NLS} (\texttt{SUNNonlinearSolver}) a \texttt{SUNNONLINSOL} object
- \texttt{LSolveFn} (\texttt{SUNNonlinSolLSolveFn}) a wrapper function to the SUNDIALS integrator’s linear solver solve function. See section 9.1.4 for the definition of \texttt{SUNNonlinSolLSolveFn}.

**Return value**
The return value \texttt{retval} (of type \texttt{int}) should be zero for a successful call, and a negative value for a failure.

**Notes**
The \texttt{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 \texttt{NULL}.

The optional function \texttt{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**
- \texttt{NLS} (\texttt{SUNNonlinearSolver}) a \texttt{SUNNONLINSOL} object.
- \texttt{CTestFn} (\texttt{SUNNonlineSolConvTestFn}) a SUNDIALS integrator’s nonlinear solver convergence test function. See section 9.1.4 for the definition of \texttt{SUNNonlinSolConvTestFn}.

**Return value**
The return value \texttt{retval} (of type \texttt{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 \texttt{NULL}.

The optional function \texttt{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**
- \texttt{NLS} (\texttt{SUNNonlinearSolver}) a \texttt{SUNNONLINSOL} object.
- \texttt{maxiters} (\texttt{int}) the maximum number of nonlinear iterations.

**Return value**
The return value \texttt{retval} (of type \texttt{int}) should be zero for a successful call, and a negative value for a failure (e.g., \texttt{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 (\texttt{SUNNonlinSolGetNumIters}) and number of convergence failures (\texttt{SUNNonlinSolGetNumConvFails}) are optional. The routine to get the current nonlinear solver iteration (\texttt{SUNNonlinSolGetCurIter}) is required when using the convergence test provided by the SUNDIALS integrator or by the ARKODE and CVODE linear solver interfaces. Otherwise, \texttt{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 spils 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.

### SUNNonlinSolLSetupFn

**Definition**
```c
typedef int (*SUNNonlinSolLSetupFn)(N_Vector y, N_Vector F, boolantype jbad, boolantype* 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 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 ignore these functions.

### SUNNonlinSolLSolveFn

**Definition**
```c
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 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 ignore these functions.

### SUNNonlinSolConvTestFn

**Definition**
```c
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.
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_RECSR: 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>
<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_RECSR</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_I LL_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:

typedef struct _generic_SUNNonlinearSolver *SUNNonlinearSolver;

struct _generic_SUNNonlinearSolver {
    void *content;
    struct _generic_SUNNonlinearSolver_Ops *ops;
};;

where the _generic_SUNNonlinearSolver_Ops structure is a list of pointers to the various actual nonlinear solver operations provided by a specific implementation. The _generic_SUNNonlinearSolver_Ops structure is defined as
struct _generic_SUNNonlinearSolver_Ops {
    SUNNonlinearSolver_Type (*gettype)(SUNNonlinearSolver);
    int (*initialize)(SUNNonlinearSolver);
    int (*setup)(SUNNonlinearSolver, N_Vector, void*);
    int (*solve)(SUNNonlinearSolver, N_Vector, N_Vector,
                 N_Vector, realtype, booleantype, void*);
    int (*free)(SUNNonlinearSolver);
    int (*setsysfn)(SUNNonlinearSolver, SUNNonlinSolSysFn);
    int (*setlsetupfn)(SUNNonlinearSolver, SUNNonlinSolLSetupFn);
    int (*setlsolvefn)(SUNNonlinearSolver, SUNNonlinSolLSolveFn);
    int (*setctestfn)(SUNNonlinearSolver, SUNNonlinSolConvTestFn);
    int (*setmaxiters)(SUNNonlinearSolver, int);
    int (*getnumiters)(SUNNonlinearSolver, long int*);
    int (*getcuriter)(SUNNonlinearSolver, int*);
    int (*getnumconvfails)(SUNNonlinearSolver, long int*);
};

The generic SUNNONLINSOL module defines and implements the nonlinear solver operations defined in Sections 9.1.1 – 9.1.3. These routines are in fact only wrappers to the nonlinear solver operations provided by a particular SUNNONLINSOL implementation, which are accessed through the ops field of the SUNNonlinearSolver structure. To illustrate this point we show below the implementation of a typical nonlinear solver operation from the generic SUNNONLINSOL module, namely SUNNonlinSolSolve, which solves the nonlinear system and returns a flag denoting a successful or failed solve:

```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 NVVECTOR module is used to interface with SUNNONLINSOL modules for solves involving sensitivity vectors stored in an NVVECTOR array. As described below, the NVVECTOR_SENSWRAPPER module is an NVVECTOR implementation where the vector content is an NVVECTOR array. This wrapper vector allows SUNNONLINSOL modules to operate on data stored as a collection of vectors.

For all SUNDIALS-provided SUNNONLINSOL modules a special constructor wrapper is provided so users do not need to interact directly with the NVVECTOR_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 NVVECTOR_SENSWRAPPER module**

This section describes the NVVECTOR_SENSWRAPPER implementation of an NVVECTOR. To access the NVVECTOR_SENSWRAPPER module, include the header file sundials/sundials_nvvector_senswrapper.h.

The NVVECTOR_SENSWRAPPER module defines an NVVECTOR 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 \( n \), \( \text{N_VLinearSum}(a,x,b,y,z) \), is
computed as 
\[ z_{j,i} = a x_{j,i} + b y_{j,i}, \quad i = 0, \ldots, n - 1, \quad j = 0, \ldots, n_v - 1. \]

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 \( n n_v \). Thus \( d = N_{\text{VDotProd}}(x, y) \) is
\[ d = \sum_{j=0}^{n-1} \sum_{i=0}^{n-1} x_{j,i} y_{j,i}. \]

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 \( m = N_{\text{VWrmsNorm}}(x, w) \) is
\[ m = \max_j \left( \sqrt{\frac{1}{n} \sum_{i=0}^{n-1} (x_{j,i} w_{j,i})^2} \right) \]

To enable usage alongside other NVVECTOR modules the NVVECTOR_SENSWRAPPER functions implementing vector operations have _SensWrapper appended to the generic vector operation name.

The NVVECTOR_SENSWRAPPER module provides the following constructors for creating an NVVECTOR_SENSWRAPPER:

```
N_VNewEmpty_SensWrapper
Call w = N_VNewEmpty_SensWrapper(count);
Description The function N_VNewEmpty_SensWrapper creates an empty NVVECTOR_SENSWRAPPER wrapper with space for \( \text{count} \) 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 NVVECTOR object if the constructor exits successfully, otherwise \( w \) will be NULL.
```

```
N_VNew_SensWrapper
Call w = N_VNew_SensWrapper(count, y);
Description The function N_VNew_SensWrapper creates an NVVECTOR_SENSWRAPPER wrapper containing \( \text{count} \) 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 NVVECTOR object if the constructor exits successfully, otherwise \( w \) will be NULL.
```

The NVVECTOR_SENSWRAPPER implementation of the NVVECTOR module defines the content field of the N_Vector to be a structure containing a N_Vector array, the number of vectors in the vector array, and a boolean flag indicating ownership of the vectors in the vector array.

```
struct _N_VectorContent_SensWrapper {
    N_Vector* vecs;
    int nvecs;
    boolean type own_vecs;
};
```

The following macros are provided to access the content of an NVVECTOR_SENSWRAPPER vector:

- \text{NV\_CONTENT\_SW}(v) - provides access to the content structure
- \text{NV\_VECS\_SW}(v) - provides access to the vector array
• **NV_NVECS_SW(v)** - provides access to the number of vectors
• **NV_OWNECS_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

\[ F(y) = 0 \]  \tag{9.1} 

given an initial guess \( y^{(0)} \), Newton’s method computes a series of approximate solutions

\[ y^{(m+1)} = y^{(m)} + \delta^{(m+1)} \]  \tag{9.2} 

where \( m \) is the Newton iteration index, and the Newton update \( \delta^{(m+1)} \) is the solution of the linear system

\[ A(y^{(m)}) \delta^{(m+1)} = -F(y^{(m)}), \]  \tag{9.3} 

in which \( A \) is the Jacobian matrix

\[ A \equiv \partial F / \partial y. \]  \tag{9.4} 

Depending on the linear solver used, the SUNNONLINSOL module will employ either a Modified Newton method, or an Inexact Newton method [5, 8, 16, 18, 32]. 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 SUNNonlinearSolverSetupFn 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 module will call the SUNNonlinearSolverSetupFn 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.

**SUNNonlinSol_Newton**

Call

\[
NLS = \text{SUNNonlinSol_Newton}(y);
\]

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

- \(y\) (NVector) 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.

**SUNNonlinSol_NewtonSens**

Call

\[
NLS = \text{SUNNonlinSol_NewtonSens}(count, y);
\]

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 \(Ns\) sensitivities the value of \(count\) is:
  - \(Ns+1\) if using a simultaneous corrector approach.
  - \(Ns\) if using a staggered corrector approach.
- \(y\) (NVector) 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**

<table>
<thead>
<tr>
<th>Call</th>
<th>retrval = SUNNonlinSolGetSysFn_Newton(NLS, SysFn);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>The function SUNNonlinSolGetSysFn_Newton returns the residual function that defines the nonlinear system.</td>
</tr>
</tbody>
</table>
| Arguments | **NLS** (SUNNonlinearSolver) a SUNNONLINSOL object  
**SysFn** (SUNNonlinSolSysFn*) the function defining the nonlinear system. |
| Return value | The return value retrval (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 interoperaing 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\_fsunnonlinsol\_newton\_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\_fsunnonlinsol\_newton\_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**

<table>
<thead>
<tr>
<th>Call</th>
<th>FSUNNEWTONINIT(code, ier);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>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.</td>
</tr>
<tr>
<td>Arguments</td>
<td><strong>code</strong> (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, and 4 for ARKODE).</td>
</tr>
<tr>
<td>Return value</td>
<td><strong>ier</strong> is a return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.</td>
</tr>
</tbody>
</table>
9.3 The SUNNonlinearSolver_FixedPoint implementation

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

\[ G(y) = y \] (9.5)

given an initial guess \( y^{(0)} \), the fixed point iteration computes a series of approximate solutions

\[ y^{(n+1)} = G(y^{(n)}) \] (9.6)

where \( n \) is the iteration index. The convergence of this iteration may be accelerated using Anderson’s method \([4, 42, 19, 36]\). With Anderson acceleration using subspace size \( m \), the series of approximate solutions can be formulated as the linear combination

\[ y^{(n+1)} = \sum_{i=0}^{m} \alpha_i^{(n)} G(y^{(n-m_{n+i})}) \] (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
\[
F_n = (f_{n-m_n}, \ldots, f_n)
\] (9.9)

with \( f_i = G(y^{(i)}) - y^{(i)} \). Due to this constraint, in the limit of \( m = 0 \) the accelerated fixed point iteration formula (9.7) simplifies to the standard fixed point iteration (9.6).

Following the recommendations made in [42], the SUNNONLINSOL\_FIXEDPOINT implementation computes the series of approximate solutions as
\[
y^{(n+1)} = G(y^{(n)}) - \sum_{i=0}^{m_n-1} \gamma_i^{(n)} \Delta g_{n-m_n+i}
\] (9.10)

with \( \Delta g_i = G(y^{(i+1)}) - G(y^{(i)}) \) and where the factors
\[
\gamma^{(n)} = (\gamma_0^{(n)}, \ldots, \gamma_{m_n-1}^{(n)})
\] (9.11)

solve the unconstrained minimization problem \( \min_\gamma \| f_n - \Delta F_n \gamma^T \|_2 \) where
\[
\Delta F_n = (\Delta f_{n-m_n}, \ldots, \Delta f_{n-1})
\] (9.12)

with \( \Delta f_i = f_{i+1} - f_i \). The least-squares problem is solved by applying a QR factorization to \( \Delta F_n = Q_n R_n \) and solving \( R_n \gamma = Q_n^T f_n \).

The acceleration subspace size \( m \) is required when constructing the SUNNONLINSOL\_FIXEDPOINT object. The default maximum number of iterations and the stopping criteria for the fixed point iteration are supplied by the SUNDIALS integrator when SUNNONLINSOL\_FIXEDPOINT is attached to it. Both the maximum number of iterations and the convergence test function may be modified by the user by calling SUNNonlinSolSetMaxIters and SUNNonlinSolSetConvTestFn functions after attaching the SUNNONLINSOL\_FIXEDPOINT object to the integrator.

### 9.3.2 SUNNonlinearSolver\_FixedPoint functions

The SUNNONLINSOL\_FIXEDPOINT module provides the following constructors for creating a SUNNonlinearSolver object.

```plaintext
SUNNonlinSol_FixedPoint
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

\textbf{SUNNonlinSol\_FixedPointSens}

Call \quad \texttt{NLS = SUNNonlinSol\_FixedPointSens(count, y, m);} \\
Description \quad The function \texttt{SUNNonlinSol\_FixedPointSens} creates a \texttt{SUNNonlinearSolver} object for use with SUNDAIS sensitivity enabled integrators (CVODES and IDAS) to solve nonlinear systems of the form $G(y) = y$. \\
Arguments \quad \texttt{count} (int) the number of vectors in the nonlinear solve. When integrating a system containing $Ns$ sensitivities the value of \texttt{count} is: \\
\begin{itemize}
  \item $Ns+1$ if using a \texttt{simultaneous} corrector approach. \\
  \item $Ns$ if using a \texttt{staggered} corrector approach. 
\end{itemize}
\texttt{y} \quad (N\_Vector) a template for cloning vectors needed within the solver. \\
\texttt{m} \quad (int) the number of acceleration vectors to use. \\
Return value \quad The return value \texttt{NLS} (of type \texttt{SUNNonlinearSolver}) will be a SUNNONLINSOL object if the constructor exits successfully, otherwise \texttt{NLS} will be \texttt{NULL}. \\
F2003 Name \quad This function is callable as \texttt{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 \texttt{SUNNonlinSolSetup}, \texttt{SUNNonlinSolSetLSetupFn}, and \texttt{SUNNonlinSolSetLSolveFn} functions, that are set to \texttt{NULL}. The SUNNONLINSOL\_FIXEDPOINT functions have the same names as those defined by the generic SUNNONLINSOL API with \texttt{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.

\textbf{SUNNonlinSolGetSysFn\_FixedPoint}

Call \quad \texttt{retval = SUNNonlinSolGetSysFn\_FixedPoint(NLS, SysFn);} \\
Description \quad The function \texttt{SUNNonlinSolGetSysFn\_FixedPoint} returns the fixed-point function that defines the nonlinear system. \\
Arguments \quad \texttt{NLS} (\texttt{SUNNonlinearSolver}) a SUNNONLINSOL object \\
\texttt{SysFn} (\texttt{SUNNonlinSolSysFn*}) the function defining the nonlinear system. \\
Return value \quad The return value \texttt{retval} (of type \texttt{int}) should be zero for a successful call, and a negative value for a failure. \\
Notes \quad 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 \texttt{SysFn}. \\
F2003 Name \quad This function is callable as \texttt{FSUNNonlinSolGetSysFn\_FixedPoint} when using the Fortran 2003 interface module.

9.3.3 SUNNonlinearSolver\_FixedPoint Fortran interfaces

The SUNNONLINSOL\_FIXEDPOINT module provides a \texttt{FORTRAN} 2003 module as well as \texttt{FORTRAN} 77 style interface functions for use from \texttt{FORTRAN} applications.
FORTRAN 2003 interface module

The sunnonlinsol_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 sunnonlinsol_fixedpoint_mod, and linking to the library lib sundials_sunnonlinsol_fixedpoint_mod.lib in addition to the C library. For details on where the library and module file sunnonlinsol_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_sunnonlinsol_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.

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

```bash
% 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 CMake-based installation

CMake-based installation provides a platform-independent build system. CMake can generate Unix and Linux Makefiles, as well as KDevelop, Visual Studio, and (Apple) XCode project files from the same configuration file. In addition, CMake also provides a GUI front end and which allows an interactive build and installation process.

The SUNDIALS build process requires CMake version 3.1.3 or higher and a working C compiler. On Unix-like operating systems, it also requires Make (and `curses`, including its development libraries, for the GUI front end to CMake, `ccmake`), while on Windows it requires Visual Studio. CMake is continually adding new features, and the latest version can be downloaded from [http://www.cmake.org](http://www.cmake.org). Build instructions for CMake (only necessary for Unix-like systems) can be found on the CMake web-site. Once CMake is installed, Linux/Unix users will be able to use `ccmake`, while Windows users will be able to use `CMakeSetup`.

As previously noted, when using CMake to configure, build and install SUNDIALS, it is always required to use a separate build directory. While in-source builds are possible, they are explicitly prohibited by the SUNDIALS CMake scripts (one of the reasons being that, unlike autotools, CMake does not provide a `make distclean` procedure and it is therefore difficult to clean-up the source tree after an in-source build). By ensuring a separate build directory, it is an easy task for the user to clean-up all traces of the build by simply removing the build directory. CMake does generate a `make clean` which will remove files generated by the compiler and linker.

#### A.1.1 Configuring, building, and installing on Unix-like systems

The default CMake configuration will build all included solvers and associated examples and will build static and shared libraries. The `instdir` defaults to `/usr/local` and can be changed by setting the `CMAKE_INSTALL_PREFIX` variable. Support for FORTRAN and all other options are disabled.

CMake can be used from the command line with the `cmake` command, or from a `curses`-based GUI by using the `ccmake` command. Examples for using both methods will be presented. For the examples shown it is assumed that there is a top level `sundials` directory with appropriate source, build and install directories:

```
% mkdir (...)sundials/instdir
% mkdir (...)sundials/builddir
% cd (...)sundials/builddir
```

**Building with the GUI**

Using CMake with the GUI follows this general process:

- Select and modify values, run configure (c key)
- New values are denoted with an asterisk
- To set a variable, move the cursor to the variable and press enter
  - If it is a boolean (ON/OFF) it will toggle the value
  - If it is string or file, it will allow editing of the string
For file and directories, the `<tab>` key can be used to complete

- Repeat until all values are set as desired and the generate option is available (`g` key)
- Some variables (advanced variables) are not visible right away
- To see advanced variables, toggle to advanced mode (`t` key)
- To search for a variable press `/` key, and to repeat the search, press the `n` key

To build the default configuration using the GUI, from the `builddir` enter the `ccmake` command and point to the `solverdir`:

```
% ccmake ../solverdir
```

The default configuration screen is shown in Figure A.1.

![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:
Figure A.2: Changing the \textit{instdir} for SUNDIALS and corresponding \texttt{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 \texttt{cmake} command. The following will build the default configuration:

\%
\make -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \ 
\make -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \ 
\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_MINISIZEREL - 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_MINISIZEREL - 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)
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 cannot 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 cannot 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

SUPERLUDIST_ENABLE - Enable SuperLU_DIST support
Default: OFF
Note: See additional information on building with SuperLU_DIST enabled in A.1.4.

SUPERLUDIST_INCLUDE_DIR - Path to SuperLU_DIST header files (typically SRC directory)

SUPERLUDIST_LIBRARY_DIR - Path to SuperLU_DIST installed library files

SUPERLUDIST_LIBRARIES - Semi-colon separated list of libraries needed for SuperLU_DIST

SUPERLUDIST_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
XSDK_INDEX_SIZE - Integer size (bits) used for indices in SUNDIALS, options are: 32 or 64
Default: 32
SUNDIALS equivalent: SUNDIALS_INDEX_SIZE

XSDK_PRECISION - Precision used in SUNDIALS, options are: double, single, or quad
Default: double
SUNDIALS equivalent: SUNDIALS_PRECISION

A.1.3 Configuration examples

The following examples will help demonstrate usage of the CMake configure options.

To configure SUNDIALS using the default C and Fortran compilers, and default mpicc and mpif77 parallel compilers, enable compilation of examples, and install libraries, headers, and example sources under subdirectories of /home/myname/sundials/, use:

```
% cmake \
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \ 
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \ 
> -DMPI_ENABLE=ON \ 
> -DFCMIX_ENABLE=ON \ 
> /home/myname/sundials/solverdir
%
% make install
%
```

To disable installation of the examples, use:

```
% cmake \
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \ 
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \ 
> -DMPI_ENABLE=ON \ 
> -DFCMIX_ENABLE=ON \ 
> -DEXAMPLES_INSTALL=OFF \ 
> /home/myname/sundials/solverdir
%
% make install
%
```

A.1.4 Working with external Libraries

The SUNDIALS suite contains many options to enable implementation flexibility when developing solutions. The following are some notes addressing specific configurations when using the supported third party libraries. When building SUNDIALS as a shared library external libraries any used with SUNDIALS must also be build as a shared library or as a static library compiled with the -fPIC flag.

Building with BLAS

SUNDIALS does not utilize BLAS directly but it may be needed by other external libraries that SUNDIALS can be built with (e.g. LAPACK, PETSc, SuperLU_MT, etc.). To enable BLAS, set the BLAS_ENABLE option to ON. If the directory containing the BLAS library is in the LD_LIBRARY_PATH environment variable, CMake will set the BLAS_LIBRARIES variable accordingly, otherwise CMake will attempt to find the BLAS library in standard system locations. To explicitly tell CMake what libraries to use, the BLAS_LIBRARIES variable can be set to the desired library. Example:

```
% cmake \
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \ 
```
A.1 CMake-based installation

```bash
> -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_F77FUNC_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:

```bash
% 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](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](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](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](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](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](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](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 the 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_LIBDIR. 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 A.1: **SUNDIALS** libraries and header files

<table>
<thead>
<tr>
<th>Library Type</th>
<th>Libraries</th>
<th>Header files</th>
<th>Module files</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SHARED</strong></td>
<td>n/a</td>
<td>sundials/sundials_config.h</td>
<td>nvector/nvector_serial.h</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sundials/sundials_fconfig.h</td>
<td>fnvector_serial_mod.mod</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sundials/sundials_types.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sundials/sundials_math.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sundials/sundials_nvector.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sundials/sundials_fnvector.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sundials/sundials_matrix.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sundials/sundials_linear_solver.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sundials/sundials_iterative.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sundials/sundials_direct.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sundials/sundials_dense.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sundials/sundials_band.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sundials/sundials_nonlinear_solver.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sundials/sundials_version.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sundials/sundials_mpi_types.h</td>
<td></td>
</tr>
</tbody>
</table>

**NVVECTOR_SERIAL**

<table>
<thead>
<tr>
<th>Libraries</th>
<th>Header files</th>
<th>Module files</th>
</tr>
</thead>
<tbody>
<tr>
<td>lib sundials_nvecserial.lib</td>
<td>nvector/nvector_serial.h</td>
<td>fnvector_serial_mod.mod</td>
</tr>
<tr>
<td>lib sundials_fnvecserial_mod.lib</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lib sundials_fnvecserial.a</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NVVECTOR_PARALLEL**

<table>
<thead>
<tr>
<th>Libraries</th>
<th>Header files</th>
<th>Module files</th>
</tr>
</thead>
<tbody>
<tr>
<td>lib sundials_nvecparallel.lib</td>
<td>nvector/nvector_parallel.h</td>
<td>fnvector_parallel_mod.mod</td>
</tr>
<tr>
<td>lib sundials_fnvecparallel.a</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NVVECTOR_OPENMP**

<table>
<thead>
<tr>
<th>Libraries</th>
<th>Header files</th>
<th>Module files</th>
</tr>
</thead>
<tbody>
<tr>
<td>lib sundials_nvecopenmp.lib</td>
<td>nvector/nvector_openmp.h</td>
<td>fnvector_openmp_mod.mod</td>
</tr>
<tr>
<td>lib sundials_fnvecopenmp_mod.lib</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lib sundials_fnvecopenmp.a</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NVVECTOR_OPENMPDEV**

<table>
<thead>
<tr>
<th>Libraries</th>
<th>Header files</th>
<th>Module files</th>
</tr>
</thead>
<tbody>
<tr>
<td>lib sundials_nvecopenmpdev.lib</td>
<td>nvector/nvector_openmpdev.h</td>
<td></td>
</tr>
</tbody>
</table>

**NVVECTOR_PTHREADS**

<table>
<thead>
<tr>
<th>Libraries</th>
<th>Header files</th>
<th>Module files</th>
</tr>
</thead>
<tbody>
<tr>
<td>lib sundials_nvecpthread.lib</td>
<td>nvector/nvector_pthreads.h</td>
<td>fvnvector_pthreads_mod.mod</td>
</tr>
<tr>
<td>lib sundials_fnvecpthread_mod.lib</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lib sundials_fnvecpthread.a</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NVVECTOR_PARHYP**

<table>
<thead>
<tr>
<th>Libraries</th>
<th>Header files</th>
<th>Module files</th>
</tr>
</thead>
<tbody>
<tr>
<td>lib sundials_nvecparhyp.lib</td>
<td>nvector/nvector_parhyp.h</td>
<td></td>
</tr>
</tbody>
</table>

*continued on next page*
### NVector PETSc Libraries
- **Libraries**: `libsundials_nvecpetsc.lib`
- **Header files**: `nvectors/nvecpetsc.h`

### NVector CUDA Libraries
- **Libraries**: `libsundials_nvecCUDA.lib`  
  `libsundials_nvecmpicuda.lib`
- **Header files**: `nvectors/nvecCUDA.h`  
  `nvectors/mpicuda.h`  
  `nvectors/cuda/ThreadPartitioning.hpp`  
  `nvectors/cuda/Vector.hpp`  
  `nvectors/cuda/VectorKernels.cuh`

### NVector RAJA Libraries
- **Libraries**: `libsundials_nvecRAJA.lib`  
  `libsundials_nvecucompiraja.lib`
- **Header files**: `nvectors/nvecRAJA.h`  
  `nvectors/mpiraja.h`  
  `nvectors/RAJA/Vector.hpp`

### NVector TRILINOS Libraries
- **Libraries**: `libsundials_nvectrilinos.lib`
- **Header files**: `nvectors/nvectrilinos.h`  
  `trilinos/SundialsTpetraVectorInterface.hpp`  
  `trilinos/SundialsTpetraVectorKernels.hpp`

### SUNMATRIX Band Libraries
- **Libraries**: `libsundials_sunmatrixband.lib`  
  `libsundials_fsunmatrixband_mod.lib`  
  `libsundials_fsunmatrixband.a`
- **Header files**: `sunmatrix/sunmatrixband.h`  
  `sunmatrix/sunmatrixband.mod.mod`

### SUNMATRIX Dense Libraries
- **Libraries**: `libsundials_sunmatrixdense.lib`  
  `libsundials_fsunmatrixdense_mod.lib`  
  `libsundials_fsunmatrixdense.a`
- **Header files**: `sunmatrix/sunmatrixdense.h`  
  `sunmatrix/sunmatrixdense.mod.mod`

### SUNMATRIX Sparse Libraries
- **Libraries**: `libsundials_sunmatrixsparse.lib`  
  `libsundials_fsunmatrixsparse_mod.lib`  
  `libsundials_fsunmatrixsparse.a`
- **Header files**: `sunmatrix/sunmatrixsparse.h`  
  `sunmatrix/sunmatrixsparse.mod.mod`

### SUNLINSOL Band Libraries
- **Libraries**: `libsundials_sunlinsolband.lib`  
  `libsundials_fsunlinsolband_mod.lib`  
  `libsundials_fsunlinsolband.a`
- **Header files**: `sunlinsol/sunlinsolband.h`  
  `sunlinsol/sunlinsolband.mod.mod`

---

*continued on next page*
## A.4 Installed libraries and exported header files

<table>
<thead>
<tr>
<th>SUNLINSOL_DENSE</th>
<th>Libraries</th>
<th>?libsundials_sunlinsoldense.lib</th>
<th>?libsundials_fsunlinsoldense_mod.lib</th>
<th>?libsundials_fsunlinsoldense.a</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header files</td>
<td>sunlinsol/sunlinsol_dense.h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Module files</td>
<td>fsunlinsol_dense_mod.mod</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Header files</td>
<td>sunlinsol/sunlinsol_klu.h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Module files</td>
<td>fsunlinsol_klu_mod.mod</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Header files</td>
<td>sunlinsol/sunlinsol_lapackband.h</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Header files</td>
<td>sunlinsol/sunlinsol_lapackdense.h</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Header files</td>
<td>sunlinsol/sunlinsol_pcg.h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Module files</td>
<td>fsunlinsol_pcg_mod.mod</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Header files</td>
<td>sunlinsol/sunlinsol_spbcgs.h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Module files</td>
<td>fsunlinsol_spbcgs_mod.mod</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Header files</td>
<td>sunlinsol/sunlinsol_spgmr.h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Module files</td>
<td>fsunlinsol_spgmr_mod.mod</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Header files</td>
<td>sunlinsol/sunlinsol_spfgmr.h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Module files</td>
<td>fsunlinsol_spfgmr_mod.mod</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

continued on next page
<table>
<thead>
<tr>
<th>Package</th>
<th>Libraries</th>
<th>Header files</th>
<th>Module files</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUNLINSOL_SPTFQMR</td>
<td>libsundials_sunlinsolstfqmr.lib</td>
<td>sunlinsol/sunlinsol_stfqmr.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>libsundials_fsunlinsolstfqmr_mod.lib</td>
<td></td>
<td>fsunlinsol_stfqmr_mod.mod</td>
</tr>
<tr>
<td></td>
<td>libsundials_fsunlinsolstfqmr.a</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Header files</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sunlinsol/sunlinsol_stfqmr.h</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Module files</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fsunlinsol_stfqmr_mod.mod</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUNLINSOL_SUPERLUMT</td>
<td>libsundials_sunlinsolsuperlumt.lib</td>
<td>sunlinsol/sunlinsol_superlumt.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>libsundials_fsunlinsolsuperlumt.a</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Header files</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sunlinsol/sunlinsol_superlumt.h</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Module files</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fsunlinsol_superlumt.mod</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUNNONLINSOL_NEWTON</td>
<td>libsundials_sunnlonlinsonewton.lib</td>
<td>sunnonlinsol/sunnlonlinsonewton.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>libsundials_fsunnlonlinsonewton.lib</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>libsundials_fsunnlonlinsonewton.a</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Header files</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sunnonlinsol/sunnlonlinsonewton.h</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Module files</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fsunnlonlinsonewton.mod</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUNNONLINSOL_FIXEDPOINT</td>
<td>libsundials_sunnlonlinsofixedpoint.lib</td>
<td>sunnonlinsol/sunnlonlinsofixedpoint.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>libsundials_fsunnlonlinsofixedpoint.a</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>libsundials_fsunnlonlinsofixedpoint_mod.lib</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Header files</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sunnonlinsol/sunnlonlinsofixedpoint.h</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Module files</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fsunnlonlinsofixedpoint_mod.mod</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVODE</td>
<td>libsundials_cvode.lib</td>
<td>cvode/cvode.h</td>
<td>fcvode_mod.mod</td>
</tr>
<tr>
<td></td>
<td>libsundials_fcvode.a</td>
<td>cvode/cvode_impl.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Header files</strong></td>
<td>cvode/cvode_direct.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>cvode/cvode_direct.h</td>
<td>cvode/cvode_spils.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>cvode/cvode_spils.h</td>
<td>cvode/cvode_bbdpre.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>cvode/cvode_bandpre.h</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Module files</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fcvode_mod.mod</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVODES</td>
<td>libsundials_cvodes.lib</td>
<td>cvodes/cvodes.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>libsundials_fcvodes.a</td>
<td>cvodes/cvodes_impl.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Header files</strong></td>
<td>cvodes/cvodes_direct.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>cvodes/cvodes_direct.h</td>
<td>cvodes/cvodes_spils.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>cvodes/cvodes_spils.h</td>
<td>cvodes/cvodes_bbdpre.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>cvodes/cvodes_bandpre.h</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Module files</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fcvodes_mod.mod</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARKODE</td>
<td>libsundials_arkode.lib</td>
<td>arkode/arkode.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>libsundials_farkode.a</td>
<td>arkode/arkode_impl.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Header files</strong></td>
<td>arkode/arkode_LS.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>arkode/arkode_LS.h</td>
<td>arkode/arkode_bandpre.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>arkode/arkode_bbdpre.h</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Module files</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>arkode/arkode_bandpre.mod</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IDA</td>
<td>libsundials_ida.lib</td>
<td>ida/ida.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>libsundials_fida.a</td>
<td>ida/ida_impl.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Header files</strong></td>
<td>ida/ida_direct.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ida/ida_direct.h</td>
<td>ida/ida_ls.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ida/ida_spils.h</td>
<td>ida/ida_bbdpre.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Module files</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ida/ida_bbdpre.h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IDAS</td>
<td>libsundials_idas.lib</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>continued on next page</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KINSOL</td>
<td>Libraries</td>
<td>Header files</td>
<td>Header files</td>
</tr>
<tr>
<td>--------</td>
<td>-----------</td>
<td>--------------</td>
<td>--------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>idas/idas.h</td>
<td>idas/idas_impl.h</td>
</tr>
<tr>
<td></td>
<td></td>
<td>idas/idas_direct.h</td>
<td>idas/idas_ls.h</td>
</tr>
<tr>
<td></td>
<td></td>
<td>idas/idas_spils.h</td>
<td>idas/idas_bbdpre.h</td>
</tr>
<tr>
<td></td>
<td></td>
<td>kinsol/kinsol.h</td>
<td>kinsol/kinsol_impl.h</td>
</tr>
<tr>
<td></td>
<td></td>
<td>kinsol/kinsol_direct.h</td>
<td>kinsol/kinsol_ls.h</td>
</tr>
<tr>
<td></td>
<td></td>
<td>kinsol/kinsol_spils.h</td>
<td>kinsol/kinsol_bbdpre.h</td>
</tr>
<tr>
<td></td>
<td></td>
<td>libsundials_kinsol.lib</td>
<td>libsundials_kinsol.a</td>
</tr>
</tbody>
</table>
Appendix B

CVODE 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 CVODE input constants

<table>
<thead>
<tr>
<th>CVODE main solver module</th>
</tr>
</thead>
<tbody>
<tr>
<td>CV_ADAMS 1 Adams-Moulton linear multistep method.</td>
</tr>
<tr>
<td>CV_BDF 2 BDF linear multistep method.</td>
</tr>
<tr>
<td>CV_NORMAL 1 Solver returns at specified output time.</td>
</tr>
<tr>
<td>CV_ONE_STEP 2 Solver returns after each successful step.</td>
</tr>
</tbody>
</table>

Iterative linear solver modules

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PREC_NONE 0</td>
<td>No preconditioning</td>
</tr>
<tr>
<td>PREC_LEFT 1</td>
<td>Preconditioning on the left only.</td>
</tr>
<tr>
<td>PREC_RIGHT 2</td>
<td>Preconditioning on the right only.</td>
</tr>
<tr>
<td>PREC_BOTH 3</td>
<td>Preconditioning on both the left and the right.</td>
</tr>
<tr>
<td>MODIFIED_GS 1</td>
<td>Use modified Gram-Schmidt procedure.</td>
</tr>
<tr>
<td>CLASSICAL_GS 2</td>
<td>Use classical Gram-Schmidt procedure.</td>
</tr>
</tbody>
</table>

B.2 CVODE output constants

<table>
<thead>
<tr>
<th>CVODE main solver module</th>
</tr>
</thead>
<tbody>
<tr>
<td>CV_SUCCESS 0 Successful function return.</td>
</tr>
<tr>
<td>CV_TSTOP_RETURN 1 CVode succeeded by reaching the specified stopping point.</td>
</tr>
<tr>
<td>CV_ROOT_RETURN 2 CVode succeeded and found one or more roots.</td>
</tr>
<tr>
<td>CV_WARNING 99 CVode succeeded but an unusual situation occurred.</td>
</tr>
<tr>
<td>CV_TOO_MUCH_WORK -1 The solver took mxstep internal steps but could not reach tout.</td>
</tr>
<tr>
<td>CV_TOO_MUCH_ACC -2 The solver could not satisfy the accuracy demanded by the user for some internal step.</td>
</tr>
<tr>
<td>Constant</td>
</tr>
<tr>
<td>--------------------------</td>
</tr>
<tr>
<td>CV_ERR_FAILURE</td>
</tr>
<tr>
<td>CV_CONV_FAILURE</td>
</tr>
<tr>
<td>CV_INIT_FAIL</td>
</tr>
<tr>
<td>CV_SETUP_FAIL</td>
</tr>
<tr>
<td>CV_SOLVE_FAIL</td>
</tr>
<tr>
<td>CV_FIRST_RHSFUNC_ERR</td>
</tr>
<tr>
<td>CV_REPTD_RHSFUNC_ERR</td>
</tr>
<tr>
<td>CV_UNREC_RHSFUNC_ERR</td>
</tr>
<tr>
<td>CV_RHSFUNC_FAIL</td>
</tr>
<tr>
<td>CV_RT_FUNC_FAIL</td>
</tr>
<tr>
<td>CV_NLS_INIT_FAIL</td>
</tr>
<tr>
<td>CV_NLS_SETUP_FAIL</td>
</tr>
<tr>
<td>CV_CONSTRAINT_FAIL</td>
</tr>
<tr>
<td>CV_MEM_FAIL</td>
</tr>
<tr>
<td>CV_MEM_NULL</td>
</tr>
<tr>
<td>CV_IILL_INPUT</td>
</tr>
<tr>
<td>CV_NO_MALLOC</td>
</tr>
<tr>
<td>CV_BAD_K</td>
</tr>
<tr>
<td>CV_BAD_T</td>
</tr>
<tr>
<td>CV_BAD_DKY</td>
</tr>
<tr>
<td>CV_TOO_CLOSE</td>
</tr>
</tbody>
</table>

**CVLS linear solver interface**

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVLS_SUCCESS</td>
<td>0</td>
<td>Successful function return.</td>
</tr>
<tr>
<td>CVLS_MEM_NULL</td>
<td>-1</td>
<td>The cvode_mem argument was NULL.</td>
</tr>
<tr>
<td>CVLS_LMEM_NULL</td>
<td>-2</td>
<td>The CVLS linear solver has not been initialized.</td>
</tr>
<tr>
<td>CVLS_IILL_INPUT</td>
<td>-3</td>
<td>The CVLS solver is not compatible with the current nvector module.</td>
</tr>
<tr>
<td>CVLS_MEM_FAIL</td>
<td>-4</td>
<td>A memory allocation request failed.</td>
</tr>
<tr>
<td>CVLS_PMEM_NULL</td>
<td>-5</td>
<td>The preconditioner module has not been initialized.</td>
</tr>
<tr>
<td>CVLS_JACFUNC_UNRECVR</td>
<td>-6</td>
<td>The Jacobian function failed in an unrecoverable manner.</td>
</tr>
<tr>
<td>CVLS_JACFUNC_RECVR</td>
<td>-7</td>
<td>The Jacobian function had a recoverable error.</td>
</tr>
<tr>
<td>CVLS_SUNMAT_FAIL</td>
<td>-8</td>
<td>An error occurred with the current sunmatrix module.</td>
</tr>
<tr>
<td>CVLS_SUNLS_FAIL</td>
<td>-9</td>
<td>An error occurred with the current sunlinsol module.</td>
</tr>
</tbody>
</table>

**CVDIAG linear solver module**
<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVDIAG_SUCCESS</td>
<td>0</td>
<td>Successful function return.</td>
</tr>
<tr>
<td>CVDIAG_MEM_NULL</td>
<td>-1</td>
<td>The cvode_mem argument was NULL.</td>
</tr>
<tr>
<td>CVDIAG_LMEM_NULL</td>
<td>-2</td>
<td>The CVDIAG linear solver has not been initialized.</td>
</tr>
<tr>
<td>CVDIAG_ILL_INPUT</td>
<td>-3</td>
<td>The CVDIAG solver is not compatible with the current NVVECTOR module.</td>
</tr>
<tr>
<td>CVDIAG_MEM_FAIL</td>
<td>-4</td>
<td>A memory allocation request failed.</td>
</tr>
<tr>
<td>CVDIAG_INV_FAIL</td>
<td>-5</td>
<td>A diagonal element of the Jacobian was 0.</td>
</tr>
<tr>
<td>CVDIAG_RHSFUNC_UNRECVR</td>
<td>-6</td>
<td>The right-hand side function failed in an unrecoverable manner.</td>
</tr>
<tr>
<td>CVDIAG_RHSFUNC_RECVR</td>
<td>-7</td>
<td>The right-hand side function had a recoverable error.</td>
</tr>
</tbody>
</table>
Bibliography


Index

Adams method, 15

BDF method, 15

BIG_REAL, 28, 111, 117

booleantype, 28

CONSTR_VEC, 98

CV_ADAMS, 34, 68

CV_BAD_DKY, 53

CV_BAD_K, 53

CV_BAD_T, 53

CV_BDF, 34, 68

CV_CONV_FAILURE, 41

CV_ERR_FAILURE, 40

CV_FIRST_RHSFUNC_ERR, 70

CV_FIRST_RHSFUNC_FAIL, 41

CV_IILL_INPUT, 34, 35, 39, 40, 44–46, 48, 52, 69

CV_LINIT_FAIL, 41

CV_LSETUP_FAIL, 41, 72, 80

CV_LSOLVE_FAIL, 41

CV_MEM_FAIL, 34, 47, 61

CV_MEM_NULL, 34–36, 39, 40, 43–48, 52, 53, 55–62,

69

CV_NO_MALLOC, 35, 36, 40, 69

CV_NORMAL, 40

CV_ONE_STEP, 40

CV_REPTD_RHSFUNC_ERR, 41

CV_RHSFUNC_FAIL, 41, 70

CV_ROOT_RETURN, 40

CV_RTFUNC_FAIL, 41, 71

CV_SUCCESS, 34–36, 39, 40, 43–48, 52, 53, 55–62,

69

CV_TOO_CLOSE, 40

CV_TOO MUCH ACC, 40

CV_TOO MUCH WORK, 40

CV_TSTOP_RETURN, 40

CV_UNREC_RHSFUNC_ERR, 41, 70

CV_WARN, 70

CVBandPreconditioner description, 76

optional output, 78–79

usage, 76–77

user-callable functions, 77–78

CVBandPrecGetNumRhsEvals, 78

CVBandPrecGetWorkSpace, 78

CVBandPrecInit, 77

CVBBDPre preconditioner

description, 79

optional output, 83–84

usage, 81–82

user-callable functions, 82–83

user-supplied functions, 79–81

CVBBDPrecGetNumGfnEvals, 84

CVBBDPrecGetWorkSpace, 83

CVBBDPrecInit, 82

CVBBDPrecReInit, 83

CVDIAG linear solver

Jacobian approximation used by, 38

selection of, 38

CVDIAG linear solver interface

memory requirements, 67

optional output, 67–68

use in FCVODE, 96

CVDIAG, 32, 37, 38

CVDIAG_IILL_INPUT, 38

CVDIAG_LMEM_NULL, 67, 68

CVDIAG_MEM_FAIL, 38

CVDIAG_MEM_NULL, 38, 67, 68

CVDIAG_SUCCESS, 38, 67, 68

CVDIAGGetLastFlag, 68

CVDIAGGetNumRhsEvals, 67

CVDIAGGetReturnFlagName, 68

CVDIAGGetWorkSpace, 67

CVDIAGGetLastFlag, 66

CVDIAGGetNumJacEvals, 63

CVDIAGGetNumRhsEvals, 64

CVDIAGGetReturnFlagName, 67

CVDIAGGetWorkspace, 63

CVDIAGGetJacFn, 73

CVDIAGSet JacFn, 49

CVDIAGSetLinearSolver, 38

CVErrHandlerFn, 70

CVEwtFn, 70

CVLS generic linear solver

SUNLINSOL compatibility, 37

CVLS linear solver interface
convergence test, 51
Jacobian approximation used by, 49
Jacobian-vector product approximation used by, 49
memory requirements, 63
optional input, 48–51
optional output, 62–67
preconditioner setup function, 50, 75
preconditioner solve function, 50, 74
CVLS_INPUT, 38, 51, 78, 82
CVLS_JACFUNC_RECVR, 72
CVLS_JACFUNC_UNRECVR, 72
CVLS_LMEM_NULL, 49–51, 63–66, 78, 82, 83
CVLS_MEM_NULL, 38, 49–51, 63–66
CVLS_SUCCESS, 38, 49–51, 63–66, 78
CVLS_SUNLS_FAIL, 38, 50, 51
CVLSJacFn, 71
CVLSJacTimesSetupFn, 74
CVLSJacTimesVecFn, 73
CVLSPrecSetupFn, 75
CVLSPrecSolveFn, 74
CVODE, 1
  motivation for writing in C, 2
  package structure, 23
  relationship to CVODE, PVODE, 1–2
  relationship to VODE, VODPK, 1
CVODE linear solver interfaces, 23–24
  CVDIAG, 38
  CVLS, 37
  selecting one, 37
CVODE linear solvers
  header files, 29
  implementation details, 24
  NVECTOR compatibility, 27
  selecting one, 37
CVode, 32, 40
  cvode/cvode.h, 29
  cvode/cvode_diag.h, 30
  cvode/cvode_ls.h, 29
  CVodeCreate, 34
  CVodeFree, 33, 34
  CVodeGetActualInitStep, 58
  CVodeGetCurrentOrder, 57
  CVodeGetCurrentStep, 58
  CVodeGetCurrentTime, 59
  CVodeGetDky, 52, 53
  CVodeGetErrWeights, 59
  CVodeGetEstLocalErrors, 60
  CVodeGetIntegratorStats, 60
  CVodeGetLastLinFlag, 66
  CVodeGetLastOrder, 57
  CVodeGetLastStep, 58
  CVodeGetLinReturnFlagName, 67
  CVodeGetLinWorkSpace, 63
  CVodeGetNonlinSolvStats, 61
  CVodeGetNumErrTestFails, 57
  CVodeGetNumGEvals, 62
  CVodeGetNumJacEvals, 63
  CVodeGetNumJtimesEvals, 66
  CVodeGetNumLinConvFails, 64
  CVodeGetNumLinIters, 64
  CVodeGetNumLinRhsEvals, 63
  CVodeGetNumLinSolvSetups, 57
  CVodeGetNumNonlinSolvConvFails, 61
  CVodeGetNumNonlinSolvIters, 61
  CVodeGetNumPrecEvals, 65
  CVodeGetNumPrecSolves, 65
  CVodeGetNumRhsEvals, 56
  CVodeGetNumStabLimOrderReds, 59
  CVodeGetNumSteps, 56
  CVodeGetReturnFlagName, 62
  CVodeGetRootInfo, 62
  CVodeGetTolScaleFactor, 59
  CVodeGetWorkSpace, 55
  CVodeInit, 34, 68
  CVodeReInit, 68
  CVodeRootInit, 39
  CVodeSetConstraints, 48
  CVodeSetEpsLin, 51
  CVodeSetErrFile, 43
  CVodeSetErrHandlerFn, 43
  CVodeSetInitStep, 45
  CVodeSetJacFn, 49
  CVodeSetLinearSolver, 32, 37, 71, 173
  CVodeSetMaxConvFails, 47
  CVodeSetMaxErrTestFails, 46
  CVodeSetMaxHnilWarns, 44
  CVodeSetMaxLinIters, 47
  CVodeSetMaxNumSteps, 44
  CVodeSetMaxOrd, 44
  CVodeSetMaxStep, 46
  CVodeSetMaxStepsBetweenJac, 48
  CVodeSetMinStep, 45
  CVodeSetNoInactiveRootWarn, 52
  CVodeSetNonlinConvCoef, 47
  CVodeSetNonlinearSolvConvFails, 39
  CVodeSetNonlinearSolver, 32, 38
  CVodeSetPreconditioner, 50, 51
  CVodeSetRootDirection, 52
  CVodeSetStabLimDet, 45
  CVodeSetStopTime, 46
  CVodeSetUserData, 43
  CVodeSTolerances, 35
  CVodeSVtolerances, 35
  CVodeWFtolerances, 35
INDEX

CVRhsFn, 34, 69
CVRootFn, 71
CVSetJacTimes, 50
CVSpilsGetLastFlag, 66
CVSpilsGetNumConvFails, 64
CVSpilsGetNumJtimesEvals, 66
CVSpilsGetNumJTSetupEvals, 65
CVSpilsGetNumLinIters, 64
CVSpilsGetNumPrecEvals, 65
CVSpilsGetNumPrecSolves, 65
CVSpilsGetNumRhsEvals, 64
CVSpilsGetReturnFlagName, 67
CVSpilsGetWorkspace, 63
CVSpilsJacTimesSetupFn, 74
CVSpilsJacTimesVecFn, 74
CVSpilsPrecSetupFn, 76
CVSpilsPrecSolveFn, 75
CVSpilsSetEpsLin, 51
CVSpilsSetJacTimes, 50
CVSpilsSetLinearSolver, 38
CVSpilsSetPreconditioner, 51

data types
Fortran, 87

eh_data, 70
error control
   order selection, 18–19
   step size selection, 18
error messages, 41
   redirecting, 43
   user-defined handler, 43, 70

FCVBANDSETJAC, 93
FCVBBDINIT, 102
FCVBBDOPT, 102
FCVBBDREINIT, 103
FCVBJAC, 92
FCVBPINIT, 100
FCVBPOPT, 101
FCVCOMMFN, 103
FCVDENSESETJAC, 92
FCVDIAG, 96
FCVDJAC, 92
FCVDKY, 96
FCVDLSSINIT, 92
FCVE, 91
FCVEWSET, 91
FCVFREE, 97
FCVFUN, 89
FCVGETERRWEIGHTS, 97
FCVGETESTLOCALERR, 97
FCVGLOCFN, 103
FCVJTIMES, 94, 103
FCVJSETUP, 94, 103

FCVLSINIT, 92
FCVLSSETJAC, 94, 101, 102
FCVLSSETPREC, 95
FCVMALLOC, 91
FCVMALLOC, 90
FCVODE, 96
FCVODE interface module
   interface to the CVBANDPRE module, 100–101
   interface to the CVBBDPRE module, 101–103
   optional input and output, 97
   rootfinding, 98–100
   usage, 89–97
   user-callable functions, 87–88
   user-supplied functions, 88

fcvode_mod, 85
FCVPSET, 95
FCVPSOL, 95
FCVREINIT, 96
FCVSETIN, 97
FCVSETTRN, 97
FCVSETVIN, 97
FCVSPARSESETJAC, 93
FCVSPILSETJAC, 94
FCVSPILSETPREC, 95
FCVSPILSINIT, 92
FCVSPJAC, 93

fnvector_serial_mod, 123
FSUNBANDLINSOLINIT, 210
FSUNDENSELINSOLINIT, 207
FSUNFIXEDPOINTINIT, 276
FSUNKLUINIT, 219
FSUNKLUREINIT, 219
FSUNKLUSERORDERING, 220
FSUNLAPACKBANDINIT, 215
FSUNLAPACKDENSEINIT, 212
fsunlinsol_band_mod, 210
fsunlinsol_dense_mod, 207
fsunlinsol_klu_mod, 219
fsunlinsol_pcgs_mod, 256
fsunlinsol_spbcgs_mod, 245
fsunlinsol_spgmr_mod, 232
fsunlinsol_sptfqr_mod, 250
FSUNMASSBANDLINSOLINIT, 210
FSUNMASSDENSELINSOLINIT, 208
FSUNMASSKLUINIT, 219
FSUNMASSKLUREINIT, 220
FSUNMASSLUSERORDERING, 220
FSUNMASSLAPACKBANDINIT, 215
FSUNMASSLAPACKDENSEINIT, 212
FSUNMASSPCGINIT, 257
FSUNMASSPCGSETMAXL, 258
FSUNMASSPCGSETPRECTYPE, 257
FSUNMASSSPBCGSINIT, 246
FSUNMASSSPBCGSSETMAXL, 247
FSUNMASSSPBCGSSETPRECTYPE, 246
FSUNMASSSPFGMRSETGSTYPE, 240
FSUNMASSSPFGMRSETMAXRS, 241
FSUNMASSSPFGMRSETPRECTYPE, 240
FSUNMASSSPGMRINIT, 233
FSUNMASSSPGMRSETGSTYPE, 233
FSUNMASSSPGMRSETMAXRS, 234
FSUNMASSSPGMRSETPRECTYPE, 234
FSUNMASSSTFQMRINIT, 251
FSUNMASSSTFQMRSETMAXL, 252
FSUNMASSSTFQMRSETPRECTYPE, 251
FSUNMASSSUPERLUMTINIT, 228
FSUNMASSUPERLUMTSETORDERING, 228
fsunmatrix_band_mod, 183
fsunmatrix_dense_mod, 177
fsunmatrix_sparse_mod, 190
FSUNNEWTONINIT, 272
fsunnonslnsol_fixedpoint_mod, 276
fsunnonslnsol_newton_mod, 272
FSUNPCGINIT, 257
FSUNPCGSETMAXL, 258
FSUNPCGSETPRECTYPE, 257
FSUNSPBCGSINIT, 245
FSUNSPBCGSSETMAXL, 246
FSUNSPBCGSSETPRECTYPE, 246
FSUNSPFGMRINIT, 239
FSUNSPFGMRSETGSTYPE, 240
FSUNSPFGMRSETMAXRS, 241
FSUNSPFGMRSETPRECTYPE, 240
FSUNSPGMRINIT, 232
FSUNSPGMRSETGSTYPE, 233
FSUNSPGMRSETMAXRS, 234
FSUNSPGMRSETPRECTYPE, 233
FSUNSPFTQMRINIT, 251
FSUNSPFTQMRSETMAXL, 252
FSUNSPFTQMRSETPRECTYPE, 251
FSUNSUPERLUMTINIT, 227
FSUNSUPERLUMTSETORDERING, 228

half-bandwidths, 77, 82
header files, 29, 76, 81
HNIL_WARNs, 98

INIT_STEP, 98
IOUT, 97, 99
itask, 32, 40

Jacobian approximation function
   band
      use in FCVODE, 92
dense
      use in FCVODE, 92
diagonal
difference quotient, 38
difference quotient, 49
Jacobian times vector
difference quotient, 49
use in FCVODE, 94
user-supplied, 50
Jacobian-vector product
user-supplied, 73–74
Jacobian-vector setup, 74
sparse
use in FCVODE, 93
user-supplied, 49, 71–73
Jacobian update frequency
optional input, 48
Jacobian-vector product
setup and solve phases, 24

lmm, 34, 68
LSODE, 1

MAX_CONVFAIL, 98
MAX_ERRFAIL, 98
MAX_NITERS, 98
MAX_NSTEPS, 98
MAX_ORD, 98
MAX_STEP, 98
maxord, 44, 68
memory requirements
cvbandpre preconditioner, 78
cvbdbpre preconditioner, 83
cvdiag linear solver interface, 67
cvls linear solver interface, 63
cvode solver, 55

MIN_STEP, 98

N_VCloneVectorArray, 107
N_VCloneVectorArray_OpenMP, 129
N_VCloneVectorArray_OpenMPDEV, 156
N_VCloneVectorArray_ParHyp, 138
N_VCloneVectorArray_Petsc, 142
N_VCloneVectorArray_Pthreads, 134
N_VCloneVectorArray_Serial, 120
N_VCloneVectorArrayEmpty, 107
N_VCloneVectorArrayEmpty_OpenMP, 130
N_VCloneVectorArrayEmpty_OpenMPDEV, 156
N_VCloneVectorArrayEmpty_ParHyp, 138
N_VCloneVectorArrayEmpty_Petsc, 142
N_VCloneVectorArrayEmpty_Pthreads, 134
N_VCloneVectorArrayEmpty_Serial, 120
N_VCopyFromDevice_Cuda, 148
N_VCopyFromDevice_OpenMPDEV, 157
N_VCopyFromDevice_Raja, 152
N_VCopyToDevice_Cuda, 148
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>N_VCopyToDevice_OpenMPDEV</td>
<td>156</td>
</tr>
<tr>
<td>N_VCopyToDevice_Raja</td>
<td>152</td>
</tr>
<tr>
<td>N_VDestroyVectorArray</td>
<td>107</td>
</tr>
<tr>
<td>N_VDestroyVectorArray_OpenMP</td>
<td>130</td>
</tr>
<tr>
<td>N_VDestroyVectorArray_OpenMPDEV</td>
<td>156</td>
</tr>
<tr>
<td>N_VDestroyVectorArray_ParHyp</td>
<td>125</td>
</tr>
<tr>
<td>N_VDestroyVectorArray_Petsc</td>
<td>138</td>
</tr>
<tr>
<td>N_VDestroyVectorArray_PThreads</td>
<td>142</td>
</tr>
<tr>
<td>N_VDestroyVectorArray_Serial</td>
<td>134</td>
</tr>
<tr>
<td>N_VDestroyVectorArray_Serial_OpenMP</td>
<td>120</td>
</tr>
<tr>
<td>N_Vector, 29, 105</td>
<td></td>
</tr>
<tr>
<td>N_VEnableConstVectorArray_Cuda</td>
<td>149</td>
</tr>
<tr>
<td>N_VEnableConstVectorArray_ManyVector</td>
<td>164</td>
</tr>
<tr>
<td>N_VEnableConstVectorArray_OpenMP</td>
<td>131</td>
</tr>
<tr>
<td>N_VEnableConstVectorArray_OpenMPDEV</td>
<td>158</td>
</tr>
<tr>
<td>N_VEnableConstVectorArray_Parallel</td>
<td>126</td>
</tr>
<tr>
<td>N_VEnableConstVectorArray_ParHyp</td>
<td>136</td>
</tr>
<tr>
<td>N_VEnableConstVectorArray_Petsc</td>
<td>143</td>
</tr>
<tr>
<td>N_VEnableConstVectorArray_Raja</td>
<td>153</td>
</tr>
<tr>
<td>N_VEnableConstVectorArray_Serial_Petsc</td>
<td>122</td>
</tr>
<tr>
<td>N_VEnableDotProdMulti_Cuda</td>
<td>149</td>
</tr>
<tr>
<td>N_VEnableDotProdMulti_ManyVector</td>
<td>163</td>
</tr>
<tr>
<td>N_VEnableDotProdMulti_OpenMP</td>
<td>131</td>
</tr>
<tr>
<td>N_VEnableDotProdMulti_OpenMPDEV</td>
<td>157</td>
</tr>
<tr>
<td>N_VEnableDotProdMulti_ParHyp</td>
<td>136</td>
</tr>
<tr>
<td>N_VEnableDotProdMulti_Petsc</td>
<td>143</td>
</tr>
<tr>
<td>N_VEnableDotProdMulti_Pthreads</td>
<td>135</td>
</tr>
<tr>
<td>N_VEnableDotProdMulti_Serial</td>
<td>121</td>
</tr>
<tr>
<td>N_VEnableFusedOps_Cuda</td>
<td>148</td>
</tr>
<tr>
<td>N_VEnableFusedOps_ManyVector</td>
<td>163</td>
</tr>
<tr>
<td>N_VEnableFusedOps_OpenMP</td>
<td>130</td>
</tr>
<tr>
<td>N_VEnableFusedOps_OpenMPDEV</td>
<td>157</td>
</tr>
<tr>
<td>N_VEnableFusedOps_ParHyp</td>
<td>139</td>
</tr>
<tr>
<td>N_VEnableFusedOps_Petsc</td>
<td>142</td>
</tr>
<tr>
<td>N_VEnableFusedOps_Pthreads</td>
<td>135</td>
</tr>
<tr>
<td>N_VEnableFusedOps_Raja</td>
<td>153</td>
</tr>
<tr>
<td>N_VEnableFusedOps_Serial</td>
<td>121</td>
</tr>
<tr>
<td>N_VEnableLinearCombination_Cuda</td>
<td>148</td>
</tr>
<tr>
<td>N_VEnableLinearCombination_ManyVector</td>
<td>163</td>
</tr>
<tr>
<td>N_VEnableLinearCombination_OpenMP</td>
<td>130</td>
</tr>
<tr>
<td>N_VEnableLinearCombination_OpenMPDEV</td>
<td>157</td>
</tr>
<tr>
<td>N_VEnableLinearCombination_Parallel</td>
<td>126</td>
</tr>
<tr>
<td>N_VEnableLinearCombination_ParHyp</td>
<td>139</td>
</tr>
<tr>
<td>N_VEnableLinearCombination_Petsc</td>
<td>143</td>
</tr>
<tr>
<td>N_VEnableLinearCombination_Pthreads</td>
<td>135</td>
</tr>
<tr>
<td>N_VEnableLinearCombination_Raja</td>
<td>153</td>
</tr>
<tr>
<td>N_VEnableLinearCombination_Raja_OpenMP</td>
<td>150</td>
</tr>
<tr>
<td>N_VEnableLinearCombination_Serial</td>
<td>121</td>
</tr>
<tr>
<td>N_VEnableLinearCombinationVectorArray_Cuda</td>
<td>149</td>
</tr>
<tr>
<td>N_VEnableLinearCombinationVectorArray_ManyVector</td>
<td>163</td>
</tr>
<tr>
<td>N_VEnableLinearCombinationVectorArray_OpenMP</td>
<td>131</td>
</tr>
<tr>
<td>N_VEnableLinearCombinationVectorArray_OpenMPDEV</td>
<td>157</td>
</tr>
<tr>
<td>N_VEnableLinearCombinationVectorArray_Parallel</td>
<td>127</td>
</tr>
<tr>
<td>N_VEnableLinearCombinationVectorArray_ParHyp</td>
<td>140</td>
</tr>
<tr>
<td>N_VEnableLinearCombinationVectorArray_Petsc</td>
<td>144</td>
</tr>
<tr>
<td>N_VEnableLinearCombinationVectorArray_Pthreads</td>
<td>136</td>
</tr>
<tr>
<td>N_VEnableLinearCombinationVectorArray_Raja</td>
<td>154</td>
</tr>
<tr>
<td>N_VEnableLinearCombinationVectorArray_Serial</td>
<td>122</td>
</tr>
<tr>
<td>N_VEnableLinearCombinationVectorArray_Serial_OpenMPDEV</td>
<td>158</td>
</tr>
<tr>
<td>N_VEnableLinearCombinationVectorArray_Serial_OpenMP</td>
<td>131</td>
</tr>
<tr>
<td>N_VEnableLinearCombinationVectorArray_Serial_OpenMPDEV</td>
<td>158</td>
</tr>
<tr>
<td>N_VEnableLinearCombinationVectorArray_Serial_Petsc</td>
<td>143</td>
</tr>
<tr>
<td>N_VEnableLinearCombinationVectorArray_Serial_Pthreads</td>
<td>136</td>
</tr>
<tr>
<td>N_VEnableLinearCombinationVectorArray_Serial_Raja</td>
<td>154</td>
</tr>
<tr>
<td>N_VEnableLinearCombinationVectorArray_Serial_OpenMPDEV_OpenMP</td>
<td>152</td>
</tr>
<tr>
<td>N_VEnableLinearCombinationVectorArray_Serial_OpenMPDEV_OpenMP_Petsc</td>
<td>144</td>
</tr>
<tr>
<td>N_VEnableLinearCombinationVectorArray_Serial_OpenMPDEV_OpenMP_Pthreads</td>
<td>136</td>
</tr>
<tr>
<td>N_VEnableLinearCombinationVectorArray_Serial_Petsc_OpenMPDEV_Petsc</td>
<td>143</td>
</tr>
<tr>
<td>N_VEnableLinearCombinationVectorArray_Serial_Petsc_OpenMPDEV_Pthreads</td>
<td>136</td>
</tr>
<tr>
<td>N_VEnableLinearCombinationVectorArray_Serial_Raja_OpenMPDEV_Petsc</td>
<td>143</td>
</tr>
<tr>
<td>N_VEnableLinearCombinationVectorArray_Serial_Raja_OpenMPDEV_Pthreads</td>
<td>136</td>
</tr>
<tr>
<td>N_VEnableLinearCombinationVectorArray_Serial_Raja_OpenMPDEV_Petsc</td>
<td>143</td>
</tr>
</tbody>
</table>
INDEX

NV_Ith_OMP, 129
NV_Ith_P, 124
NV_Ith_PT, 133
NV_Ith_S, 119
NV_LENGTH_OMP, 128
NV_LENGTH_OMPDEV, 155
NV_LENGTH_PT, 133
NV_LENGTH_S, 119
NV_LENGTH_P, 124
NV_LENGTH_OMPDEV, 128
NV_LENGTH_PT, 133
NV_LENGTH_S, 119
NV_LOCLENGTH_P, 124
NV_NUM_THREADS_OMP, 128
NV_NUM_THREADS_PT, 133
NV_NUM_THREADS_S, 119
NV_NUM_THREADS_P, 124
NV_NUM_THREADS_OMPDEV, 155
NV_NUM_THREADS_S, 119
NV_OWNPDATA_OMP, 128
NV_OWNPDATA_OMPDEV, 155
NV_OWNPDATA_P, 124
NV_OWNPDATA_S, 119
NV_OWNPDATA_PT, 133
NV_OWNPDATA_S, 119
NVECTOR module, 105
nvector_openmp_mod, 132
nvector_pthreads_mod, 137

optional input
  generic linear solver interface, 48–51
  iterative linear solver, 50–51
  matrix-based linear solver, 49
  matrix-free linear solver, 49–50
  rootfinding, 52
  solver, 43–48

optional output
  band-block-diagonal preconditioner, 83–84
  banded preconditioner, 78–79
  diagonal linear solver interface, 67–68
  generic linear solver interface, 62–67
  interpolated solution, 52
  solver, 55–62
  version, 53–55

output mode, 19, 40

portability, 28
  Fortran, 86
Preconditioner setup routine
  use in FCVODE, 94
Preconditioner solve routine
  use in FCVODE, 94
Preconditioner update frequency
  optional input, 48
preconditioning
  advice on, 19, 24
  band-block diagonal, 79
  banded, 76
  setup and solve phases, 24
  user-supplied, 50–51, 74, 75
PVODE, 1

RCONST, 28
realtype, 28
SUNSparseMatrix_Data, 189
SUNSparseMatrix_IndexPointers, 189
SUNSparseMatrix_IndexValues, 189
SUNSparseMatrix_NNZ, 73, 189
SUNSparseMatrix_NP, 189
SUNSparseMatrix_Print, 188
SUNSparseMatrix_Realloc, 188
SUNSparseMatrix_Reallocate, 188
SUNSparseMatrix_Rows, 188
SUNSparseMatrix_SparseType, 189
tolerances, 16, 36, 70

UNIT_ROUNDOFF, 28
User main program
  CVBANDPRE usage, 76
  CVBBDPRE usage, 81
  FCVBBD usage, 102
  FCVBP usage, 100
  FCVODE usage, 89
  IVP solution, 30
user_data, 43, 70–72, 80

VODE, 1
VODPK, 1

weighted root-mean-square norm, 16