User Documentation for CVODES v4.1.0
(SUNDIALS v4.1.0)

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

February 12, 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.
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 ..................................... 13
    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
  2.5 Pure quadrature integration .................................... 22
  2.6 Forward sensitivity analysis ................................... 22
    2.6.1 Forward sensitivity methods .............................. 23
    2.6.2 Selection of the absolute tolerances for sensitivity variables 24
    2.6.3 Evaluation of the sensitivity right-hand side ............ 24
    2.6.4 Quadratures depending on forward sensitivities .......... 25
  2.7 Adjoint sensitivity analysis .................................... 25
    2.7.1 Checkpointing scheme .................................... 27
  2.8 Second-order sensitivity analysis .............................. 28

3 Code Organization 31
  3.1 SUNDIALS organization ......................................... 31
  3.2 CVODES organization ........................................... 31

4 Using CVODES for IVP Solution 35
  4.1 Access to library and header files .......................... 35
  4.2 Data Types .................................................. 36
    4.2.1 Floating point types .................................... 36
    4.2.2 Integer types used for vector and matrix indices ..... 36
  4.3 Header files ................................................ 37
  4.4 A skeleton of the user’s main program ...................... 38
  4.5 User-callable functions ....................................... 41
    4.5.1 CVODES initialization and deallocation functions .... 42
    4.5.2 CVODES tolerance specification functions .......... 43
    4.5.3 Linear solver interface functions ..................... 44
4.5.4 Nonlinear solver interface function .............................................. 46
4.5.5 Rootfinding initialization function ............................................... 47
4.5.6 CVODES solver function ............................................................. 47
4.5.7 Optional input functions ............................................................. 49
  4.5.7.1 Main solver optional input functions ........................................ 49
  4.5.7.2 Linear solver interface optional input functions ......................... 56
  4.5.7.3 Rootfinding optional input functions ....................................... 59
4.5.8 Interpolated output function ....................................................... 60
4.5.9 Optional output functions ........................................................... 60
  4.5.9.1 SUNDIALS version information ............................................... 61
  4.5.9.2 Main solver optional output functions ..................................... 61
  4.5.9.3 Rootfinding optional output functions ..................................... 68
  4.5.9.4 cvls linear solver interface optional output functions .............. 69
  4.5.9.5 Diagonal linear solver interface optional output functions .......... 73
4.5.10 CVODES reinitialization function .............................................. 74

4.6 User-supplied functions ............................................................... 75
  4.6.1 ODE right-hand side ................................................................. 75
  4.6.2 Error message handler function ............................................... 76
  4.6.3 Error weight function ............................................................... 77
  4.6.4 Rootfinding function ............................................................... 77
  4.6.5 Jacobian construction (matrix-based linear solvers) ...................... 77
  4.6.6 Jacobian-vector product (matrix-free linear solvers) ................. 79
  4.6.7 Jacobian-vector product setup (matrix-free linear solvers) ........... 80
  4.6.8 Preconditioner solve (iterative linear solvers) ......................... 81
  4.6.9 Preconditioner setup (iterative linear solvers) .......................... 81

4.7 Integration of pure quadrature equations ........................................ 82
  4.7.1 Quadrature initialization and deallocation functions ................... 84
  4.7.2 CVODES solver function ............................................................ 85
  4.7.3 Quadrature extraction functions ............................................... 85
  4.7.4 Optional inputs for quadrature integration .................................. 86
  4.7.5 Optional outputs for quadrature integration ................................ 87
  4.7.6 User-supplied function for quadrature integration ...................... 88

4.8 Preconditioner modules ............................................................... 89
  4.8.1 A serial banded preconditioner module ....................................... 89
  4.8.2 A parallel band-block-diagonal preconditioner module .................. 91

5 Using CVODES for Forward Sensitivity Analysis .................................. 97
  5.1 A skeleton of the user’s main program ......................................... 97
  5.2 User-callable routines for forward sensitivity analysis .................... 100
    5.2.1 Forward sensitivity initialization and deallocation functions .......... 100
    5.2.2 Forward sensitivity tolerance specification functions ............... 103
    5.2.3 Forward sensitivity nonlinear solver interface functions ............ 105
    5.2.4 CVODES solver function ......................................................... 106
    5.2.5 Forward sensitivity extraction functions .................................. 106
    5.2.6 Optional inputs for forward sensitivity analysis ....................... 108
    5.2.7 Optional outputs for forward sensitivity analysis ...................... 110
  5.3 User-supplied routines for forward sensitivity analysis .................... 114
    5.3.1 Sensitivity equations right-hand side (all at once) .................... 114
    5.3.2 Sensitivity equations right-hand side (one at a time) .................. 115
  5.4 Integration of quadrature equations depending on forward sensitivities .... 116
    5.4.1 Sensitivity-dependent quadrature initialization and deallocation .... 118
    5.4.2 CVODES solver function ......................................................... 119
    5.4.3 Sensitivity-dependent quadrature extraction functions ............... 119
    5.4.4 Optional inputs for sensitivity-dependent quadrature integration .... 121
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4.5 Optional outputs for sensitivity-dependent quadrature integration</td>
<td>123</td>
</tr>
<tr>
<td>5.4.6 User-supplied function for sensitivity-dependent quadrature integration</td>
<td>124</td>
</tr>
<tr>
<td>5.5 Note on using partial error control</td>
<td>125</td>
</tr>
<tr>
<td>6 Using CVODES for Adjoint Sensitivity Analysis</td>
<td>127</td>
</tr>
<tr>
<td>6.1 A skeleton of the user’s main program</td>
<td>127</td>
</tr>
<tr>
<td>6.2 User-callable functions for adjoint sensitivity analysis</td>
<td>131</td>
</tr>
<tr>
<td>6.2.1 Adjoint sensitivity allocation and deallocation functions</td>
<td>131</td>
</tr>
<tr>
<td>6.2.2 Forward integration function</td>
<td>132</td>
</tr>
<tr>
<td>6.2.3 Backward problem initialization functions</td>
<td>133</td>
</tr>
<tr>
<td>6.2.4 Tolerance specification functions for backward problem</td>
<td>135</td>
</tr>
<tr>
<td>6.2.5 Linear solver initialization functions for backward problem</td>
<td>136</td>
</tr>
<tr>
<td>6.2.6 Backward integration function</td>
<td>137</td>
</tr>
<tr>
<td>6.2.7 Adjoint sensitivity optional input</td>
<td>139</td>
</tr>
<tr>
<td>6.2.8 Optional input functions for the backward problem</td>
<td>139</td>
</tr>
<tr>
<td>6.2.8.1 Main solver optional input functions</td>
<td>139</td>
</tr>
<tr>
<td>6.2.8.2 Linear solver interface optional input functions</td>
<td>140</td>
</tr>
<tr>
<td>6.2.9 Optional output functions for the backward problem</td>
<td>143</td>
</tr>
<tr>
<td>6.2.10 Backward integration of quadrature equations</td>
<td>144</td>
</tr>
<tr>
<td>6.2.10.1 Backward quadrature initialization functions</td>
<td>144</td>
</tr>
<tr>
<td>6.2.10.2 Backward quadrature extraction function</td>
<td>146</td>
</tr>
<tr>
<td>6.2.10.3 Optional input/output functions for backward quadrature integration</td>
<td>146</td>
</tr>
<tr>
<td>6.3 User-supplied functions for adjoint sensitivity analysis</td>
<td>147</td>
</tr>
<tr>
<td>6.3.1 ODE right-hand side for the backward problem</td>
<td>147</td>
</tr>
<tr>
<td>6.3.2 ODE right-hand side for the backward problem depending on the forward sensitivities</td>
<td>147</td>
</tr>
<tr>
<td>6.3.3 Quadrature right-hand side for the backward problem</td>
<td>148</td>
</tr>
<tr>
<td>6.3.4 Sensitivity-dependent quadrature right-hand side for the backward problem</td>
<td>149</td>
</tr>
<tr>
<td>6.3.5 Jacobian construction for the backward problem (matrix-based linear solvers)</td>
<td>150</td>
</tr>
<tr>
<td>6.3.6 Jacobian-vector product for the backward problem (matrix-free linear solvers)</td>
<td>151</td>
</tr>
<tr>
<td>6.3.7 Jacobian-vector product setup for the backward problem (matrix-free linear solvers)</td>
<td>153</td>
</tr>
<tr>
<td>6.3.8 Preconditioner solve for the backward problem (iterative linear solvers)</td>
<td>154</td>
</tr>
<tr>
<td>6.3.9 Preconditioner setup for the backward problem (iterative linear solvers)</td>
<td>155</td>
</tr>
<tr>
<td>6.4 Using CVODES preconditioner modules for the backward problem</td>
<td>157</td>
</tr>
<tr>
<td>6.4.1 Using the banded preconditioner CVBANDPRE</td>
<td>157</td>
</tr>
<tr>
<td>6.4.2 Using the band-block-diagonal preconditioner CVBBDPRE</td>
<td>158</td>
</tr>
<tr>
<td>6.4.2.1 Initialization of CVBBDPRE</td>
<td>158</td>
</tr>
<tr>
<td>6.4.2.2 User-supplied functions for CVBBDPRE</td>
<td>159</td>
</tr>
<tr>
<td>7 Description of the NVVECTOR module</td>
<td>161</td>
</tr>
<tr>
<td>7.1 NVVECTOR functions used by CVODES</td>
<td>170</td>
</tr>
<tr>
<td>7.2 The NVVECTOR_SERIAL implementation</td>
<td>171</td>
</tr>
<tr>
<td>7.2.1 NVVECTOR_SERIAL accessor macros</td>
<td>171</td>
</tr>
<tr>
<td>7.2.2 NVVECTOR_SERIAL functions</td>
<td>172</td>
</tr>
<tr>
<td>7.2.3 NVVECTOR_SERIAL Fortran interfaces</td>
<td>175</td>
</tr>
<tr>
<td>7.3 The NVVECTOR_PARALLEL implementation</td>
<td>176</td>
</tr>
<tr>
<td>7.3.1 NVVECTOR_PARALLEL accessor macros</td>
<td>176</td>
</tr>
<tr>
<td>7.3.2 NVVECTOR_PARALLEL functions</td>
<td>177</td>
</tr>
<tr>
<td>7.3.3 NVVECTOR_PARALLEL Fortran interfaces</td>
<td>180</td>
</tr>
<tr>
<td>7.4 The NVVECTOR_OPENMP implementation</td>
<td>180</td>
</tr>
<tr>
<td>7.4.1 NVVECTOR_OPENMP accessor macros</td>
<td>181</td>
</tr>
<tr>
<td>7.4.2 NVVECTOR_OPENMP functions</td>
<td>181</td>
</tr>
<tr>
<td>7.4.3 NVVECTOR_OPENMP Fortran interfaces</td>
<td>185</td>
</tr>
</tbody>
</table>
### 7.5 The NVECTOR_PTHREADS implementation
- 7.5.1 NVECTOR_PTHREADS accessor macros
- 7.5.2 NVECTOR_PTHREADS functions
- 7.5.3 NVECTOR_PTHREADS Fortran interfaces

### 7.6 The NVECTOR_PARHYPP implementation
- 7.6.1 NVECTOR_PARHYPP functions

### 7.7 The NVECTOR_PETSC implementation
- 7.7.1 NVECTOR_PETSC functions

### 7.8 The NVECTOR_CUDA implementation
- 7.8.1 NVECTOR_CUDA functions

### 7.9 The NVECTOR_RAJA implementation
- 7.9.1 NVECTOR_RAJA functions

### 7.10 The NVECTOR_OPENMPDEV implementation
- 7.10.1 NVECTOR_OPENMPDEV accessor macros
- 7.10.2 NVECTOR_OPENMPDEV functions

### 7.11 The NVECTOR_TRILINOS implementation

### 7.12 NVECTOR Examples

### 8 Description of the SUNMatrix module
- 8.1 SUNMatrix functions used by CVODES
- 8.2 The SUNMatrix.Dense implementation
  - 8.2.1 SUNMatrix.Dense accessor macros
  - 8.2.2 SUNMatrix.Dense functions
  - 8.2.3 SUNMatrix.Dense Fortran interfaces
- 8.3 The SUNMatrix.Band implementation
  - 8.3.1 SUNMatrix.Band accessor macros
  - 8.3.2 SUNMatrix.Band functions
  - 8.3.3 SUNMatrix.Band Fortran interfaces
- 8.4 The SUNMatrix.Sparse implementation
  - 8.4.1 SUNMatrix.Sparse accessor macros
  - 8.4.2 SUNMatrix.Sparse functions
  - 8.4.3 SUNMatrix.Sparse Fortran interfaces

### 9 Description of the SUNLinearSolver module
- 9.1 The SUNLinearSolver API
  - 9.1.1 SUNLinearSolver core functions
  - 9.1.2 SUNLinearSolver set functions
  - 9.1.3 SUNLinearSolver get functions
  - 9.1.4 Functions provided by SUNDIALS packages
  - 9.1.5 SUNLinearSolver return codes
  - 9.1.6 The generic SUNLinearSolver module
- 9.2 Compatibility of SUNLinearSolver modules
- 9.3 Implementing a custom SUNLinearSolver module
  - 9.3.1 Intended use cases
- 9.4 CVODES SUNLinearSolver interface
  - 9.4.1 Lagged matrix information
  - 9.4.2 Iterative linear solver tolerance
- 9.5 The SUNLinearSolver_Dense implementation
  - 9.5.1 SUNLinearSolver_Dense description
  - 9.5.2 SUNLinearSolver_Dense functions
  - 9.5.3 SUNLinearSolver_Dense Fortran interfaces
  - 9.5.4 SUNLinearSolver_Dense content
- 9.6 The SUNLinearSolver_Band implementation
  - 9.6.1 SUNLinearSolver_Band description
10 Description of the SUNNonlinearSolver module

10.1 The SUNNonlinearSolver API
10.1.1 SUNNonlinearSolver core functions
10.1.2 SUNNonlinearSolver set functions
10.1.3 SUNNonlinearSolver get functions
10.1.4 Functions provided by SUNDIALS integrators
10.1.5 SUNNonlinearSolver return codes
10.1.6 The generic SUNNonlinearSolver module
10.1.7 Usage with sensitivity enabled integrators
10.1.8 Implementing a Custom SUNNonlinearSolver Module

10.2 The SUNNonlinearSolver Newton implementation
10.2.1 SUNNonlinearSolver Newton description
10.2.2 SUNNonlinearSolver Newton functions
10.2.3 SUNNonlinearSolver Newton Fortran interfaces
10.2.4 SUNNonlinearSolver Newton content

10.3 The SUNNonlinearSolver FixedPoint implementation
10.3.1 SUNNonlinearSolver FixedPoint description
10.3.2 SUNNonlinearSolver FixedPoint functions
10.3.3 SUNNonlinearSolver FixedPoint Fortran interfaces
10.3.4 SUNNonlinearSolver FixedPoint content

A SUNDIALS Package Installation Procedure
A.1 CMake-based installation
A.1.1 Configuring, building, and installing on Unix-like systems
A.1.2 Configuration options (Unix/Linux)
A.1.3 Configuration examples
A.1.4 Working with external Libraries
A.1.5 Testing the build and installation
A.2 Building and Running Examples
A.3 Configuring, building, and installing on Windows
A.4 Installed libraries and exported header files

B CVODES Constants
B.1 CVODES input constants
B.2 CVODES output constants

Bibliography

Index
# List of Tables

4.1 **SUNDIALS** linear solver interfaces and vector implementations that can be used for each. 41
4.2 Optional inputs for **CVODES** and **CVLS** ........................................... 50
4.3 Optional outputs from **CVODES**, **CVLS**, and **CVDIAG** .......................... 62

5.1 Forward sensitivity optional inputs .......................................................... 108
5.2 Forward sensitivity optional outputs ......................................................... 110

7.1 Vector Identifications associated with vector kernels supplied with SUNDIALS. 163
7.2 Description of the NVECTOR operations .................................................. 164
7.3 Description of the NVECTOR fused operations ......................................... 167
7.4 Description of the NVECTOR vector array operations ............................... 168
7.5 List of vector functions usage by CVODES code modules ......................... 217

8.1 Identifiers associated with matrix kernels supplied with SUNDIALS. .......... 220
8.2 Description of the SUNMatrix operations ............................................... 220
8.3 SUNDIALS matrix interfaces and vector implementations that can be used for each. 221
8.4 List of matrix functions usage by CVODES code modules ......................... 222

9.1 Description of the SUNLinearSolver error codes ...................................... 248
9.2 SUNDIALS matrix-based linear solvers and matrix implementations that can be used for each. .............................................................. 249
9.3 List of linear solver function usage in the CVLS interface ......................... 252

10.1 Description of the SUNNonlinearSolver return codes ........................... 311

A.1 SUNDIALS libraries and header files ....................................................... 339
## List of Figures

2.1 Illustration of the checkpointing algorithm for generation of the forward solution during the integration of the adjoint system ................................. 27
3.1 High-level diagram of the SUNDIALS suite ............................................ 32
3.2 Organization of the SUNDIALS suite ...................................................... 33
3.3 Overall structure diagram of the CVODES package ................................. 34
8.1 Diagram of the storage for a SUNMATRIX_BAND object ......................... 228
8.2 Diagram of the storage for a compressed-sparse-column matrix ................. 235
A.1 Initial ccmake configuration screen ...................................................... 325
A.2 Changing the instdir ............................................................................. 326
Chapter 1

Introduction

cvodes [42] is part of a software family called sundials: SUite of Nonlinear and DIfferential/ALgebraic equation Solvers [26]. This suite consists of cvode, arkode, kinsol, and ida, and variants of these with sensitivity analysis capabilities. cvodes is a solver for stiff and nonstiff initial value problems (IVPs) for systems of ordinary differential equation (ODEs). In addition to solving stiff and nonstiff ODE systems, cvodes has sensitivity analysis capabilities, using either the forward or the adjoint methods.

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 [5] and vodpk [8]. 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 [38]. 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 [6]. The capabilities of both vode and vodpk have been combined in the C-language package cvode [13].

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) [41], FGMRES (Flexible Generalized Minimum RESidual) [40], Bi-CGStab (Bi-Conjugate Gradient Stabilized) [44], TFQMR (Transpose-Free Quasi-Minimal Residual) [20], and PCG (Preconditioned Conjugate Gradient) [21] 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 [10], the parallel variant of CVODE.

CVODES is written with a functionality that is a superset of that of the pair CVODE/PVODE. Sensitivity analysis capabilities, both forward and adjoint, have been added to the main integrator. Enabling forward sensitivity computations in CVODES will result in the code integrating the so-called sensitivity equations simultaneously with the original IVP, yielding both the solution and its sensitivity with respect to parameters in the model. Adjoint sensitivity analysis, most useful when the gradients of relatively few functionals of the solution with respect to many parameters are sought, involves integration of the original IVP forward in time followed by the integration of the so-called adjoint equations backward in time. CVODES provides the infrastructure needed to integrate any final-condition ODE dependent on the solution of the original IVP (in particular the adjoint system).

Development of CVODES was concurrent with a redesign of the vector operations module across the SUNDIALS suite. The key feature of the NVcctor module is that it is written in terms of abstract vector operations with the actual vector functions attached by a particular implementation (such as serial or parallel) of NVcctor. This allows writing the SUNDIALS solvers in a manner independent of the actual NVcctor implementation (which can be user-supplied), as well as allowing more than one NVектор module to be linked into an executable file. SUNDIALS (and thus CVODES) is supplied with serial, MPI-parallel, and both OpenMP and Pthreads thread-parallel NVECTOR implementations.

There were several motivations for choosing the C language for CVODE, and later for CVODES. 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. Finally, we prefer C over C++ for CVODES 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 v4.1.0

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

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

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

The implementation header file cvodes_impl.h is no longer installed. This means users who are directly manipulating the CVodeMem structure will need to update their code to use CVODES’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 CVODES library, libsundials_cvodes.

Changes in v4.0.1

No changes were made in this release.
1.2 Changes from previous versions

Changes in v4.0.0

CVODES’ 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 SUNLINSOL 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 SUNLINSOL implementations are included in Chapter 9. All CVODES 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 routine names may still be used; these will be deprecated in future releases, so we recommend that users migrate to the new names soon.

The names of all constructor routines for SUNDIALS-provided SUNLINSOL implementations have been updated to follow the naming convention SUNLinSol_• where • is the name of the linear solver. The new names are SUNLinSol_Band, SUNLinSol_Dense, SUNLinSol_KLU, SUNLinSol_LapackBand, SUNLinSol_LapackDense, SUNLinSol_PCG, SUNLinSol_SPBCGS, 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 CVODES example programs and the standalone linear solver examples have been updated to use the new naming convention.

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

SUNDIALS integrators have been updated to utilize generic nonlinear solver modules defined through the SUNNONLINSOL API. This API will ease the addition of new nonlinear solver options and allow for external or user-supplied nonlinear solvers. The SUNNONLINSOL API and SUNDIALS provided modules are described in Chapter 10 and follow the same object oriented design and implementation used by the NVECTOR, SUNMATRIX, and SUNLINSOL modules. Currently two SUNNONLINSOL implementations are provided, SUNNONLINSOL_NEWTON and SUNNONLINSOL_FIXEDPOINT. These replicate the previous integrator specific implementations of a Newton iteration and a fixed-point iteration (previously referred to as a functional iteration), respectively. Note the SUNNONLINSOL_FIXEDPOINT module can optionally utilize Anderson’s method to accelerate convergence. Example programs using each of these nonlinear solver modules in a standalone manner have been added and all CVODES example programs have been updated to use generic SUNNONLINSOL modules.

With the introduction of SUNNONLINSOL modules, the input parameter iter to CVodeCreate has been removed along with the function CVodeSetIterType and the constants CV_NEWTON and CV_FUNCTIONAL. Instead of specifying the nonlinear iteration type when creating the CVODES memory structure, CVODES 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 Chapters 4, 5, and 6. CVODES functions for setting the nonlinear solver options (e.g., CVodeSetMaxNonlinIters) or getting nonlinear solver statistics (e.g., 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 NVECTOR API. These optional operations are disabled by default and may be activated by calling vector specific routines after creating an NVECTOR (see Chapter 7 for more details). The new operations are intended to increase data reuse in vector operations, reduce parallel communication on distributed memory systems, and lower the number of kernel launches on systems with accelerators. The fused operations are N_VLinearCombination, N_VScaleAddMulti, and N_VDotProdMulti and the vector array operations are N_VLinearCombinationVectorArray, N_VScaleVectorArray, N_VConstVectorArray, N_VWrmsNormVectorArray, N_VWrmsNormMaskVectorArray, N_VScaleAddMultiVectorArray, and N_VLinearCombinationVectorArray. If an NVECTOR implementation defines any of these operations as NULL, then standard NVECTOR operations will automatically be called as necessary to complete the computation.
Multiple updates to NVECTOR_CUDA were made:

- Changed `NVGetLength_Cuda` to return the global vector length instead of the local vector length.
- Added `NVGetLocalLength_Cuda` to return the local vector length.
- Added `NVGetMPIComm_Cuda` to return the MPI communicator used.
- Removed the accessor functions in the namespace suncudavec.
- Changed the `NVMake_Cuda` function to take a host data pointer and a device data pointer instead of an `N_VectorContent_Cuda` object.
- Added the ability to set the `cudaStream_t` used for execution of the NVECTOR_CUDA kernels. See the function `NVSetCudaStreams_Cuda`.
- Added `NVNewManaged_Cuda`, `NVMakeManaged_Cuda`, and `N_VIsManagedMemory_Cuda` functions to accommodate using managed memory with the NVECTOR_CUDA.

Multiple changes to NVECTOR_RAJA were made:

- Changed `NVGetLength_Raja` to return the global vector length instead of the local vector length.
- Added `NVGetLocalLength_Raja` to return the local vector length.
- Added `NVGetMPIComm_Raja` to return the MPI communicator used.
- Removed the accessor functions in the namespace suncudavec.

A new NVECTOR implementation for leveraging OpenMP 4.5+ device offloading has been added, NVECTOR_OPENMPDEV. See §7.10 for more details.

Two changes were made in the 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.

Changes in v3.2.1

The changes in this minor release include the following:

- Fixed a bug in the CUDA NVECTOR where the `NVInvTest` 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.
1.2 Changes from previous versions

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 $N_{\text{MinQuotient}}$, $N_{\text{VConstr}}$, and $N_{\text{VCompare}}$ that were not previously required by CVODE and CVODES.

Fixed a thread-safety issue when using adjoint sensitivity analysis.

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

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

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

Several changes were made to the build system:

- CMake 3.1.3 is now the minimum required CMake version.
- Deprecate the behavior of the SUNDIALS_INDEX_TYPE CMake option and added the SUNDIALS_INDEX_SIZE CMake option to select the sunindextype integer size.
- The native CMake FindMPI module is now used to locate an MPI installation.
- If MPI is enabled and MPI compiler wrappers are not set, the build system will check if CMAKE_<language>COMPILER can compile MPI programs before trying to locate and use an MPI installation.
- The previous options for setting MPI compiler wrappers and the executable for running MPI programs have been deprecated. The new options that align with those used in native CMake FindMPI module are MPI_<language>COMPILER, MPI_<language>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 new example, `cvRoberts_FSA_dns_Switch.c`, which demonstrates switching on/off forward sensitivity computations. This example came from the usage notes page of the SUNDIALS website.

• The misnamed function `CVSpilsSetJacTimesSetupFnBS` has been deprecated and replaced by `CVSpilsSetJacTimesBS`. The deprecated function `CVSpilsSetJacTimesSetupFnBS` will be removed in the next major release.

• 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 `summatrix_sparse.c` where we had used `int` instead of `sunindextype` in one location.

• Added missing `#include <stdio.h>` in `nvector` and `sunmatrix` header files.

• Fixed an indexing bug in the CUDA nvector implementation of `N_WrmsNormMask` and revised the RAJA nvector implementation of `N_WrmsNormMask` to work with mask arrays using values other than zero or one. Replaced `double` with `realtype` in the RAJA vector test functions.

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

**Changes in v3.1.0**

Added nvector print functions that write vector data to a specified file (e.g., `N_VPrintFile_Serial`). Added `make test` and `make test_install` options to the build system for testing SUNDIALS after building with `make` and installing with `make install` respectively.

**Changes in v3.0.0**

All interfaces to matrix structures and linear solvers have been reworked, and all example programs have been updated. The goal of the redesign of these interfaces was to provide more encapsulation and 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, SuperLU_MT, 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 web site, https://software.llnl.gov/RAJA/. These additions are accompanied by additions to various interface functions and to user documentation.

All indices for data structures were updated to a new sunindextype that can be configured to be a 32- or 64-bit integer data index type. sunindextype is defined to be int32_t or int64_t when portable types are supported, otherwise it is defined as int or long int. The Fortran interfaces continue to use long int for indices, except for their sparse matrix interface that now uses the new sunindextype. This new flexible capability for index types includes interfaces to PETSc, hypre, SuperLU_MT, and KLU with either 32-bit or 64-bit capabilities depending how the user configures SUNDIALS.

To avoid potential namespace conflicts, the macros defining 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, `NVGetVectorID`, that returns the `nvector` module name.

A bug was fixed in the interpolation functions used in solving backward problems for adjoint sensitivity analysis.

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.

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.

In interpolation routines for backward problems, added logic to bypass sensitivity interpolation if input sensitivity argument is NULL.

New examples were added for use of sparse direct solvers within sensitivity integrations and for use of OpenMP.

Minor corrections and additions were made to the `cvodes` solver, 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 `cvodes` 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 `cvodes`.

Otherwise, only relatively minor modifications were made to the `cvodes` 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 `smu = MIN(N-1,mu+ml)` was changed to `smu = mu + ml` to correct an illegal input error for `DGBTRF/DGBTRS`.

Some minor changes were made in order to minimize the differences between the sources for private functions in `cvodes` and `cvode`.

An option was added in the case of Adjoint Sensitivity Analysis with dense or banded Jacobian: With a call to `CVDlsSetDenseJacFnBS` or `CVDlsSetBandJacFnBS`, the user can specify a user-supplied Jacobian function of type `CVDls***JacFnBS`, for the case where the backward problem depends on the forward sensitivities.
In CVodeQuadSensInit, the line `cv_mem->cv_fQS_data = ...` was corrected (missing Q).

In the User Guide, a paragraph was added in Section 6.2.1 on CVodeAdjReInit, and a paragraph was added in Section 6.2.9 on CVodeGetAdjY. In the example cvsRoberts_ASAi_dns, the output was revised to include the use of CVodeGetAdjY.

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

For the Adjoint Sensitivity Analysis case in which the backward problem depends on the forward sensitivities, options have been added to allow for user-supplied pset, psolve, and jtimes functions.

In order to avoid possible name conflicts, the mathematical macro and function names MIN, MAX, SQRT, RABS, RSqrt, REXP, RPOWERI, and RPOWERR were changed to SUNMIN, SUNMAX, SUNSQR, SUNRABS, SUNRSQRT, SUNREXP, SRPOWERI, and SUNRPOWERR, respectively. These names occur in both the solver and example programs.

In the example cvsHessian_ASA_FSA, an error was corrected in the function fB2: y2 in place of y3 in the third term of Ith(yBdot,6).

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

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

Changes in v2.7.0

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

Errors in the logic for the integration of backward problems were identified and fixed.

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 related to the integration of ODE IVP problems 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.

This version also includes several new features related to sensitivity analysis, among which are: (a) support for integration of quadrature equations depending on both the states and forward sensitivity (and thus support for forward sensitivity analysis of quadrature equations), (b) support for simultaneous integration of multiple backward problems based on the same underlying ODE (e.g., for use in an forward-over-adjoint method for computing second order derivative information), (c) support for backward integration of ODEs and quadratures depending on both forward states and sensitivities (e.g., for use in computing second-order derivative information), and (d) support for reinitialization of the adjoint module.

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. Moreover, the prototypes of all functions related to integration of backward problems were
modified to support the simultaneous integration of multiple problems. All backward problems defined
by the user are internally managed through a linked list and identified in the user interface through
a unique identifier.

**Changes in v2.5.0**

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

In the adjoint solver module, the following two bugs were fixed: in CVodeF the solver was sometimes
incorrectly taking an additional step before returning control to the user (in `CV_NORMAL` mode) thus
leading to a failure in the interpolated output function; in CVodeB, while searching for the current check
point, the solver was sometimes reaching outside the integration interval resulting in a segmentation
fault.

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 (SPTFQMR)
linear solver modules, respectively (for details see Chapter 4). At the same time, function type names
for Scaled Preconditioned Iterative Linear Solvers were added for the user-supplied Jacobian-times-
vector and preconditioner setup and solve functions.

A new interpolation method was added to the CVODES adjoint module. The function `CVadjMalloc`
has an additional argument which can be used to select the desired interpolation scheme.

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 (`cvodes` 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**

A minor bug was fixed in the interpolation functions of the adjoint CVODES module.

**Changes in v2.2.0**

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

Changes in v2.1.2
A bug was fixed in the CVode function that was potentially leading to erroneous behaviour of the rootfinding procedure on the integration first step.

Changes in v2.1.1
This CVODES release includes bug fixes related to forward sensitivity computations (possible loss of accuracy on a BDF order increase and incorrect logic in testing user-supplied absolute tolerances). In addition, we have added the option of activating and deactivating forward sensitivity calculations on successive CVODES runs without memory allocation/deallocation.

Other changes in this minor SUNDIALS release affect the build system.

Changes in v2.1.0
The major changes from the previous version involve a redesign of the user interface across the entire SUNDIALS suite. We have eliminated the mechanism of providing optional inputs and extracting optional statistics from the solver through the iopt and ropt arrays. Instead, CVODES now provides a set of routines (with prefix CVodeSet) to change the default values for various quantities controlling the solver and a set of extraction routines (with prefix CVodeGet) to extract statistics after return from the main solver routine. Similarly, each linear solver module provides its own set of Set- and Get-type routines. For more details see §4.5.7 and §4.5.9.

Additionally, the interfaces to several user-supplied routines (such as those providing Jacobians, preconditioner information, and sensitivity right hand sides) were simplified by reducing the number of arguments. The same information that was previously accessible through such arguments can now be obtained through Get-type functions.

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 CVODES (and all of 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 CVODES. 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 [43]. In addition, to solve a forward sensitivity problem the user should read §2.6, followed by Chapter 5 through §5.2.5 only, and look at examples in [43].

In a different direction, a more expert user with an IVP problem may want to (a) use a package preconditioner (§4.8), (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 NVECTOR module (Chapter 7), or even (e) supply new SUNLINSOL and/or SUNMATRIX modules (Chapters 8 and 9). An advanced user with a forward sensitivity problem may also want to (a) provide his/her own sensitivity equations right-hand side routine (§5.3), (b) perform multiple runs with the same number of sensitivity parameters (§5.2.1), or (c) extract additional diagnostic information (§5.2.5). A user with an adjoint sensitivity problem needs to understand the IVP solution approach at the desired level and also go through §2.7 for a short mathematical description of the adjoint approach, Chapter 6 for the usage of the adjoint module in CVODES, and the examples in [43].

The structure of this document is as follows:

• In Chapter 2, we give short descriptions of the numerical methods implemented by CVODES for the solution of initial value problems for systems of ODEs, continue with short descriptions of
preconditioning (§2.2), stability limit detection (§2.3), and rootfinding (§2.4), and conclude with an overview of the mathematical aspects of sensitivity analysis, both forward (§2.6) and adjoint (§2.7).

• The following chapter describes the structure of the SUNDIALS suite of solvers (§3.1) and the software organization of the CVODES solver (§3.2).

• Chapter 4 is the main usage document for CVODES for simulation applications. It includes a complete description of the user interface for the integration of ODE initial value problems. Readers that are not interested in using CVODES for sensitivity analysis can then skip the next two chapters.

• Chapter 5 describes the usage of CVODES for forward sensitivity analysis as an extension of its IVP integration capabilities. We begin with a skeleton of the user main program, with emphasis on the steps that are required in addition to those already described in Chapter 4. Following that we provide detailed descriptions of the user-callable interface routines specific to forward sensitivity analysis and of the additional optional user-defined routines.

• Chapter 6 describes the usage of CVODES for adjoint sensitivity analysis. We begin by describing the CVODES checkpointing implementation for interpolation of the original IVP solution during integration of the adjoint system backward in time, and with an overview of a user’s main program. Following that we provide complete descriptions of the user-callable interface routines for adjoint sensitivity analysis as well as descriptions of the required additional user-defined routines.

• Chapter 7 gives a brief overview of the generic NVVECTOR module shared among the various components of SUNDIALS, and details on the NVVECTOR implementations provided with SUNDIALS.

• Chapter 8 gives a brief overview of the generic SUNMATRIX module shared among the various components of SUNDIALS, and details on the SUNMATRIX implementations provided with SUNDIALS: a dense implementation (§8.2), a banded implementation (§8.3) and a sparse implementation (§8.4).

• Chapter 9 gives a brief overview of the generic SUNLINSOL module shared among the various components of SUNDIALS. This chapter contains details on the SUNLINSOL implementations provided with SUNDIALS. The chapter also contains details on the SUNLINSOL implementations provided with SUNDIALS that interface with external linear solver libraries.

• Finally, in the appendices, we provide detailed instructions for the installation of CVODES, within the structure of SUNDIALS (Appendix A), as well as a list of all the constants used for input to and output from CVODES 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 CVODES, are written in all capitals. Usage and installation instructions that constitute important warnings are marked with a triangular symbol in the margin.

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 disclamer 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)
<table>
<thead>
<tr>
<th>Code Number</th>
<th>Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCRL-CODE-155951</td>
<td>CVODE</td>
</tr>
<tr>
<td>UCRL-CODE-155950</td>
<td>CVODES</td>
</tr>
<tr>
<td>UCRL-CODE-155952</td>
<td>IDA</td>
</tr>
<tr>
<td>UCRL-CODE-237203</td>
<td>IDAS</td>
</tr>
<tr>
<td>LLNL-CODE-665877</td>
<td>KINSOL</td>
</tr>
</tbody>
</table>
Chapter 2

Mathematical Considerations

cvodes solves ODE initial value problems (IVPs) in real $N$-space, which we write in the abstract form
\[
\dot{y} = f(t, y), \quad y(t_0) = y_0,
\]  
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. cvodes 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.

Additionally, if (2.1) depends on some parameters $p \in \mathbb{R}^{N_p}$, i.e.
\[
\dot{y} = f(t, y, p)
\]
\[
y(t_0) = y_0(p),
\]  
(cvodes can also compute first order derivative information, performing either forward sensitivity analysis or adjoint sensitivity analysis. In the first case, cvodes computes the sensitivities of the solution with respect to the parameters $p$, while in the second case, cvodes computes the gradient of a derived function with respect to the parameters $p$.

2.1 IVP solution

The methods used in cvodes are variable-order, variable-step multistep methods, based on formulas of the form
\[
\sum_{i=0}^{K_1} \alpha_{n,i} y^{n-i} + h_n \sum_{i=0}^{K_2} \beta_{n,i} \dot{y}^{n-i} = 0.
\]  
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, cvodes 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 [9] 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
\[
F(y^n) \equiv y^n - h_n \beta_{n,0} f(t_n, y^n) - a_n = 0,
\]  
or as a fixed-point problem
\[
G(y^n) \equiv h_n \beta_{n,0} f(t_n, y^n) + a_n = y^n.
\]
where \( a_n = \sum_{i>0}(\alpha_{n,i}y^{n-i} + h_n\beta_{n,i}y^{n-i}) \). CVODES provides several nonlinear solver choices as well as the option of using a user-defined nonlinear solver (see Chapter 10). By default CVODES solves (2.4) with a Newton iteration which requires the solution of linear systems

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

in which

\[
M \approx I - \gamma J, \quad J = \partial f/\partial y, \quad \text{and} \quad \gamma = h_n\beta_{n,0}.
\]

The exact variation of the Newton iteration depends on the choice of linear solver and is discussed below and in §10.2. For nonstiff systems, a fixed-point iteration (previously referred to as a functional iteration in this guide) for solving (2.5) is also available. This involves evaluations of \( f \) only and can (optionally) use Anderson’s method [3, 45, 18, 35] to accelerate convergence (see §10.3 for more details). For any nonlinear solver, the initial guess for the iteration is a predicted value \( y^{n(0)} \) computed explicitly from the available history data.

For nonlinear solvers that require the solution of the linear system (2.6) (e.g., the default Newton iteration), CVODES provides several linear solver choices, including the option of a user-supplied linear solver module (see Chapter 9). 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 [14, 1], or the thread-enabled SuperLU_MT sparse solver library [34, 16, 2] (serial or threaded vector modules only) [Note that users will need to download and install the KLU or SUPERLUMT packages independent of CVODES],
- 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 [6].

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, CVODES uses a weighted root-mean-square norm, denoted ||·||\text{WRMS}, for all error-like quantities. The multiplicative weights used are based on the current solution and on the relative and absolute tolerances input by the user, namely

\[
W_i = 1/\left[\text{RTOL} \cdot |y_i| + \text{ATOL}_i\right].
\]
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 cases 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 $Jv$ 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 (2.4), 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

$$J_{ij} = \frac{f_i(t, y + \sigma_j e_j) - f_i(t, y)}{\sigma_j},$$

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 (2.8). 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

$$Jv = \frac{[f(t, y + \sigma v) - f(t, y)]}{\sigma}.$$  \hspace{1cm} (2.9)

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

A critical part of CVODES — 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

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

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

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

The local error test is simply $\|LTE\| \leq 1$. 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

$$\|\Delta_n\| \leq \epsilon \equiv 1/|C'|.$$

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

$$(h'/h)^{q+1}\|\Delta_n\| = \epsilon/6.$$  \hspace{1cm} (2.9)

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, CVODES 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
\[
h'/h = (\varepsilon/6\|\Delta_{n}\|)^{1/(q+1)} \equiv \eta_q.
\]
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, \( \text{LTE}(q') \), 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 CVODES described above, as inherited from VODE and VODPK, are documented in [5, 8, 25]. They are also summarized in [26].

CVODES 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, CVODES 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, CVODES 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 CVODES 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 \( (2.6) \) (e.g., the default Newton iteration), CVODES 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 \((AP^{-1})P_x = b\); or on both sides, as \((P_L^{-1}AP_R^{-1})P_{Rx} = P_L^{-1}b\). The Krylov method is then applied to a system with the matrix \( P^{-1}A \), or \( AP^{-1} \), or \( P_L^{-1}AP_R^{-1} \), instead of \( A \). In order to improve the convergence of the Krylov iteration, the preconditioner matrix \( P \), or the product \( P_LP_R \) 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 \( P_L \) and \( P_R \), should be reasonably efficient to evaluate and solve. Finding a good point in this tradeoff between rapid convergence and low cost can be very difficult. Good choices are often problem-dependent (for example, see [6] for an extensive study of preconditioners for reaction-transport systems).

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 \( P_LP_R \)). 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 \textsc{cvodes} are based on approximations to the system Jacobian, $J = \partial f/\partial y$. Since the matrix involved is $M = I - \gamma J$, any approximation \( \bar{J} \) to $J$ yields a matrix that is of potential use as a preconditioner, namely $P = I - \gamma \bar{J}$. Because the linear solver 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.

\section*{2.3 BDF stability limit detection}

\textsc{cvodes} includes an algorithm, \textsc{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, \textsc{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 ($h \sim 1/\nu$, where $\nu$ 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 \textsc{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 [23]. The algorithm supplements (but differs greatly from) the existing algorithms in \textsc{cvodes} for choosing step size and order based on estimated local truncation errors. The \textsc{stald} algorithm works directly with history data that is readily available in \textsc{cvodes}. 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 \textsc{stald} algorithm has been tested in combination with the \textsc{vode} solver on linear
advection-dominated advection-diffusion problems \cite{24}, where it works well. The implementation in CVODES 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 CVODES solver has been augmented to include a rootfinding feature. This means that, while integrating the Initial Value Problem (2.1), CVODES 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 CVODES. 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 \cite{22}. In addition, each time a step is computed, CVODES 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 $t$, CVODES 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, CVODES stops and reports an error. This way, each time CVODES 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, CVODES has an interval $(t_{lo}, t_{hi})$ in which roots of the $g_i(t)$ are to be sought, such that $t_{hi}$ is further ahead in the direction of integration, and all $g_i(t_{lo}) \neq 0$. The endpoint $t_{hi}$ is either $t_n$, the end of the time step last taken, or the next requested output time $t_{out}$ if this comes sooner. The endpoint $t_{lo}$ is either $t_{n-1}$, 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

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

Whenever sign changes are seen in two or more root functions, the one deemed most likely to have its root occur first is the one with the largest value of $|g_i(t_{hi})|/|g_i(t_{hi}) - g_i(t_{lo})|$, corresponding to the closest to $t_{lo}$ of the secant method values. At each pass through the loop, a new value $t_{mid}$ is set, strictly within the search interval, and the values of $g_i(t_{mid})$ are checked. Then either $t_{lo}$ or $t_{hi}$ is reset to $t_{mid}$ according to which subinterval is found to 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 $t_{\text{mid}}$ the secant method value. Thereafter, $\alpha$ is reset according to the side of the subinterval (low vs. high, i.e., toward $t_{\text{lo}}$ vs. toward $t_{\text{hi}}$) 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 $t_{\text{mid}}$ is closer to $t_{\text{lo}}$ when $\alpha < 1$ and closer to $t_{\text{hi}}$ when $\alpha > 1$. If the above value of $t_{\text{mid}}$ is within $\tau/2$ of $t_{\text{lo}}$ or $t_{\text{hi}}$, 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 $\tau/2$.

2.5 Pure quadrature integration

In many applications, and most notably during the backward integration phase of an adjoint sensitivity analysis run (see §2.7) it is of interest to compute integral quantities of the form

$$z(t) = \int_{t_0}^{t} q(\tau, y(\tau), p) d\tau. \quad (2.10)$$

The most effective approach to compute $z(t)$ is to extend the original problem with the additional ODEs (obtained by applying Leibnitz’s differentiation rule):

$$\dot{z} = q(t, y, p), \quad z(t_0) = 0. \quad (2.11)$$

Note that this is equivalent to using a quadrature method based on the underlying linear multistep polynomial representation for $y(t)$.

This can be done at the “user level” by simply exposing to CVODES the extended ODE system (2.2)+(2.10). However, in the context of an implicit integration solver, this approach is not desirable since the nonlinear solver module will require the Jacobian (or Jacobian-vector product) of this extended ODE. Moreover, since the additional states $z$ do not enter the right-hand side of the ODE (2.10) and therefore the right-hand side of the extended ODE system, it is much more efficient to treat the ODE system (2.10) separately from the original system (2.2) by “taking out” the additional states $z$ from the nonlinear system (2.4) that must be solved in the correction step of the LMM. Instead, “corrected” values $z^n$ are computed explicitly as

$$z^n = -\frac{1}{\alpha_{n,0}} \left( h_{n,0|0} q(t_n, y_n, p) + h_n \sum_{i=1}^{K_2} \beta_{n,i} z^{n-i} + \sum_{i=1}^{K_1} \alpha_{n,i} z^{n-i} \right),$$

once the new approximation $y^n$ is available.

The quadrature variables $z$ can be optionally included in the error test, in which case corresponding relative and absolute tolerances must be provided.

2.6 Forward sensitivity analysis

Typically, the governing equations of complex, large-scale models depend on various parameters, through the right-hand side vector and/or through the vector of initial conditions, as in (2.2). In addition to numerically solving the ODEs, it may be desirable to determine the sensitivity of the results with respect to the model parameters. Such sensitivity information can be used to estimate which parameters are most influential in affecting the behavior of the simulation or to evaluate optimization gradients (in the setting of dynamic optimization, parameter estimation, optimal control, etc.).

The solution sensitivity with respect to the model parameter $p_i$ is defined as the vector $s_i(t) = \partial y(t)/\partial p_i$ and satisfies the following forward sensitivity equations (or sensitivity equations for short):

$$\dot{s}_i = \frac{\partial f}{\partial y} s_i + \frac{\partial f}{\partial p_i}, \quad s_i(t_0) = \frac{\partial y_0(p)}{\partial p_i}, \quad (2.12)$$

obtained by applying the chain rule of differentiation to the original ODEs (2.2).
When performing forward sensitivity analysis, CVODES carries out the time integration of the combined system, (2.2) and (2.12), by viewing it as an ODE system of size \( N(N_s + 1) \), where \( N_s \) is the number of model parameters \( p_i \), with respect to which sensitivities are desired (\( N_s \leq N_p \)). However, major improvements in efficiency can be made by taking advantage of the special form of the sensitivity equations as linearizations of the original ODEs. In particular, for stiff systems, for which CVODES employs a Newton iteration, the original ODE system and all sensitivity systems share the same Jacobian matrix, and therefore the same iteration matrix \( M \) in (2.7).

The sensitivity equations are solved with the same linear multistep formula that was selected for the original ODEs and, if Newton iteration was selected, the same linear solver is used in the correction phase for both state and sensitivity variables. In addition, CVODES offers the option of including (full error control) or excluding (partial error control) the sensitivity variables from the local error test.

### 2.6.1 Forward sensitivity methods

In what follows we briefly describe three methods that have been proposed for the solution of the combined ODE and sensitivity system for the vector \( \hat{y} = [y, s_1, \ldots, s_{N_s}] \).

- **Staggered Direct**

  In this approach [12], the nonlinear system (2.4) is first solved and, once an acceptable numerical solution is obtained, the sensitivity variables at the new step are found by directly solving (2.12) after the (BDF or Adams) discretization is used to eliminate \( \dot{s}_i \). Although the system matrix of the above linear system is based on exactly the same information as the matrix \( M \) in (2.7), it must be updated and factored at every step of the integration, in contrast to an evaluation of \( M \) which is updated only occasionally. For problems with many parameters (relative to the problem size), the staggered direct method can outperform the methods described below [33]. However, the computational cost associated with matrix updates and factorizations makes this method unattractive for problems with many more states than parameters (such as those arising from semidiscretization of PDEs) and is therefore not implemented in CVODES.

- **Simultaneous Corrector**

  In this method [36], the discretization is applied simultaneously to both the original equations (2.2) and the sensitivity systems (2.12) resulting in the following nonlinear system

  \[
  \hat{F}(\hat{y}_n) \equiv \hat{y}_n - h_n \beta_n,0 \hat{f}(t_n, \hat{y}_n) - \hat{a}_n = 0, 
  \]

  where \( \hat{f} = [f(t,y,p), \ldots, (\partial f / \partial y)(t,y,p)s_i + (\partial f / \partial p_i)(t,y,p), \ldots] \), and \( \hat{a}_n \) is comprised of the terms in the discretization that depend on the solution at previous integration steps. This combined nonlinear system can be solved using a modified Newton method as in (2.6) by solving the corrector equation

  \[
  \hat{M}[\hat{y}_{n(m+1)} - \hat{y}_{n(m)}] = -\hat{F}(\hat{y}_{n(m)}) \tag{2.13}
  \]

  at each iteration, where

  \[
  \hat{M} = \begin{bmatrix}
  M \\
  -\gamma J_1 & M \\
  -\gamma J_2 & 0 & M \\
  \vdots & \vdots & \ddots & \ddots \\
  -\gamma J_{N_s} & 0 & \ldots & 0 & M
  \end{bmatrix},
  \]

  \( M \) is defined as in (2.7), and \( J_i = (\partial/\partial y)[(\partial f/\partial y)s_i + (\partial f/\partial p_i)] \). It can be shown that 2-step quadratic convergence can be retained by using only the block-diagonal portion of \( \hat{M} \) in the corrector equation (2.13). This results in a decoupling that allows the reuse of \( M \) without additional matrix factorizations. However, the products \( \partial f / \partial y \) and the vectors \( \partial f / \partial p_i \) must still be reevaluated at each step of the iterative process (2.13) to update the sensitivity portions of the residual \( \hat{G} \).
• Staggered corrector

In this approach [19], as in the staggered direct method, the nonlinear system (2.4) is solved first using the Newton iteration (2.6). Then a separate Newton iteration is used to solve the sensitivity system (2.12):

\[
M[s_i^{(m+1)} - s_i^{(m)\text{}}] = -\left[ s_i^{(m)} - \gamma \left( \frac{\partial f}{\partial y}(t_n, y^n, p)s_i^{n(m)} + \frac{\partial f}{\partial p_i}(t_n, y^n, p) \right) - a_{i,n} \right], \quad (2.14)
\]

where \(a_{i,n} = \sum_{j>0}(\alpha_n j s_i^{n-j} + h_n \beta_n j s_i^{n-j})\). In other words, a modified Newton iteration is used to solve a linear system. In this approach, the vectors \(\partial f/\partial p_i\) need be updated only once per integration step, after the state correction phase (2.6) has converged. Note also that Jacobian-related data can be reused at all iterations (2.14) to evaluate the products \((\partial f/\partial y)s_i\),

CVODES implements the simultaneous corrector method and two flavors of the staggered corrector method which differ only if the sensitivity variables are included in the error control test. In the full error control case, the first variant of the staggered corrector method requires the convergence of the iterations (2.14) for all \(N_s\) sensitivity systems and then performs the error test on the sensitivity variables. The second variant of the method will perform the error test for each sensitivity vector \(s_i, (i = 1, 2, \ldots, N_s)\) individually, as they pass the convergence test. Differences in performance between the two variants may therefore be noticed whenever one of the sensitivity vectors \(s_i\) fails a convergence or error test.

An important observation is that the staggered corrector method, combined with a Krylov linear solver, effectively results in a staggered direct method. Indeed, the Krylov solver requires only the action of the matrix \(M\) on a vector and this can be provided with the current Jacobian information. Therefore, the modified Newton procedure (2.14) will theoretically converge after one iteration.

2.6.2 Selection of the absolute tolerances for sensitivity variables

If the sensitivities are included in the error test, CVODES provides an automated estimation of absolute tolerances for the sensitivity variables based on the absolute tolerance for the corresponding state variable. The relative tolerance for sensitivity variables is set to be the same as for the state variables. The selection of absolute tolerances for the sensitivity variables is based on the observation that the sensitivity vector \(s_i\) will have units of \([y_i]/[p_i]\). With this, the absolute tolerance for the \(j\)-th component of the sensitivity vector \(s_i\) is set to \(\text{ATOL}_j/\bar{p}_i\), where \(\text{ATOL}_j\) are the absolute tolerances for the state variables and \(\bar{p}\) is a vector of scaling factors that are dimensionally consistent with the model parameters \(p\) and give an indication of their order of magnitude. This choice of relative and absolute tolerances is equivalent to requiring that the weighted root-mean-square norm of the sensitivity vector \(s_i\) with weights based on \(s_i\) be the same as the weighted root-mean-square norm of the vector of scaled sensitivities \(\bar{s}_i = |\bar{p}_i|s_i\) with weights based on the state variables (the scaled sensitivities \(\bar{s}_i\) being dimensionally consistent with the state variables). However, this choice of tolerances for the \(s_i\) may be a poor one, and the user of CVODES can provide different values as an option.

2.6.3 Evaluation of the sensitivity right-hand side

There are several methods for evaluating the right-hand side of the sensitivity systems (2.12): analytic evaluation, automatic differentiation, complex-step approximation, and finite differences (or directional derivatives). CVODES provides all the software hooks for implementing interfaces to automatic differentiation (AD) or complex-step approximation; future versions will include a generic interface to AD-generated functions. At the present time, besides the option for analytical sensitivity right-hand sides (user-provided), CVODES can evaluate these quantities using various finite difference-based approximations to evaluate the terms \((\partial f/\partial y)s_i\) and \((\partial f/\partial p_i)\), or using directional derivatives to evaluate \([(\partial f/\partial y)s_i + (\partial f/\partial p_i)]\). As is typical for finite differences, the proper choice of perturbations is a delicate matter. CVODES takes into account several problem-related features: the
relative ODE error tolerance RTOL, the machine unit roundoff $U$, the scale factor $\bar{p}_i$, and the weighted root-mean-square norm of the sensitivity vector $s_i$.

Using central finite differences as an example, the two terms $(\partial f / \partial y)_{s_i}$ and $\partial f / \partial p_i$ in the right-hand side of (2.12) can be evaluated either separately:

$$\frac{\partial f}{\partial y}_{s_i} \approx \frac{f(t,y + \sigma_y s_i, p) - f(t,y - \sigma_y s_i, p)}{2\sigma_y}, \quad (2.15)$$

$$\frac{\partial f}{\partial p_i} \approx \frac{f(t,y, p + \sigma e_i) - f(t,y, p - \sigma e_i)}{2\sigma_i}, \quad (2.15')$$

$$\sigma_i = |\bar{p}_i|\sqrt{\max(\text{RTOL}, U)}, \quad \sigma_y = \frac{1}{\max(1/\sigma_i, \|s_i\|_{\text{WRMS}}/|\bar{p}_i|)},$$

or simultaneously:

$$\frac{\partial f}{\partial y}_{s_i} + \frac{\partial f}{\partial p_i} \approx \frac{f(t,y + \sigma s_i, p + \sigma e_i) - f(t,y - \sigma s_i, p - \sigma e_i)}{2\sigma}$$

$$\sigma = \min(\sigma_i, \sigma_y), \quad (2.16)$$

or by adaptively switching between (2.15)+(2.15') and (2.16), depending on the relative size of the finite difference increments $\sigma_i$ and $\sigma_y$. In the adaptive scheme, if $\rho = \max(\sigma_i/\sigma_y, \sigma_y/\sigma_i)$, we use separate evaluations if $\rho > \rho_{\text{max}}$ (an input value), and simultaneous evaluations otherwise.

These procedures for choosing the perturbations ($\sigma_i$, $\sigma_y$, $\sigma$) and switching between finite difference and directional derivative formulas have also been implemented for one-sided difference formulas. Forward finite differences can be applied to $(\partial f / \partial y)_{s_i}$ and $(\partial f / \partial p_i)$ separately, or the single directional derivative formula

$$\frac{\partial f}{\partial y}_{s_i} + \frac{\partial f}{\partial p_i} \approx \frac{f(t,y + \sigma s_i, p + \sigma e_i) - f(t,y - \sigma s_i, p - \sigma e_i)}{\sigma}$$

can be used. In CVODES, the default value of $\rho_{\text{max}} = 0$ indicates the use of the second-order centered directional derivative formula (2.16) exclusively. Otherwise, the magnitude of $\rho_{\text{max}}$ and its sign (positive or negative) indicates whether this switching is done with regard to (centered or forward) finite differences, respectively.

### 2.6.4 Quadratures depending on forward sensitivities

If pure quadrature variables are also included in the problem definition (see §2.5), CVODES does not carry their sensitivities automatically. Instead, we provide a more general feature through which integrals depending on both the states $y$ of (2.2) and the state sensitivities $s_i$ of (2.12) can be evaluated. In other words, CVODES provides support for computing integrals of the form:

$$\bar{z}(t) = \int_{t_0}^{t} \tilde{q}(\tau, y(\tau), s_1(\tau), \ldots, s_{N_p}(\tau), p) \, d\tau.$$

If the sensitivities of the quadrature variables $z$ of (2.10) are desired, these can then be computed by using:

$$\tilde{q}_i = q_y s_i + q_p, \quad i = 1, \ldots, N_p,$$

as integrands for $\bar{z}$, where $q_y$ and $q_p$ are the partial derivatives of the integrand function $q$ of (2.10).

As with the quadrature variables $z$, the new variables $\bar{z}$ are also excluded from any nonlinear solver phase and “corrected” values $\bar{z}^\alpha$ are obtained through explicit formulas.

### 2.7 Adjoint sensitivity analysis

In the forward sensitivity approach described in the previous section, obtaining sensitivities with respect to $N_x$ parameters is roughly equivalent to solving an ODE system of size $(1 + N_x)N$. This can become prohibitively expensive, especially for large-scale problems, if sensitivities with respect
to many parameters are desired. In this situation, the *adjoint sensitivity method* is a very attractive alternative, provided that we do not need the solution sensitivities \( s_i \), but rather the gradients with respect to model parameters of a relatively few derived functionals of the solution. In other words, if \( y(t) \) is the solution of (2.2), we wish to evaluate the gradient \( dG/dp \) of
\[
G(p) = \int_{t_0}^{T} g(t,y,p) dt ,
\]
(2.17)
or, alternatively, the gradient \( dg/dp \) of the function \( g(t,y,p) \) at the final time \( T \). The function \( g \) must be smooth enough that \( \partial g/\partial y \) and \( \partial g/\partial p \) exist and are bounded.

In what follows, we only sketch the analysis for the sensitivity problem for both \( G \) and \( g \). For details on the derivation see [11]. Introducing a Lagrange multiplier \( \lambda \), we form the augmented objective function
\[
I(p) = G(p) - \int_{t_0}^{T} \lambda^* (\dot{y} - f(t,y,p)) dt ,
\]
(2.18)
where \(*\) denotes the conjugate transpose. The gradient of \( G \) with respect to \( p \) is
\[
\frac{dG}{dp} = \frac{dI}{dp} = \int_{t_0}^{T} (g_p + g_y s) dt - \int_{t_0}^{T} \lambda^* (\dot{s} - f_y s - f_p) dt ,
\]
(2.19)
where subscripts on functions \( f \) or \( g \) are used to denote partial derivatives and \( s = [s_1, \ldots, s_N] \) is the matrix of solution sensitivities. Applying integration by parts to the term \( \lambda^* \dot{s} \), and by requiring that \( \lambda \) satisfy
\[
\dot{\lambda} = - \left( \frac{\partial f}{\partial y} \right)^* \lambda - \left( \frac{\partial g}{\partial y} \right)^* \lambda(T) = 0 ,
\]
(2.20)
the gradient of \( G \) with respect to \( p \) is nothing but
\[
\frac{dG}{dp} = \lambda^* (t_0) s(t_0) + \int_{t_0}^{T} (g_p + \lambda^* f_p) dt .
\]
(2.21)
The gradient of \( g(T,y,p) \) with respect to \( p \) can be then obtained by using the Leibnitz differentiation rule. Indeed, from (2.17),
\[
\frac{dg}{dp} (T) = \frac{d}{dT} \frac{dG}{dp}
\]
and therefore, taking into account that \( dG/dp \) in (2.21) depends on \( T \) both through the upper integration limit and through \( \lambda \), and that \( \lambda(T) = 0 \),
\[
\frac{dg}{dp} (T) = \mu^* (t_0) s(t_0) + g_p(T) + \int_{t_0}^{T} \mu^* f_p dt ,
\]
(2.22)
where \( \mu \) is the sensitivity of \( \lambda \) with respect to the final integration limit \( T \). Thus \( \mu \) satisfies the following equation, obtained by taking the total derivative with respect to \( T \) of (2.20):
\[
\dot{\mu} = - \left( \frac{\partial f}{\partial y} \right)^* \mu
\]
\[
\mu(T) = \left( \frac{\partial g}{\partial y} \right)^*_{t=T} .
\]
(2.23)
The final condition on \( \mu(T) \) follows from \( (\partial \lambda/\partial t) + (\partial \lambda/\partial T) = 0 \) at \( T \), and therefore, \( \mu(T) = -\dot{\lambda}(T) \).

The first thing to notice about the adjoint system (2.20) is that there is no explicit specification of the parameters \( p \); this implies that, once the solution \( \lambda \) is found, the formula (2.21) can then be
used to find the gradient of $G$ with respect to any of the parameters $p$. The same holds true for the system (2.23) and the formula (2.22) for gradients of $g(T, y, p)$. The second important remark is that the adjoint systems (2.20) and (2.23) are terminal value problems which depend on the solution $y(t)$ of the original IVP (2.2). Therefore, a procedure is needed for providing the states $y$ obtained during a forward integration phase of (2.2) to CVODES during the backward integration phase of (2.20) or (2.23). The approach adopted in CVODES, based on checkpointing, is described below.

2.7.1 Checkpointing scheme

During the backward integration, the evaluation of the right-hand side of the adjoint system requires, at the current time, the states $y$ which were computed during the forward integration phase. Since CVODES implements variable-step integration formulas, it is unlikely that the states will be available at the desired time and so some form of interpolation is needed. The CVODES implementation being also variable-order, it is possible that during the forward integration phase the order may be reduced as low as first order, which means that there may be points in time where only $y$ and $\dot{y}$ are available. These requirements therefore limit the choices for possible interpolation schemes. CVODES implements two interpolation methods: a cubic Hermite interpolation algorithm and a variable-degree polynomial interpolation method which attempts to mimic the BDF interpolant for the forward integration.

However, especially for large-scale problems and long integration intervals, the number and size of the vectors $y$ and $\dot{y}$ that would need to be stored make this approach computationally intractable. Thus, CVODES settles for a compromise between storage space and execution time by implementing a so-called checkpointing scheme. At the cost of at most one additional forward integration, this approach offers the best possible estimate of memory requirements for adjoint sensitivity analysis. To begin with, based on the problem size $N$ and the available memory, the user decides on the number $N_d$ of data pairs $(y, \dot{y})$ if cubic Hermite interpolation is selected, or on the number $N_d$ of $y$ vectors in the case of variable-degree polynomial interpolation, that can be kept in memory for the purpose of interpolation. Then, during the first forward integration stage, after every $N_d$ integration steps a checkpoint is formed by saving enough information (either in memory or on disk) to allow for a hot restart, that is a restart which will exactly reproduce the forward integration. In order to avoid storing Jacobian-related data at each checkpoint, a reevaluation of the iteration matrix is forced before each checkpoint. At the end of this stage, we are left with $N_c$ checkpoints, including one at $t_0$. During the backward integration stage, the adjoint variables are integrated from $T$ to $t_0$ going from one checkpoint to the previous one. The backward integration from checkpoint $i + 1$ to checkpoint $i$ is preceded by a forward integration from $i$ to $i + 1$ during which the $N_d$ vectors $y$ (and, if necessary $\dot{y}$) are generated and stored in memory for interpolation\(^1\) (see Fig. 2.1).

This approach transfers the uncertainty in the number of integration steps in the forward integration phase to uncertainty in the final number of checkpoints. However, $N_c$ is much smaller than the number of steps taken during the forward integration, and there is no major penalty for writing/reading the checkpoint data to/from a temporary file. Note that, at the end of the first forward integration stage, interpolation data are available from the last checkpoint to the end of the interval of integration. If no checkpoints are necessary ($N_d$ is larger than the number of integration steps taken in the solution of (2.2)), the total cost of an adjoint sensitivity computation can be as low as one forward plus one backward integration. In addition, CVODES provides the capability of reusing a set of checkpoints for multiple backward integrations, thus allowing for efficient computation of gradients of several functionals (2.17).

\(^1\)The degree of the interpolation polynomial is always that of the current BDF order for the forward interpolation at the first point to the right of the time at which the interpolated value is sought (unless too close to the $i$-th checkpoint, in which case it uses the BDF order at the right-most relevant point). However, because of the FLC BDF implementation (see §2.1), the resulting interpolation polynomial is only an approximation to the underlying BDF interpolant.

The Hermite cubic interpolation option is present because it was implemented chronologically first and it is also used by other adjoint solvers (e.g. DASPKADJOIN). The variable-degree polynomial is more memory-efficient (it requires only half of the memory storage of the cubic Hermite interpolation) and is more accurate. The accuracy differences are minor when using BDF (since the maximum method order cannot exceed 5), but can be significant for the Adams method for which the order can reach 12.
Finally, we note that the adjoint sensitivity module in CVODES provides the necessary infrastructure to integrate backwards in time any ODE terminal value problem dependent on the solution of the IVP (2.2), including adjoint systems (2.20) or (2.23), as well as any other quadrature ODEs that may be needed in evaluating the integrals in (2.21) or (2.22). In particular, for ODE systems arising from semi-discretization of time-dependent PDEs, this feature allows for integration of either the discretized adjoint PDE system or the adjoint of the discretized PDE.

2.8 Second-order sensitivity analysis

In some applications (e.g., dynamically-constrained optimization) it may be desirable to compute second-order derivative information. Considering the ODE problem (2.2) and some model output functional, \( g(y) \) then the Hessian \( \frac{d^2 g}{dp^2} \) can be obtained in a forward sensitivity analysis setting as

\[
\frac{d^2 g}{dp^2} = \left( g_y \otimes I_{N_p} \right) y_{pp} + y_p^T g_{yy} y_p,
\]

where \( \otimes \) is the Kronecker product. The second-order sensitivities are solution of the matrix ODE system:

\[
\dot{y}_{pp} = \left( f_y \otimes I_{N_p} \right) \cdot y_{pp} + \left( I_N \otimes y_p^T \right) \cdot f_{yy} y_p
\]

\[
y_{pp}(t_0) = \frac{\partial^2 y_0}{\partial p^2},
\]

where \( y_p \) is the first-order sensitivity matrix, the solution of \( N_p \) systems (2.12), and \( y_{pp} \) is a third-order tensor. It is easy to see that, except for situations in which the number of parameters \( N_p \) is very small, the computational cost of this so-called forward-over-forward approach is exorbitant as it requires the solution of \( N_p + N_p^2 \) additional ODE systems of the same dimension \( N \) as (2.2).

A much more efficient alternative is to compute Hessian-vector products using a so-called forward-over-adjoint approach. This method is based on using the same “trick” as the one used in computing gradients of pointwise functionals with the adjoint method, namely applying a formal directional forward derivation to one of the gradients of (2.21) or (2.22). With that, the cost of computing a full Hessian is roughly equivalent to the cost of computing the gradient with forward sensitivity analysis. However, Hessian-vector products can be cheaply computed with one additional adjoint solve. Consider for example, \( G(p) = \int_{t_0}^{t_f} g(t, y) \ dt \). It can be shown that the product between the Hessian of \( G \) (with respect to the parameters \( p \)) and some vector \( u \) can be computed as

\[
\frac{\partial^2 G}{\partial p^T} u = \left( \lambda^T \otimes I_{N_p} \right) y_{pp} u + y_p^T H_t \big|_{t=t_0},
\]

\footnote{For the sake of simplifity in presentation, we do not include explicit dependencies of \( g \) on time \( t \) or parameters \( p \). Moreover, we only consider the case in which the dependency of the original ODE (2.2) on the parameters \( p \) is through its initial conditions only. For details on the derivation in the general case, see [37].}
where $\lambda$, $\mu$, and $s$ are solutions of

\[
\begin{align*}
- \dot{\mu} &= f_y^T \mu + (\lambda^T \otimes I_n) f_{yy}s + g_{yy}s; \quad \mu(t_f) = 0 \\
- \dot{\lambda} &= f_y^T \lambda + g_y^T; \quad \lambda(t_f) = 0 \\
\dot{s} &= f_y s; \quad s(t_0) = y_{0P}u
\end{align*}
\]

(2.24)

In the above equation, $s = y_{P}u$ is a linear combination of the columns of the sensitivity matrix $y_P$. The forward-over-adjoint approach hinges crucially on the fact that $s$ can be computed at the cost of a forward sensitivity analysis with respect to a single parameter (the last ODE problem above) which is possible due to the linearity of the forward sensitivity equations (2.12).

Therefore, the cost of computing the Hessian-vector product is roughly that of two forward and two backward integrations of a system of ODEs of size $N$. For more details, including the corresponding formulas for a pointwise model functional output, see [37].

To allow the forward-over-adjoint approach described above, CVODES provides support for:

- the integration of multiple backward problems depending on the same underlying forward problem (2.2), and

- the integration of backward problems and computation of backward quadratures depending on both the states $y$ and forward sensitivities (for this particular application, $s$) of the original problem (2.2).
Chapter 3

Code Organization

3.1 SUNDIALS organization

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

The various solvers of this family share many subordinate modules. For this reason, it is organized as a family, with a directory structure that exploits that sharing (see Figs. 3.1 and 3.2). The following is a list of the solver packages presently available, and the basic functionality of each:

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

3.2 CVODES organization

The CVODES 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 CVODES package is shown in Figure 3.3. The basic elements of the structure are a module for the basic integration algorithm (including forward sensitivity analysis), a module for adjoint sensitivity analysis, and support for the solution of nonlinear and linear systems that arise in the case of a stiff system. 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.

CVODES utilizes generic linear and nonlinear solver modules defined by the SUNLINSOL API (see Chapter 9) and SUNNONLINSOL API (see Chapter 10), respectively. As such, CVODES 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.
In addition, if forward sensitivity analysis is turned on, the main module will integrate the forward sensitivity equations simultaneously with the original IVP. The sensitivity variables may be included in the local error control mechanism of the main integrator. CVODES provides three different strategies for dealing with the correction stage for the sensitivity variables: CV_SIMULTANEOUS, CV STAGGERED and CV STAGGERED1 (see §2.6 and §5.2.1). The CVODES package includes an algorithm for the approximation of the sensitivity equations right-hand sides by difference quotients, but the user has the option of supplying these right-hand sides directly.

The adjoint sensitivity module (file cvodea.c) provides the infrastructure needed for the backward integration of any system of ODEs which depends on the solution of the original IVP, in particular the adjoint system and any quadratures required in evaluating the gradient of the objective functional. This module deals with the setup of the checkpoints, the interpolation of the forward solution during the backward integration, and the backward integration of the adjoint equations.

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 9). These solvers may utilize a SUNMATRIX object (see Chapter 8) for storing Jacobian information, or they may be matrix-free. Since CVODES can operate on any valid SUNLINSOL implementation, the set of linear solver modules available to CVODES will expand as new SUNLINSOL modules are developed.

Additionally, CVODES 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, CVODES 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, CVODES 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.
3.2 CVODES organization

Figure 3.2: Organization of the SUNDIALS suite
Figure 3.3: Overall structure diagram of the CVODES package. Modules specific to CVODES begin with “CV” (CVLS, CVDIAG, CVBBDPRE, CVBANDPRE, and CVNLS), all other items correspond to generic solver and auxiliary modules. Note also that the LAPACK, KLU and SUPERLU_MT support is through interfaces to external packages. Users will need to download and compile those packages independently.

For preconditioned iterative methods, the preconditioning must be supplied by the user, again in two phases: setup and solve. While there is no default choice of preconditioner analogous to the difference-quotient approximation in the direct case, the references [6, 8], together with the example and demonstration programs included with CVODES, offer considerable assistance in building preconditioners.

CVODES’ 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.

CVODES 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, CVBBDPRE, works in conjunction with NVECTOR_PARALLEL and generates a preconditioner that is a block-diagonal matrix with each block being a banded matrix.

All state information used by CVODES 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 CVODES 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 CVODES memory structure. The reentrancy of CVODES 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 CVODES for IVP Solution

This chapter is concerned with the use of CVODES 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 CVODES user-callable functions and user-supplied functions. This usage is essentially equivalent to using CVODE [27].

The sample programs described in the companion document [43] 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 CVODES package.

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 8) and each SUNLINSOL module (Chapter 9). 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 8 and 9 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.

CVODES 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 CVODES, 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 CVODES. The relevant library files are

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

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

- \texttt{incdir/include/cvodes}
- \texttt{incdir/include/sundials}
- \texttt{incdir/include/nvector}
• `incdir/include/sunmatrix`
• `incdir/include/sunlinsol`
• `incdir/include/sunnonlinsol`

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

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

### 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 `boolentype`, which is used for certain logic operations within SUNDIALS.

#### 4.2.1 Floating point types

The type `realtype` can be `float`, `double`, or `long double`, with the default being `double`. The user can change the precision of the SUNDIALS solvers arithmetic at the configuration stage (see §A.1.2).

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

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

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

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

A user program which uses the type `realtype` and the `RCONST` macro to handle floating-point constants is precision-independent except for any calls to precision-specific standard math library functions. (Our example programs use both `realtype` and `RCONST`.) Users can, however, use the type `double`, `float`, or `long double` in their code (assuming that this usage is consistent with the typedef for `realtype`). Thus, a previously existing piece of ANSI C code can use SUNDIALS without modifying the code to use `realtype`, so long as the SUNDIALS libraries use the correct precision (for details see §A.1.2).

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

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

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

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

4.3 Header files

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

- cvodes/cvodes.h, the main header file for CVODES, which defines the several types and various constants, and includes function prototypes. This includes the header file for CVLS, cvodes/cvodes_ls.h.

Note that cvodes.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 7 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 sundials_sunnonsol_***.h where *** is the name of the nonlinear solver module (see Chapter 10 for more information). This file in turn includes the header file sundials_nonlinearsolver.h which defines the abstract SUNNonlinearSolver data type.

If using a nonlinear solver that requires the solution of a linear system of the form (2.6) (e.g., the default Newton iteration), then a linear solver module header file will be required. The header files corresponding to the various SUNDIALS-provided linear solver modules available for use with CVODES are:

- Direct linear solvers:
  - sundials_sunnlsol_dense.h, which is used with the dense linear solver module, SUNLINSOL DENSE;
  - sundials_sunnlsol_band.h, which is used with the banded linear solver module, SUNLINSOL BAND;
  - sundials_sunnlsol_lapackdense.h, which is used with the LAPACK dense linear solver module, SUNLINSOL LAPACKDENSE;
  - sundials_sunnlsol_lapackband.h, which is used with the LAPACK banded linear solver module, SUNLINSOL LAPACKBAND;
  - sundials_sunnlsol_klu.h, which is used with the KLU sparse linear solver module, SUNLINSOL KLU;
  - sundials_sunnlsol_superlumt.h, which is used with the SUPERLUMT sparse linear solver module, SUNLINSOL SUPERLUMT;

- Iterative linear solvers:
  - sundials_sunnlsol_spgmr.h, which is used with the scaled, preconditioned GMRES Krylov linear solver module, SUNLINSOL SPGMR;
Using CVODES for IVP Solution

- `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;

- `cvodes/cvodes_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 `summatrix/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 `summatrix/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 `summatrix/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 `cvsDiurnal_kryp` example (see [43]), 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 NVVECTOR, SUNMATRIX, SUNLINSOL, and SUNNONLINSOL implementations used. For the steps that are not, refer to Chapters 7, 8, 9, and 10 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 NVVECTOR 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 §7.2-7.5 for details.
For the hypre and PETSc vector wrappers, first create and initialize the underlying vector, and then create an NVVECTOR wrapper with a call of the form \( y_0 = \text{N_VMake}(***(yvec)) \), where \( yvec \) is a hypre or PETSc vector. Note that calls like \( \text{N_New}(***(...) \) and \( \text{N_VGetArrayPointer}(***(...) \) are not available for these vector wrappers. See §7.6 and §7.7 for details.

If using either the CUDA- or RAJA-based vector implementations use a call of the form \( y_0 = \text{N_VMake}(***(..., c) \) where \( c \) is a pointer to a suncudavec or sunrajaavec vector class if this class already exists. Otherwise, create a new vector by making a call of the form \( y_0 = \text{N_VNew}(***(...) \), and then set its elements by accessing the underlying data where it is located with a call of the form \( \text{N_VGetDeviceArrayPointer}(***(...) \) or \( \text{N_VGetHostArrayPointer}(***(...) \). Note that the vector class will allocate memory on both the host and device when instantiated. See §7.8-7.9 for details.

4. Create cvodes object

Call \( \text{cvode\_mem} = \text{CVodeCreate}(1mm) \) to create the CVODES memory block and to specify the linear multistep method. \text{CVodeCreate} returns a pointer to the CVODES memory structure. See §4.5.1 for details.

5. Initialize CVODES solver

Call \( \text{CVodeInit}(...) \) to provide required problem specifications, allocate internal memory for CVODES, and initialize CVODES. \text{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 \( \text{CVodeSStolerances}(...) \) or \( \text{CVodeSVtolerances}(...) \) 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 \( \text{CVodeWFTolerances} \) 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 SUNMATRIX implementation.

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

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

or

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

or

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

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

8. Create linear solver object

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

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

\[
\text{SUNLinearSolver } LS = \text{SUNLinSol}_*(...);
\]
where * can be replaced with “Dense”, “SPGMR”, or other options, as discussed in §4.5.3 and Chapter 9.

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 9 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):
        ier = CVodeSetLinearSolver(...);
    Alternately, if the CVODES-specific diagonal linear solver module, CVDIAG, is desired, initialize the linear solver module and attach it to CVODES with the call
        ier = CVDiag(...);

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

12. Create nonlinear solver object (optional)
    If using a non-default nonlinear solver (see §4.5.4), then create the desired nonlinear solver object by calling the appropriate constructor function defined by the particular SUNNONLINLSOL implementation (e.g., NLS = SUNNonlinSol_***(...); where *** is the name of the nonlinear solver (see Chapter 10 for details).

13. Attach nonlinear solver module (optional)
    If using a non-default nonlinear solver, then initialize the nonlinear solver interface by attaching the nonlinear solver object by calling ier = CVodeSetNonlinearSolver(cvode_mem, NLS); (see §4.5.4 for details).

14. Set nonlinear solver optional inputs (optional)
    Call the appropriate set functions for the selected nonlinear solver module to change optional inputs specific to that nonlinear solver. These must be called after CVodeInit if using the default nonlinear solver or after attaching a new nonlinear solver to CVODE, otherwise the optional inputs will be overridden by CVODES defaults. See Chapter 10 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 ier = CVode(cvode_mem, tout, yout, &tret, itask). Here itask specifies the return mode. The vector yout (which can be the same as the vector 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 yout) by calling the appropriate destructor function defined by the NVECTOR implementation:
4.5 User-callable functions

This section describes the CVODES 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 CVODES. In any case, refer to §4.4 for the correct order of these calls.

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

SUNDIALS provides some linear solvers only as a means for users to get problems running and not as highly efficient solvers. For example, if solving a dense system, we suggest using the LAPACK solvers if the size of the linear system is > 50,000. (Thanks to A. Nicolai for his testing and recommendation.) Table 4.1 shows the linear solver interfaces available as SUNLINSOL modules and the vector implementations required for use. As an example, one cannot use the dense direct solver interfaces with the MPI-based vector implementation. However, as discussed in Chapter 9 the SUNDIALS packages operate on generic SUNLINSOL objects, allowing a user to develop their own solvers should they so desire.
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 CVODES 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 CVODES memory block created and allocated by the first two calls.

#### CVodeCreate

**Call**

```c
CVodeCreate(lmm);  
```

**Description**
The function `CVodeCreate` instantiates a CVODES 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 CVODES see the description of `CVodeSetNonlinearSolver`.

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

#### CVodeInit

**Call**

```c
CVodeInit(cvode_mem, f, t0, y0);  
```

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

**Arguments**
- `cvode_mem` *(void *) pointer to the CVODES 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 CVODES 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.

#### CVodeFree

**Call**

```c
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 CVODES memory block (of type `void *`).

**Return value**
The function `CVodeFree` has no return value.
4.5 User-callable functions

4.5.2 CVODES 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.

**CVodeSSStolerances**

Call
flag = CVodeSSStolerances(cvode_mem, reltol, abstol);

Description The function CVodeSSStolerances specifies scalar relative and absolute tolerances.

Arguments
- cvode_mem (void *) pointer to the CVODES 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 CVodeSSStolerances was successful.
- CV_MEM_NULL The CVODES memory block was not initialized through a previous call to CVodeCreate.
- CV_NO_MALLOC The allocation function CVodeInit has not been called.
- CV_ILL_INPUT One of the input tolerances was negative.

**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 CVODES 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 CVODES memory block was not initialized through a previous call to CVodeCreate.
- CV_NO_MALLOC The allocation function CVodeInit has not been called.
- CV_ILL_INPUT The relative error tolerance was negative or the absolute tolerance had a negative component.

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

**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. (2.8).

Arguments
- cvode_mem (void *) pointer to the CVODES 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 CVODES memory block was not initialized through a previous call to CVodeCreate.
Using CVODES for IVP Solution

CV_NO_MALLOC The allocation function CVodeInit has not been called.

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

1. The scalar relative tolerance \texttt{reltol} is to be set to control relative errors. So \texttt{reltol} = $10^{-4}$ means that errors are controlled to .01%. We do not recommend using \texttt{reltol} larger than $10^{-3}$. On the other hand, \texttt{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 \texttt{abstol} (whether scalar or vector) need to be set to control absolute errors when any components of the solution vector \texttt{y} may be so small that pure relative error control is meaningless. For example, if \texttt{y[i]} starts at some nonzero value, but in time decays to zero, then pure relative error control on \texttt{y[i]} makes no sense (and is overly costly) after \texttt{y[i]} is below some noise level. Then \texttt{abstol} (if scalar) or \texttt{abstol[i]} (if a vector) needs to be set to that noise level. If the different components have different noise levels, then \texttt{abstol} should be a vector. See the example \texttt{cvsRoberts_dns} in the CVODES package, and the discussion of it in the CVODES Examples document [43]. In that problem, the three components vary between 0 and 1, and have different noise levels; hence the \texttt{abstol} vector. It is impossible to give any general advice on \texttt{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 \texttt{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 \texttt{y} returned by CVODES, with magnitude comparable to \texttt{abstol} or less, is equivalent to zero as far as the computation is concerned.

3. The user’s right-hand side routine \texttt{f} should never change a negative value in the solution vector \texttt{y} to a non-negative value, as a ”solution” to this problem. This can cause instability. If the \texttt{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 \texttt{y} vector) for the purposes of computing \texttt{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 (2.6) (e.g., the default Newton iteration), there are two CVODES linear solver interfaces currently available for this task: \texttt{CVLS} and \texttt{CVDIAG}.
The first corresponds to the main linear solver interface in CVODES, 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 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 SPPGMR 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 (2.7).

The cvdiag linear solver is also a direct linear solver, but it only uses a diagonal approximation to $J$.

To specify a generic linear solver to CVODES, after the call to CVodeCreate but before any calls to CVodes, 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

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


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 8 and 9.

Once this solver object has been constructed, the user should attach it to CVODES via a call to CVodeSetLinearSolver. The first argument passed to this function is the CVODES 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 CVODES integrator, and allows the user to specify additional parameters and routines pertinent to their choice of linear solver.

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

```c
CVodeSetLinearSolver
```

<table>
<thead>
<tr>
<th>Call</th>
<th><code>flag = CVodeSetLinearSolver(cvode_mem, LS, J);</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>The function CVodeSetLinearSolver attaches a generic SUNLINSOL object LS and corresponding template Jacobian SUNMATRIX object J to CVODES, initializing the CVLS linear solver interface.</td>
</tr>
<tr>
<td>Arguments</td>
<td><code>cvode_mem</code> (void *) pointer to the CVODES memory block.</td>
</tr>
<tr>
<td></td>
<td><code>LS</code> (SUNLinearSolver) SUNLINSOL object to use for solving linear systems of the form (2.6).</td>
</tr>
<tr>
<td></td>
<td><code>J</code> (SUNMatrix) SUNMATRIX object for used as a template for the Jacobian (or NULL if not applicable).</td>
</tr>
<tr>
<td>Return value</td>
<td>The return value <code>flag</code> (of type <code>int</code>) is one of</td>
</tr>
<tr>
<td></td>
<td><code>CVLS_SUCCESS</code> The CVLS initialization was successful.</td>
</tr>
</tbody>
</table>
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 nVECTOR 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 8 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.

CVDiag

Call
flag = CVDiag(cvode_mem);

Description
The function CVDiag selects the CVDIAG linear solver.
The user’s main program must include the cvodes_diag.h header file.

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

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

CVDIAG_SUCCESS The CVDIAG initialization was successful.
CVDIAG_MEM_NULL The cvode_mem pointer is NULL.
CVDIAG_ILL_INPUT The CVDIAG solver is not compatible with the current nVECTOR module.
CVDIAG_MEM_FAIL A memory allocation request failed.

Notes
The CVDIAG solver is the simplest of all of the available CVODES linear solver interfaces.
The CVDIAG solver uses an approximate diagonal Jacobian formed by way of a difference quotient. The user does not have the option of supplying a function to compute an approximate diagonal Jacobian.

4.5.4 Nonlinear solver interface function

By default CVODES uses the SUNNONLINSOL implementation of Newton’s method defined by the SUNNONLINSOL_NEWTON module (see §10.2). To specify a different nonlinear solver in CVODES, 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 CVODES, CVodeSetNonlinearSolver must be called after CVodeInit. If any calls to CVode have been made, then CVODES 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 CVODES memory pointer returned by CVodeCreate and the second argument is the SUNNONLINSOL object to use for
solving the nonlinear system (2.4) or (2.5). A call to this function attaches the nonlinear solver to the main CVODES integrator.

### CVodeSetNonlinearSolver

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

**Description**
The function `CVodeSetNonlinearSolver` attaches a SUNNONLINSOL object (NLS) to CVODES.

**Arguments**
- `cvode_mem` (void *) pointer to the CVODES memory block.
- `NLS` (SUNNonlinearSolver) SUNNONLINSOL object to use for solving nonlinear systems (2.4) or (2.5).

**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.

**Notes**
- When forward sensitivity analysis capabilities are enabled and the `CV_STAGGERED` or `CV_STAGGERED1` corrector method is used this function sets the nonlinear solver method for correcting state variables (see §5.2.3 for more details).

### 4.5.5 Rootfinding initialization function

While solving the IVP, CVODES 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`.

### 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 CVODES 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.

### 4.5.6 CVODES 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 CVODES 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 CVODES 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 interpolates 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 g_i were found to have a root.

CV_MEM_NULL The cvode_mem argument was NULL.

CV_NO_MALLOC The CVODES 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 t_0 and the output time t_out 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 (MXNCF = 10) during one internal time step, or with |h| = h_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.
The inequality constraints were violated and the solver was unable to recover.

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

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

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.

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.

The rootfinding function failed.

The vector \( y_{out} \) can occupy the same space as the vector \( y_0 \) of initial conditions that was passed to \texttt{CVodeInit}.

In the \texttt{CV\_ONE\_STEP} mode, \( tout \) 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 \texttt{CVodeSetStopTime}), then \texttt{CVode} returns the solution at \( t_{stop} \). Once the integrator returns at a stop time, any future testing for \( t_{stop} \) is disabled (and can be reenabled only through a new call to \texttt{CVodeSetStopTime}).

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

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

### 4.5.7 Optional input functions

There are numerous optional input parameters that control the behavior of the \texttt{CVODES} solver. \texttt{CVODES} 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 \texttt{CVODES} which are then described in detail in the remainder of this section, beginning with those for the main \texttt{CVODES} 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 \texttt{CVODES}, 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.

#### 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 \texttt{CVodeSetErrFile} or \texttt{CVodeSetErrHandlerFn} is to be called, that call should be first, in order to take effect for any later error message.

### CVodeSetErrFile

**Call**

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

**Description**

The function \texttt{CVodeSetErrFile} specifies a pointer to the file where all \texttt{CVODES} messages should be directed when the default \texttt{CVODES} error handler function is used.
Table 4.2: Optional inputs for CVODES 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 $t_{out}$</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 $t_{stop}$</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

Arguments

- `cvode_mem` (void *) pointer to the CVODES 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 CVODES 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.

```c
CVodeSetErrHandlerFn
```

Call

```c
flag = CVodeSetErrHandlerFn(cvode_mem, ehfun, 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 CVODES 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 CVODES solver memory is NULL will always be directed to `stderr`.
- 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.

```c
CVodeSetUserData
```

Call

```c
flag = CVodeSetUserData(cvode_mem, user_data);
```

Description

The function `CVodeSetUserData` specifies the user data block `user_data` and attaches it to the main CVODES memory block.

Arguments

- `cvode_mem` (void *) pointer to the CVODES 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.

```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 CVODES 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 \( \leq 0 \), 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 CVODES memory block, its value cannot be increased past its previous value. An input value greater than the default will result in the default value.

**CVodeSetMaxNumSteps**

Call 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 CVODES 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 CVODES using the default value (500). Passing mxsteps < 0 disables the test (not recommended).

**CVodeSetMaxHnilWarns**

Call 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 CVODES memory block.
- mxhnil (int) maximum number of warning messages (> 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. A negative value for mxhnil indicates that no warning messages should be issued.

**CVodeSetStabLimDet**

Call 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 CVODES memory block.
- stldet (boolean type) flag controlling stability limit detection (SUNTRUE = on; SUNFALSE = off).

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 linear multistep method is not set to CV_BDF.
4.5 User-callable functions

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.

**CVodeSetInitStep**

Call flag = CVodeSetInitStep(cvode_mem, hin);

Description The function CVodeSetInitStep specifies the initial step size.

Arguments cvode_mem (void *) pointer to the CVODES 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

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

Notes By default, CVODES estimates the initial step size to be the solution h of the equation \[ \|0.5h^2\dot{y}\|_{WRMS} = 1 \], where \( \dot{y} \) is an estimated second derivative of the solution at \( t_0 \).

**CVodeSetMinStep**

Call 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 CVODES 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

- CV_SUCCESS The optional value has been successfully set.
- CV_MEM_NULL The cvode_mem pointer is NULL.
- CV_ILL_INPUT Either hmin is nonpositive or it exceeds the maximum allowable step size.

Notes The default value is 0.0.

**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 CVODES 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_ILL_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 \( \infty \).
**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 CVODES 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_ILL_INPUT` The value of `tstop` is not beyond the current `t` value, `tn`.

**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`).

**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 CVODES 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.

**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 CVODES 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_MEM_FAIL` The `sunnonlinsol` module is NULL.

**Notes**
The default value is 3.

**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 CVODES 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.

**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 CVODES 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.

Notes The default value is 0.1.

**CVodeSetIterType**

Call `flag = CVodeSetIterType(cvode_mem, iter);`

Description The function CVodeSetIterType resets the nonlinear solver iteration type to `iter`.

Arguments
- `cvode_mem` (void *) pointer to the CVODES memory block.
- `iter` (int) specifies the type of nonlinear solver iteration and may be either CV_NEWTON or CV_FUNCTIONAL.

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 `iter` value passed is neither CV_NEWTON nor CV_FUNCTIONAL.

Notes The nonlinear solver iteration type is initially specified in the call to CVodeCreate (see §4.5.1). This function call is needed only if `iter` is being changed from its value in the prior call to CVodeCreate.

**CVodeSetConstraints**

Call `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 CVODES 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 ≥ 0.0`.
  - -1.0 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`.
  - -2.0 then `y_i` will be constrained to be `y_i < 0.0`.

Return value The return value of 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 or the simultaneous corrector option has been selected when doing forward sensitivity analysis.

Notes The presence of a non-NULL constraints vector that is not 0.0 in all components will cause constraint checking to be performed. However, a call with 0.0 in all components of constraints will result in an illegal input return. A NULL constraints vector will disable constraint checking.
Constraint checking when doing forward sensitivity analysis with the simultaneous corrector option is currently disallowed and will result in an illegal input return.

4.5.7.2 Linear solver interface optional input functions

The mathematical explanation of the linear solver methods available to CVODES 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, CVODES 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, CVODES 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 $J(t,y) = \frac{df(t,y)}{dy}$.

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.

```c
CVodeSetMaxStepsBetweenJac
```

Call `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.

Arguments `cvode_mem (void *)` pointer to the CVODES 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.

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.
4.5 User-callable functions

**CVodeSetJacFn**

Call

```c
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 CVODES 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.

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 `CVLlsJacTimesVecFn` and can be specified through a call to `CVodeSetJacTimes` (see §4.6.6 for specification details). The evaluation and processing of any Jacobian-related data needed by the user’s Jacobian-times-vector function is done in the optional user-supplied function `jtsetup` (see §4.6.7 for specification details).

The pointer `user_data` received through `CVodeSetUserData` (or a pointer to `NULL` if `user_data` was not specified) is passed to the Jacobian-times-vector setup and product functions, `jtsetup` and `jtimes`, each time they are called. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied functions without using global data in the program.

**CVodeSetJacTimes**

Call

```c
flag = CVodeSetJacTimes(cvode_mem, jtsetup, jtimes);
```

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

Arguments
- `cvode_mem` (void *) pointer to the CVODES 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_SUNLS_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.

When using an iterative linear solver, the user may supply a preconditioning operator to aid in solution of the system. This operator consists of two user-supplied functions, psetup and psolve, that are supplied to CVODES using the function CVodeSetPreconditioner. The psetup function supplied to this routine should handle evaluation and preprocessing of any Jacobian data needed by the user’s preconditioner solve function, psolve. The user data pointer received through CVodeSetUserData (or a pointer to NULL if user data was not specified) is passed to the psetup and psolve functions. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied preconditioner functions without using global data in the program. Also, as described in §2.1, the 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 CVodeSetEpsLin function.

```
CVodeSetPreconditioner
```

Call

flag = CVodeSetPreconditioner(cvode_mem, psetup, psolve);

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

Arguments

- `cvode_mem` (void *) pointer to the CVODES 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_SUNLS_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.
4.5 User-callable functions

**CVodeSetEpsLin**

Call `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 CVODES 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_ILL_INPUT** 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-compatibility. However, this will be deprecated in future releases, so we recommend that users transition to the new routine name soon.

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 `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 CVODES 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_ILL_INPUT** rootfinding has not been activated through a call to **CVodeRootInit**.

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

**CVodeSetNoInactiveRootWarn**

Call `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 CVODES memory block.

Return value The return value `flag` (of type `int`) is one of
- **CV_SUCCESS** The optional value has been successfully set.
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 CVODES 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**
The return value `flag` (of type int) is one of
- `CV_SUCCESS` `CVodeGetDky` succeeded.
- `CV_BAD_K` $k$ is not in the range 0, 1, ..., $q_u$.
- `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.

4.5.9 Optional output functions

CVODES provides an extensive set of functions that can be used to obtain solver performance information. Table 4.3 lists all optional output functions in CVODES, 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 CVODES 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 `nmiters/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/nmiters` (in the case of a matrix-based linear solver), and the ratio `npevals/nmiters` (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, \( n\text{jevals}/n\text{niters} \) 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 \( n\text{liters}/n\text{niters} \) 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

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

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

Arguments

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

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` sets 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

`cvodes` 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 `cvodes` 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 `cvodes` nonlinear solver used. As a convenience, additional information extraction functions provide the optional outputs in groups. These optional output functions are described next.
Table 4.3: Optional outputs from CVODES, CVLS, and CVDIAG

<table>
<thead>
<tr>
<th>Optional output</th>
<th>CVODES main solver</th>
<th>Function name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of CVODES real and integer workspaces</td>
<td>CVodeGetWorkSpace</td>
<td>cvodes</td>
</tr>
<tr>
<td>Cumulative number of internal steps</td>
<td>CVodeGetNumSteps</td>
<td>cvodes</td>
</tr>
<tr>
<td>No. of calls to r.h.s. function</td>
<td>CVodeGetNumRhsEvals</td>
<td>cvodes</td>
</tr>
<tr>
<td>No. of calls to linear solver setup function</td>
<td>CVodeGetNumLinSolvSetups</td>
<td>cvodes</td>
</tr>
<tr>
<td>No. of local error test failures that have occurred</td>
<td>CVodeGetNumErrTestFails</td>
<td>cvodes</td>
</tr>
<tr>
<td>Order used during the last step</td>
<td>CVodeGetLastOrder</td>
<td>cvodes</td>
</tr>
<tr>
<td>Order to be attempted on the next step</td>
<td>CVodeGetCurrentOrder</td>
<td>cvodes</td>
</tr>
<tr>
<td>No. of order reductions due to stability limit detection</td>
<td>CVodeGetNumStabLimOrderReds</td>
<td>cvodes</td>
</tr>
<tr>
<td>Actual initial step size used</td>
<td>CVodeGetActualInitStep</td>
<td>cvodes</td>
</tr>
<tr>
<td>Step size used for the last step</td>
<td>CVodeGetLastStep</td>
<td>cvodes</td>
</tr>
<tr>
<td>Step size to be attempted on the next step</td>
<td>CVodeGetCurrentStep</td>
<td>cvodes</td>
</tr>
<tr>
<td>Current internal time reached by the solver</td>
<td>CVodeGetCurrentTime</td>
<td>cvodes</td>
</tr>
<tr>
<td>Suggested factor for tolerance scaling</td>
<td>CVodeGetTolScaleFactor</td>
<td>cvodes</td>
</tr>
<tr>
<td>Error weight vector for state variables</td>
<td>CVodeGetErrWeights</td>
<td>cvodes</td>
</tr>
<tr>
<td>No. of nonlinear solver iterations</td>
<td>CVodeGetNumNonlinSolvIters</td>
<td>cvodes</td>
</tr>
<tr>
<td>No. of nonlinear convergence failures</td>
<td>CVodeGetNumNonlinSolvConvFails</td>
<td>cvodes</td>
</tr>
<tr>
<td>All CVODES integrator statistics</td>
<td>CVodeGetIntegratorStats</td>
<td>cvodes</td>
</tr>
<tr>
<td>cvodes nonlinear solver statistics</td>
<td>CVodeGetNonlinSolvStats</td>
<td>cvodes</td>
</tr>
<tr>
<td>Array showing roots found</td>
<td>CVodeGetRootInfo</td>
<td>cvodes</td>
</tr>
<tr>
<td>No. of calls to user root function</td>
<td>CVodeGetNumGEvals</td>
<td>cvodes</td>
</tr>
<tr>
<td>Name of constant associated with a return flag</td>
<td>CVodeGetReturnFlagName</td>
<td>cvodes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Optional output</th>
<th>CVLS linear solver interface</th>
<th>Function name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of real and integer workspaces</td>
<td>CVodeGetWorkSpace</td>
<td>cvodes</td>
</tr>
<tr>
<td>No. of Jacobian evaluations</td>
<td>CVodeGetNumJacEvals</td>
<td>cvodes</td>
</tr>
<tr>
<td>No. of r.h.s. calls for finite diff. Jacobian[-vector] evals.</td>
<td>CVodeGetNumLinRhsEvals</td>
<td>cvodes</td>
</tr>
<tr>
<td>No. of linear iterations</td>
<td>CVodeGetNumLinIters</td>
<td>cvodes</td>
</tr>
<tr>
<td>No. of linear convergence failures</td>
<td>CVodeGetNumLinConvFails</td>
<td>cvodes</td>
</tr>
<tr>
<td>No. of preconditioner evaluations</td>
<td>CVodeGetNumPrecEvals</td>
<td>cvodes</td>
</tr>
<tr>
<td>No. of preconditioner solves</td>
<td>CVodeGetNumPrecSolves</td>
<td>cvodes</td>
</tr>
<tr>
<td>No. of Jacobian-vector setup evaluations</td>
<td>CVodeGetNumJSetupEval</td>
<td>cvodes</td>
</tr>
<tr>
<td>No. of Jacobian-vector product evaluations</td>
<td>CVodeGetNumJtimesEval</td>
<td>cvodes</td>
</tr>
<tr>
<td>Last return from a linear solver function</td>
<td>CVodeGetLastLinFlag</td>
<td>cvodes</td>
</tr>
<tr>
<td>Name of constant associated with a return flag</td>
<td>CVodeGetLinReturnFlagName</td>
<td>cvodes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Optional output</th>
<th>CVDIAG linear solver interface</th>
<th>Function name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of CVDIAG real and integer workspaces</td>
<td>CVDiagGetWorkSpace</td>
<td>cvdiag</td>
</tr>
<tr>
<td>No. of r.h.s. calls for finite diff. Jacobian evals.</td>
<td>CVDiagGetNumRhsEvals</td>
<td>cvdiag</td>
</tr>
<tr>
<td>Last return from a CVDIAG function</td>
<td>CVDiagGetLastFlag</td>
<td>cvdiag</td>
</tr>
<tr>
<td>Name of constant associated with a return flag</td>
<td>CVDiagGetReturnFlagName</td>
<td>cvdiag</td>
</tr>
</tbody>
</table>
4.5 User-callable functions

CVodeGetWorkSpace

Call

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

Description

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

Arguments

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

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

leniw (long int) the number of integer values in the CVODES 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 \( \text{maxord} \), and the number \( \text{nrtfn} \) of root functions (see §4.5.5), the actual size of the real workspace, in realtype words, is given by the following:

- base value: \( \text{lenrw} = 96 + (\text{maxord}+5) \times N_r + 3 \times \text{nrtfn} \);
- using CVodeSVtolerances: \( \text{lenrw} = \text{lenrw} + N_r \);
- with constraint checking (see CVodeSetConstraints): \( \text{lenrw} = \text{lenrw} + N_r \);

where \( N_r \) is the number of real words in one \( \text{N_Vector} \) (\( \approx N \)).

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

- base value: \( \text{leniw} = 40 + (\text{maxord}+5) \times N_i + \text{nrtfn} \);
- using CVodeSVtolerances: \( \text{leniw} = \text{leniw} + N_i \);
- with constraint checking: \( \text{leniw} = \text{leniw} + N_i \);

where \( N_i \) is the number of integer words in one \( \text{N_Vector} \) (\( = 1 \) for \( \text{nvector_serial} \) and \( 2 \times \text{npes} \) for \( \text{nvector_parallel} \) and \( \text{npes} \) processors).

For the default value of \( \text{maxord} \), no rootfinding, no constraints, and without using CVodeSVtolerances, these lengths are given roughly by:

- For the Adams method: \( \text{lenrw} = 96 + 17N \) and \( \text{leniw} = 57 \)
- For the BDF method: \( \text{lenrw} = 96 + 10N \) and \( \text{leniw} = 50 \)

Note that additional memory is allocated if quadratures and/or forward sensitivity integration is enabled. See §4.7.1 and §5.2.1 for more details.

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 CVODES memory block.

nsteps (long int) number of steps taken by CVODES.

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.
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 CVODES 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.

CVodeGetNumLinSolvSetups

Call 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 CVODES 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.

CVodeGetNumErrTestFails

Call 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 CVODES 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.

CVodeGetLastOrder

Call 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 CVODES 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.
4.5 User-callable functions

**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 CVODES 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.

**CVodeGetLastStep**

Call

```c
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 CVODES 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.

**CVodeGetCurrentStep**

Call

```c
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 CVODES 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.

**CVodeGetActualInitStep**

Call

```c
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 CVODES 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 CVODES to ensure that the step size is within the prescribed bounds ($h_{\text{min}} \leq h_0 \leq h_{\text{max}}$), or to satisfy the local error test condition.
Using CVODES for IVP Solution

**CVodeGetCurrentTime**

Call  
\[
\text{flag} = \text{CVodeGetCurrentTime} (\text{cvode\_mem}, \&\text{tcur});
\]

Description  
The function \text{CVodeGetCurrentTime} returns the current internal time reached by the solver.

Arguments  
\text{cvode\_mem} \ (\text{void *}) \ \text{pointer to the CVODES memory block.}
\text{tcur} \ (\text{realtype}) \ \text{current internal time reached.}

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

**CVodeGetNumStabLimOrderReds**

Call  
\[
\text{flag} = \text{CVodeGetNumStabLimOrderReds} (\text{cvode\_mem}, \&\text{nslred});
\]

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

Arguments  
\text{cvode\_mem} \ (\text{void *}) \ \text{pointer to the CVODES memory block.}
\text{nslred} \ (\text{long int}) \ \text{number of order reductions due to stability limit detection.}

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

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

**CVodeGetTolScaleFactor**

Call  
\[
\text{flag} = \text{CVodeGetTolScaleFactor} (\text{cvode\_mem}, \&\text{tolsfac});
\]

Description  
The function \text{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  
\text{cvode\_mem} \ (\text{void *}) \ \text{pointer to the CVODES memory block.}
\text{tolsfac} \ (\text{realtype}) \ \text{suggested scaling factor for user-supplied tolerances.}

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

**CVodeGetErrWeights**

Call  
\[
\text{flag} = \text{CVodeGetErrWeights} (\text{cvode\_mem}, \text{eweight});
\]

Description  
The function \text{CVodeGetErrWeights} returns the solution error weights at the current time. These are the reciprocals of the \(W_i\) given by (2.8).

Arguments  
\text{cvode\_mem} \ (\text{void *}) \ \text{pointer to the CVODES memory block.}
\text{eweight} \ (\text{N\_Vector}) \ \text{solution error weights at the current time.}

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

Notes  
The user must allocate memory for \text{eweight}. 

⚠️
### CVodeGetEstLocalErrors

**Call**
```c
flag = CVodeGetEstLocalErrors(cvode_mem, ele);
```

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

**Arguments**
- **cvode_mem** (void *) pointer to the CVODES 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 `cvode_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 `eweight[i]*ele[i]`.

### CVodeGetIntegratorStats

**Call**
```c
flag = CVodeGetIntegratorStats(cvode_mem, &nsteps, &nfevals,
    &nlinsetups, &netfails, &qlast, &qcur,
    &hinused, &hlast, &hcur, &tcur);
```

**Description**
The function `CVodeGetIntegratorStats` returns the CVODES integrator statistics as a group.

**Arguments**
- **cvode_mem** (void *) pointer to the CVODES memory block.
- **nsteps** (long int) number of steps taken by CVODES.
- **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** (realtype) actual value of initial step size.
- **hlast** (realtype) step size taken on the last internal step.
- **hcur** (realtype) step size to be attempted on the next internal step.
- **tcur** (realtype) current internal time reached.

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

### CVodeGetNumNonlinSolvIters

**Call**
```c
flag = CVodeGetNumNonlinSolvIters(cvode_mem, &nniters);
```

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

**Arguments**
- **cvode_mem** (void *) pointer to the CVODES memory block.
- **nniters** (long int) number of nonlinear iterations performed.

**Return value**
The return value `flag` (of type `int`) is one of:...
CVODES for IVP Solution

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.

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 CVODES 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.

CVodeGetNonlinSolvStats
Call flag = CVodeGetNonlinSolvStats(cvode_mem, &nniters, &nncfails);
Description The function CVodeGetNonlinSolvStats returns the CVODES nonlinear solver statistics as a group.
Arguments cvode_mem (void *) pointer to the CVODES memory block.
nniters (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.

CVodeGetReturnFlagName
Call name = CVodeGetReturnFlagName(flag);
Description The function CVodeGetReturnFlagName returns the name of the CVODES constant corresponding to flag.
Arguments The only argument, of type int, is a return flag from a CVODES 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 CVODES memory block.
rootsfound (int *) array of length nrtfn with the indices of the user functions \( g_i \) found to have a root. For \( i = 0, \ldots, \text{nrtfn} - 1 \), \( \text{rootsfound}[i] \neq 0 \) if \( g_i \) has a root, and = 0 if not.
4.5 User-callable functions

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 \(g_i\) for which a root was found, the sign of \(\text{rootsfound}[i]\) indicates the direction of zero-crossing. A value of +1 indicates that \(g_i\) is increasing, while a value of −1 indicates a decreasing \(g_i\).

The user must allocate memory for the vector \(\text{rootsfound}\).

### 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 CVODES memory block.
- ngevals (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.

### 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 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**  
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 CVODES memory block.
- lenrwLS (long int) the number of realltype 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 allocated within this interface and to memory allocated by the SUNLINSOL object attached to it. The template Jacobian matrix allocated by the user outside of CVLS is not included in this report.

The previous routines CVDlsGetWorkspace and CVSpilsGetWorkspace 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.
Using CVODES for IVP Solution

**CVodeGetNumJacEvals**

Call  
\[
\text{flag} = \text{CVodeGetNumJacEvals}(\text{cvode\_mem, njevals});
\]

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

Arguments  
- \( \text{cvode\_mem} \) (void *) pointer to the CVODES memory block.
- \( \text{njevals} \) (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, this will be deprecated in future releases, so we recommend that users transition to the new routine name soon.

**CVodeGetNumLinRhsEvals**

Call  
\[
\text{flag} = \text{CVodeGetNumLinRhsEvals}(\text{cvode\_mem, nfevalsLS});
\]

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  
- \( \text{cvode\_mem} \) (void *) pointer to the CVODES memory block.
- \( \text{nfevalsLS} \) (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 CVSplsGetNumRhsEvals 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.

**CVodeGetNumLinIters**

Call  
\[
\text{flag} = \text{CVodeGetNumLinIters}(\text{cvode\_mem, nliters});
\]

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

Arguments  
- \( \text{cvode\_mem} \) (void *) pointer to the CVODES memory block.
- \( \text{nliters} \) (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 CVSplsGetNumLinIters is now a wrapper for this routine, and may still be used for backward-compatibility. However, this will be deprecated in future releases, so we recommend that users transition to the new routine name soon.
4.5 User-callable functions

**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.

**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 CVODES 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.

**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 CVODES 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.
Using CVODES for IVP Solution

**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**
code

- `cvode_mem` (void *) pointer to the CVODES 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.

**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**
code

- `cvode_mem` (void *) pointer to the CVODES 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.

**CVodeGetLastLinFlag**

**Call**

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

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

**Arguments**
code

- `cvode_mem` (void *) pointer to the CVODES 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 `CVLS_SETUP_FAIL`) when using the SUNLINSOL_DENSE or SUNLINSOL_BAND modules, then the value of `lsflag` is equal to the column index (numbered from one) at which a zero diagonal element was encountered during the LU factorization of the (dense or banded) Jacobian matrix.

If the CVLS setup function failed when using another SUNLINSOL module, then `lsflag` will be `SUNLS_PSET_FAIL_UNREC`, `SUNLSASET_FAIL_UNREC`, or `SUNLS_PACKAGE_FAIL_UNREC`. 
If the cvls solve function failed (i.e., CVode returned CV_LSOLVE_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_TIMES_FAIL_UNREC, indicating an unrecoverable failure in the \( J v \) function;
- SUNLS_PSOLVE_FAIL_UNREC, indicating that the preconditioner solve function \( psolve \) failed unrecoverably;
- SUNLS_GS_FAIL, indicating a failure in the Gram-Schmidt procedure (spgmr and spfgmr only);
- SUNLS_QRSOL_FAIL, indicating that the matrix \( R \) was found to be singular during the QR solve phase (spgmr and spfgmr only); or
- SUNLS_PACKAGE_FAIL_UNREC, indicating an unrecoverable failure in an external iterative linear solver package.

The previous routines CVDlsGetLastFlag 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.

### CVodeGetLinReturnFlagName

```c
Call 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

```c
Call 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 CVODES 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 cvode_mem 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.
Using CVODES for IVP Solution

CVDiagGetNumRhsEvals

Call 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 CVODES 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).

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 CVODES 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).

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 CVODES reinitialization function

The function CVodeReInit reinitializes the main CVODES solver for the solution of a new problem, where a prior call to CVodeInit 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.
4.6 User-supplied functions

if the multistep method parameter \texttt{lmm} is unchanged (or changed from \texttt{CV_ADAMS} to \texttt{CV_BDF}) and the default value for \texttt{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 \texttt{cvls} interface routines, as described in §4.5.3. Otherwise, all solver inputs set previously remain in effect.

One important use of the \texttt{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 \texttt{CVodeReInit}. To stop when the location of the discontinuity is known, simply make that location a value of \texttt{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 \emph{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 \texttt{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.

\begin{verbatim}
CVodeReInit
Call flag = CVodeReInit(cvode_mem, t0, y0);  
Description The function CVodeReInit provides required problem specifications and reinitializes CVODES.
Arguments cvode_mem (void *) pointer to the CVODES 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 CVODES memory block was not initialized through a previous call to CVodeCreate.
   CV_NO_MALLOC Memory space for the CVODES 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.
\end{verbatim}

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.

\subsection*{4.6.1 ODE right-hand side}

The user must provide a function of type \texttt{CVRhsFn} defined as follows:

\begin{verbatim}
CVRhsFn
Definition typedef int (*CVRhsFn)(realtype t, N_Vector y, N_Vector ydot, void *user_data);
\end{verbatim}
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 \( f(t, y) \).
- \( \text{user_data} \) is the \( \text{user_data} \) pointer passed to \text{CVodeSetUserData}.

Return value

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

Notes

Allocation of memory for \( ydot \) is handled within \text{CVODES}.

A recoverable failure error return from the \text{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, \text{CVODES} 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.) However, if the user program also includes quadrature integration, the state variables can be checked for legality in the call to \text{CVQuadRhsFn}, which is called at the converged solution of the nonlinear system, and therefore \text{CVODES} can be flagged to attempt to recover from such a situation. Also, if sensitivity analysis is performed with one of the staggered methods, the ODE right-hand side function is called at the converged solution of the nonlinear system, and a recoverable error at that point can be flagged, and \text{CVODES} will then try to correct it.

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 \text{CVRhsFn} (in which case \text{CVODES} returns \text{CV_FIRST_RHSFUNC_ERR}). The other is when a recoverable error is reported by \text{CVRhsFn} after an error test failure, while the linear multistep method order is equal to 1 (in which case \text{CVODES} returns \text{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 \text{errfp} (see \text{CVodeSetErrFile}), the user may provide a function of type \text{CVErrHandlerFn} to process any such messages. The function type \text{CVErrHandlerFn} is defined as follows:

\[\text{CVErrHandlerFn}\]

Definition

\[
\text{typedef void (*CVErrHandlerFn)(int error\_code, const char *module, const char *function, char *msg, void *eh\_data);}
\]

Purpose

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

Arguments

- \text{error\_code} is the error code.
- \text{module} is the name of the \text{CVODES} module reporting the error.
- \text{function} is the name of the function in which the error occurred.
- \text{msg} is the error message.
- \text{eh\_data} is a pointer to user data, the same as the \text{eh\_data} parameter passed to \text{CVodeSetErrHandlerFn}.
Return value A CVErrHandlerFn function has no return value.

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

4.6.3 Error weight function

As an alternative to providing the relative and absolute tolerances, the user may provide a function of type CVEwtFn to compute a vector ewt containing the weights in the WRMS norm $\|v\|_{WRMS} = \sqrt{(1/N) \sum_{i=1}^{N} (W_i \cdot v_i)^2}$. These weights will be used in place of those defined by Eq. (2.8). The function type CVEwtFn is defined as follows:

```
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 cvodes.

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:

```
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 cvodes.

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 \( J = \frac{\partial f}{\partial y} \) (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 \( \text{SUNMatrix} \)).
\( user\_data \) is a pointer to user data, the same as the \( user\_data \) parameter passed to 
\text{CVodeSetUserData}.

\( tmp1 \)
\( tmp2 \)
\( tmp3 \) are pointers to memory allocated for variables of type \( \text{N\_Vector} \) 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 oc-
curred (in which case CVODES will attempt to correct, while CVLS sets \( \text{last}\_flag \) to
\text{CVLS\_JACFUNC\_RECVR}), or a negative value if it failed unrecoverably (in which case the
integration is halted, CVode returns \text{CVLSETUP\_FAIL} and CVLS sets \( \text{last}\_flag \) to
\text{CVLS\_JACFUNC\_UNRECVR}).

notes Information regarding the structure of the specific \text{sunmatrix} structure (e.g. number
of rows, upper/lower bandwidth, sparsity type) may be obtained through using the
implementation-specific \text{sunmatrix} interface functions (see Chapter 8 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 \text{cv\_mem}
to \( user\_data \) and then use the \text{CVodeGet\*} functions described in \S 4.5.9.2. The unit
roundoff can be accessed as \text{UNIT\_ROUNDOFF} defined in \text{sundials\_types.h}.

dense:
A user-supplied dense Jacobian function must load the \( N \) by \( N \) dense matrix \( Jac \) with
an approximation to the Jacobian matrix \( J(t,y) \) at the point \( (t, y) \). The accessor
macros \text{SM\_ELEMENT\_D} and \text{SM\_COLUMN\_D} allow the user to read and write dense matrix
elements without making explicit references to the underlying representation of the \text{sun-
matrix\_dense} type. \text{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 state-
ment \text{SM\_ELEMENT\_D}(J, m-1, n-1) = J_{m,n}. Alternatively, \text{SM\_COLUMN\_D}(J, j) returns
a pointer to the first element of the \( j \)-th column of \( Jac \) (with \( j = 0 \ldots N - 1 \)), and the
elements of the \( j \)-th column can then be accessed using ordinary array indexing. Con-
sequently, \( J_{m,n} \) can be loaded using the statements \text{col}_n = \text{SM\_COLUMN\_D}(J, n-1); \text{col}_n[m-1] = J_{m,n}. For large problems, it is more efficient to use \text{SM\_COLUMN\_D} than to
use \text{SM\_ELEMENT\_D}. Note that both of these macros number rows and columns starting
from 0. The \text{SUNMATRIX\_DENSE} type and accessor macros are documented in \S 8.2.

banded:
A user-supplied banded Jacobian function must load the \( N \) by \( N \) 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 -mupper ≤ m-n ≤ mlower. Alternatively, SM_COLUMN_B(J, j) returns a pointer to the diagonal element of the j-th column of Jac, and if we assign this address to realtype *col_j, then the i-th element of the j-th column is given by SM_COLUMN_ELEMENT_B(col_j, i, j), counting from 0. Thus, for (m, n) within the band, J_{m,n} can be loaded by setting col_n = SM_COLUMN_B(J, n-1); SM_COLUMN_ELEMENT_B(col_n, m-1, n-1) = J_{m,n}. The elements of the j-th column can also be accessed via ordinary array indexing, but this approach requires knowledge of the underlying storage for a band matrix of type SUNMATRIX_BAND. The array col_n can be indexed from -mupper to mlower. For large problems, it is more efficient to use SM_COLUMN_B and SM_COLUMN_ELEMENT_B than to use the SM_ELEMENT_B macro. As in the dense case, these macros all number rows and columns starting from 0. The SUNMATRIX_BAND type and accessor macros are documented in §8.3.

sparse:
A user-supplied sparse Jacobian function must load the N by N compressed-sparse-column or compressed-sparse-row matrix Jac with an approximation to the Jacobian matrix J(t, y) at the point (t, y). Storage for Jac already exists on entry to this function, although the user should ensure that sufficient space is allocated in Jac to hold the nonzero values to be set; if the existing space is insufficient the user may reallocate the data and index arrays as needed. The amount of allocated space in a SUNMATRIX_SPARSE object may be accessed using the macro SM_NNZ_S or the routine SUNSparseMatrix_NNZ. The SUNMATRIX_SPARSE type and accessor macros are documented in §8.4.

The previous function type CVDlsJacFn is identical to 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 NULL-valued SUNMATRIX was supplied to CVodeSetLinearSolver), the user may provide a function of type 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 typedef int (*CVLsJacTimesVecFn)(N_Vector v, N_Vector Jv, realtype t, N_Vector y, N_Vector fy, void *user_data, N_Vector tmp);

Purpose This function computes the product Jv = (∂f/∂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).
user_data is a pointer to user data, the same as the user_data parameter passed to CVodeSetUserData.
tmp is a pointer to memory allocated for a variable of type N_Vector which can be used for work space.

Return value The value 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 current value of $J$, i.e. as evaluated at the current $(t, y)$. If the user’s 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 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 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 User-supplied functions

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 \texttt{CVLsPrecSolveFn}, defined as follows:

\begin{verbatim}
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 \( \text{Res} = r - Pz \) of the system should be made less than \( \delta \) in the weighted \( l_2 \) norm, i.e., \( \sqrt{\sum (\text{Res}_i \cdot \text{ewt}_i)^2} < \delta \). To obtain the \texttt{N_Vector} \texttt{ewt}, call \texttt{CVodeGetErrWeights} (see \S4.5.9.2).
lr is an input flag indicating whether the preconditioner solve function is to use the left preconditioner (\( lr = 1 \)) or the right preconditioner (\( lr = 2 \));
user_data is a pointer to user data, the same as the user_data parameter passed to the function \texttt{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 \texttt{CVSpilsPrecSolveFn} is identical to \texttt{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.
\end{verbatim}

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 \texttt{CVLsPrecSetupFn}, defined as follows:

\begin{verbatim}
CVLsPrecSetupFn
Definition typedef int (*CVLsPrecSetupFn)(realtype t, N_Vector y, N_Vector fy,
boolean_type jok, boolean_type *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 \( f(t, y) \).
\end{verbatim}
**Using CVODES for IVP Solution**

- **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 = SUNFALSE` means that the Jacobian-related data must be recomputed from scratch. `jok = 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 = SUNTRUE` can only occur after a call with `jok = 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 $\gamma$ 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 Integration of pure quadrature equations

CVODES allows the ODE system to include *pure quadratures*. In this case, it is more efficient to treat the quadratures separately by excluding them from the nonlinear solution stage. To do this, begin by excluding the quadrature variables from the vector y and excluding the quadrature equations from within `res`. Thus a separate vector $y_Q$ of quadrature variables is to satisfy $(d/dt)y_Q = I_Q(t,y)$. The following is an overview of the sequence of calls in a user’s main program in this situation. Steps that are unchanged from the skeleton program presented in §4.4 are grayed out.

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

2. **Set problem dimensions, etc.**

   Set the problem size $N$ (excluding quadrature variables), and the number of quadrature variables $N_q$.

   If appropriate, set the local vector length $N_{local}$ (excluding quadrature variables), and the local number of quadrature variables $N_{qlocal}$.
3. Set vector of initial values
4. Create CVODES object
5. Initialize CVODES solver
6. Specify integration tolerances
7. Create matrix object
8. Create linear solver object
9. Set linear solver optional inputs
10. Attach linear solver module
11. Set optional inputs
12. Attach nonlinear solver module
13. Set nonlinear solver optional inputs
14. Set vector yQ0 of initial values for quadrature variables
   Typically, the quadrature variables should be initialized to 0.
15. Initialize quadrature integration
   Call CVodeQuadInit to specify the quadrature equation right-hand side function and to allocate
   internal memory related to quadrature integration. See §4.7.1 for details.
16. Set optional inputs for quadrature integration
   Call CVodeSetQuadErrCon to indicate whether or not quadrature variables should be used in the
   step size control mechanism, and to specify the integration tolerances for quadrature variables.
   See §4.7.4 for details.
17. Advance solution in time
18. Extract quadrature variables
   Call CVodeGetQuad to obtain the values of the quadrature variables at the current time. See §4.7.3
   for details.
19. Get optional outputs
20. Get quadrature optional outputs
   Call CVodeGetQuad* functions to obtain optional output related to the integration of quadratures.
   See §4.7.5 for details.
21. Deallocate memory for solution vector and for the vector of quadrature variables
22. Free solver memory
23. Free nonlinear solver memory
24. Free linear solver and matrix memory
25. Finalize MPI, if used

CVodeQuadInit can be called and quadrature-related optional inputs (step 16 above) can be set
anywhere between steps 4 and 17.
4.7.1 Quadrature initialization and deallocation functions

The function `CVodeQuadInit` activates integration of quadrature equations and allocates internal memory related to these calculations. The form of the call to this function is as follows:

```c
flag = CVodeQuadInit(cvode_mem, fQ, yQ0);
```

**Description** The function `CVodeQuadInit` provides required problem specifications, allocates internal memory, and initializes quadrature integration.

**Arguments**
- `cvode_mem` (void *) pointer to the CVODES memory block returned by `CVodeCreate`.
- `fQ` (CVQuadRhsFn) is the C function which computes $f_Q$, the right-hand side of the quadrature equations. This function has the form $f_Q(t, y, y\dot{}, f_Q\text{data})$ (for full details see §4.7.6).
- `yQ0` (N_Vector) is the initial value of $y_Q$ (typically $y_Q0$ has all zero components).

**Return value** The return value `flag` (of type int) will be one of the following:
- `CV_SUCCESS` The call to `CVodeQuadInit` was successful.
- `CV_MEM_NULL` The CVODES memory was not initialized by a prior call to `CVodeCreate`.
- `CV_MEM_FAIL` A memory allocation request failed.

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

In terms of the number of quadrature variables $N_q$ and maximum method order `maxord`, the size of the real workspace is increased as follows:

- Base value: $lenrw = lenrw + (maxord+5)N_q$
- If using `CVodeSVtolerances` (see `CVodeSetQuadErrCon`): $lenrw = lenrw + N_q$

The size of the integer workspace is increased as follows:

- Base value: $leniw = leniw + (maxord+5)N_q$
- If using `CVodeSVtolerances`: $leniw = leniw + N_q$

The function `CVodeQuadReInit`, useful during the solution of a sequence of problems of same size, reinitializes the quadrature-related internal memory and must follow a call to `CVodeQuadInit` (and maybe a call to `CVodeReInit`). The number $N_q$ of quadratures is assumed to be unchanged from the prior call to `CVodeQuadInit`. The call to the `CVodeQuadReInit` function has the following form:

```c
flag = CVodeQuadReInit(cvode_mem, yQ0);
```

**Description** The function `CVodeQuadReInit` provides required problem specifications and reinitializes the quadrature integration.

**Arguments**
- `cvode_mem` (void *) pointer to the CVODES memory block.
- `yQ0` (N_Vector) is the initial value of $y_Q$.

**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 CVODES memory was not initialized by a prior call to `CVodeCreate`.
- `CV_NO_QUAD` Memory space for the quadrature integration was not allocated by a prior call to `CVodeQuadInit`.

**Notes** If an error occurred, `CVodeQuadReInit` also sends an error message to the error handler function.
4.7 Integration of pure quadrature equations

CVodeQuadFree

Call CVodeQuadFree(cvode_mem);

Description The function CVodeQuadFree frees the memory allocated for quadrature integration.

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

Return value The function CVodeQuadFree has no return value.

Notes In general, CVodeQuadFree need not be called by the user as it is invoked automatically by CVodeFree.

4.7.2 CVODES solver function

Even if quadrature integration was enabled, the call to the main solver function CVode is exactly the same as in §4.5.6. However, in this case the return value flag can also be one of the following:

CV_QRHSFUNC_FAIL The quadrature right-hand side function failed in an unrecoverable manner.
CV_FIRST_QRHSFUNC_FAIL The quadrature right-hand side function failed at the first call.
CV_REPTD_QRHSFUNC_ERR Convergence test failures occurred too many times due to repeated recoverable errors in the quadrature right-hand side function. This value will also be returned if the quadrature right-hand side function had repeated recoverable errors during the estimation of an initial step size (assuming the quadrature variables are included in the error tests).
CV_UNREC_QRHSFUNC_ERR The quadrature right-hand function had a recoverable error, but no recovery was possible. This failure mode is rare, as it can occur only if the quadrature right-hand side function fails recoverably after an error test failed while at order one.

4.7.3 Quadrature extraction functions

If quadrature integration has been initialized by a call to CVodeQuadInit, or reinitialized by a call to CVodeQuadReInit, then CVODES computes both a solution and quadratures at time \( t \). However, CVode will still return only the solution \( y \) in yout. Solution quadratures can be obtained using the following function:

CVodeGetQuad

Call flag = CVodeGetQuad(cvode_mem, &tret, yQ);

Description The function CVodeGetQuad returns the quadrature solution vector after a successful return from CVode.

Arguments cvode_mem (void *) pointer to the memory previously allocated by CVodeInit.
tret (realtype) the time reached by the solver (output).
yQ (N_Vector) the computed quadrature vector. This vector must be allocated by the user.

Return value The return value flag of CVodeGetQuad is one of:
CV_SUCCESS CVodeGetQuad was successful.
CV_MEM_NULL cvode_mem was NULL.
CV_NO_QUAD Quadrature integration was not initialized.
CV_BAD_DKY yQ is NULL.

Notes In case of an error return, an error message is also sent to the error handler function.

The function CVodeGetQuadDky computes the \( k \)-th derivatives of the interpolating polynomials for the quadrature variables at time \( t \). This function is called by CVodeGetQuad with \( k = 0 \) and with the current time at which CVode has returned, but may also be called directly by the user.
Using CVODES for IVP Solution

CVodeGetQuadDky

Call
flag = CVodeGetQuadDky(cvode_mem, t, k, dkyQ);

Description
The function CVodeGetQuadDky returns derivatives of the quadrature solution vector after a successful return from CVode.

Arguments
cvode_mem (void *) pointer to the memory previously allocated by CVodeInit.
t (realtype) the time at which quadrature information is requested. The time t must fall within the interval defined by the last successful step taken by CVODES.
k (int) order of the requested derivative. This must be ≤ qlast.
dkyQ (N_Vector) the vector containing the derivative. This vector must be allocated by the user.

Return value
The return value flag of CVodeGetQuadDky is one of:
CV_SUCCESS CVodeGetQuadDky succeeded.
CV_MEM_NULL The pointer to cvode_mem was NULL.
CV_NO_QUAD Quadrature integration was not initialized.
CV_BAD_DKY The vector dkyQ is NULL.
CV_BAD_K k is not in the range 0, 1, ..., qlast.
CV_BAD_T The time t is not in the allowed range.

Notes
In case of an error return, an error message is also sent to the error handler function.

4.7.4 Optional inputs for quadrature integration

CVODES provides the following optional input functions to control the integration of quadrature equations.

CVodeSetQuadErrCon

Call
flag = CVodeSetQuadErrCon(cvode_mem, errconQ);

Description
The function CVodeSetQuadErrCon specifies whether or not the quadrature variables are to be used in the step size control mechanism within CVODES. If they are, the user must call CVodeQuadSSStolerances or CVodeQuadSVtolerances to specify the integration tolerances for the quadrature variables.

Arguments
cvode_mem (void *) pointer to the CVODES memory block.
errconQ (boolean type) specifies whether quadrature variables are included (SUNTRUE) or not (SUNFALSE) in the error control mechanism.

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_NO_QUAD Quadrature integration has not been initialized.

Notes
By default, errconQ is set to SUNFALSE.
It is illegal to call CVodeSetQuadErrCon before a call to CVodeQuadInit.

If the quadrature variables are part of the step size control mechanism, one of the following functions must be called to specify the integration tolerances for quadrature variables.

CVodeQuadSSStolerances

Call
flag = CVodeQuadSSStolerances(cvode_mem, reltolQ, abstolQ);

Description
The function CVodeQuadSSStolerances specifies scalar relative and absolute tolerances.

Arguments
cvode_mem (void *) pointer to the CVODES memory block.
4.7 Integration of pure quadrature equations

reltolQ (realtype) is the scalar relative error tolerance.
abstolQ (realtype) is the scalar absolute error tolerance.

Return value The return value flag (of type int) is one of:
CV_SUCCESS The optional value has been successfully set.
CV_NO_QUAD Quadrature integration was not initialized.
CV_MEM_NULL The cvode_mem pointer is NULL.
CV_ILL_INPUT One of the input tolerances was negative.

CVodeQuadSVtolerances
Call flag = CVodeQuadSVtolerances(cvode_mem, reltolQ, abstolQ);
Description The function CVodeQuadSVtolerances specifies scalar relative and vector absolute tolerances.
Arguments cvode_mem (void *) pointer to the CVODES memory block.
reltolQ (realtype) is the scalar relative error tolerance.
abstolQ (N_Vector) is the vector absolute error tolerance.

Return value The return value flag (of type int) is one of:
CV_SUCCESS The optional value has been successfully set.
CV_NO_QUAD Quadrature integration was not initialized.
CV_MEM_NULL The cvode_mem pointer is NULL.
CV_ILL_INPUT One of the input tolerances was negative.

4.7.5 Optional outputs for quadrature integration

CVODES provides the following functions that can be used to obtain solver performance information related to quadrature integration.

CVodeGetQuadNumRhsEvals
Call flag = CVodeGetQuadNumRhsEvals(cvode_mem, &nfQevals);
Description The function CVodeGetQuadNumRhsEvals returns the number of calls made to the user's quadrature right-hand side function.
Arguments cvode_mem (void *) pointer to the CVODES memory block.
fQevals (long int) number of calls made to the user's fQ 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.
CV_NO_QUAD Quadrature integration has not been initialized.

CVodeGetQuadNumErrTestFails
Call flag = CVodeGetQuadNumErrTestFails(cvode_mem, &nQetfails);
Description The function CVodeGetQuadNumErrTestFails returns the number of local error test failures due to quadrature variables.
Arguments cvode_mem (void *) pointer to the CVODES memory block.
nQetfails (long int) number of error test failures due to quadrature variables.

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_NO_QUAD Quadrature integration has not been initialized.
CVodeGetQuadErrWeights

**Call**

```c
flag = CVodeGetQuadErrWeights(cvode_mem, eQweight);
```

**Description**
The function `CVodeGetQuadErrWeights` returns the quadrature error weights at the current time.

**Arguments**
- `cvode_mem` (void *) pointer to the CVODES memory block.
- `eQweight` (N_Vector) quadrature 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 `cvode_mem` pointer is NULL.
- `CV_NO_QUAD` Quadrature integration has not been initialized.

**Notes**
The user must allocate memory for `eQweight`.

If quadratures were not included in the error control mechanism (through a call to `CVodeSetQuadErrCon` with `errconQ = SUNTRUE`), `CVodeGetQuadErrWeights` does not set the `eQweight` vector.

CVodeGetQuadStats

**Call**

```c
flag = CVodeGetQuadStats(cvode_mem, &nfQevals, &nQetfails);
```

**Description**
The function `CVodeGetQuadStats` returns the CVODES integrator statistics as a group.

**Arguments**
- `cvode_mem` (void *) pointer to the CVODES memory block.
- `nfQevals` (long int) number of calls to the user’s `fQ` function.
- `nQetfails` (long int) number of error test failures due to quadrature variables.

**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_NO_QUAD` Quadrature integration has not been initialized.

4.7.6 User-supplied function for quadrature integration

For integration of quadrature equations, the user must provide a function that defines the right-hand side of the quadrature equations (in other words, the integrand function of the integral that must be evaluated). This function must be of type `CVQuadRhsFn` defined as follows:

```c
typedef int (*CVQuadRhsFn)(realtype t, N_Vector y, N_Vector yQdot, void *user_data);
```

This function computes the quadrature equation 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)`.
- `yQdot` is the output vector `fQ(t,y)`.
- `user_data` is the `user_data` pointer passed to `CVodeSetUserData`.

**Return value**
A `CVQuadRhsFn` should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODES will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and `CV_QRHSFUNC_FAIL` is returned).
4.8 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, CVODES provides a banded preconditioner in the module CVBANDPRE and a band-block-diagonal preconditioner module CVBBDPRE.

4.8.1 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) and sub-diagonals ($m_l$, 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 cvodes_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 CVODES object
5. Initialize CVODES 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

Notes

Allocation of memory for $yQdot$ is automatically handled within CVODES.

Both $y$ and $yQdot$ are of type N_Vector, but they typically have different internal representations. It is the user’s responsibility to access the vector data consistently (including the use of the correct accessor macros from each NVECTOR implementation).

For the sake of computational efficiency, the vector functions in the two NVECTOR implementations provided with CVODES do not perform any consistency checks with respect to their N_Vector arguments (see §7.2 and §7.3).

There are two situations in which recovery is not possible even if CVQuadRhsFn function returns a recoverable error flag. One is when this occurs at the very first call to the CVQuadRhsFn (in which case CVODES returns CV_FIRST_QRHSFUNC_ERR). The other is when a recoverable error is reported by CVQuadRhsFn after an error test failure, while the linear multistep method order is equal to 1 (in which case CVODES returns CV_UNREC_QRHSFUNC_ERR).
9. Attach linear solver module

10. **Initialize the cvbandpre preconditioner module**

   Specify the upper and lower half-bandwidths (\( \mu \) and \( \ml \), respectively) and call
   
   \[ \text{flag} = \text{CVBandPrecInit}(\text{cvode}\_\text{mem}, N, \mu, \ml); \]

   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 \text{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, \text{CVBandPrecGetWorkSpace} and \text{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:

```
CVBandPrecInit
```

**Call**

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

**Description**

The function \text{CVBandPrecInit} initializes the cvbandpre preconditioner and allocates required (internal) memory for it.

**Arguments**

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

**Return value**

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

- \text{CVLS\_SUCCESS} The call to CVBandPrecInit was successful.
- \text{CVLS\_MEM\_NULL} The cvode\_mem pointer was NULL.
- \text{CVLS\_MEM\_FAIL} A memory allocation request has failed.
- \text{CVLS\_LMEM\_NULL} A CVLS linear solver memory was not attached.
- \text{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 \(-\ml \leq j - i \leq \mu\).

The following three optional output functions are available for use with the cvbandpre module:
4.8 Preconditioner modules

CVBandPrecGetWorkSpace

Call flag = CVBandPrecGetWorkSpace(cvode_mem, &lenrwBP, &leniwBP);

Description The function CVBandPrecGetWorkSpace returns the sizes of the CVBANDPRE real and integer workspaces.

Arguments
cvode_mem (void *) pointer to the CVODES memory block.
lenrwBP (long int) the number of realtype values in the CVBANDPRE workspace.
leniwBP (long int) the number of integer values in the CVBANDPRE workspace.

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

CVLS_SUCCESS The optional output values have been successfully set.
CVLS_PMEM_NULL The CVBANDPRE preconditioner has not been initialized.

Notes The workspace requirements reported by this routine correspond only to memory allocated within the 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 CVodeGetLinWorkSpace.

CVBandPrecGetNumRhsEvals

Call flag = CVBandPrecGetNumRhsEvals(cvode_mem, &nfevalsBP);

Description The function 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
cvode_mem (void *) pointer to the CVODES memory block.
nfevalsBP (long int) the number of calls to the user right-hand side function.

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

CVLS_SUCCESS The optional output value has been successfully set.
CVLS_PMEM_NULL The CVBANDPRE preconditioner has not been initialized.

Notes The counter nfevalsBP is distinct from the counter nfevalsLS returned by the corresponding function CVodeGetNumLinRhsEvals and nfevals returned by CVodeGetNumRhsEvals. The total number of right-hand side function evaluations is the sum of all three of these counters.

4.8.2 A parallel band-block-diagonal preconditioner module

A principal reason for using a parallel ODE solver such as CVODES 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 (2.6) 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 CVODES package. This module works with the parallel vector module NVECTOR_PARALLEL and is usable with any of the Krylov iterative linear solvers through the 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 CVBBBDPRE.
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 \( \bar{y}_m \) denote \( y_m \) augmented with those other components on which \( g_m \) depends. Then we have

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

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

The preconditioner associated with this decomposition has the form

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

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

\[
Px = b
\]

reduces to solving each of the equations

\[
P_m x_m = b_m
\]

and this is done by banded LU factorization of \( P_m \) followed by a banded backsolve.

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

The \texttt{cvbbdpre} module calls two user-provided functions to construct \( P \): a required function \texttt{gloc} (of type \texttt{CVLocalFn}) which approximates the right-hand side function \( g(t, y) \approx f(t, y) \) and which is computed locally, and an optional function \texttt{cfn} (of type \texttt{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 \texttt{user\_data} that is passed by the user to \texttt{CVodeSetUserData} and that was passed to the user’s function \( f \). The user is responsible for providing space (presumably within \texttt{user\_data}) for components of \( y \) that are communicated between processes by \texttt{cfn}, and that are then used by \texttt{gloc}, which should not do any communication.

\texttt{CVLocalFn}

\begin{verbatim}
typedef int (*CVLocalFn)(sunindextype Nlocal, realtype t, N_Vector y, N_Vector glocal, void *user_data);

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

Arguments:

\begin{itemize}
  \item \texttt{Nlocal} is the local vector length.
  \item \texttt{t} is the value of the independent variable.
  \item \texttt{y} is the dependent variable.
\end{itemize}
\end{verbatim}
4.8 Preconditioner modules

\textit{glocal} is the output vector.
\textit{user\_data} is a pointer to user data, the same as the \textit{user\_data} parameter passed to \texttt{CVodeSetUserData}.

Return value \ A \texttt{CVLocalFn} should return 0 if successful, a positive value if a recoverable error occurred (in which case \texttt{CVODES} will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and \texttt{CVode} returns \texttt{CV\_LSETUP\_FAIL}).

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

\texttt{CVCommFn}

\begin{verbatim}
typedef int (*CVCommFn)(sunindextype Nlocal, realtype t, N_Vector y, void *user_data);
\end{verbatim}

\textbf{Definition} \ A \texttt{CVCommFn} should return 0 if successful, a positive value if a recoverable error occurred (in which case \texttt{CVODES} will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and \texttt{CVode} returns \texttt{CV\_LSETUP\_FAIL}).

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

Besides the header files required for the integration of the ODE problem (see §4.3), to use the \texttt{cvbbdpre} module, the main program must include the header file \texttt{cvodes\_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 \texttt{CVODES} object
5. Initialize \texttt{CVODES} solver
6. Specify integration tolerances
7. Create linear solver object
   \textbf{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
```
flag = CVBBDPrecInit(cvode_mem, local_N, mudq, mldq,
                      mukeep, mlkeep, dqrely, gloc, cfn);
```

Description
The function `CVBBDPrecInit` initializes and allocates (internal) memory for the CVBBDPRE preconditioner.

Arguments
- `cvode_mem` (void *) pointer to the CVODES 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.
4.8 Preconditioner modules

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.

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 CVODES for a subsequent problem, a call to CVBBDPrecReInit can be made to change any of the following: the half-bandwidths mudq and mldq used in the difference-quotient Jacobian approximations, the relative increment dqrely, or one of the user-supplied functions gloc and cfn. If there is a change in any of the linear solver inputs, an additional call to the “Set” routines provided by the SUNLINSOL module, and/or one or more of the corresponding CVLS “set” functions, must also be made (in the proper order).

**CVBBDPrecReInit**

Call flag = CVBBDPrecReInit(cvode_mem, mudq, mldq, dqrely);

Description The function CVBBDPrecReInit re-initializes the CVBBDPre preconditioner.

Arguments
cvoke_mem (void *) pointer to the CVODES 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.

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

- CVLS_SUCCESS The call to CVBBDPrecReInit was successful.
- CVLS_MEM_NULL The cvode_mem pointer was NULL.
CVLS_LMEM_NULL  A CVLS linear solver memory was not attached.
CVLS_PMEM_NULL  The function CVBBDPRecInit was not previously called.

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

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

**CVBBDPrecGetWorkSpace**

Call  

```c
flag = CVBBDPrecGetWorkSpace(cvode_mem, &lenrwBBDP, &leniwBBDP);
```

Description  The function CVBBDPrecGetWorkSpace returns the local CVBBDPRE real and integer workspace sizes.

Arguments  

cvode_mem (void *) pointer to the CVODES memory block.
lenrwBBDP (long int) local number of realtype values in the CVBBDPRE workspace.
leniwBBDP (long int) local number of integer values in the CVBBDPRE workspace.

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 was NULL.
CVLS_PMEM_NULL  The CVBBDPRE preconditioner has not been initialized.

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 CVodeGetLinWorkSpace.

**CVBBDPrecGetNumGfnEvals**

Call  

```c
flag = CVBBDPrecGetNumGfnEvals(cvode_mem, &ngevalsBBDP);
```

Description  The function CVBBDPrecGetNumGfnEvals returns the number of calls made to the user-supplied gloc function due to the finite difference approximation of the Jacobian blocks used within the preconditioner setup function.

Arguments  

cvode_mem (void *) pointer to the CVODES memory block.
ngevalsBBDP (long int) the number of calls made to the user-supplied gloc 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 was NULL.
CVLS_PMEM_NULL  The CVBBDPRE preconditioner has not been initialized.

Notes  The ngevalsBBDP gloc evaluations, the costs associated with CVBBDPRE also include nlinsetups LU factorizations, nlinsetups calls to cfn, npsolves banded backsolve calls, and nfevalsLS right-hand side function evaluations, where nlinsetups is an optional CVODES output and npsolves and nfevalsLS are linear solver optional outputs (see §4.5.9).
Chapter 5

Using CVODES for Forward Sensitivity Analysis

This chapter describes the use of CVODES to compute solution sensitivities using forward sensitivity analysis. One of our main guiding principles was to design the CVODES user interface for forward sensitivity analysis as an extension of that for IVP integration. Assuming a user main program and user-defined support routines for IVP integration have already been defined, in order to perform forward sensitivity analysis the user only has to insert a few more calls into the main program and (optionally) define an additional routine which computes the right-hand side of the sensitivity systems (2.12). The only departure from this philosophy is due to the CVRhsFn type definition (§4.6.1). Without changing the definition of this type, the only way to pass values of the problem parameters to the ODE right-hand side function is to require the user data structure f_data to contain a pointer to the array of real parameters p.

CVODES 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.

We begin with a brief overview, in the form of a skeleton user program. Following that are detailed descriptions of the interface to the various user-callable routines and of the user-supplied routines that were not already described in Chapter 4.

5.1 A skeleton of the user’s main program

The following is a skeleton of the user’s main program (or calling program) as an application of CVODES. The user program is to have these steps in the order indicated, unless otherwise noted. For the sake of brevity, we defer many of the details to the later sections. As in §4.4, most steps are independent of the NVECTOR, SUNMATRIX, SUNLINSOL, and SUNNONLINSOL implementations used. For the steps that are not, refer to Chapters 7, 8, 9, and 10 for the specific name of the function to be called or macro to be referenced.

Differences between the user main program in §4.4 and the one below start only at step (16). Steps that are unchanged from the skeleton program presented in §4.4 are grayed out.

First, note that no additional header files need be included for forward sensitivity analysis beyond those for IVP solution (§4.4).

1. Initialize parallel or multi-threaded environment, if appropriate
2. Set problem dimensions etc.
3. Set vector of initial values
4. Create CVODES object
5. Initialize CVODES solver
6. Specify integration tolerances
7. Create matrix object
8. Create linear solver object
9. Set linear solver optional inputs
10. Attach linear solver module
11. Set optional inputs
12. Create nonlinear solver object
13. Attach nonlinear solver module
14. Set nonlinear solver optional inputs
15. Initialize quadrature problem, if not sensitivity-dependent

16. Define the sensitivity problem

- Number of sensitivities (required)
  Set $N_s = N_s$, the number of parameters with respect to which sensitivities are to be computed.

- Problem parameters (optional)
  If CVODES is to evaluate the right-hand sides of the sensitivity systems, set $p$, an array of $N_p$ real parameters upon which the IVP depends. Only parameters with respect to which sensitivities are (potentially) desired need to be included. Attach $p$ to the user data structure `user_data`. For example, `user_data->p = p;`
  If the user provides a function to evaluate the sensitivity right-hand side, $p$ need not be specified.

- Parameter list (optional)
  If CVODES is to evaluate the right-hand sides of the sensitivity systems, set $plist$, an array of $N_s$ integers to specify the parameters $p$ with respect to which solution sensitivities are to be computed. If sensitivities with respect to the $j$-th parameter $p[j]$ are desired ($0 \leq j < N_p$), set $plist_i = j$, for some $i = 0, \ldots, N_s - 1$.
  If $plist$ is not specified, CVODES will compute sensitivities with respect to the first $N_s$ parameters; i.e., $plist_i = i (i = 0, \ldots, N_s - 1)$.
  If the user provides a function to evaluate the sensitivity right-hand side, $plist$ need not be specified.

- Parameter scaling factors (optional)
  If CVODES is to estimate tolerances for the sensitivity solution vectors (based on tolerances for the state solution vector) or if CVODES is to evaluate the right-hand sides of the sensitivity systems using the internal difference-quotient function, the results will be more accurate if order of magnitude information is provided.
  Set $pbar$, an array of $N_s$ positive scaling factors. Typically, if $p_i \neq 0$, the value $\bar{p}_i = |p_{plist_i}|$ can be used.
  If $pbar$ is not specified, CVODES will use $\bar{p}_i = 1.0$.
  If the user provides a function to evaluate the sensitivity right-hand side and specifies tolerances for the sensitivity variables, $pbar$ need not be specified.

Note that the names for $p$, $pbar$, $plist$, as well as the field $p$ of `user_data` are arbitrary, but they must agree with the arguments passed to `CVodeSetSensParams` below.
17. **Set sensitivity initial conditions**

Set the $N_s$ vectors $y_{0}^i$ of initial values for sensitivities (for $i = 0, \ldots, N_s - 1$), using the appropriate functions defined by the particular NVECTOR implementation chosen.

First, create an array of $N_s$ vectors by making the appropriate call

\[ y_{0} = N_{V}^*\text{CloneVectorArray}_\text{***}(N_s, y_0); \]

or

\[ y_{0} = N_{V}^*\text{CloneVectorArrayEmpty}_\text{***}(N_s, y_0); \]

Here the argument $y_0$ serves only to provide the $N_{\text{Vector}}$ type for cloning.

Then, for each $i = 0, \ldots, N_s - 1$, load initial values for the $i$-th sensitivity vector $y_{0}^i$.

18. **Activate sensitivity calculations**

Call $\text{flag} = \text{CVodeSensInit}$ or $\text{CVodeSensInit1}$ to activate forward sensitivity computations and allocate internal memory for CVODES related to sensitivity calculations (see §5.2.1).

19. **Set sensitivity tolerances**

Call $\text{CVodeSensSSStolerances}$, $\text{CVodeSensSVtolerances}$ or $\text{CVodeEEtolerances}$. (See §5.2.2).

20. **Set sensitivity analysis optional inputs**

Call $\text{CVodeSetSens}$* routines to change from their default values any optional inputs that control the behavior of CVODES in computing forward sensitivities. (See §5.2.6.)

21. **Create sensitivity nonlinear solver object** *(optional)*

If using a non-default nonlinear solver (see §5.2.3), then create the desired nonlinear solver object by calling the appropriate constructor function defined by the particular SUNNONLINSOL implementation e.g.,

\[ \text{NLSSens} = \text{SUNNonlinSol}_\text{***Sens}(\ldots); \]

for the CV_SIMULTANEOUS or CV_STAGGERED options or

\[ \text{NLSSens} = \text{SUNNonlinSol}_\text{***(\ldots);} \]

for the CV_STAGGERED1 option where *** is the name of the nonlinear solver and ... are constructor specific arguments (see Chapter 10 for details).

22. **Attach the sensitivity nonlinear solver module** *(optional)*

If using a non-default nonlinear solver, then initialize the nonlinear solver interface by attaching the nonlinear solver object by calling

\[ \text{ier} = \text{CVodeSetNonlinearSolverSensSim}(cvode\_mem, \text{NLSSens}); \]

when using the CV_SIMULTANEOUS corrector method,

\[ \text{ier} = \text{CVodeSetNonlinearSolverSensStg}(cvode\_mem, \text{NLSSens}); \]

when using the CV_STAGGERED corrector method, or

\[ \text{ier} = \text{CVodeSetNonlinearSolverSensStg1}(cvode\_mem, \text{NLSSens}); \]

when using the CV_STAGGERED1 corrector method (see §5.2.3 for details).
23. Set sensitivity nonlinear solver optional inputs *(optional)*

Call the appropriate set functions for the selected nonlinear solver module to change optional inputs specific to that nonlinear solver. These must be called after CVodeSensInit if using the default nonlinear solver or after attaching a new nonlinear solver to CVODES, otherwise the optional inputs will be overridden by CVODE defaults. See Chapter 10 for more information on optional inputs.

24. Specify rootfinding

25. Advance solution in time

26. Extract sensitivity solution

After each successful return from CVode, the solution of the original IVP is available in the $y$ argument of CVode, while the sensitivity solution can be extracted into $yS$ (which can be the same as $yS0$) by calling one of the routines CVodeGetSens, CVodeGetSens1, CVodeGetSensDky, or CVodeGetSensDky1 (see §5.2.5).

27. Get optional outputs

28. Deallocate memory for solution vector

29. Deallocate memory for sensitivity vectors

Upon completion of the integration, deallocate memory for the vectors $yS0$ using the appropriate destructor:

```
N_VDestroyVectorArray_***(&yS0, Ns);
```

If $yS$ was created from `realtype` arrays $yS_i$, it is the user’s responsibility to also free the space for the arrays $yS0_i$.

30. Free user data structure

31. Free solver memory

32. Free nonlinear solver memory

33. Free vector specification memory

34. Free linear solver and matrix memory

35. Finalize MPI, if used

### 5.2 User-callable routines for forward sensitivity analysis

This section describes the CVODES functions, in addition to those presented in §4.5, that are called by the user to setup and solve a forward sensitivity problem.

#### 5.2.1 Forward sensitivity initialization and deallocation functions

Activation of forward sensitivity computation is done by calling `CVodeSensInit` or `CVodeSensInit1`, depending on whether the sensitivity right-hand side function returns all sensitivities at once or one by one, respectively. The form of the call to each of these routines is as follows:

```c
CVodeSensInit
```

Call

```c
flag = CVodeSensInit(cvode_mem, Ns, ism, fS, yS0);
```
5.2 User-callable routines for forward sensitivity analysis

Description The routine CVodeSensInit activates forward sensitivity computations and allocates internal memory related to sensitivity calculations.

Arguments
- cvode_mem (void *) pointer to the CVODES memory block returned by CVodeCreate.
- Ns (int) the number of sensitivities to be computed.
- ism (int) a flag used to select the sensitivity solution method. Its value can be CV_SIMULTANEOUS or CV_STAGGERED:
  - In the CV_SIMULTANEOUS approach, the state and sensitivity variables are corrected at the same time. If the default Newton nonlinear solver is used, this amounts to performing a modified Newton iteration on the combined nonlinear system;
  - In the CV_STAGGERED approach, the correction step for the sensitivity variables takes place at the same time for all sensitivity equations, but only after the correction of the state variables has converged and the state variables have passed the local error test;
- fS (CVSensRhsFn) is the C function which computes all sensitivity ODE right-hand sides at the same time. For full details see §5.3.
- yS0 (N_Vector *) a pointer to an array of Ns vectors containing the initial values of the sensitivities.

Return value The return value flag (of type int) will be one of the following:
- CV_SUCCESS The call to CVodeSensInit was successful.
- CV_MEM_NULL The CVODES 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 CVodeSensInit has an illegal value.

Notes
- Passing fS=NULL indicates using the default internal difference quotient sensitivity right-hand side routine.
- If an error occurred, CVodeSensInit also sends an error message to the error handler function.
- It is illegal here to use ism = CV_STAGGERED1. This option requires a different type for fS and can therefore only be used with CVodeSensInit1 (see below).

CVodeSensInit1

Call

flag = CVodeSensInit1(cvode_mem, Ns, ism, fS1, yS0);

Description The routine CVodeSensInit1 activates forward sensitivity computations and allocates internal memory related to sensitivity calculations.

Arguments
- cvode_mem (void *) pointer to the CVODES memory block returned by CVodeCreate.
- Ns (int) the number of sensitivities to be computed.
- ism (int) a flag used to select the sensitivity solution method. Its value can be CV_SIMULTANEOUS, CV_STAGGERED, or CV_STAGGERED1:
  - In the CV_SIMULTANEOUS approach, the state and sensitivity variables are corrected at the same time. If the default Newton nonlinear solver is used, this amounts to performing a modified Newton iteration on the combined nonlinear system;
  - In the CV_STAGGERED approach, the correction step for the sensitivity variables takes place at the same time for all sensitivity equations, but only after the correction of the state variables has converged and the state variables have passed the local error test;
In the CV_STAGGERED1 approach, all corrections are done sequentially, first for the state variables and then for the sensitivity variables, one parameter at a time. If the sensitivity variables are not included in the error control, this approach is equivalent to CV_STAGGERED. Note that the CV_STAGGERED1 approach can be used only if the user-provided sensitivity right-hand side function is of type CVSensRhs1Fn (see §5.3).

\(fS1(\text{CVSensRhs1Fn})\) is the C function which computes the right-hand sides of the sensitivity ODE, one at a time. For full details see §5.3.

\(yS0(\text{N_Vector} \, \ast)\) a pointer to an array of \(Ns\) vectors containing the initial values of the sensitivities.

Return value The return value \(\text{flag}\) (of type \(\text{int}\)) will be one of the following:

- CV_SUCCESS The call to CVodeSensInit1 was successful.
- CV_MEM_NULL The CVODES 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 CVodeSensInit1 has an illegal value.

Notes Passing \(fS1\)=NULL indicates using the default internal difference quotient sensitivity right-hand side routine.

If an error occurred, CVodeSensInit1 also sends an error message to the error handler function.

In terms of the problem size \(N\), number of sensitivity vectors \(Ns\), and maximum method order \(\text{maxord}\), the size of the real workspace is increased as follows:

- Base value: \(lenrw = lenrw + (\text{maxord}+5)NsN\)
- With CVodeSensSVtolerances: \(lenrw = lenrw + NsN\)

the size of the integer workspace is increased as follows:

- Base value: \(leniw = leniw + (\text{maxord}+5)NsN_i\)
- With CVodeSensSVtolerances: \(leniw = leniw + NsN_i\)

where \(N_i\) is the number of integers in one \text{N_Vector}.

The routine CVodeSensReInit, useful during the solution of a sequence of problems of same size, reinitializes the sensitivity-related internal memory. The call to it must follow a call to CVodeSensInit or CVodeSensInit1 (and maybe a call to CVodeReInit). The number \(Ns\) of sensitivities is assumed to be unchanged since the call to the initialization function. The call to the CVodeSensReInit function has the form:

```
CVodeSensReInit
Call flag = CVodeSensReInit(cvode_mem, ism, yS0);
Description The routine CVodeSensReInit reinitializes forward sensitivity computations.
Arguments cvode_mem (void *) a pointer to the CVODES memory block returned by CVodeCreate.
ism (int) a flag used to select the sensitivity solution method. Its value can be CV_SIMULTANEOUS, CV_STAGGERED, or CV_STAGGERED1.
yS0 (N_Vector * ) a pointer to an array of Ns variables of type N_Vector containing the initial values of the sensitivities.
Return value The return value \(\text{flag}\) (of type \(\text{int}\)) will be one of the following:
- CV_SUCCESS The call to CVodeReInit was successful.
- CV_MEM_NULL The CVODES memory block was not initialized through a previous call to CVodeCreate.
```
5.2 User-callable routines for forward sensitivity analysis

CV_NO_SENS  Memory space for sensitivity integration was not allocated through a previous call to CVodeSensInit.
CV_IILL_INPUT An input argument to CVodeSensReInit has an illegal value.
CV_MEM_FAIL  A memory allocation request has failed.

Notes All arguments of CVodeSensReInit are the same as those of the functions CVodeSensInit and CVodeSensInit1.
If an error occurred, CVodeSensReInit also sends a message to the error handler function.
CVodeSensReInit potentially does some minimal memory allocation (for the sensitivity absolute tolerance) and for arrays of counters used by the CV_STAGGERED1 method.
The value of the input argument \( ism \) must be compatible with the type of the sensitivity ODE right-hand side function. Thus if the sensitivity module was initialized using CVodeSensInit, then it is illegal to pass \( ism = CV_{STAGGERED1} \) to CVodeSensReInit.

To deallocate all forward sensitivity-related memory (allocated in a prior call to CVodeSensInit or CVodeSensInit1), the user must call

```c
CVodeSensFree(cvode_mem);
```
Description The function CVodeSensFree frees the memory allocated for forward sensitivity computations by a previous call to CVodeSensInit or CVodeSensInit1.
Arguments The argument is the pointer to the CVODES memory block (of type void *).
Return value The function CVodeSensFree has no return value.
Notes In general, CVodeSensFree need not be called by the user, as it is invoked automatically by CVodeFree.
After a call to CVodeSensFree, forward sensitivity computations can be reactivated only by calling CVodeSensInit or CVodeSensInit1 again.

To activate and deactivate forward sensitivity calculations for successive CVODES runs, without having to allocate and deallocate memory, the following function is provided:

```c
CVodeSensToggleOff(cvode_mem);
```
Description The function CVodeSensToggleOff deactivates forward sensitivity calculations. It does not deallocate sensitivity-related memory.
Arguments `cvode_mem` (void *) pointer to the memory previously returned by CVodeCreate.
Return value The return value `flag` of CVodeSensToggle is one of:
- CV_SUCCESS CVodeSensToggleOff was successful.
- CV_MEM_NULL cvode_mem was NULL.
Notes Since sensitivity-related memory is not deallocated, sensitivities can be reactivated at a later time (using CVodeSensReInit).

5.2.2 Forward sensitivity tolerance specification functions
One of the following three functions must be called to specify the integration tolerances for sensitivities. Note that this call must be made after the call to CVodeSensInit/CVodeSensInit1.
Using CVODES for Forward Sensitivity Analysis

**CVodeSensSSStolerances**

Call

\[
\text{flag} = \text{CVodeSensSSStolerances}(\text{cvode}, \text{reltolS}, \text{abstolS});
\]

Description The function \text{CVodeSensSSStolerances} specifies scalar relative and absolute tolerances.

Arguments

\- \text{cvode} \text{(void *)} pointer to the CVODES memory block returned by \text{CVodeCreate}.
\- \text{reltolS} \text{(realtype)} is the scalar relative error tolerance.
\- \text{abstolS} \text{(realtype*)} is a pointer to an array of length \(N_s\) containing the scalar absolute error tolerances, one for each parameter.

Return value The return flag \text{flag} (of type \text{int}) will be one of the following:

\- \text{CV_SUCCESS} The call to \text{CVodeSSStolerances} was successful.
\- \text{CV_MEM_NULL} The CVODES memory block was not initialized through a previous call to \text{CVodeCreate}.
\- \text{CV_NO_SENS} The sensitivity allocation function (\text{CVodeSensInit} or \text{CVodeSensInit1}) has not been called.
\- \text{CV_Ill_INPUT} One of the input tolerances was negative.

**CVodeSensSVtolerances**

Call

\[
\text{flag} = \text{CVodeSensSVtolerances}(\text{cvode}, \text{reltolS}, \text{abstolS});
\]

Description The function \text{CVodeSensSVtolerances} specifies scalar relative tolerance and vector absolute tolerances.

Arguments

\- \text{cvode} \text{(void *)} pointer to the CVODES memory block returned by \text{CVodeCreate}.
\- \text{reltolS} \text{(realtype)} is the scalar relative error tolerance.
\- \text{abstolS} \text{(N_Vector*)} is an array of \(N_s\) variables of type \text{N_Vector}. The \text{N_Vector} \text{from abstolS[is]} specifies the vector tolerances for \(i\)-th sensitivity.

Return value The return flag \text{flag} (of type \text{int}) will be one of the following:

\- \text{CV_SUCCESS} The call to \text{CVodeSVtolerances} was successful.
\- \text{CV_MEM_NULL} The CVODES memory block was not initialized through a previous call to \text{CVodeCreate}.
\- \text{CV_NO_SENS} The allocation function for sensitivities has not been called.
\- \text{CV_Ill_INPUT} The relative error tolerance was negative or an absolute tolerance vector had a negative component.

Notes This choice of tolerances is important when the absolute error tolerance needs to be different for each component of any vector \(yS[i]\).

**CVodeSensEEtolerances**

Call

\[
\text{flag} = \text{CVodeSensEEtolerances}(\text{cvode_mem});
\]

Description When \text{CVodeSensEEtolerances} is called, CVODES will estimate tolerances for sensitivity variables based on the tolerances supplied for states variables and the scaling factors \(\bar{p}\).

Arguments \text{cvode_mem} \text{(void *)} pointer to the CVODES memory block returned by \text{CVodeCreate}.

Return value The return flag \text{flag} (of type \text{int}) will be one of the following:

\- \text{CV_SUCCESS} The call to \text{CVodeSensEEtolerances} was successful.
\- \text{CV_MEM_NULL} The CVODES memory block was not initialized through a previous call to \text{CVodeCreate}.
\- \text{CV_NO_SENS} The sensitivity allocation function has not been called.

Notes
5.2.3 Forward sensitivity nonlinear solver interface functions

As in the pure ODE case, when computing solution sensitivities using forward sensitivity analysis cvodes uses the SUNNONLINSOL implementation of Newton’s method defined by the SUNNONLINSOL_NEWTON module (see §10.2) by default. To specify a different nonlinear solver in CVODES, the user’s program must create a SUNNONLINSOL object by calling the appropriate constructor routine. The user must then attach the SUNNONLINSOL object to CVODES by calling CVodeSetNonlinearSolverSensSim when using the CV_SIMULTANEOUS corrector option, or CVodeSetNonlinearSolver when using the §4.5.4 and CVodeSetNonlinearSolverStg or CVodeSetNonlinearSolverStg1 when using the CV_STAGGERED or CV_STAGGERED1 corrector option respectively, as documented below.

When changing the nonlinear solver in CVODES, CVodeSetNonlinearSolver must be called after CVodeInit; similarly CVodeSetNonlinearSolverSensSim, CVodeSetNonlinearSolverStg, and CVodeSetNonlinearSolverStg1 must be called after CVodeSensInit. If any calls to CVode have been made, then CVODES 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 routines CVodeSetNonlinearSolverSensSim, CVodeSetNonlinearSolverStg, and CVodeSetNonlinearSolverStg1 is the CVODES memory pointer returned by CVodeCreate and the second argument is the SUNNONLINSOL object to use for solving the nonlinear systems (2.4) or (2.5). A call to this function attaches the nonlinear solver to the main CVODES integrator.

```c
CVodeSetNonlinearSolverSensSim
```

Call

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

Description The function CVodeSetNonlinearSolverSensSim attaches a SUNNONLINSOL object (NLS) to CVODES when using the CV_SIMULTANEOUS approach to correct the state and sensitivity variables at the same time.

Arguments

- `cvode_mem` (void *) pointer to the CVODES memory block.
- `NLS` (SUNNonlinearSolver) SUNNONLINSOL object to use for solving nonlinear systems (2.4) or (2.5).

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.

```c
CVodeSetNonlinearSolverSensStg
```

Call

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

Description The function CVodeSetNonlinearSolverSensStg attaches a SUNNONLINSOL object (NLS) to CVODES when using the CV_STAGGERED approach to correct all the sensitivity variables after the correction of the state variables.

Arguments

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

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.
CVODES solver function

Even if forward sensitivity analysis was enabled, the call to the main solver function `CVode` is exactly the same as in §4.5.6. However, in this case the return value `flag` can also be one of the following:

- **CV_SRHSFUNC_FAIL**  
  The sensitivity right-hand side function failed in an unrecoverable manner.

- **CV_FIRST_SRHSFUNC_ERR**  
  The sensitivity right-hand side function failed at the first call.

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

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

5.2.5 Forward sensitivity extraction functions

If forward sensitivity computations have been initialized by a call to `CVodeSensInit/CVodeSensInit1`, or reinitialized by a call to `CVSensReInit`, then CVODES computes both a solution and sensitivities at time $t$. However, CVode will still return only the solution $y$ in `yout`. Solution sensitivities can be obtained through one of the following functions:
5.2 User-callable routines for forward sensitivity analysis

**CVodeGetSens**

Call  
\[
\text{flag} = \text{CVodeGetSens}(\text{cvode\_mem}, \&\text{tret}, \text{yS});
\]

Description  
The function `CVodeGetSens` returns the sensitivity solution vectors after a successful return from `CVode`.

Arguments  
- `cvode\_mem` (void *) pointer to the memory previously allocated by `CVodeInit`.
- `tret` (realtype *) the time reached by the solver (output).
- `yS` (N\_Vector *) array of computed forward sensitivity vectors. This vector array must be allocated by the user.

Return value  
The return value `flag` of `CVodeGetSens` is one of:
- `CV\_SUCCESS` : `CVodeGetSens` was successful.
- `CV\_MEM\_NULL` : `cvode\_mem` was `NULL`.
- `CV\_NO\_SENS` : Forward sensitivity analysis was not initialized.
- `CV\_BAD\_DKY` : `yS` is `NULL`.

Notes  
Note that the argument `tret` is an output for this function. Its value will be the same as that returned at the last `CVode` call.

The function `CVodeGetSensDky` computes the k-th derivatives of the interpolating polynomials for the sensitivity variables at time t. This function is called by `CVodeGetSens` with k = 0, but may also be called directly by the user.

**CVodeGetSensDky**

Call  
\[
\text{flag} = \text{CVodeGetSensDky}(\text{cvode\_mem}, \text{t}, \text{k}, \text{dkyS});
\]

Description  
The function `CVodeGetSensDky` returns derivatives of the sensitivity solution vectors after a successful return from `CVode`.

Arguments  
- `cvode\_mem` (void *) pointer to the memory previously allocated by `CVodeInit`.
- `t` (realtype) specifies the time at which sensitivity information is requested. The time \( t \) must fall within the interval defined by the last successful step taken by `CVODES`.
- `k` (int) order of derivatives.
- `dkyS` (N\_Vector *) array of \( N \) vectors containing the derivatives on output. The space for `dkyS` must be allocated by the user.

Return value  
The return value `flag` of `CVodeGetSensDky` is one of:
- `CV\_SUCCESS` : `CVodeGetSensDky` succeeded.
- `CV\_MEM\_NULL` : `cvode\_mem` was `NULL`.
- `CV\_NO\_SENS` : Forward sensitivity analysis was not initialized.
- `CV\_BAD\_DKY` : One of the vectors `dkyS` is `NULL`.
- `CV\_BAD\_K` : k is not in the range 0, 1, ..., `qlast`.
- `CV\_BAD\_T` : The time \( t \) is not in the allowed range.

Forward sensitivity solution vectors can also be extracted separately for each parameter in turn through the functions `CVodeGetSens1` and `CVodeGetSensDky1`, defined as follows:

**CVodeGetSens1**

Call  
\[
\text{flag} = \text{CVodeGetSens1}(\text{cvode\_mem}, \&\text{tret}, \text{is}, \text{yS});
\]

Description  
The function `CVodeGetSens1` returns the \( \text{is} \)-th sensitivity solution vector after a successful return from `CVode`.

Arguments  
- `cvode\_mem` (void *) pointer to the memory previously allocated by `CVodeInit`.
- `tret` (realtype *) the time reached by the solver (output).
Using CVODES for Forward Sensitivity Analysis

108

is  (int) specifies which sensitivity vector is to be returned (0 ≤ is < Ns).
yS  (N_Vector) the computed forward sensitivity vector. This vector array must
be allocated by the user.

Return value The return value flag of CVodeGetSens1 is one of:
CV_SUCCESS CVodeGetSens1 was successful.
CV_MEM_NULL cvoode_mem was NULL.
CV_NO_SENS Forward sensitivity analysis was not initialized.
CV_BAD_IS The index is is not in the allowed range.
CV_BAD_DKY yS is NULL.
CV_BAD_T The time t is not in the allowed range.

Notes Note that the argument tret is an output for this function. Its value will be the same
as that returned at the last CVode call.

CVodeGetSensDky1

Call  flag = CVodeGetSensDky1(cvode_mem, t, k, is, dkyS);

Description The function CVodeGetSensDky1 returns the k-th derivative of the is-th sensitivity
solution vector after a successful return from CVode.

Arguments cvode_mem (void *) pointer to the memory previously allocated by CVodeInit.
t (realtype) specifies the time at which sensitivity information is requested.  
The time t must fall within the interval defined by the last successful step 
taken by CVODES.
k (int) order of derivative.
is (int) specifies the sensitivity derivative vector to be returned (0 ≤ is < Ns).
dkyS (N_Vector) the vector containing the derivative. The space for dkyS must
be allocated by the user.

Return value The return value flag of CVodeGetSensDky1 is one of:
CV_SUCCESS CVodeGetQuadDky1 succeeded.
CV_MEM_NULL The pointer to cvoode_mem was NULL.
CV_NO_SENS Forward sensitivity analysis was not initialized.
CV_BAD_DKY dkyS or one of the vectors dkyS[i] is NULL.
CV_BAD_IS The index is is not in the allowed range.
CV_BAD_K k is not in the range 0, 1,..., qlast.
CV_BAD_T The time t is not in the allowed range.

5.2.6 Optional inputs for forward sensitivity analysis

Optional input variables that control the computation of sensitivities can be changed from their default
values through calls to CVodeSetSens* functions. Table 5.1 lists all forward sensitivity optional input
functions in CVODES which are described in detail in the remainder of this section.

Table 5.1: Forward sensitivity optional inputs

<table>
<thead>
<tr>
<th>Optional input</th>
<th>Routine name</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sensitivity scaling factors</td>
<td>CVodeSetSensParams</td>
<td>NULL</td>
</tr>
<tr>
<td>DQ approximation method</td>
<td>CVodeSetSensDQMethod</td>
<td>centered/0.0</td>
</tr>
<tr>
<td>Error control strategy</td>
<td>CVodeSetSensErrCon</td>
<td>SUNFALSE</td>
</tr>
<tr>
<td>Maximum no. of nonlinear iterations</td>
<td>CVodeSetSensMaxNonlinIters</td>
<td>3</td>
</tr>
</tbody>
</table>
5.2 User-callable routines for forward sensitivity analysis

**CVodeSetSensParams**

Call: `flag = CVodeSetSensParams(cvode_mem, p, pbar, plist);`

Description: The function `CVodeSetSensParams` specifies problem parameter information for sensitivity calculations.

Arguments:
- `cvode_mem` (void *) pointer to the CVODES memory block.
- `p` (realtype *) a pointer to the array of real problem parameters used to evaluate \( f(t, y, p) \). If non-NULL, `p` must point to a field in the user's data structure `user_data` passed to the right-hand side function. (See §5.1).
- `pbar` (realtype *) an array of \( N_s \) positive scaling factors. If non-NULL, `pbar` must have all its components \( > 0.0 \). (See §5.1).
- `plist` (int *) an array of \( N_s \) non-negative indices to specify which components \( p[i] \) to use in estimating the sensitivity equations. If non-NULL, `plist` must have all components \( \geq 0 \). (See §5.1).

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_NO_SENS` Forward sensitivity analysis was not initialized.
- `CV_Ill_INPUT` An argument has an illegal value.

Notes: This function must be preceded by a call to `CVodeSensInit` or `CVodeSensInit1`.

**CVodeSetSensDQMethod**

Call: `flag = CVodeSetSensDQMethod(cvode_mem, DQtype, DQrhomax);`

Description: The function `CVodeSetSensDQMethod` specifies the difference quotient strategy in the case in which the right-hand side of the sensitivity equations are to be computed by CVODES.

Arguments:
- `cvode_mem` (void *) pointer to the CVODES memory block.
- `DQtype` (int) specifies the difference quotient type. Its value can be `CV_CENTERED` or `CV_FORWARD`.
- `DQrhomax` (realtype) positive value of the selection parameter used in deciding switching between a simultaneous or separate approximation of the two terms in the sensitivity right-hand side.

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` An argument has an illegal value.

Notes: If `DQrhomax = 0.0`, then no switching is performed. The approximation is done simultaneously using either centered or forward finite differences, depending on the value of `DQtype`. For values of `DQrhomax \geq 1.0`, the simultaneous approximation is used whenever the estimated finite difference perturbations for states and parameters are within a factor of `DQrhomax`, and the separate approximation is used otherwise. Note that a value `DQrhomax < 1.0` will effectively disable switching. See §2.6 for more details.

The default value are `DQtype=CV_CENTERED` and `DQrhomax= 0.0`.

**CVodeSetSensErrCon**

Call: `flag = CVodeSetSensErrCon(cvode_mem, errconS);`

Description: The function `CVodeSetSensErrCon` specifies the error control strategy for sensitivity variables.
Using CVODES for Forward Sensitivity Analysis

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

errconS (booleantype) specifies whether sensitivity variables are to be included (SUNTRUE) or not (SUNFALSE) in the error control mechanism.

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  By default, errconS is set to SUNFALSE. If errconS=SUNTRUE then both state variables and sensitivity variables are included in the error tests. If errconS=SUNFALSE then the sensitivity variables are excluded from the error tests. Note that, in any event, all variables are considered in the convergence tests.

CVodeSetSensMaxNonlinIters

Call  
flag = CVodeSetSensMaxNonlinIters(cvode_mem, maxcorS);

Description  The function CVodeSetSensMaxNonlinIters specifies the maximum number of nonlinear solver iterations for sensitivity variables per step.

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

maxcorS (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_MEM_FAIL The SUNNONLINSOL module is NULL.

Notes  The default value is 3.

5.2.7 Optional outputs for forward sensitivity analysis

Optional output functions that return statistics and solver performance information related to forward sensitivity computations are listed in Table 5.2 and described in detail in the remainder of this section.

CVodeGetSensNumRhsEvals

Call  
flag = CVodeGetSensNumRhsEvals(cvode_mem, &nfSevals);

Description  The function CVodeGetSensNumRhsEvals returns the number of calls to the sensitivity right-hand side function.

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

<table>
<thead>
<tr>
<th>Optional output</th>
<th>Routine name</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of calls to sensitivity r.h.s. function</td>
<td>CVodeGetSensNumRhsEvals</td>
</tr>
<tr>
<td>No. of calls to r.h.s. function for sensitivity</td>
<td>CVodeGetNumRhsEvalSens</td>
</tr>
<tr>
<td>No. of sensitivity local error test failures</td>
<td>CVodeGetSensNumErrTestFails</td>
</tr>
<tr>
<td>No. of calls to lin. solv. setup routine for sens.</td>
<td>CVodeGetSensNumLinSolvSetups</td>
</tr>
<tr>
<td>Error weight vector for sensitivity variables</td>
<td>CVodeGetSensErrWeights</td>
</tr>
<tr>
<td>No. of sens. nonlinear solver iterations</td>
<td>CVodeGetSensNumNonlinSolvIters</td>
</tr>
<tr>
<td>No. of sens. convergence failures</td>
<td>CVodeGetSensNumNonlinSolvConvFails</td>
</tr>
<tr>
<td>No. of staggered nonlinear solver iterations</td>
<td>CVodeGetStgrSensNumNonlinSolvIters</td>
</tr>
<tr>
<td>No. of staggered convergence failures</td>
<td>CVodeGetStgrSensNumNonlinSolvConvFails</td>
</tr>
</tbody>
</table>
nfSevals (long int) number of calls to the sensitivity right-hand side 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.
- CV_NO_SENS: Forward sensitivity analysis was not initialized.

Notes In order to accommodate any of the three possible sensitivity solution methods, the default internal finite difference quotient functions evaluate the sensitivity right-hand sides one at a time. Therefore, nfSevals will always be a multiple of the number of sensitivity parameters (the same as the case in which the user supplies a routine of type CVSensRhs1Fn).

CTXodeGetNumRhsEvalsSens
Call flag = CVodeGetNumRhsEvalsSens(cvode_mem, &nfevalsS);
Description The function CVodeGetNumRhsEvalsSens returns the number of calls to the user’s right-hand side function due to the internal finite difference approximation of the sensitivity right-hand sides.
Arguments cvode_mem (void *) pointer to the CVODES memory block.
- nfevalsS (long int) number of calls to the user’s ODE right-hand side function for the evaluation of sensitivity right-hand sides.

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_NO_SENS: Forward sensitivity analysis was not initialized.

Notes This counter is incremented only if the internal finite difference approximation routines are used for the evaluation of the sensitivity right-hand sides.

CVodeGetSensNumErrTestFails
Call flag = CVodeGetSensNumErrTestFails(cvode_mem, &nSetfails);
Description The function CVodeGetSensNumErrTestFails returns the number of local error test failures for the sensitivity variables that have occurred.
Arguments cvode_mem (void *) pointer to the CVODES memory block.
- nSetfails (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.
- CV_NO_SENS: Forward sensitivity analysis was not initialized.

Notes This counter is incremented only if the sensitivity variables have been included in the error test (see CVodeSetSensErrCon in §5.2.6). Even in that case, this counter is not incremented if the ism=CV_SIMULTANEOUS sensitivity solution method has been used.

CVodeGetSensNumLinSolvSetups
Call flag = CVodeGetSensNumLinSolvSetups(cvode_mem, &nlinsetupsS);
Description The function CVodeGetSensNumLinSolvSetups returns the number of calls to the linear solver setup function due to forward sensitivity calculations.
Arguments cvode_mem (void *) pointer to the CVODES memory block.
Using CVODES for Forward Sensitivity Analysis

nlinsetupsS (long int) number of calls 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.
CV_NO_SENS  Forward sensitivity analysis was not initialized.

Notes This counter is incremented only if a nonlinear solver requiring a linear solve has been used and if either the ism = CV_STAGGERED or the ism = CV_STAGGERED1 sensitivity solution method has been specified (see §5.2.1).

CVodeGetSensStats

Call  flag = CVodeGetSensStats(cvode_mem, &nfSevals, &nfevalsS, &nSetfails, &nlinsetupsS);

Description The function CVodeGetSensStats returns all of the above sensitivity-related solver statistics as a group.

Arguments cvode_mem (void *) pointer to the CVODES memory block.
nfSevals (long int) number of calls to the sensitivity right-hand side function.
nfevalsS (long int) number of calls to the ODE right-hand side function for sensitivity evaluations.
nSetfails (long int) number of error test failures.
nlinsetupsS (long int) number of calls to the linear solver setup function.

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_NO_SENS  Forward sensitivity analysis was not initialized.

CVodeGetSensErrWeights

Call  flag = CVodeGetSensErrWeights(cvode_mem, eSweight);

Description The function CVodeGetSensErrWeights returns the sensitivity error weight vectors at the current time. These are the reciprocals of the $W_i$ of (2.8) for the sensitivity variables.

Arguments cvode_mem (void *) pointer to the CVODES memory block.
eSweight (N_Vector *) pointer to the array of error weight vectors.

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_NO_SENS  Forward sensitivity analysis was not initialized.

Notes The user must allocate memory for eweightS.

CVodeGetSensNumNonlinSolvIters

Call  flag = CVodeGetSensNumNonlinSolvIters(cvode_mem, &nSniters);

Description The function CVodeGetSensNumNonlinSolvIters returns the number of nonlinear iterations performed for sensitivity calculations.

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

Return value The return value flag (of type int) is one of:
5.2 User-callable routines for forward sensitivity analysis

CV_SUCCESS The optional output value has been successfully set.
CV_MEM_NULL The cvode_mem pointer is NULL.
CV_NOSENS Forward sensitivity analysis was not initialized.
CV_MEM_FAIL The SUNNONLINSOL module is NULL.

Notes
This counter is incremented only if ism was CV_STAGGERED or CV_STAGGERED1 (see §5.2.1).
In the CV_STAGGERED1 case, the value of nSniters is the sum of the number of nonlinear iterations performed for each sensitivity equation. These individual counters can be obtained through a call to CVodeGetStgrSensNumNonlinSolvIters (see below).

### Call
```c
flag = CVodeGetSensNumNonlinSolvConvFails(cvode_mem, &nSncfails);
```

### Description
The function CVodeGetSensNumNonlinSolvConvFails returns the number of nonlinear convergence failures that have occurred for sensitivity calculations.

### Arguments
- cvode_mem (void *) pointer to the CVODES memory block.
- nSncfails (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_NOSENS Forward sensitivity analysis was not initialized.
- CV_MEM_FAIL The SUNNONLINSOL module is NULL.

### Notes
This counter is incremented only if ism was CV_STAGGERED or CV_STAGGERED1 (see §5.2.1).
In the CV_STAGGERED1 case, the value of nSncfails is the sum of the number of nonlinear convergence failures that occurred for each sensitivity equation. These individual counters can be obtained through a call to CVodeGetStgrSensNumNonlinConvFails (see below).

### Call
```c
flag = CVodeGetSensNonlinSolvStats(cvode_mem, &nSniters, &nSncfails);
```

### Description
The function CVodeGetSensNonlinSolvStats returns the sensitivity-related nonlinear solver statistics as a group.

### Arguments
- cvode_mem (void *) pointer to the CVODES memory block.
- nSniters (long int) number of nonlinear iterations performed.
- nSncfails (long int) number of nonlinear convergence failures.

### 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_NOSENS Forward sensitivity analysis was not initialized.
- CV_MEM_FAIL The SUNNONLINSOL module is NULL.

### Call
```c
flag = CVodeGetStgrSensNumNonlinSolvIters(cvode_mem, nSTGR1niters);
```

### Description
The function CVodeGetStgrSensNumNonlinSolvIters returns the number of nonlinear iterations performed for each sensitivity equation separately, in the CV_STAGGERED1 case.

### Arguments
- cvode_mem (void *) pointer to the CVODES memory block.
Using CVODES for Forward Sensitivity Analysis

nSTGR1nitters (long int *) an array (of dimension Ns) which will be set with the number of nonlinear iterations performed for each sensitivity system individually.

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_NO_SENS Forward sensitivity analysis was not initialized.

Notes The user must allocate space for nSTGR1nitters.

CVodeGetStgrSensNumNonlinSolvConvFails

Call flag = CVodeGetStgrSensNumNonlinSolvConvFails(cvode_mem, nSTGR1ncfails);

Description The function CVodeGetStgrSensNumNonlinSolvConvFails returns the number of nonlinear convergence failures that have occurred for each sensitivity equation separately, in the CV_STAGGERED1 case.

Arguments cvode_mem (void *) pointer to the CVODES memory block.
- nSTGR1ncfails (long int *) an array (of dimension Ns) which will be set with the number of nonlinear convergence failures for each sensitivity system individually.

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_NO_SENS Forward sensitivity analysis was not initialized.

Notes The user must allocate space for nSTGR1ncfails.

5.3 User-supplied routines for forward sensitivity analysis

In addition to the required and optional user-supplied routines described in §4.6, when using CVODES for forward sensitivity analysis, the user has the option of providing a routine that calculates the right-hand side of the sensitivity equations (2.12).

By default, CVODES uses difference quotient approximation routines for the right-hand sides of the sensitivity equations. However, CVODES allows the option for user-defined sensitivity right-hand side routines (which also provides a mechanism for interfacing CVODES to routines generated by automatic differentiation).

5.3.1 Sensitivity equations right-hand side (all at once)

If the CV_SIMULTANEOUS or CV_STAGGERED approach was selected in the call to CVodeSensInit or CVodeSensInit1, the user may provide the right-hand sides of the sensitivity equations (2.12), for all sensitivity parameters at once, through a function of type CVSensRhsFn defined by:

CVSensRhsFn

definition typedef int (*CVSensRhsFn)(int Ns, realtype t,
N_Vector y, N_Vector ydot,
N_Vector *yS, N_Vector *ySdot,
void *user_data,
N_Vector tmp1, N_Vector tmp2);

Purpose This function computes the sensitivity right-hand side for all sensitivity equations at once. It must compute the vectors \( (\partial f/\partial y)_{s_i}(t) + (\partial f/\partial p_i) \) and store them in ySdot[1].
### Arguments

- **Ns** is the number of sensitivities.
- **t** is the current value of the independent variable.
- **y** is the current value of the state vector, \( y(t) \).
- **ydot** is the current value of the right-hand side of the state equations.
- **yS** contains the current values of the sensitivity vectors.
- **ySdot** is the output of `CVSensRhsFn`. On exit it must contain the sensitivity right-hand side vectors.
- **user_data** is a pointer to user data, the same as the `user_data` parameter passed to `CVodeSetUserData`.
- **tmp1**
- **tmp2** are `N_Vector`s of length `N` which can be used as temporary storage.

### Return value

A `CVSensRhsFn` should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODES will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and `CV_SRHSFUNC_FAIL` is returned).

### Notes

A sensitivity right-hand side function of type `CVSensRhsFn` is not compatible with the `CV_STAGGERED1` approach.

Allocation of memory for `ySdot` is handled within CVODES.

There are two situations in which recovery is not possible even if `CVSensRhsFn` function returns a recoverable error flag. One is when this occurs at the very first call to the `CVSensRhsFn` (in which case CVODES returns `CV_FIRST_SRHSFUNC_ERR`). The other is when a recoverable error is reported by `CVSensRhsFn` after an error test failure, while the linear multistep method order is equal to 1 (in which case CVODES returns `CV_UNREC_SRHSFUNC_ERR`).

### 5.3.2 Sensitivity equations right-hand side (one at a time)

Alternatively, the user may provide the sensitivity right-hand sides, one sensitivity parameter at a time, through a function of type `CVSensRhs1Fn`. Note that a sensitivity right-hand side function of type `CVSensRhs1Fn` is compatible with any valid value of the argument `ism` to `CVodeSensInit` and `CVodeSensInit1`, and is _required_ if `ism = CV_STAGGERED1` in the call to `CVodeSensInit1`. The type `CVSensRhs1Fn` is defined by

**CVSensRhs1Fn**

**Definition**

```c
typedef int (*CVSensRhs1Fn)(int Ns, realtype t,
                        N_Vector y, N_Vector ydot,
                        int iS, N_Vector yS, N_Vector ySdot,
                        void *user_data,
                        N_Vector tmp1, N_Vector tmp2);
```

**Purpose**

This function computes the sensitivity right-hand side for one sensitivity equation at a time. It must compute the vector \((\partial f/\partial y)s_i(t) + (\partial f/\partial p_i)\) for \(i = iS\) and store it in `ySdot`.

**Arguments**

- **Ns** is the number of sensitivities.
- **t** is the current value of the independent variable.
- **y** is the current value of the state vector, \( y(t) \).
- **ydot** is the current value of the right-hand side of the state equations.
- **iS** is the index of the parameter for which the sensitivity right-hand side must be computed (\(0 \leq iS < Ns\)).
- **yS** contains the current value of the `iS`-th sensitivity vector.
ySdot is the output of CVSensRhs1Fn. On exit it must contain the iS-th sensitivity right-hand side vector.

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

tmp1
tmp2 are N_Vectors of length N which can be used as temporary storage.

Return value A CVSensRhs1Fn should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODES will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and CV_SRHSFUNC_FAIL is returned).

Notes Allocation of memory for ySdot is handled within CVODES.

There are two situations in which recovery is not possible even if CVSensRhs1Fn function returns a recoverable error flag. One is when this occurs at the very first call to the CVSensRhs1Fn (in which case CVODES returns CV_FIRST_SRHSFUNC_ERR). The other is when a recoverable error is reported by CVSensRhs1Fn after an error test failure, while the linear multistep method order equal to 1 (in which case CVODES returns CV_UNREC_SRHSFUNC_ERR).

5.4 Integration of quadrature equations depending on forward sensitivities

CVODES provides support for integration of quadrature equations that depends not only on the state variables but also on forward sensitivities.

The following is an overview of the sequence of calls in a user’s main program in this situation. Steps that are unchanged from the skeleton program presented in §5.1 are grayed out.

1. Initialize parallel or multi-threaded environment, if appropriate
2. Set problem dimensions etc.
3. Set vectors of initial values
4. Create CVODES object
5. Initialize CVODES solver
6. Specify integration tolerances
7. Create matrix object
8. Create linear solver object
9. Set linear solver optional inputs
10. Attach linear solver module
11. Set optional inputs
12. Create nonlinear solver object
13. Attach nonlinear solver module
14. Set nonlinear solver optional inputs
15. Initialize sensitivity-independent quadrature problem
16. Define the sensitivity problem
17. Set sensitivity initial conditions
18. Activate sensitivity calculations
19. Set sensitivity tolerances
20. Set sensitivity analysis optional inputs
21. Create sensitivity nonlinear solver object
22. Attach the sensitivity nonlinear solver module
23. Set sensitivity nonlinear solver optional inputs
24. **Set vector of initial values for quadrature variables**
   Typically, the quadrature variables should be initialized to 0.
25. **Initialize sensitivity-dependent quadrature integration**
   Call CVodeQuadSensInit to specify the quadrature equation right-hand side function and to allocate internal memory related to quadrature integration. See §5.4.1 for details.
26. **Set optional inputs for sensitivity-dependent quadrature integration**
   Call CVodeSetQuadSensErrCon to indicate whether or not quadrature variables should be used in the step size control mechanism. If so, one of the CVodeQuadSens*tolerances functions must be called to specify the integration tolerances for quadrature variables. See §5.4.4 for details.
27. Advance solution in time
28. **Extract sensitivity-dependent quadrature variables**
   Call CVodeGetQuadSens, CVodeGetQuadSens1, CVodeGetQuadSensDky or CVodeGetQuadSensDky1 to obtain the values of the quadrature variables or their derivatives at the current time. See §5.4.3 for details.
29. Get optional outputs
30. Extract sensitivity solution
31. **Get sensitivity-dependent quadrature optional outputs**
   Call CVodeGetQuadSens* functions to obtain desired optional output related to the integration of sensitivity-dependent quadratures. See §5.4.5 for details.
32. Deallocate memory for solutions vector
33. Deallocate memory for sensitivity vectors
34. **Deallocate memory for sensitivity-dependent quadrature variables**
35. **Free solver memory**
36. Free nonlinear solver memory
37. Free vector specification memory
38. Free linear solver and matrix memory
39. Finalize MPI, if used
118 Using CVODES for Forward Sensitivity Analysis

Note: CVodeQuadSensInit (step 25 above) can be called and quadrature-related optional inputs (step 26 above) can be set anywhere between steps 16 and 27.

5.4.1 Sensitivity-dependent quadrature initialization and deallocation

The function CVodeQuadSensInit activates integration of quadrature equations depending on sensitivities and allocates internal memory related to these calculations. If rhsQS is input as NULL, then CVODES uses an internal function that computes difference quotient approximations to the functions \( \bar{q}_i = q_y s_i + q_p, \) in the notation of (2.10). The form of the call to this function is as follows:

\[
\text{CVodeQuadSensInit}
\]

Call

\[
\text{flag} = \text{CVodeQuadSensInit}(\text{cvode_mem}, \text{rhsQS}, \text{yQS0});
\]

Description The function CVodeQuadSensInit provides required problem specifications, allocates internal memory, and initializes quadrature integration.

Arguments

- cvode_mem (void *) pointer to the CVODES memory block returned by CVodeCreate.
- rhsQS (CVQuadSensRhsFn) is the C function which computes \( f_{QS} \), the right-hand side of the sensitivity-dependent quadrature equations (for full details see §5.4.6).
- yQS0 (NVector *) contains the initial values of sensitivity-dependent quadratures.

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

- CV_SUCCESS The call to CVodeQuadSensInit was successful.
- CVODE_MEM_NULL The CVODES memory was not initialized by a prior call to CVodeCreate.
- CVODE_MEM_FAIL A memory allocation request failed.
- CV_NOSENS The sensitivities were not initialized by a prior call to CVodeSensInit or CVodeSensInit1.
- CV_Ill_INPUT The parameter yQS0 is NULL.

Notes Before calling CVodeQuadSensInit, the user must enable the sensitivities by calling CVodeSensInit or CVodeSensInit1.

If an error occurred, CVodeQuadSensInit also sends an error message to the error handler function.

In terms of the number of quadrature variables \( N_q \) and maximum method order \( \text{maxord} \), the size of the real workspace is increased as follows:

- Base value: \( \text{lenrw} = \text{lenrw} + (\text{maxord} \times 5)N_q \)

and the size of the integer workspace is increased as follows:

- Base value: \( \text{leniw} = \text{leniw} + (\text{maxord} \times 5)N_q \)

The function CVodeQuadSensReInit, useful during the solution of a sequence of problems of same size, reinitializes quadrature-related internal memory and must follow a call to CVodeQuadSensInit. The number \( N_q \) of quadratures as well as the number \( N_s \) of sensitivities are assumed to be unchanged from the prior call to CVodeQuadSensInit. The call to the CVodeQuadSensReInit function has the form:
5.4 Integration of quadrature equations depending on forward sensitivities

**CVodeQuadSensReInit**

**Call**

\[ \text{flag} = \text{CVodeQuadSensReInit}(\text{cvode_mem}, yQS0); \]

**Description**
The function `CVodeQuadSensReInit` provides required problem specifications and reinitializes the sensitivity-dependent quadrature integration.

**Arguments**

- `cvode_mem` (void *) pointer to the CVODES memory block.
- `yQS0` (N_Vector *) contains the initial values of sensitivity-dependent quadratures.

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

- CV_SUCCESS: The call to `CVodeQuadSensReInit` was successful.
- CVODE_MEM_NULL: The CVODES memory was not initialized by a prior call to `CVodeCreate`.
- CV_NO_SENS: Memory space for the sensitivity calculation was not allocated by a prior call to `CVodeSensInit` or `CVodeSensInit1`.
- CV_NO_QUADSENS: Memory space for the sensitivity quadratures integration was not allocated by a prior call to `CVodeQuadSensInit`.
- CV_Ill_INPUT: The parameter `yQS0` is NULL.

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

**CVodeQuadSensFree**

**Call**

\[ \text{CVodeQuadSensFree}(\text{cvode_mem}); \]

**Description**
The function `CVodeQuadSensFree` frees the memory allocated for sensitivity quadrature integration.

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

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

**Notes**
In general, `CVodeQuadSensFree` need not be called by the user, as it is invoked automatically by `CVodeFree`.

### 5.4.2 CVODES solver function

Even if quadrature integration was enabled, the call to the main solver function `CVode` is exactly the same as in §4.5.6. However, in this case the return value `flag` can also be one of the following:

- CV_QSRHFSFUNC_ERR: The sensitivity quadrature right-hand side function failed in an unrecoverable manner.
- CV_FIRST_QSRHFSFUNC_ERR: The sensitivity quadrature right-hand side function failed at the first call.
- CV_REPTD_QSRHFSFUNC_ERR: Convergence test failures occurred too many times due to repeated recoverable errors in the quadrature right-hand side function. This flag will also be returned if the quadrature right-hand side function had repeated recoverable errors during the estimation of an initial step size (assuming the sensitivity quadrature variables are included in the error tests).

### 5.4.3 Sensitivity-dependent quadrature extraction functions

If sensitivity-dependent quadratures have been initialized by a call to `CVodeQuadSensInit`, or reinitialized by a call to `CVodeQuadSensReInit`, then CVODES computes a solution, sensitivity vectors, and quadratures depending on sensitivities at time \( t \). However, `CVode` will still return only the solution \( y \). Sensitivity-dependent quadratures can be obtained using one of the following functions:
CVodeGetQuadSens

Call  

\[ \text{flag} = \text{CVodeGetQuadSens}(\text{cvode\_mem}, \&tret, \text{yQS}); \]

Description  
The function CVodeGetQuadSens returns the quadrature sensitivities solution vectors after a successful return from CVode.

Arguments  
- \text{cvode\_mem} (void *) pointer to the memory previously allocated by CVodeInit.
- \text{tret} (realtype) the time reached by the solver (output).
- \text{yQS} (N\_Vector *) array of \(N_s\) computed sensitivity-dependent quadrature vectors. This vector array must be allocated by the user.

Return value  
The return value \text{flag} of CVodeGetQuadSens is one of:
- CV\_SUCCESS: CVodeGetQuadSens was successful.
- CVODE\_MEM\_NULL: cvode\_mem was NULL.
- CV\_NO\_SENS: Sensitivities were not activated.
- CV\_NO\_QUADSENS: Quadratures depending on the sensitivities were not activated.
- CV\_BAD\_DKY: yQS or one of the \text{yQS}[i] is NULL.

The function CVodeGetQuadSensDky computes the \(k\)-th derivatives of the interpolating polynomials for the sensitivity-dependent quadrature variables at time \(t\). This function is called by CVodeGetQuadSens with \(k = 0\), but may also be called directly by the user.

CVodeGetQuadSensDky

Call  

\[ \text{flag} = \text{CVodeGetQuadSensDky}(\text{cvode\_mem}, t, k, \text{dkyQS}); \]

Description  
The function CVodeGetQuadSensDky returns derivatives of the quadrature sensitivities solution vectors after a successful return from CVode.

Arguments  
- \text{cvode\_mem} (void *) pointer to the memory previously allocated by CVodeInit.
- \(t\) (realtype) the time at which information is requested. The time \(t\) must fall within the interval defined by the last successful step taken by CVODES.
- \(k\) (int) order of the requested derivative.
- \text{dkyQS} (N\_Vector *) array of \(N_s\) the vector containing the derivatives on output. This vector array must be allocated by the user.

Return value  
The return value \text{flag} of CVodeGetQuadSensDky is one of:
- CV\_SUCCESS: CVodeGetQuadSensDky succeeded.
- CVODE\_MEM\_NULL: The pointer to cvode\_mem was NULL.
- CV\_NO\_SENS: Sensitivities were not activated.
- CV\_NO\_QUADSENS: Quadratures depending on the sensitivities were not activated.
- CV\_BAD\_DKY: dkyQS or one of the vectors dkyQS[i] is NULL.
- CV\_BAD\_K: \(k\) is not in the range 0, 1, ..., qlast.
- CV\_BAD\_T: The time \(t\) is not in the allowed range.

Quadrature sensitivity solution vectors can also be extracted separately for each parameter in turn through the functions CVodeGetQuadSens1 and CVodeGetQuadSensDky1, defined as follows:

CVodeGetQuadSens1

Call  

\[ \text{flag} = \text{CVodeGetQuadSens1}(\text{cvode\_mem}, \&tret, \text{is}, \text{yQS}); \]

Description  
The function CVodeGetQuadSens1 returns the \(\text{is}\)-th sensitivity of quadratures after a successful return from CVode.

Arguments  
- \text{cvode\_mem} (void *) pointer to the memory previously allocated by CVodeInit.
- \text{tret} (realtype) the time reached by the solver (output).
- \text{is} (int) specifies which sensitivity vector is to be returned (\(0 \leq \text{is} < N_s\)).
5.4 Integration of quadrature equations depending on forward sensitivities

\( y_{QS} \) \( (\text{N}_{Vector}) \) the computed sensitivity-dependent quadrature vector. This vector array must be allocated by the user.

Return value The return value flag of CVodeGetQuadSens1 is one of:
- CV_SUCCESS CVodeGetQuadSens1 was successful.
- CVODE_MEM_NULL cvode_mem was NULL.
- CV_NO_SENS Forward sensitivity analysis was not initialized.
- CV_NO_QUADSENS Quadratures depending on the sensitivities were not activated.
- CV_BAD_IS The index is is not in the allowed range.
- CV_BAD_DKY yQS is NULL.

\[ \text{CVodeGetQuadSensDky1} \]

Call \( \text{flag} = \text{CVodeGetQuadSensDky1}(\text{cvode_mem}, t, k, is, dkyQS); \)

Description The function CVodeGetQuadSensDky1 returns the \( k \)-th derivative of the \( is \)-th sensitivity solution vector after a successful return from CVode.

Arguments
- \( \text{cvode_mem} \) (void *) pointer to the memory previously allocated by CVodeInit.
- \( t \) (realtype) specifies the time at which sensitivity information is requested. The time \( t \) must fall within the interval defined by the last successful step taken by cvodes.
- \( k \) (int) order of derivative.
- \( is \) (int) specifies the sensitivity derivative vector to be returned (\( 0 \leq is < N_s \)).
- \( dkyQS \) (N_Vector) the vector containing the derivative on output. The space for \( dkyQS \) must be allocated by the user.

Return value The return value flag of CVodeGetQuadSensDky1 is one of:
- CV_SUCCESS CVodeGetQuadDky1 succeeded.
- CVODE_MEM_NULL cvode_mem was NULL.
- CV_NO_SENS Forward sensitivity analysis was not initialized.
- CV_NO_QUADSENS Quadratures depending on the sensitivities were not activated.
- CV_BAD_DKY dkyQS is NULL.
- CV_BAD_IS The index is is not in the allowed range.
- CV_BAD_K \( k \) is not in the range \( 0, 1, \ldots, q\text{last} \).
- CV_BAD_T The time \( t \) is not in the allowed range.

5.4.4 Optional inputs for sensitivity-dependent quadrature integration

cvodes provides the following optional input functions to control the integration of sensitivity-dependent quadrature equations.

\[ \text{CVodeSetQuadSensErrCon} \]

Call \( \text{flag} = \text{CVodeSetQuadSensErrCon}(\text{cvode_mem}, \text{errconQS}) \)

Description The function CVodeSetQuadSensErrCon specifies whether or not the quadrature variables are to be used in the step size control mechanism. If they are, the user must call one of the functions CVodeQuadSensSS tolerances, CVodeQuadSensSVtolerances, or CVodeQuadSensEE tolerances to specify the integration tolerances for the quadrature variables.

Arguments
- \( \text{cvode_mem} \) (void *) pointer to the CVODES memory block.
- \( \text{errconQS} \) (booleantype) specifies whether sensitivity quadrature variables are to be included (\text{SUNTRUE}) or not (\text{SUNFALSE}) in the error control mechanism.
Return value The return value flag (of type int) is one of:
- **CV_SUCCESS** The optional value has been successfully set.
- **CVODE_MEM_NULL** cvode_mem is NULL.
- **CV_NO_SENS** Sensitivities were not activated.
- **CV_NO_QUADSENS** Quadratures depending on the sensitivities were not activated.

Notes
- By default, *errconQS* is set to SUNFALSE.
- It is illegal to call `CVodeSetQuadSensErrCon` before a call to `CVodeQuadSensInit`.

If the quadrature variables are part of the step size control mechanism, one of the following functions must be called to specify the integration tolerances for quadrature variables.

### CVodeQuadSensSSStolerances

**Call**

```c
flag = CVodeQuadSensSSStolerances(cvode_mem, reltolQS, abstolQS);
```

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

**Arguments**
- `cvode_mem` (void *) pointer to the CVODES memory block.
- `reltolQS` (realtype) is the scalar relative error tolerance.
- `abstolQS` (realtype*) is a pointer to an array containing the *Ns* scalar absolute error tolerances.

**Return value**
The return value flag (of type int) is one of:
- **CV_SUCCESS** The optional value has been successfully set.
- **CVODE_MEM_NULL** The cvode_mem pointer is NULL.
- **CV_NO_SENS** Sensitivities were not activated.
- **CV_NO_QUADSENS** Quadratures depending on the sensitivities were not activated.
- **CV_NO_INPUT** One of the input tolerances was negative.

### CVodeQuadSensSVtolerances

**Call**

```c
flag = CVodeQuadSensSVtolerances(cvode_mem, reltolQS, abstolQS);
```

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

**Arguments**
- `cvode_mem` (void *) pointer to the CVODES memory block.
- `reltolQS` (realtype) is the scalar relative error tolerance.
- `abstolQS` (N Vector*) is an array of *Ns* variables of type N Vector. The N Vector abstolS[is] specifies the vector tolerances for is-th quadrature sensitivity.

**Return value**
The return value flag (of type int) is one of:
- **CV_SUCCESS** The optional value has been successfully set.
- **CV_NO_QUAD** Quadrature integration was not initialized.
- **CVODE_MEM_NULL** The cvode_mem pointer is NULL.
- **CV_NO_SENS** Sensitivities were not activated.
- **CV_NO_QUADSENS** Quadratures depending on the sensitivities were not activated.
- **CV_NO_INPUT** One of the input tolerances was negative.
5.4 Integration of quadrature equations depending on forward sensitivities

**CVodeQuadSensEEtolerances**

Call  
\[ \text{flag} = \text{CVodeQuadSensEEtolerances}(\text{cvode\_mem}); \]

Description  
A call to the function `CVodeQuadSensEEtolerances` specifies that the tolerances for the sensitivity-dependent quadratures should be estimated from those provided for the pure quadrature variables.

Arguments  
- `cvode\_mem` (void *) pointer to the CVODES memory block.

Return value  
The return value `flag` (of type `int`) is one of:
- `CV\_SUCCESS`: The optional value has been successfully set.
- `CVODE\_MEM\_NULL`: The `cvode\_mem` pointer is NULL.
- `CV\_NO\_SENS`: Sensitivities were not activated.
- `CV\_NO\_QUADSENS`: Quadratures depending on the sensitivities were not activated.

Notes  
When `CVodeQuadSensEEtolerances` is used, before calling `CVode`, integration of pure quadratures must be initialized (see 4.7.1) and tolerances for pure quadratures must be also specified (see 4.7.4).

5.4.5 Optional outputs for sensitivity-dependent quadrature integration

CVODES provides the following functions that can be used to obtain solver performance information related to quadrature integration.

**CVodeGetQuadSensNumRhsEvals**

Call  
\[ \text{flag} = \text{CVodeGetQuadSensNumRhsEvals}(\text{cvode\_mem}, &\text{nrhsQSevals}); \]

Description  
The function `CVodeGetQuadSensNumRhsEvals` returns the number of calls made to the user’s quadrature right-hand side function.

Arguments  
- `cvode\_mem` (void *) pointer to the CVODES memory block.
- `nrhsQSevals` (long int) number of calls made to the user’s `rhsQS` function.

Return value  
The return value `flag` (of type `int`) is one of:
- `CV\_SUCCESS`: The optional output value has been successfully set.
- `CVODE\_MEM\_NULL`: The `cvode\_mem` pointer is NULL.
- `CV\_NO\_QUADSENS`: Sensitivity-dependent quadrature integration has not been initialized.

**CVodeGetQuadSensNumErrTestFails**

Call  
\[ \text{flag} = \text{CVodeGetQuadSensNumErrTestFails}(\text{cvode\_mem}, &\text{nQSetfails}); \]

Description  
The function `CVodeGetQuadSensNumErrTestFails` returns the number of local error test failures due to quadrature variables.

Arguments  
- `cvode\_mem` (void *) pointer to the CVODES memory block.
- `nQSetfails` (long int) number of error test failures due to quadrature variables.

Return value  
The return value `flag` (of type `int`) is one of:
- `CV\_SUCCESS`: The optional output value has been successfully set.
- `CVODE\_MEM\_NULL`: The `cvode\_mem` pointer is NULL.
- `CV\_NO\_QUADSENS`: Sensitivity-dependent quadrature integration has not been initialized.
CVodeGetQuadSensErrWeights

Call
\[
\text{flag} = \text{CVodeGetQuadSensErrWeights(cvode\_mem, eQSweight)};
\]

Description The function CVodeGetQuadSensErrWeights returns the quadrature error weights at the current time.

Arguments
- cvode\_mem \((\text{void } *)\) pointer to the CVODES memory block.
- eQSweight \((\text{N\_Vector } *)\) array of quadrature error weight vectors 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.
- CVODE\_MEM\_NULL The cvode\_mem pointer is NULL.
- CV\_NO\_QUADSENS Sensitivity-dependent quadrature integration has not been initialized.

Notes The user must allocate memory for eQSweight.

If quadratures were not included in the error control mechanism (through a call to CVodeSetQuadSensErrCon with errconQS = SUNTRUE), then this function does not set the eQSweight array.

CVodeGetQuadSensStats

Call
\[
\text{flag} = \text{CVodeGetQuadSensStats(cvode\_mem, &nrhsQSevals, &nQSetfails)};
\]

Description The function CVodeGetQuadSensStats returns the CVODES integrator statistics as a group.

Arguments
- cvode\_mem \((\text{void } *)\) pointer to the CVODES memory block.
- nrhsQSevals \((\text{long int})\) number of calls to the user’s rhsQS function.
- nQSetfails \((\text{long int})\) number of error test failures due to quadrature variables.

Return value The return value flag (of type int) is one of
- CV\_SUCCESS the optional output values have been successfully set.
- CVODE\_MEM\_NULL the cvode\_mem pointer is NULL.
- CV\_NO\_QUADSENS Sensitivity-dependent quadrature integration has not been initialized.

5.4.6 User-supplied function for sensitivity-dependent quadrature integration

For the integration of sensitivity-dependent quadrature equations, the user must provide a function that defines the right-hand side of those quadrature equations. For the sensitivities of quadratures (2.10) with integrand \(q\), the appropriate right-hand side functions are given by: \(q_i = q_y s_i + q_p\). This user function must be of type CVQuadSensRhsFn defined as follows:

CVQuadSensRhsFn

Definition
\[
\text{typedef int (*)(CVQuadSensRhsFn)(int Ns, realtype t, N\_Vector y,}
\]
\[
\text{N\_Vector yS, N\_Vector yQdot,}
\]
\[
\text{N\_Vector *rhsvalQS, void *user\_data,}
\]
\[
\text{N\_Vector tmp1, N\_Vector tmp2)};
\]

Purpose This function computes the sensitivity quadrature equation right-hand side for a given value of the independent variable \(t\) and state vector \(y\).

Arguments
- Ns is the number of sensitivity vectors.
- t is the current value of the independent variable.
- y is the current value of the dependent variable vector, \(y(t)\).
- yS is an array of Ns variables of type N\_Vector containing the dependent sensitivity vectors \(s_i\).
5.5 Note on using partial error control

\( y'Q \) is the current value of the quadrature right-hand side, \( q \).

\( \text{rhsvalQS} \) array of \( N_s \) vectors to contain the right-hand sides.

\( \text{user-data} \) is the \( \text{user-data} \) pointer passed to \( \text{CVodeSetUserData} \).

\( \text{tmp1} \)

\( \text{tmp2} \) are \( \text{N_Vector} \)s which can be used as temporary storage.

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

Notes

Allocation of memory for \( \text{rhsvalQS} \) is automatically handled within \( \text{CVODES} \).

Here \( y \) is of type \( \text{N_Vector} \) and \( yS \) is a pointer to an array containing \( N_s \) vectors of type \( \text{N_Vector} \). It is the user’s responsibility to access the vector data consistently (including the use of the correct accessor macros from each \( \text{NVECTOR} \) implementation).

For the sake of computational efficiency, the vector functions in the two \( \text{NVECTOR} \) implementations provided with \( \text{CVODES} \) do not perform any consistency checks with respect to their \( \text{N_Vector} \) arguments (see §7.2 and §7.3).

There are two situations in which recovery is not possible even if \( \text{CVQuadSensRhsFn} \) function returns a recoverable error flag. One is when this occurs at the very first call to the \( \text{CVQuadSensRhsFn} \) (in which case \( \text{CVODES} \) returns \( \text{CV_FIRST_QS_RHSFUNC_ERR} \)). The other is when a recoverable error is reported by \( \text{CVQuadSensRhsFn} \) after an error test failure, while the linear multistep method order is equal to 1 (in which case \( \text{CVODES} \) returns \( \text{CV_UNREC_QS_RHSFUNC_ERR} \)).

5.5 Note on using partial error control

For some problems, when sensitivities are excluded from the error control test, the behavior of \( \text{CVODES} \) may appear at first glance to be erroneous. One would expect that, in such cases, the sensitivity variables would not influence in any way the step size selection. A comparison of the solver diagnostics reported for \( \text{cvsdenx} \) and the second run of the \( \text{cvsfdwdenx} \) example in [43] indicates that this may not always be the case.

The short explanation of this behavior is that the step size selection implemented by the error control mechanism in \( \text{CVODES} \) is based on the magnitude of the correction calculated by the nonlinear solver. As mentioned in §5.2.1, even with partial error control selected (in the call to \( \text{CVodeSetSensErrCon} \)), the sensitivity variables are included in the convergence tests of the nonlinear solver.

When using the simultaneous corrector method (§2.6), the nonlinear system that is solved at each step involves both the state and sensitivity equations. In this case, it is easy to see how the sensitivity variables may affect the convergence rate of the nonlinear solver and therefore the step size selection. The case of the staggered corrector approach is more subtle. After all, in this case (\( \text{ism} = \text{CV_STAGGERED} \) or \( \text{CV_STAGGERED1} \) in the call to \( \text{CVodeSensInit/CVodeSensInit1} \)), the sensitivity variables at a given step are computed only once the solver for the nonlinear state equations has converged. However, if the nonlinear system corresponding to the sensitivity equations has convergence problems, \( \text{CVODES} \) will attempt to improve the initial guess by reducing the step size in order to provide a better prediction of the sensitivity variables. Moreover, even if there are no convergence failures in the solution of the sensitivity system, \( \text{CVODES} \) may trigger a call to the linear solver’s setup routine which typically involves reevaluation of Jacobian information (Jacobian approximation in the case of \( \text{CVDENSE} \) and \( \text{CVBAND} \), or preconditioner data in the case of the Krylov solvers). The new Jacobian information will be used by subsequent calls to the nonlinear solver for the state equations and, in this way, potentially affect the step size selection.

When using the simultaneous corrector method it is not possible to decide whether nonlinear solver convergence failures or calls to the linear solver setup routine have been triggered by convergence problems due to the state or the sensitivity equations. When using one of the staggered corrector methods however, these situations can be identified by carefully monitoring the diagnostic information.
provided through optional outputs. If there are no convergence failures in the sensitivity nonlinear solver, and none of the calls to the linear solver setup routine were made by the sensitivity nonlinear solver, then the step size selection is not affected by the sensitivity variables.

Finally, the user must be warned that the effect of appending sensitivity equations to a given system of ODEs on the step size selection (through the mechanisms described above) is problem-dependent and can therefore lead to either an increase or decrease of the total number of steps that CVODES takes to complete the simulation. At first glance, one would expect that the impact of the sensitivity variables, if any, would be in the direction of increasing the step size and therefore reducing the total number of steps. The argument for this is that the presence of the sensitivity variables in the convergence test of the nonlinear solver can only lead to additional iterations (and therefore a smaller final iteration error), or to additional calls to the linear solver setup routine (and therefore more up-to-date Jacobian information), both of which will lead to larger steps being taken by CVODES. However, this is true only locally. Overall, a larger integration step taken at a given time may lead to step size reductions at later times, due to either nonlinear solver convergence failures or error test failures.
Chapter 6

Using CVODES for Adjoint Sensitivity Analysis

This chapter describes the use of CVODES to compute sensitivities of derived functions using adjoint sensitivity analysis. As mentioned before, the adjoint sensitivity module of CVODES provides the infrastructure for integrating backward in time any system of ODEs that depends on the solution of the original IVP, by providing various interfaces to the main CVODES integrator, as well as several supporting user-callable functions. For this reason, in the following sections we refer to the backward problem and not to the adjoint problem when discussing details relevant to the ODEs that are integrated backward in time. The backward problem can be the adjoint problem (2.20) or (2.23), and can be augmented with some quadrature differential equations.

CVODES 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.

We begin with a brief overview, in the form of a skeleton user program. Following that are detailed descriptions of the interface to the various user-callable functions and of the user-supplied functions that were not already described in Chapter 4.

6.1 A skeleton of the user’s main program

The following is a skeleton of the user’s main program as an application of CVODES. The user program is to have these steps in the order indicated, unless otherwise noted. For the sake of brevity, we defer many of the details to the later sections. As in §4.4, most steps are independent of the NVECTOR, SUNMATRIX, SUNLINSOL, and SUNNONLINSOL implementations used. For the steps that are not, refer to Chapters 7, 8, 9, and 10 for the specific name of the function to be called or macro to be referenced.

Steps that are unchanged from the skeleton programs presented in §4.4, §5.1, and §5.4, are grayed out.

1. Include necessary header files

The cvodes.h header file also defines additional types, constants, and function prototypes for the adjoint sensitivity module user-callable functions. In addition, the main program should include an NVECTOR implementation header file (for the particular implementation used), and, if a nonlinear solver requiring a linear solver (e.g., the default Newton iteration) will be used, the header file of the desired linear solver module.

2. Initialize parallel or multi-threaded environment, if appropriate

Forward problem

3. Set problem dimensions etc. for the forward problem
4. Set initial conditions for the forward problem
5. Create CVODES object for the forward problem
6. Initialize CVODES for the forward problem
7. Specify integration tolerances for forward problem
8. Create matrix object for the forward problem
9. Create linear solver object for the forward problem
10. Set linear solver optional inputs for the forward problem
11. Attach linear solver module for the forward problem
12. Set optional inputs for the forward problem
13. Create nonlinear solver object for the forward problem
14. Attach nonlinear solver module for the forward problem
15. Set nonlinear solver optional inputs for the forward problem
16. Initialize quadrature problem or problems for forward problems, using CVodeQuadInit and/or CVodeQuadSensInit.
17. Initialize forward sensitivity problem
18. Specify root finding
19. Allocate space for the adjoint computation
   Call CVodeAdjInit() to allocate memory for the combined forward-backward problem (see §6.2.1 for details). This call requires Nd, the number of steps between two consecutive checkpoints. CVodeAdjInit also specifies the type of interpolation used (see §2.7.1).
20. Integrate forward problem
   Call CVodeF, a wrapper for the CVODES main integration function CVode, either in CV_NORMAL mode to the time tout or in CV_ONE_STEP mode inside a loop (if intermediate solutions of the forward problem are desired (see §6.2.2)). The final value of tret is then the maximum allowable value for the endpoint T of the backward problem.

Backward problem(s)

21. Set problem dimensions etc. for the backward problem
   This generally includes the backward problem vector length NB, and possibly the local vector length NBlocal.
22. Set initial values for the backward problem
   Set the endpoint time tB0 = T, and set the corresponding vector yB0 at which the backward problem starts.
23. Create the backward problem
   Call CVodeCreateB, a wrapper for CVodeCreate, to create the CVODES memory block for the new backward problem. Unlike CVodeCreate, the function CVodeCreateB does not return a pointer to the newly created memory block (see §6.2.3). Instead, this pointer is attached to the internal adjoint memory block (created by CVodeAdjInit) and returns an identifier called which that the user must later specify in any actions on the newly created backward problem.
24. **Allocate memory for the backward problem**

Call CVodeInitB (or CVodeInitBS, when the backward problem depends on the forward sensitivities). The two functions are actually wrappers for CVodeInit and allocate internal memory, specify problem data, and initialize CVODEs at tB0 for the backward problem (see §6.2.3).

25. **Specify integration tolerances for backward problem**

Call CVodeSS tolerancesB(...) or CVodeSV tolerancesB(...) to specify a scalar relative tolerance and scalar absolute tolerance or scalar relative tolerance and a vector of absolute tolerances, respectively. The functions are wrappers for CVodeSS tolerances and CVodeSV tolerances, but they require an extra argument which, the identifier of the backward problem returned by CVodeCreateB. See §6.2.4 for more information.

26. **Create matrix object for the backward problem**

If a nonlinear solver requiring a linear solve will be used (e.g., the default Newton iteration) and the linear solver will be a direct linear solver, then a template Jacobian matrix must be created by calling the appropriate constructor function defined by the particular SUNMATRIX implementation. For the SUNDIALS-supplied SUNMATRIX implementations, the matrix object may be created using a call of the form

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

or

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

or

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

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

Note also that it is not required to use the same matrix type for both the forward and the backward problems.

27. **Create linear solver object for the backward problem**

If a nonlinear solver requiring a linear solver is chosen (e.g., the default Newton iteration), then the desired linear solver object for the backward problem must be created by calling the appropriate constructor function defined by the particular SUNLINSOL implementation. For any of the SUNDIALS-supplied SUNLINSOL implementations, the linear solver object may be created using a call of the form

```c
SUNLinearSolver LS = SUNLinSol*(...);
```

where * can be replaced with “Dense”, “SPGMR”, or other options, as discussed in §4.5.3 and Chapter 9.

Note that it is not required to use the same linear solver module for both the forward and the backward problems; for example, the forward problem could be solved with the SUNLINSOL_DENSE linear solver module and the backward problem with SUNLINSOL_SPGMR linear solver module.

28. **Set linear solver interface optional inputs for the backward problem**

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 9 for details.

29. **Attach linear solver module for the backward problem**

If a nonlinear solver requiring a linear solver is chosen for the backward problem (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):

\[
\text{ier} = \text{CVodeSetLinearSolverB}(\ldots);
\]

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

\[
\text{ier} = \text{CVDiagB}(\ldots);
\]

30. **Set optional inputs for the backward problem**

Call CVodeSet\textbullet\textcircled{B} functions to change from their default values any optional inputs that control the behavior of CVODES. Unlike their counterparts for the forward problem, these functions take an extra argument which, the identifier of the backward problem returned by CVodeCreate\textbullet\textcircled{B} (see §6.2.8).

31. **Create nonlinear solver object for the backward problem (optional)**

If using a non-default nonlinear solver for the backward problem, then create the desired nonlinear solver object by calling the appropriate constructor function defined by the particular SUNNONLINSOL implementation (e.g., NLSB = SUNNonlinSol_\textbullet\textcircled{\ldots}; where \textbullet\textcircled{\ldots} is the name of the nonlinear solver (see Chapter 10 for details).

32. **Attach nonlinear solver module for the backward problem (optional)**

If using a non-default nonlinear solver for the backward problem, then initialize the nonlinear solver interface by attaching the nonlinear solver object by calling

\[
\text{ier} = \text{CVodeSetNonlinearSolverB}(\text{cvode_mem, NLSB});
\]

(see §4.5.4 for details).

33. **Initialize quadrature calculation**

If additional quadrature equations must be evaluated, call CVodeQuadInit\textbullet\textcircled{B} or CVodeQuadInit\textbullet\textcircled{BS} (if quadrature depends also on the forward sensitivities) as shown in §6.2.10.1. These functions are wrappers around CVodeQuadInit and can be used to initialize and allocate memory for quadrature integration. Optionally, call CVodeSetQuad\textbullet\textcircled{B} functions to change from their default values optional inputs that control the integration of quadratures during the backward phase.

34. **Integrate backward problem**

Call CVode\textbullet\textcircled{B}, a second wrapper around the CVODES main integration function CVode, to integrate the backward problem from \textcircled{t\textbf{B}}0 (see §6.2.6). This function can be called either in CV\_\textcircled{NORMAL} or CV\_\textcircled{ONE\_STEP} mode. Typically, CVode\textbullet\textcircled{B} will be called in CV\_\textcircled{NORMAL} mode with an end time equal to the initial time \textcircled{t\textbf{0}} of the forward problem.

35. **Extract quadrature variables**

If applicable, call CVodeGetQuad\textbullet\textcircled{B}, a wrapper around CVodeGetQuad, to extract the values of the quadrature variables at the time returned by the last call to CVode\textbullet\textcircled{B}. See §6.2.10.2.

36. **Deallocate memory**

Upon completion of the backward integration, call all necessary deallocation functions. These include appropriate destructors for the vectors \textcircled{y} and \textcircled{yB}, a call to CVodeFree to free the CVODES memory block for the forward problem. If one or more additional Adjoint Sensitivity Analyses are to be done for this problem, a call to CVodeAdjFree (see §6.2.1) may be made to free and deallocate memory allocated for the backward problems, followed by a call to CVodeAdjInit.

37. **Free the nonlinear solver memory for the forward and backward problems**

38. **Free linear solver and matrix memory for the forward and backward problems**

39. **Finalize MPI, if used**
The above user interface to the adjoint sensitivity module in CVODES was motivated by the desire to keep it as close as possible in look and feel to the one for ODE IVP integration. Note that if steps (21)-(35) are not present, a program with the above structure will have the same functionality as one described in §4.4 for integration of ODEs, albeit with some overhead due to the checkpointing scheme.

If there are multiple backward problems associated with the same forward problem, repeat steps (21)-(35) above for each successive backward problem. In the process, each call to CVodeCreateB creates a new value of the identifier which.

6.2 User-callable functions for adjoint sensitivity analysis

6.2.1 Adjoint sensitivity allocation and deallocation functions

After the setup phase for the forward problem, but before the call to CVodeF, memory for the combined forward-backward problem must be allocated by a call to the function CVodeAdjInit. The form of the call to this function is

```c
CVodeAdjInit
Call flag = CVodeAdjInit(cvode_mem, Nd, interpType);
Description The function CVodeAdjInit updates CVODES memory block by allocating the internal memory needed for backward integration. Space is allocated for the Nd = N_d interpolation data points, and a linked list of checkpoints is initialized.
Arguments cvode_mem (void *) is the pointer to the CVODES memory block returned by a previous call to CVodeCreate.
Nd (long int) is the number of integration steps between two consecutive checkpoints.
interpType (int) specifies the type of interpolation used and can be CV_POLYNOMIAL or CV_HERMITE, indicating variable-degree polynomial and cubic Hermite interpolation, respectively (see §2.7.1).
Return value The return value flag (of type int) is one of:
CV_SUCCESS CVodeAdjInit was successful.
CV_MEM_FAIL A memory allocation request has failed.
CV_MEM_NULL cvode_mem was NULL.
CV_ILL_INPUT One of the parameters was invalid: Nd was not positive or interpType is not one of the CV_POLYNOMIAL or CV_HERMITE.
Notes The user must set Nd so that all data needed for interpolation of the forward problem solution between two checkpoints fits in memory. CVodeAdjInit attempts to allocate space for (2Nd+3) variables of type N_Vector.
If an error occurred, CVodeAdjInit also sends a message to the error handler function.
```

```c
CVodeAdjReInit
Call flag = CVodeAdjReInit(cvode_mem);
Description The function CVodeAdjReInit reinitializes the CVODES memory block for ASA, assuming that the number of steps between check points and the type of interpolation remain unchanged.
Arguments cvode_mem (void *) is the pointer to the CVODES memory block returned by a previous call to CVodeCreate.
Return value The return value flag (of type int) is one of:
CV_SUCCESS CVodeAdjReInit was successful.
```
CV_MEM_NULL cvode_mem was NULL.

CV_NO_ADJ The function CVodeAdjInit was not previously called.

Notes The list of checkpoints (and associated memory) is deleted. The list of backward problems is kept. However, new backward problems can be added to this list by calling CVodeCreateB. If a new list of backward problems is also needed, then free the adjoint memory (by calling CVodeAdjFree) and reinitialize ASA with CVodeAdjInit.

The CVODES memory for the forward and backward problems can be reinitialized separately by calling CVodeReInit and CVodeReInitB, respectively.

```c
CVodeAdjFree
```

Call CVodeAdjFree(cvode_mem);

Description The function CVodeAdjFree frees the memory related to backward integration allocated by a previous call to CVodeAdjInit.

Arguments The only argument is the CVODES memory block pointer returned by a previous call to CVodeCreate.

Return value The function CVodeAdjFree has no return value.

Notes This function frees all memory allocated by CVodeAdjInit. This includes workspace memory, the linked list of checkpoints, memory for the interpolation data, as well as the CVODES memory for the backward integration phase. Unless one or more further calls to CVodeAdjInit are to be made, CVodeAdjFree should not be called by the user, as it is invoked automatically by CVodeFree.

### 6.2.2 Forward integration function

The function CVodeF is very similar to the CVODES function CVode (see §4.5.6) in that it integrates the solution of the forward problem and returns the solution in $y$. At the same time, however, CVodeF stores checkpoint data every $N_d$ integration steps. CVodeF can be called repeatedly by the user. Note that CVodeF is used only for the forward integration pass within an Adjoint Sensitivity Analysis. It is not for use in Forward Sensitivity Analysis; for that, see Chapter 5. The call to this function has the form

```c
CVodeF
```

Call flag = CVodeF(cvode_mem, tout, yret, &tret, itask, &ncheck);

Description The function CVodeF integrates the forward problem over an interval in $t$ and saves checkpointing data.

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

tout (realtype) the next time at which a computed solution is desired.

yret (N_Vector) the computed solution vector $y$.

tret (realtype) the time reached by the solver (output).

itask (int) a flag indicating the job of the solver for the next step. The CV_NORMAL task is to have the solver take internal steps until it has reached or just passed the user-specified tout parameter. The solver then interpolates in order to return an approximate value of $y(tout)$. The CV_ONE_STEP option tells the solver to just take one internal step and return the solution at the point reached by that step.

ncheck (int) the number of (internal) checkpoints stored so far.
Return value
On return, CVodeF returns the vector \( y_{\text{ret}} \) and a corresponding independent variable value \( t = t_{\text{ret}} \), such that \( y_{\text{ret}} \) is the computed value of \( y(t) \). Additionally, it returns in \( n_{\text{check}} \) the number of internal checkpoints saved; the total number of checkpoint intervals is \( n_{\text{check}} + 1 \). The return value \( \text{flag} \) (of type int) will be one of the following. For more details see §4.5.6.

- **CV\_SUCCESS**: CVodeF succeeded.
- **CV\_TSTOP\_RETURN**: CVodeF succeeded by reaching the optional stopping point.
- **CV\_ROOT\_RETURN**: CVodeF succeeded and found one or more roots. In this case, \( t_{\text{ret}} \) is the location of the root. If \( n_{\text{rtfn}} > 1 \), call CVodeGetRootInfo to see which \( g_i \) were found to have a root.
- **CV\_NO\_MALLOC**: The function CVodeInit has not been previously called.
- **CV\_ILL\_INPUT**: One of the inputs to CVodeF is illegal.
- **CV\_TOO\_MUCH\_WORK**: The solver took \( mx_{\text{step}} \) internal steps but could not reach \( tout \).
- **CV\_TOO\_MUCH\_ACC**: The solver could not satisfy the accuracy demanded by the user for some internal step.
- **CV\_ERR\_FAILURE**: Error test failures occurred too many times during one internal time step or occurred with \( |h| = h_{\text{min}} \).
- **CV\_CONV\_FAILURE**: Convergence test failures occurred too many times during one internal time step or occurred with \( |h| = h_{\text{min}} \).
- **CV\_LSETUP\_FAIL**: The linear solver’s setup function failed in an unrecoverable manner.
- **CV\_LSOLVE\_FAIL**: The linear solver’s solve function failed in an unrecoverable manner.
- **CV\_NO\_ADJ**: The function CVodeAdjInit has not been previously called.
- **CV\_MEM\_FAIL**: A memory allocation request has failed (in an attempt to allocate space for a new checkpoint).

Notes
All failure return values are negative and therefore a test \( \text{flag} < 0 \) will trap all CVodeF failures.

At this time, CVodeF stores checkpoint information in memory only. Future versions will provide for a safeguard option of dumping checkpoint data into a temporary file as needed. The data stored at each checkpoint is basically a snapshot of the CVODES internal memory block and contains enough information to restart the integration from that time and to proceed with the same step size and method order sequence as during the forward integration.

In addition, CVodeF also stores interpolation data between consecutive checkpoints so that, at the end of this first forward integration phase, interpolation information is already available from the last checkpoint forward. In particular, if no checkpoints were necessary, there is no need for the second forward integration phase.

It is illegal to change the integration tolerances between consecutive calls to CVodeF, as this information is not captured in the checkpoint data.

### 6.2.3 Backward problem initialization functions

The functions CVodeCreateB and CVodeInitB (or CVodeInitBS) must be called in the order listed. They instantiate a CVODES solver object, provide problem and solution specifications, and allocate internal memory for the backward problem.

**CVodeCreateB**

Call

```c
flag = CVodeCreateB(cvode_mem, lmmB, &which);
```

Description
The function CVodeCreateB instantiates a CVODES solver object and specifies the solution method for the backward problem.

Arguments
- `cvode_mem` (void *) pointer to the CVODES memory block returned by CVodeCreate.
lmmB (int) specifies the linear multistep method and may be one of two possible values: CV_ADAMS or CV_BDF.

which (int) contains the identifier assigned by CVODES for the newly created backward problem. Any call to CVode*B functions requires such an identifier.

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

- CV_SUCCESS The call to CVodeCreateB was successful.
- CV_MEM_NULL cvode_mem was NULL.
- CV_NO_ADJ The function CVodeAdjInit has not been previously called.
- CV_MEM_FAIL A memory allocation request has failed.

There are two initialization functions for the backward problem – one for the case when the backward problem does not depend on the forward sensitivities, and one for the case when it does. These two functions are described next.

The function CVodeInitB initializes the backward problem when it does not depend on the forward sensitivities. It is essentially a wrapper for CVodeInit with some particularization for backward integration, as described below.

```c
CVodeInitB
```

Call
flag = CVodeInitB(cvode_mem, which, rhsB, tB0, yB0);

Description The function CVodeInitB provides problem specification, allocates internal memory, and initializes the backward problem.

Arguments
- cvode_mem (void *) pointer to the CVODES memory block returned by CVodeCreate.
- which (int) represents the identifier of the backward problem.
- rhsB (CVRhsFnB) is the C function which computes fB, the right-hand side of the backward ODE problem. This function has the form rhsB(t, y, yB, yBdot, user_dataB) (for full details see §6.3.1).
- tB0 (realtype) specifies the endpoint T where final conditions are provided for the backward problem, normally equal to the endpoint of the forward integration.
- yB0 (N_Vector) is the initial value (at t = tB0) of the backward solution.

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

- CV_SUCCESS The call to CVodeInitB was successful.
- CV_NO_MALLOC The function CVodeInit has not been previously called.
- CV_MEM_NULL cvode_mem was NULL.
- CV_NO_ADJ The function CVodeAdjInit has not been previously called.
- CV_BAD_TB0 The final time tB0 was outside the interval over which the forward problem was solved.
- CV_I LL_INPUT The parameter which represented an invalid identifier, or either yB0 or rhsB was NULL.

Notes The memory allocated by CVodeInitB is deallocated by the function CVodeAdjFree.

For the case when backward problem also depends on the forward sensitivities, user must call CVodeInitBS instead of CVodeInitB. Only the third argument of each function differs between these two functions.

```c
CVodeInitBS
```

Call
flag = CVodeInitBS(cvode_mem, which, rhsBS, tB0, yB0);

Description The function CVodeInitBS provides problem specification, allocates internal memory, and initializes the backward problem.

Arguments
- cvode_mem (void *) pointer to the CVODES memory block returned by CVodeCreate.
6.2 User-callable functions for adjoint sensitivity analysis

which (int) represents the identifier of the backward problem.

rhsBS (CVRhsFnBS) is the C function which computes \( fB \), the right-hand side of the backward ODE problem. This function has the form \( \text{rhsBS}(t, y, yS, yB, yBdot, \text{user\_dataB}) \) (for full details see §6.3.2).

tB0 (realtype) specifies the endpoint \( T \) where final conditions are provided for the backward problem.

yB0 (NVector) is the initial value (at \( t = tB0 \)) of the backward solution.

Return value The return value \( \text{flag} \) (of type int) will be one of the following:

CV\_SUCCESS The call to CVodeInitB was successful.

CV\_NO\_MALLOC The function CVodeInit has not been previously called.

CV\_MEM\_NULL cvode_mem was NULL.

CV\_NO\_ADJ The function CVodeAdjInit has not been previously called.

CV\_BAD\_TB0 The final time \( tB0 \) was outside the interval over which the forward problem was solved.

CV\_ILL\_INPUT The parameter which represented an invalid identifier, either yB0 or rhsBS was NULL, or sensitivities were not active during the forward integration.

Notes The memory allocated by CVodeInitBS is deallocated by the function CVodeAdjFree.

The function CVodeReInitB reinitializes CVODES for the solution of a series of backward problems, each identified by a value of the parameter which. CVodeReInitB is essentially a wrapper for CVodeReInit, and so all details given for CVodeReInit in §4.5.10 apply here. Also note that CVodeReInitB can be called to reinitialize the backward problem even it has been initialized with the sensitivity-dependent version CVodeInitBS. Before calling CVodeReInitB for a new backward problem, call any desired solution extraction functions CVodeGet** associated with the previous backward problem. The call to the CVodeReInitB function has the form

\[
\text{CVodeReInitB} \quad \text{flag} = \text{CVodeReInitB}(\text{cvode\_mem}, \text{which}, \text{tB0}, \text{yB0})
\]

Description The function CVodeReInitB reinitializes a CVODES backward problem.

Arguments cvode_mem (void *) pointer to CVODES memory block returned by CVodeCreate.

which (int) represents the identifier of the backward problem.

tB0 (realtype) specifies the endpoint \( T \) where final conditions are provided for the backward problem.

yB0 (N\_Vector) is the initial value (at \( t = tB0 \)) of the backward solution.

Return value The return value \( \text{flag} \) (of type int) will be one of the following:

CV\_SUCCESS The call to CVodeReInitB was successful.

CV\_NO\_MALLOC The function CVodeInit has not been previously called.

CV\_MEM\_NULL The cvode_mem memory block pointer was NULL.

CV\_NO\_ADJ The function CVodeAdjInit has not been previously called.

CV\_BAD\_TB0 The final time \( tB0 \) is outside the interval over which the forward problem was solved.

CV\_ILL\_INPUT The parameter which represented an invalid identifier, or \( yB0 \) was NULL.

6.2.4 Tolerance specification functions for backward problem

One of the following two functions must be called to specify the integration tolerances for the backward problem. Note that this call must be made after the call to CVodeInitB or CVodeInitBS.
Using CVODES for Adjoint Sensitivity Analysis

**CVodeSS tolerances**

**Call**

\[
\text{flag} = \text{CVodeSS tolerances}(\text{cvoode mem}, \text{which}, \text{reltolB}, \text{abstolB});
\]

**Description**
The function CVodeSS tolerances specifies scalar relative and absolute tolerances.

**Arguments**
- `cvoode mem` (void *) pointer to the CVODES memory block returned by CVodeCreate.
- `which` (int) represents the identifier of the backward problem.
- `reltolB` (realtype) is the scalar relative error tolerance.
- `abstolB` (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 CVodeSS tolerances was successful.
- **CV_MEM_NULL** The CVODES memory block was not initialized through a previous call to CVodeCreate.
- **CV_NO_MALLOC** The allocation function CVodeInit has not been called.
- **CV_NO_ADJ** The function CVodeAdjInit has not been previously called.
- **CV_ILL_INPUT** One of the input tolerances was negative.

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

### 6.2.5 Linear solver initialization functions for backward problem

All CVODES linear solver modules available for forward problems are available for the backward problem. They should be created as for the forward problem and then attached to the memory structure for the backward problem using the following functions.

**CVodeSetLinearSolver**

**Call**

\[
\text{flag} = \text{CVodeSetLinearSolver}(\text{cvoode mem}, \text{which}, \text{LS}, \text{A});
\]

**Description**
The function CVodeSetLinearSolver attaches a generic SUNLINSOL object LS and corresponding template Jacobian SUNMATRIX object A to CVODES, initializing the CVLS linear solver interface for solution of the backward problem.

**Arguments**
- `cvoode mem` (void *) pointer to the CVODES memory block.
6.2 User-callable functions for adjoint sensitivity analysis

which (int) represents the identifier of the backward problem returned by CVodeCreateB. 

LS (SUNLinearSolver) SUNLINSOL object to use for solving linear systems for the backward problem. 

A (SUNMatrix) SUNMATRIX object for used as a template for the Jacobian for the backward problem (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 solver is not compatible with the current NVECTOR module. 

CVLS_MEM_FAIL A memory allocation request failed. 

CVLS_NO_ADJ The function CVAdjInit has not been previously called. 

CVLS_ILL_INPUT The parameter which represented an invalid identifier. 

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 8 for further information). 

The previous routines CVDlsSetLinearSolverB and CVSpilsSetLinearSolverB 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. 

CVDiagB 

Call flag = CVDiagB(cvode_mem, which); 

Description The function CVDiagB selects the CVDIAG linear solver for the solution of the backward problem. 

The user’s main program must include the cvodes_diag.h header file. 

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

which (int) represents the identifier of the backward problem returned by CVodeCreateB. 

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

CVDIAG_SUCCESS The CVDIAG initialization was successful. 

CVDIAG_MEM_NULL The cvode_mem pointer is NULL. 

CVDIAG_ILL_INPUT The CVDIAG solver is not compatible with the current NVECTOR module. 

CVDIAG_MEM_FAIL A memory allocation request failed. 

Notes The CVDIAG solver is the simplest of all of the available CVODES linear solver interfaces. The CVDIAG solver uses an approximate diagonal Jacobian formed by way of a difference quotient. The user does not have the option of supplying a function to compute an approximate diagonal Jacobian. 

6.2.6 Backward integration function 

The function CVodeB performs the integration of the backward problem. It is essentially a wrapper for the CVODES main integration function CVode and, in the case in which checkpoints were needed, it evolves the solution of the backward problem through a sequence of forward-backward integration pairs between consecutive checkpoints. The first run of each pair integrates the original IVP forward
in time and stores interpolation data; the second run integrates the backward problem backward in
time and performs the required interpolation to provide the solution of the IVP to the backward
problem.

The function CVodeB does not return the solution $y_B$ itself. To obtain that, call the function
CVodeGetB, which is also described below.

The CVodeB function does not support rootfinding, unlike CVodeF, which supports the finding of
roots of functions of $(t,y)$. If rootfinding was performed by CVodeF, then for the sake of efficiency, it
should be disabled for CVodeB by first calling CVodeRootInit with $nrtfn = 0$.

The call to CVodeB has the form

```c
CVodeB
Call
flag = CVodeB(cvode_mem, tBout, itaskB);
```

**Description** The function CVodeB integrates the backward ODE problem.

**Arguments**
- `cvode_mem` (void *) pointer to the CVODES memory returned by CVodeCreate.
- `tBout` (realtype) the next time at which a computed solution is desired.
- `itaskB` (int) a flag indicating the job of the solver for the next step. The CV_NORMAL
task is to have the solver take internal steps until it has reached or just
passed the user-specified value `tBout`. The solver then interpolates in order
to return an approximate value of $y_B(tBout)$. The CV_ONE_STEP option tells
the solver to take just one internal step in the direction of `tBout` and return.

**Return value** The return value `flag` (of type int) will be one of the following. For more details see
§4.5.6.
- CV_SUCCESS CVodeB succeeded.
- CV_MEM_NULL `cvode_mem` was NULL.
- CV_NO_ADJ The function CVodeAdjInit has not been previously called.
- CV_NO_BCK No backward problem has been added to the list of backward prob-
lems by a call to CVodeCreateB
- CV_NO_FWD The function CVodeF has not been previously called.
- CV_ILLEGAL_INPUT One of the inputs to CVodeB is illegal.
- CV_BAD_ITASK The `itaskB` argument has an illegal value.
- CV_TOO_MUCH_WORK The solver took `mxstep` internal steps but could not reach tBout.
- CV_TOO_MUCH_ACC The solver could not satisfy the accuracy demanded by the user for
some internal step.
- CV_ERR_FAILURE Error test failures occurred too many times during one internal time
step.
- CV_CONV_FAILURE Convergence test failures occurred too many times during one internal time
step.
- CV_LSETUP_FAIL The linear solver’s setup function failed in an unrecoverable manner.
- CV_SOLVE_FAIL The linear solver’s solve function failed in an unrecoverable manner.
- CV_BCKMEM_NULL The solver memory for the backward problem was not created with
a call to CVodeCreateB.
- CV_BAD_TBOUT The desired output time `tBout` is outside the interval over which the
forward problem was solved.
- CV_REIFWD_FAIL Reinitialization of the forward problem failed at the first checkpoint
(corresponding to the initial time of the forward problem).
- CV_FWD_FAIL An error occurred during the integration of the forward problem.

**Notes** All failure return values are negative and therefore a test `flag < 0` will trap all CVodeB
failures.
In the case of multiple checkpoints and multiple backward problems, a given call to CVodeB in CV_ONE_STEP mode may not advance every problem one step, depending on the relative locations of the current times reached. But repeated calls will eventually advance all problems to tBout.

To obtain the solution \( y_B \) to the backward problem, call the function CVodeGetB as follows:

```c
CVodeGetB
Call
flag = CVodeGetB(cvode_mem, which, &tret, yB);
Description
The function CVodeGetB provides the solution \( y_B \) of the backward ODE problem.
Arguments
 cvode_mem (void *) pointer to the CVODES memory returned by CVodeCreate.
 which (int) the identifier of the backward problem.
 tret (realtype) the time reached by the solver (output).
 yB (N_Vector) the backward solution at time tret.
Return value
The return value flag (of type int) will be one of the following.
 CV_SUCCESS CVodeGetB was successful.
 CV_MEM_NULL cvode_mem is NULL.
 CV_NO_ADJ The function CVodeAdjInit has not been previously called.
 CV_ILL_INPUT The parameter which is an invalid identifier.
Notes
The user must allocate space for yB.
To obtain the solution associated with a given backward problem at some other time within the last integration step, first obtain a pointer to the proper CVODES memory structure by calling CVodeGetAdjCVodeBmem and then use it to call CVodeGetDky.

6.2.7 Adjoint sensitivity optional input

At any time during the integration of the forward problem, the user can disable the checkpointing of the forward sensitivities by calling the following function:

```c
CVodeAdjSetNoSensi
Call
flag = CVodeAdjSetNoSensi(cvode_mem);
Description
The function CVodeAdjSetNoSensi instructs CVodeF not to save checkpointing data for forward sensitivities anymore.
Arguments
 cvode_mem (void *) pointer to the CVODES memory block.
Return value
The return value flag (of type int) is one of:
 CV_SUCCESS The call to CVodeCreateB was successful.
 CV_MEM_NULL cvode_mem was NULL.
 CV_NO_ADJ The function CVodeAdjInit has not been previously called.

6.2.8 Optional input functions for the backward problem

6.2.8.1 Main solver optional input functions

The adjoint module in CVODES provides wrappers for most of the optional input functions defined in §4.5.7.1. The only difference is that the user must specify the identifier which of the backward problem within the list managed by CVODES.

The optional input functions defined for the backward problem are:
flag = CVodeSetNonlinearSolverB(cvode_mem, which, NLSB);
flag = CVodeSetUserDataB(cvode_mem, which, user_dataB);
flag = CVodeSetMaxOrdB(cvode_mem, which, maxordB);
flag = CVodeSetMaxNumStepsB(cvode_mem, which, mxstepsB);
flag = CVodeSetInitStepB(cvode_mem, which, hinB);
flag = CVodeSetMinStepB(cvode_mem, which, hminB);
flag = CVodeSetMaxStepB(cvode_mem, which, hmaxB);
flag = CVodeSetStabLimDetB(cvode_mem, which, stldetB);
flag = CVodeSetConstraintsB(cvode_mem, which, constraintsB);

Their return value flag (of type int) can have any of the return values of their counterparts, but it can also be CV_NO_ADJ if CVodeAdjInit has not been called, or CV_ILL_INPUT if which was an invalid identifier.

### 6.2.8.2 Linear solver interface optional input functions

When using matrix-based linear solver modules, the CVLS solver interface needs a function to compute an approximation to the Jacobian for the backward problem. This Jacobian evaluation function can be attached through a call to either CVodeSetJacFnB or CVodeSetJacFnBS, with the second used when the backward problem depends on the forward sensitivities.

#### CVodeSetJacFnB

**Call**

flag = CVodeSetJacFnB(cvode_mem, which, jacB);

**Description**
The function CVodeSetJacFnB specifies the Jacobian approximation function to be used for the backward problem.

**Arguments**
- cvode_mem (void *) pointer to the CVODES memory returned by CVodeCreate.
- which (int) represents the identifier of the backward problem.
- jacB (CVLsJacFnB) user-defined Jacobian approximation function.

**Return value**
The return value flag (of type int) is one of:
- CVLS_SUCCESS CVodeSetJacFnB succeeded.
- CVLS_MEM_NULL cvode_mem was NULL.
- CVLS_NO_ADJ The function CVodeAdjInit has not been previously called.
- CVLS_LMEM_NULL The linear solver has not been initialized with a call to CVodeSetLinearSolverB.
- CVLS_ILL_INPUT The parameter which represented an invalid identifier.

**Notes**
The function type CVLsJacFnB is described in §6.3.5.

The previous routine CVDlsSetJacFnB 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.

#### CVodeSetJacFnBS

**Call**

flag = CVodeSetJacFnBS(cvode_mem, which, jacBS);

**Description**
The function CVodeSetJacFnBS specifies the Jacobian approximation function to be used for the backward problem, in the case where the backward problem depends on the forward sensitivities.

**Arguments**
- cvode_mem (void *) pointer to the CVODES memory returned by CVodeCreate.
- which (int) represents the identifier of the backward problem.
- jacBS (CVLsJacFnBS) user-defined Jacobian approximation function.

**Return value**
The return value flag (of type int) is one of:
6.2 User-callable functions for adjoint sensitivity analysis

CVLS_SUCCESS CVodeSetJacFnBS succeeded.
CVLS_MEM_NULL cvode_mem was NULL.
CVLS_NO_ADJ The function CVodeAdjInit has not been previously called.
CVLS_LMEM_NULL The linear solver has not been initialized with a call to CVodeSetLinearSolverB.
CVLS_ILL_INPUT The parameter which represented an invalid identifier.

Notes The function type CVLsJacFnBS is described in §6.3.5.

The previous routine CVLsSetJacFnBs 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.

[CVodeSetJacTimesB]

Call flag = CVodeSetJacTimesB(cvode_mem, which, jsetupB, jtvB);

Description The function CVodeSetJacTimesB specifies the Jacobian-vector setup and product functions to be used.

Arguments cvode_mem (void *) pointer to the CVODES memory block.
which (int) the identifier of the backward problem.
jsetupB (CVLsJacTimesSetupFnB) user-defined function to set up the Jacobian-vector product. Pass NULL if no setup is necessary.
jtvB (CVLsJacTimesVecFnB) 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 cvode_mem was NULL.
CVLS_LMEM_NULL The CVLS linear solver has not been initialized.
CVLS_NO_ADJ The function CVodeAdjInit has not been previously called.
CVLS_ILL_INPUT The parameter which represented an invalid identifier.

Notes The function types CVLsJacTimesVecFnB and CVLsJacTimesSetupFnB are described in §6.3.6.

The previous routine CVSpilsSetJacTimesB 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.

[CVodeSetJacTimesBS]

Call flag = CVodeSetJacTimesBS(cvode_mem, which, jtvBS);

Description The function CVodeSetJacTimesBS specifies the Jacobian-vector setup and product functions to be used, in the case where the backward problem depends on the forward sensitivities.

Arguments cvode_mem (void *) pointer to the CVODES memory block.
which (int) the identifier of the backward problem.
jsetupBS (CVLsJacTimesSetupFnBS) user-defined function to set up the Jacobian-vector product. Pass NULL if no setup is necessary.
jtvBS (CVLsJacTimesVecFnBS) 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 cvode_mem was NULL.
CVLS_LMEM_NULL The CVLS linear solver has not been initialized.
The function CVodeAdjInit has not been previously called.

The parameter which represented an invalid identifier.

The function types CVLsJacTimesVecFnBS and CVLsJacTimesSetupFnBS are described in §6.3.6.

The previous routine CVSpilsSetJacTimesBS 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.

CVodeSetPreconditionerB

Call flag = CVodeSetPreconditionerB(cvode_mem, which, psetupB, psolveB);

Description The function CVodeSetPrecSolveFnB specifies the preconditioner setup and solve functions for the backward integration.

Arguments cvode_mem (void *) pointer to the CVODES memory block.
which (int) the identifier of the backward problem.
psetupB (CVLsPrecSetupFnB) user-defined preconditioner setup function.
psolveB (CVLsPrecSolveFnB) user-defined preconditioner solve 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 cvode_mem was NULL.
CVLS_LMEM_NULL The CVLS linear solver has not been initialized.
CVLS_NO_ADJ The function CVodeAdjInit has not been previously called.
CVLS_Ill_INPUT The parameter which represented an invalid identifier.

Notes The function types CVLsPrecSolveFnB and CVLsPrecSetupFnB are described in §6.3.8 and §6.3.9, respectively. The psetupB argument may be NULL if no setup operation is involved in the preconditioner.

The previous routine CVSpilsSetPrecSolveFnB 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.

CVodeSetPreconditionerBS

Call flag = CVodeSetPreconditionerBS(cvode_mem, which, psetupBS, psolveBS);

Description The function CVodeSetPrecSolveFnBS specifies the preconditioner setup and solve functions for the backward integration, in the case where the backward problem depends on the forward sensitivities.

Arguments cvode_mem (void *) pointer to the CVODES memory block.
which (int) the identifier of the backward problem.
psetupBS (CVLsPrecSetupFnBS) user-defined preconditioner setup function.
psolveBS (CVLsPrecSolveFnBS) user-defined preconditioner solve 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 cvode_mem was NULL.
CVLS_LMEM_NULL The CVLS linear solver has not been initialized.
CVLS_NO_ADJ The function CVodeAdjInit has not been previously called.
CVLS_Ill_INPUT The parameter which represented an invalid identifier.
Notes  The function types CVodePrecSolveFnBS and CVodePrecSetupFnBS are described in §6.3.8 and §6.3.9, respectively. The psetupBS argument may be NULL if no setup operation is involved in the preconditioner.

The previous routine CVSpilsSetPrecSolveFnBS 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.

CVodeSetEpsLinB

Call  flag = CVodeSetEpsLinB(cvode_mem, which, eplifacB);

Description  The function CVodeSetEpsLinB specifies the factor by which the Krylov linear solver’s convergence test constant is reduced from the nonlinear iteration test constant. This routine can be used in both the cases whereby the backward problem does and does not depend on the forward sensitivities.

Arguments  cvode_mem (void *) pointer to the CVODES memory block.
which (int) the identifier of the backward problem.
eplifacB (realtype) value of the convergence test constant reduction factor (≥ 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  cvode_mem was NULL.
CVLS_LMEM_NULL  The CVLS linear solver has not been initialized.
CVLS_NO_ADJ  The function CVodeAdjInit has not been previously called.
CVLS_ILL_INPUT  The parameter which represented an invalid identifier, or eplifacB was negative.

Notes  The default value is 0.05. Passing a value eplifacB= 0.0 also indicates using the default value.

The previous routine CVSpilsSetEpsLinB 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.

6.2.9 Optional output functions for the backward problem

The user of the adjoint module in CVODES has access to any of the optional output functions described in §4.5.9, both for the main solver and for the linear solver modules. The first argument of these CVodeGet* and CVode*Get* functions is the pointer to the CVODES memory block for the backward problem. In order to call any of these functions, the user must first call the following function to obtain this pointer.

CVodeGetAdjCVodeBmem

Call  cvode_memB = CVodeGetAdjCVodeBmem(cvode_mem, which);

Description  The function CVodeGetAdjCVodeBmem returns a pointer to the CVODES memory block for the backward problem.

Arguments  cvode_mem (void *) pointer to the CVODES memory block created by CVodeCreate.
which (int) the identifier of the backward problem.

Return value  The return value, cvode_memB (of type void *), is a pointer to the CVODES memory for the backward problem.

Notes  The user should not modify cvode_memB in any way.

Optional output calls should pass cvode_memB as the first argument; for example, to get the number of integration steps: flag = CVodeGetNumSteps(cvodes_memB, &nsteps).
To get values of the *forward* solution during a backward integration, use the following function. The input value of \( t \) would typically be equal to that at which the backward solution has just been obtained with \texttt{CVodeGetB}. In any case, it must be within the last checkpoint interval used by \texttt{CVodeB}.

```c
CVodeGetAdjY
```

**Call**

\[
\text{flag} = \text{CVodeGetAdjY} (\text{cvode_mem, } t, y);
\]

**Description**
The function \texttt{CVodeGetAdjY} returns the interpolated value of the forward solution \( y \) during a backward integration.

**Arguments**
- \texttt{cvode_mem} (\texttt{void *} pointer to the CVODES memory block created by \texttt{CVodeCreate}).
- \( t \) (\texttt{realtype}) value of the independent variable at which \( y \) is desired (input).
- \( y \) (\texttt{N_Vector}) forward solution \( y(t) \).

**Return value**
The return value \( \text{flag} \) (of type \texttt{int}) is one of:
- \texttt{CV_SUCCESS} \texttt{CVodeGetAdjY} was successful.
- \texttt{CV_MEM_NULL} \texttt{cvode_mem} was NULL.
- \texttt{CV_GETY_BADT} The value of \( t \) was outside the current checkpoint interval.

**Notes**
The user must allocate space for \( y \).

```c
CVodeGetAdjCheckPointsInfo
```

**Call**

\[
\text{flag} = \text{CVodeGetAdjCheckPointsInfo} (\text{cvode_mem, } \text{CVadjCheckPointRec *ckpnt});
\]

**Description**
The function \texttt{CVodeGetAdjCheckPointsInfo} loads an array of \( ncheck+1 \) records of type \texttt{CVadjCheckPointRec}. The user must allocate space for the array \( \text{ckpnt} \).

**Arguments**
- \texttt{cvode_mem} (\texttt{void *} pointer to the CVODES memory block created by \texttt{CVodeCreate}).
- \( \text{ckpnt} \) (\texttt{CVadjCheckPointRec *}) array of \( ncheck+1 \) checkpoint records, each of type \texttt{CVadjCheckPointRec}.

**Return value**
The return value is \texttt{CV_SUCCESS} if successful, or \texttt{CV_MEM_NULL} if \texttt{cvode_mem} is NULL, or \texttt{CV_NO_ADJ} if ASA was not initialized.

**Notes**
The members of each record \( \text{ckpnt}[i] \) are:

- \( \text{ckpnt}[i].\text{my_addr} \) (\texttt{void *}) address of current checkpoint in \texttt{cvode_mem->cv_adj_mem}
- \( \text{ckpnt}[i].\text{next_addr} \) (\texttt{void *}) address of next checkpoint
- \( \text{ckpnt}[i].t0 \) (\texttt{realtype}) start of checkpoint interval
- \( \text{ckpnt}[i].t1 \) (\texttt{realtype}) end of checkpoint interval
- \( \text{ckpnt}[i].\text{nstep} \) (\texttt{long int}) step counter at checkpoint \( t0 \)
- \( \text{ckpnt}[i].\text{order} \) (\texttt{int}) method order at checkpoint \( t0 \)
- \( \text{ckpnt}[i].\text{step} \) (\texttt{realtype}) step size at checkpoint \( t0 \)

### 6.2.10 Backward integration of quadrature equations

Not only the backward problem but also the backward quadrature equations may or may not depend on the forward sensitivities. Accordingly, either \texttt{CVodeQuadInitB} or \texttt{CVodeQuadInitBS} should be used to allocate internal memory and to initialize backward quadratures. For any other operation (extraction, optional input/output, reinitialization, deallocation), the same function is callable regardless of whether or not the quadratures are sensitivity-dependent.

#### 6.2.10.1 Backward quadrature initialization functions

The function \texttt{CVodeQuadInitB} initializes and allocates memory for the backward integration of quadrature equations that do not depend on forward sensitivities. It has the following form:
6.2 User-callable functions for adjoint sensitivity analysis

CVodeQuadInitB

Call $\text{flag} = \text{CVodeQuadInitB}(\text{cvode\_mem, which, rhsQB, yQB0})$;

Description The function CVodeQuadInitB provides required problem specifications, allocates internal memory, and initializes backward quadrature integration.

Arguments
- cvode\_mem (void *) pointer to the CVODES memory block.
- which (int) the identifier of the backward problem.
- rhsQB (CVQuadRhsFnB) is the C function which computes $fQB$, the right-hand side of the backward quadrature equations. This function has the form rhsQB(t, y, yB, qBdot, user\_dataB) (see §6.3.3).
- yQB0 (N\_Vector) is the value of the quadrature variables at $tB0$.

Return value The return value flag (of type int) will be one of the following:
- CV\_SUCCESS The call to CVodeQuadInitB was successful.
- CV\_MEM\_NULL cvode\_mem was NULL.
- CV\_NO\_ADJ The function CVodeAdjInit has not been previously called.
- CV\_MEM\_FAIL A memory allocation request has failed.
- CV\_ILL\_INPUT The parameter which is an invalid identifier.

The integration of quadrature equations during the backward phase can be re-initialized by calling the following function. Before calling CVodeQuadReInitB for a new backward problem, call any desired solution extraction functions CVodeGet** associated with the previous backward problem.

CVodeQuadInitBS

Call $\text{flag} = \text{CVodeQuadInitBS}(\text{cvode\_mem, which, rhsQBS, yQBS0})$;

Description The function CVodeQuadInitBS provides required problem specifications, allocates internal memory, and initializes backward quadrature integration.

Arguments
- cvode\_mem (void *) pointer to the CVODES memory block.
- which (int) the identifier of the backward problem.
- rhsQBS (CVQuadRhsFnBS) is the C function which computes $fQBS$, the right-hand side of the backward quadrature equations. This function has the form rhsQBS(t, y, yS, yB, qBdot, user\_dataB) (see §6.3.4).
- yQBS0 (N\_Vector) is the value of the sensitivity-dependent quadrature variables at $tB0$.

Return value The return value flag (of type int) will be one of the following:
- CV\_SUCCESS The call to CVodeQuadInitBS was successful.
- CV\_MEM\_NULL cvode\_mem was NULL.
- CV\_NO\_ADJ The function CVodeAdjInit has not been previously called.
- CV\_MEM\_FAIL A memory allocation request has failed.
- CV\_ILL\_INPUT The parameter which is an invalid identifier.

CVodeQuadReInitB

Call $\text{flag} = \text{CVodeQuadReInitB}(\text{cvode\_mem, which, yQB0})$;

Description The function CVodeQuadReInitB re-initializes the backward quadrature integration.

Arguments
- cvode\_mem (void *) pointer to the CVODES memory block.
- which (int) the identifier of the backward problem.
- yQB0 (N\_Vector) is the value of the quadrature variables at $tB0$. 
Return value The return value flag (of type int) will be one of the following:

- **CV_SUCCESS** The call to CVodeQuadReInitB was successful.
- **CV_MEM_NULL** cvode_mem was NULL.
- **CV_NO_ADJ** The function CVodeAdjInit has not been previously called.
- **CV_MEM_FAIL** A memory allocation request has failed.
- **CV_NO_QUAD** Quadrature integration was not activated through a previous call to CVodeQuadInitB.
- **CV_Ill_INPUT** The parameter which is an invalid identifier.

Notes The function CVodeQuadReInitB can be called after a call to either CVodeQuadInitB or CVodeQuadInitBS.

### 6.2.10.2 Backward quadrature extraction function

To extract the values of the quadrature variables at the last return time of CVodeB, CVODES provides a wrapper for the function CVodeGetQuad (see §4.7.3). The call to this function has the form

```c
flag = CVodeGetQuadB(cvode_mem, which, &tret, yQB);
```

**Call**

Description The function CVodeGetQuadB returns the quadrature solution vector after a successful return from CVodeB.

**Arguments**

- `cvode_mem` (void *) pointer to the CVODES memory.
- `tret` (realtype) the time reached by the solver (output).
- `yQB` (N_Vector) the computed quadrature vector.

**Return value** The return value flag of CVodeGetQuadB is one of:

- **CV_SUCCESS** CVodeGetQuadB was successful.
- **CV_MEM_NULL** cvode_mem is NULL.
- **CV_NO_ADJ** The function CVodeAdjInit has not been previously called.
- **CV_NO_QUAD** Quadrature integration was not initialized.
- **CV_BAD_DKY** yQB was NULL.
- **CV_Ill_INPUT** The parameter which is an invalid identifier.

**Notes** The user must allocate space for yQB.

To obtain the quadratures associated with a given backward problem at some other time within the last integration step, first obtain a pointer to the proper CVODES memory structure by calling CVodeGetAdjCVodeBmem and then use it to call CVodeGetQuadDky.

### 6.2.10.3 Optional input/output functions for backward quadrature integration

Optional values controlling the backward integration of quadrature equations can be changed from their default values through calls to one of the following functions which are wrappers for the corresponding optional input functions defined in §4.7.4. The user must specify the identifier which of the backward problem for which the optional values are specified.

```c
flag = CVodeSetQuadErrConB(cvode_mem, which, errconQ);
flag = CVodeQuadSStolerancesB(cvode_mem, which, reltolQ, abstolQ);
flag = CVodeQuadSVtolerancesB(cvode_mem, which, reltolQ, abstolQ);
```

Their return value flag (of type int) can have any of the return values of its counterparts, but it can also be **CV_NO_ADJ** if the function CVodeAdjInit has not been previously called or **CV_Ill_INPUT** if the parameter which was an invalid identifier.
Access to optional outputs related to backward quadrature integration can be obtained by calling the corresponding CVodeGetQuad* functions (see §4.7.5). A pointer cvode_memB to the CVODES memory block for the backward problem, required as the first argument of these functions, can be obtained through a call to the functions CVodeGetAdjCVodeMem (see §6.2.9).

6.3 User-supplied functions for adjoint sensitivity analysis

In addition to the required ODE right-hand side function and any optional functions for the forward problem, when using the adjoint sensitivity module in CVODES, the user must supply one function defining the backward problem ODE and, optionally, functions to supply Jacobian-related information and one or two functions that define the preconditioner (if an iterative SUNLINSOL module is selected) for the backward problem. Type definitions for all these user-supplied functions are given below.

6.3.1 ODE right-hand side for the backward problem

If the backward problem does not depend on the forward sensitivities, the user must provide a rhsB function of type CVRhsFnB defined as follows:

```c
typedef int (*CVRhsFnB)(realtype t, N_Vector y, N_Vector yB, N_Vector yBdot, void *user_dataB);
```

**Definition**
This function evaluates the right-hand side \( f_B(t,y,y_B) \) of the backward problem ODE system. This could be either (2.20) or (2.23).

**Arguments**
- \( t \) is the current value of the independent variable.
- \( y \) is the current value of the forward solution vector.
- \( y_B \) is the current value of the backward dependent variable vector.
- \( y_Bdot \) is the output vector containing the right-hand side \( f_B \) of the backward ODE problem.
- `user_dataB` is a pointer to user data, same as passed to CVodeSetUserDataB.

**Return value**
A CVRhsFnB should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODES will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and CVodeB returns CV_RHSFUNC_FAIL).

**Notes**
Allocation of memory for \( y_Bdot \) is handled within CVODES. The \( y, y_B, \) and \( y_Bdot \) arguments are all of type N_Vector, but \( y_B \) and \( y_Bdot \) typically have different internal representations from \( y \). It is the user’s responsibility to access the vector data consistently (including the use of the correct accessor macros from each NVVECTOR implementation). For the sake of computational efficiency, the vector functions in the two NVVECTOR implementations provided with CVODES do not perform any consistency checks with respect to their N_Vector arguments (see §7.2 and §7.3).

The `user_dataB` pointer is passed to the user’s rhsB function every time it is called and can be the same as the `user_data` pointer used for the forward problem.

Before calling the user’s rhsB function, CVODES needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, CVODES triggers an unrecoverable failure in the right-hand side function which will halt the integration and CVodeB will return CV_RHSFUNC_FAIL.

6.3.2 ODE right-hand side for the backward problem depending on the forward sensitivities

If the backward problem does depend on the forward sensitivities, the user must provide a rhsBS function of type CVRhsFnBS defined as follows:
Using CVODES for Adjoint Sensitivity Analysis

**CVRhsFnBS**

**Definition**

```c
typedef int (*CVRhsFnBS)(realtype t, N_Vector y, N_Vector *yS,
                        N_Vector yB, N_Vector yBdot, void *user_dataB);
```

**Purpose**

This function evaluates the right-hand side $f_B(t, y, y_B, s)$ of the backward problem ODE system. This could be either (2.20) or (2.23).

**Arguments**

- `t` is the current value of the independent variable.
- `y` is the current value of the forward solution vector.
- `yS` is a pointer to an array of $N_v$ vectors containing the sensitivities of the forward solution.
- `yB` is the current value of the backward dependent variable vector.
- `yBdot` is the output vector containing the right-hand side $f_B$ of the backward ODE problem.
- `user_dataB` is a pointer to user data, same as passed to `CVodeSetUserDataB`.

**Return value**

A `CVRhsFnBS` should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODES will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and `CVodeB` returns `CV_RHSFUNC_FAIL`).

**Notes**

Allocation of memory for `qBdot` is handled within CVODES.

The `y`, `yB`, and `yBdot` arguments are all of type `N_Vector`, but `yB` and `yBdot` typically have different internal representations from `y`. Likewise for each `yS[i]`. It is the user’s responsibility to access the vector data consistently (including the use of the correct accessor macros from each `NVECTOR` implementation). For the sake of computational efficiency, the vector functions in the two `NVECTOR` implementations provided with CVODES do not perform any consistency checks with respect to their `N_Vector` arguments (see §7.2 and §7.3).

The `user_dataB` pointer is passed to the user’s `rhsBS` function every time it is called and can be the same as the `user_data` pointer used for the forward problem.

Before calling the user’s `rhsBS` function, CVODES needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, CVODES triggers an unrecoverable failure in the right-hand side function which will halt the integration and `CVodeB` will return `CV_RHSFUNC_FAIL`.

### 6.3.3 Quadrature right-hand side for the backward problem

The user must provide an `fQB` function of type `CVQuadRhsFnB` defined by

**CVQuadRhsFnB**

**Definition**

```c
typedef int (*CVQuadRhsFnB)(realtype t, N_Vector y, N_Vector yB,
                           N_Vector qBdot, void *user_dataB);
```

**Purpose**

This function computes the quadrature equation right-hand side for the backward problem.

**Arguments**

- `t` is the current value of the independent variable.
- `y` is the current value of the forward solution vector.
- `yB` is the current value of the backward dependent variable vector.
- `qBdot` is the output vector containing the right-hand side $f_B$ of the backward quadrature equations.
- `user_dataB` is a pointer to user data, same as passed to `CVodeSetUserDataB`.

**Return value**

A `CVQuadRhsFnB` should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODES will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and `CVodeB` returns `CV_QRHSFUNC_FAIL`).
6.3 User-supplied functions for adjoint sensitivity analysis

Notes

Allocation of memory for \texttt{rhsvalBQ} is handled within CVODES.

The \( y \), \( yB \), and \( qBdot \) arguments are all of type \( N\text{\_Vector} \), but they typically do not all have the same representation. It is the user’s responsibility to access the vector data consistently (including the use of the correct accessor macros from each \texttt{NVECTOR} implementation). For the sake of computational efficiency, the vector functions in the two \texttt{NVECTOR} implementations provided with CVODES do not perform any consistency checks with respect to their \( N\text{\_Vector} \) arguments (see §7.2 and §7.3).

The \texttt{user\_dataB} pointer is passed to the user’s \( fQB \) function every time it is called and can be the same as the \texttt{user\_data} pointer used for the forward problem.

Before calling the user’s \( fQB \) function, CVODES needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, CVODES triggers an unrecoverable failure in the quadrature right-hand side function which will halt the integration and \texttt{CVodeB} will return \texttt{CV\_QRHSFUNC\_FAIL}.

6.3.4 Sensitivity-dependent quadrature right-hand side for the backward problem

The user must provide an \( fQBS \) function of type \texttt{CVQuadRhsFnBS} defined by

\[
\text{CVQuadRhsFnBS}
\]

\begin{verbatim}
definition int (*CVQuadRhsFnBS)(realtype t, N\_Vector y, N\_Vector *yS, N\_Vector yB, N\_Vector qBdot, void *user\_dataB);
\end{verbatim}

\textbf{Purpose} This function computes the quadrature equation right-hand side for the backward problem.

\textbf{Arguments}

\begin{tabular}{ll}
\( t \) & is the current value of the independent variable. \\
\( y \) & is the current value of the forward solution vector. \\
\( yS \) & a pointer to an array of \( Ns \) vectors containing the sensitivities of the forward solution. \\
\( yB \) & is the current value of the backward dependent variable vector. \\
\( qBdot \) & is the output vector containing the right-hand side \( fQBS \) of the backward quadrature equations. \\
\texttt{user\_dataB} & is a pointer to user data, same as passed to \texttt{CVodeSetUserDataB}.
\end{tabular}

\textbf{Return value} A \texttt{CVQuadRhsFnBS} should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODES will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and \texttt{CVodeB} returns \texttt{CV\_QRHSFUNC\_FAIL}).

\textbf{Notes} Allocation of memory for \texttt{qBdot} is handled within CVODES.

The \( y \), \( yS \), and \( qBdot \) arguments are all of type \( N\text{\_Vector} \), but they typically do not all have the same internal representation. Likewise for each \( yS[i] \). It is the user’s responsibility to access the vector data consistently (including the use of the correct accessor macros from each \texttt{NVECTOR} implementation). For the sake of computational efficiency, the vector functions in the two \texttt{NVECTOR} implementations provided with CVODES do not perform any consistency checks with respect to their \( N\text{\_Vector} \) arguments (see §7.2 and §7.3).

The \texttt{user\_dataB} pointer is passed to the user’s \( fQBS \) function every time it is called and can be the same as the \texttt{user\_data} pointer used for the forward problem.

Before calling the user’s \( fQBS \) function, CVODES needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the
interpolation, CVODES triggers an unrecoverable failure in the quadrature right-hand side function which will halt the integration and CVodeB will return CV_QRHSFUNC_FAIL.

### 6.3.5 Jacobian construction for the backward problem (matrix-based linear solvers)

If a matrix-based linear solver module is used for the backward problem (i.e., a non-NULL SUNMATRIX object was supplied to CVodeSetLinearSolverB), the user may provide a function of type CVLSJacFnB or CVLsJacFnBS (see §6.2.8), defined as follows:

#### CVLSJacFnB

**Definition**

```c
typedef int (*CVLSJacFnB)(realtype t, N_Vector y, N_Vector yB, N_Vector fyB, SUNMatrix JacB, void *user_dataB, N_Vector tmp1B, N_Vector tmp2B, N_Vector tmp3B);
```

**Purpose**

This function computes the Jacobian of the backward problem (or an approximation to it).

**Arguments**

- `t` is the current value of the independent variable.
- `y` is the current value of the forward solution vector.
- `yB` is the current value of the backward dependent variable vector.
- `fyB` is the current value of the backward right-hand side function $f_B$.
- `JacB` is the output approximate Jacobian matrix.
- `user_dataB` is a pointer to user data – the same as passed to CVodeSetUserDataB.
- `tmp1B` and `tmp2B` are pointers to memory allocated for variables of type `N_Vector` which can be used by the CVLSJacFnB function as temporary storage or work space.

**Return value**

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

**Notes**

- A user-supplied Jacobian function must load the matrix `JacB` with an approximation to the Jacobian matrix at the point $(t, y, yB)$, where $y$ is the solution of the original IVP at time $t$, and $yB$ is the solution of the backward problem at the same time.
- 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 8 for details). Only nonzero elements need to be loaded into `JacB` as this matrix is set to zero before the call to the Jacobian function.

Before calling the user’s CVLSJacFnB, CVODES needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, CVODES triggers an unrecoverable failure in the Jacobian function which will halt the integration (CVodeB returns CV_LSETUP_FAIL and CVLS sets `last_flag` to `CVLS_JACFUNC_UNRECVR`).

The previous function type CVDlsJacFnB is identical to CVLSJacFnB, 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.
6.3 User-supplied functions for adjoint sensitivity analysis

**CVLsJacFnBS**

**Definition**

\[
\text{typedef int } (*\text{CVLsJacFnBS})(\text{realtype } t, \text{N} \text{Vector } y, \\
\text{N} \text{Vector } *yS, \text{N} \text{Vector } yB, \text{N} \text{Vector } fyB, \\
\text{SUNMatrix } \text{JacB}, \text{void } *\text{user} \text{dataB}, \\
\text{N} \text{Vector } \text{tmp1B}, \text{N} \text{Vector } \text{tmp2B}, \\
\text{N} \text{Vector } \text{tmp3B});
\]

**Purpose**
This function computes the Jacobian of the backward problem (or an approximation to it), in the case where the backward problem depends on the forward sensitivities.

**Arguments**
- \(t\) is the current value of the independent variable.
- \(y\) is the current value of the forward solution vector.
- \(yS\) a pointer to an array of \(N_s\) vectors containing the sensitivities of the forward solution.
- \(yB\) is the current value of the backward dependent variable vector.
- \(fyB\) is the current value of the backward right-hand side function \(f_B\).
- \(\text{JacB}\) is the output approximate Jacobian matrix.
- \(\text{user} \text{dataB}\) is a pointer to user data – the same as passed to \text{CVodeSetUserDataB}.
- \(\text{tmp1B}\)
- \(\text{tmp2B}\)
- \(\text{tmp3B}\) are pointers to memory allocated for variables of type \text{N} \text{Vector} which can be used by \text{CVLsJacFnBS} as temporary storage or work space.

**Return value**
A \text{CVLsJacFnBS} should return 0 if successful, a positive value if a recoverable error occurred (in which case \text{CVODES} will attempt to correct, while \text{CVLS} sets \text{last} \text{flag} to \text{CVLS} \text{JACFUNC} \text{RECVR}), or a negative value if it failed unrecoverably (in which case the integration is halted, \text{CVodeB} returns \text{CV} \text{LSETUP} \text{FAIL} and \text{CVLS} sets \text{last} \text{flag} to \text{CVLS} \text{JACFUNC} \text{UNRECVR}).

**Notes**
A user-supplied Jacobian function must load the matrix \(\text{JacB}\) with an approximation to the Jacobian matrix at the point \((t, y, yS, yB)\), where \(y\) is the solution of the original IVP at time \(tt\), \(yS\) is the vector of forward sensitivities at time \(tt\), and \(yB\) is the solution of the backward problem at the same time. 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 8 for details). Only nonzero elements need to be loaded into \(\text{JacB}\) as this matrix is set to zero before the call to the Jacobian function.

Before calling the user’s \text{CVLsJacFnBS}, \text{CVODES} needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, \text{CVODES} triggers an unrecoverable failure in the Jacobian function which will halt the integration (\text{CVodeB} returns \text{CV} \text{LSETUP} \text{FAIL} and \text{CVLS} sets \text{last} \text{flag} to \text{CVLS} \text{JACFUNC} \text{UNRECVR}).

The previous function type \text{CVDlsJacFnBS} is identical to \text{CVLsJacFnBS}, 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.

6.3.6 Jacobian-vector product for the backward problem (matrix-free linear solvers)

If a matrix-free linear solver is to be used for the backward problem (i.e., a \text{NULL}-valued SUNMATRIX was supplied to \text{CVodeSetLinearSolverB} in the steps described in §6.1), the user may provide a function of type \text{CVLsJacTimesVecFnB} or \text{CVLSsJacTimesVecFnBS} 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.
CVLsJacTimesVecFnB

**Definition**

```c
typedef int (*CVLsJacTimesVecFnB)(N_Vector vB, N_Vector JvB,
                               realtype t, N_Vector y, N_Vector yB,
                               N_Vector fyB, void *user_dataB,
                               N_Vector tmpB);
```

**Purpose**

This function computes the action of the Jacobian \( J_B \) for the backward problem on a given vector \( v_B \).

**Arguments**

- \( v_B \) is the vector by which the Jacobian must be multiplied to the right.
- \( Jv_B \) is the computed output vector \( J_Bv_B \).
- \( t \) is the current value of the independent variable.
- \( y \) is the current value of the forward solution vector.
- \( y_B \) is the current value of the backward dependent variable vector.
- \( fy_B \) is the current value of the backward right-hand side function \( f_B \).
- \( user_dataB \) is a pointer to user data – the same as passed to \( CVodeSetUserDataB \).
- \( tmpB \) is a pointer to memory allocated for a variable of type \( N_Vector \) which can be used by \( CVLsJacTimesVecFn \) as temporary storage or work space.

**Return value**

The return value of a function of type \( CVLsJacTimesVecFnB \) should be 0 if successful or nonzero if an error was encountered, in which case the integration is halted.

**Notes**

A user-supplied Jacobian-vector product function must load the vector \( Jv_B \) with the product of the Jacobian of the backward problem at the point \( (t, y, y_B) \) and the vector \( v_B \). Here, \( y \) is the solution of the original IVP at time \( t \) and \( y_B \) is the solution of the backward problem at the same time. The rest of the arguments are equivalent to those passed to a function of type \( CVLsJacTimesVecFn \) (see §4.6.6). If the backward problem is the adjoint of \( \dot{y} = f(t, y) \), then this function is to compute \(-\frac{\partial f}{\partial y}^T v_B \).

The previous function type \( CVSpilsJacTimesVecFnB \) is identical to \( CVLsJacTimesVecFnB \), 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.

CVLsJacTimesVecFnBS

**Definition**

```c
typedef int (*CVLsJacTimesVecFnBS)(N_Vector vB, N_Vector JvB,
                                   realtype t, N_Vector y, N_Vector *yS,
                                   N_Vector yB, N_Vector fyB,
                                   void *user_dataB, N_Vector tmpB);
```

**Purpose**

This function computes the action of the Jacobian \( J_B \) for the backward problem on a given vector \( v_B \), in the case where the backward problem depends on the forward sensitivities.

**Arguments**

- \( v_B \) is the vector by which the Jacobian must be multiplied to the right.
- \( Jv_B \) is the computed output vector \( J_Bv_B \).
- \( t \) is the current value of the independent variable.
- \( y \) is the current value of the forward solution vector.
- \( y_S \) is a pointer to an array containing the forward sensitivity vectors.
- \( y_B \) is the current value of the backward dependent variable vector.
- \( fy_B \) is the current value of the backward right-hand side function \( f_B \).
- \( user_dataB \) is a pointer to user data – the same as passed to \( CVodeSetUserDataB \).
- \( tmpB \) is a pointer to memory allocated for a variable of type \( N_Vector \) which can be used by \( CVLsJacTimesVecFn \) as temporary storage or work space.

**Return value**

The return value of a function of type \( CVLsJacTimesVecFnBS \) should be 0 if successful or nonzero if an error was encountered, in which case the integration is halted.
Notes A user-supplied Jacobian-vector product function must load the vector \( JvB \) with the product of the Jacobian of the backward problem at the point \((t, y, yB)\) and the vector \( vB \). Here, \( y \) is the solution of the original IVP at time \( t \) and \( yB \) is the solution of the backward problem at the same time. The rest of the arguments are equivalent to those passed to a function of type \( CVLsJacTimesVecFn \) (see §4.6.6).

The previous function type \( CVSpilsJacTimesVecFnB \) is identical to \( CVLsJacTimesVecFnB \), 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.

### 6.3.7 Jacobian-vector product setup for the backward problem (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 \( CVLsJacTimesSetupFnB \) or \( CVLsJacTimesSetupFnBS \), defined as follows:

```c
CVLsJacTimesSetupFnB
```

**Definition**

```c
typedef int (*CVLsJacTimesSetupFnB)(realtype t,
                                  N_Vector y, N_Vector yB,
                                  N_Vector fyB, void *user_dataB);
```

**Purpose**

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

**Arguments**

- \( t \) is the current value of the independent variable.
- \( y \) is the current value of the dependent variable vector, \( y(t) \).
- \( yB \) is the current value of the backward dependent variable vector.
- \( fyB \) is the current value of the right-hand-side for the backward problem.
- \( user_dataB \) is a pointer to user data — the same as the \( user_dataB \) parameter passed to \( CVSetUserDataB \).

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

If the user’s \( CVLsJacTimesVecFnB \) function uses difference quotient approximations, it may need to access quantities not in the call list. These include the current stepsize, the error weights, etc. To obtain these, the user will need to add a pointer to \( cvode_mem \) to \( user_dataB \) and then use the \( CVGet* \) functions described in §4.5.9.2. The unit roundoff can be accessed as \( \text{UNIT\_ROUNDOFF} \) defined in \texttt{sundials\_types.h}.

The previous function type \( CVSpilsJacTimesSetupFnB \) is identical to \( CVLsJacTimesSetupFnB \), 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.
Definition  typedef int (*CVLsJacTimesSetupFnBS)(realtype t,
N_Vector y, N_Vector *yS,
N_Vector yB, N_Vector fyB,
void *user_dataB);

Purpose  This function preprocesses and/or evaluates Jacobian data needed by the Jacobian-
times-vector routine for the backward problem, in the case that the backward problem
depends on the forward sensitivities.

Arguments  t  is the current value of the independent variable.
            y  is the current value of the dependent variable vector, \( y(t) \).
            yS  a pointer to an array of \( N_s \) vectors containing the sensitivities of the forward
                 solution.
            yB  is the current value of the backward dependent variable vector.
            fyB  is the current value of the right-hand-side function for the backward problem.
            user_dataB  is a pointer to user data — the same as the user_dataB parameter passed
to CVSetUserDataB.

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 backward
problem residual user function with the same \((t, y, yS, yB)\) arguments. Thus, the
setup function can use any auxiliary data that is computed and saved during the eval-
uation of the right-hand-side function.

If the user’s CVLsJacTimesVecFnBS function uses difference quotient approximations, it
may need to access quantities not in the call list. These include the current stepsize, the
error weights, etc. To obtain these, the user will need to add a pointer to cvode_mem to
user_dataB and then use the CVGet* 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 CVSplsJacTimesSetupFnBS is identical to
CVLsJacTimesSetupFnBS, 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.

6.3.8  Preconditioner solve for the backward problem (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 a right
preconditioner matrix. Here \( P \) should approximate (at least crudely) the matrix \( M_B = I - \gamma_B J_B \),
where \( J_B = \frac{\partial f_B}{\partial y_B} \). If preconditioning is done on both sides, the product of the two preconditioner
matrices should approximate \( M_B \). This function must be of one of the following two types:

Definition  typedef int (*CVLsPrecSolveFnB)(realtype t,
N_Vector y, N_Vector yB, N_Vector fyB,
N_Vector rvecB, N_Vector zvecB,
realtype gammaB, realtype deltaB,
void *user_dataB);

Purpose  This function solves the preconditioning system \( Pz = r \) for the backward problem.

Arguments  t  is the current value of the independent variable.
           y  is the current value of the forward solution vector.
           yB  is the current value of the backward dependent variable vector.
6.3 User-supplied functions for adjoint sensitivity analysis

6.3.9 Preconditioner setup for the backward problem (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 one of the following two types:

- **fyB** is the current value of the backward right-hand side function \( f_B \).
- **rvecB** is the right-hand side vector \( r \) of the linear system to be solved.
- **zvecB** is the computed output vector.
- **gammaB** is the scalar appearing in the matrix, \( M_B = I - \gamma_B J_B \).
- **deltaB** is an input tolerance to be used if an iterative method is employed in the solution.

**user_dataB** is a pointer to user data — the same as the **user_dataB** parameter passed to CVodeSetUserDataB.

**Return value**

The return value of a preconditioner solve function for the backward problem 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 CVSpilsPrecSolveFnB is identical to CVLsPrecSolveFnB, 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.

**CVLsPrecSolveFnBS**

**Definition**

```c
typedef int (*CVLsPrecSolveFnBS)(realtype t, N_Vector y, N_Vector *yS, N_Vector yB, N_Vector fyB, N_Vector rvecB, N_Vector zvecB, realtype gammaB, realtype deltaB, void *user_dataB);
```

**Purpose**

This function solves the preconditioning system \( Pz = r \) for the backward problem, in the case where the backward problem depends on the forward sensitivities.

**Arguments**

- **t** is the current value of the independent variable.
- **y** is the current value of the forward solution vector.
- **yS** is a pointer to an array containing the forward sensitivity vectors.
- **yB** is the current value of the backward dependent variable vector.
- **fyB** is the current value of the backward right-hand side function \( f_B \).
- **rvecB** is the right-hand side vector \( r \) of the linear system to be solved.
- **zvecB** is the computed output vector.
- **gammaB** is the scalar appearing in the matrix, \( M_B = I - \gamma_B J_B \).
- **deltaB** is an input tolerance to be used if an iterative method is employed in the solution.

**user_dataB** is a pointer to user data — the same as the **user_dataB** parameter passed to CVodeSetUserDataB.

**Return value**

The return value of a preconditioner solve function for the backward problem 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 CVSpilsPrecSolveFnBS is identical to CVLsPrecSolveFnBS, 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.
CVLsPrecSetupFnB

Definition typedef int (*CVLsPrecSetupFnB)(realtype t, N_Vector y, 
N_Vector yB, N_Vector fyB, 
booleantype jokB, booleantype *jcurPtrB, 
realtype gammaB, void *user_dataB);

Purpose This function preprocesses and/or evaluates Jacobian-related data needed by the pre-
conditioner for the backward problem.

Arguments The arguments of a CVLsPrecSetupFnB are as follows:
- t is the current value of the independent variable.
- y is the current value of the forward solution vector.
- yB is the current value of the backward dependent variable vector.
- fyB is the current value of the backward right-hand side function f_B.
- jokB is an input flag indicating whether Jacobian-related data needs to be recom-
puted (jokB=SUNFALSE) or information saved from a previous invocation can be safely used (jokB=SUNTRUE).
- jcurPtr is an output flag which must be set to SUNTRUE if Jacobian-relatd data was recomputed or SUNFALSE otherwise.
- gammaB is the scalar appearing in the matrix M_B = I - γ_B J_B.
- user_dataB is a pointer to user data — the same as the user_dataB parameter passed to CVodeSetUserDataB.

Return value The return value of a preconditioner setup function for the backward problem 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 CVSpilsPrecSetupFnB is identical to CVLsPrecSetupFnB, 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.

CVLsPrecSetupFnBS

Definition typedef int (*CVLsPrecSetupFnBS)(realtype t, N_Vector y, N_Vector *yS, 
N_Vector yB, N_Vector fyB, 
booleantype jokB, booleantype *jcurPtrB, 
realtype gammaB, void *user_dataB);

Purpose This function preprocesses and/or evaluates Jacobian-related data needed by the pre-
conditioner for the backward problem, in the case where the backward problem depends
on the forward sensitivities.

Arguments The arguments of a CVLsPrecSetupFnBS are as follows:
- t is the current value of the independent variable.
- y is the current value of the forward solution vector.
- yS is a pointer to an array containing the forward sensitivity vectors.
- yB is the current value of the backward dependent variable vector.
- fyB is the current value of the backward right-hand side function f_B.
- jokB is an input flag indicating whether Jacobian-related data needs to be recom-
puted (jokB=SUNFALSE) or information saved from a previous invocation can be safely used (jokB=SUNTRUE).
- jcurPtr is an output flag which must be set to SUNTRUE if Jacobian-relatd data was recomputed or SUNFALSE otherwise.
- gammaB is the scalar appearing in the matrix M_B = I - γ_B J_B.
6.4 Using CVODES preconditioner modules for the backward problem

As on the forward integration phase, the efficiency of Krylov iterative methods for the solution of linear systems can be greatly enhanced through preconditioning. Both preconditioner modules provided with SUNDAIS, the serial banded preconditioner CVBANDPRE and the parallel band-block-diagonal preconditioner module CVBBDPRE, provide interface functions through which they can be used on the backward integration phase.

6.4.1 Using the banded preconditioner CVBANDPRE

The adjoint module in CVODES offers an interface to the banded preconditioner module CVBANDPRE described in section §4.8.1. This preconditioner, usable only in a serial setting, provides a band matrix preconditioner based on difference quotients of the backward problem right-hand side function \( f_B \). It generates a banded approximation to the Jacobian with \( m_{LB} \) sub-diagonals and \( m_{UB} \) super-diagonals to be used with one of the Krylov linear solvers.

In order to use the CVBANDPRE module in the solution of the backward problem, the user need not define any additional functions. Instead, after an iterative SUNLINSOL object has been attached to CVODES via a call to CVodeSetLinearSolverB, the following call to the CVBANDPRE module initialization function must be made.

```c
CVBandPrecInitB
```

Call

\[
\text{flag} = \text{CVBandPrecInitB}(\text{cvode\_mem}, \text{which}, \text{nB}, \text{muB}, \text{mlB});
\]

Description

The function \text{CVBandPrecInitB} initializes and allocates memory for the CVBANDPRE preconditioner for the backward problem. It creates, allocates, and stores (internally in the CVODES solver block) a pointer to the newly created CVBANDPRE memory block.

Arguments

- \text{cvode\_mem} (void *) pointer to the CVODES memory block.
- \text{which} (int) the identifier of the backward problem.
- \text{nB} (sunindextype) backward problem dimension.
- \text{muB} (sunindextype) upper half-bandwidth of the backward problem Jacobian approximation.
- \text{mlB} (sunindextype) lower half-bandwidth of the backward problem Jacobian approximation.

Return value

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

- \text{CVLS\_SUCCESS} The call to CVodeBandPrecInitB was successful.
- \text{CVLS\_MEM\_FAIL} A memory allocation request has failed.
- \text{CVLS\_MEM\_NULL} The \text{cvode\_mem} argument was \text{NULL}.
- \text{CVLS\_LMEM\_NULL} No linear solver has been attached.
- \text{CVLS\_ILL\_INPUT} An invalid parameter has been passed.

For more details on CVBANDPRE see §4.8.1.
6.4.2 Using the band-block-diagonal preconditioner CVBBDPRE

The adjoint module in CVODES offers an interface to the band-block-diagonal preconditioner module CVBBDPRE described in section §4.8.2. This generates a preconditioner that is a block-diagonal matrix with each block being a band matrix and can be used with one of the Krylov linear solvers and with the MPI-parallel vector module NVECTOR_PARALLEL.

In order to use the CVBBDPRE module in the solution of the backward problem, the user must define one or two additional functions, described at the end of this section.

6.4.2.1 Initialization of CVBBDPRE

The CVBBDPRE module is initialized by calling the following function, after an iterative SUNLINSOL object has been attached to CVODES via a call to CVodeSetLinearSolverB.

```c
CVBBDPrecInitB
```

Call `flag = CVBBDPrecInitB(cvode_mem, which, NlocalB, mudqB, mldqB, mukeepB, mlkeepB, dqrelyB, glocB, gcommB);`

Description The function CVBBDPrecInitB initializes and allocates memory for the CVBBDPRE preconditioner for the backward problem. It creates, allocates, and stores (internally in the CVODES solver block) a pointer to the newly created CVBBDPRE memory block.

Arguments

- `cvode_mem` (void *) pointer to the CVODES memory block.
- `which` (int) the identifier of the backward problem.
- `NlocalB` (sunindextype) local vector dimension for the backward problem.
- `mudqB` (sunindextype) upper half-bandwidth to be used in the difference-quotient Jacobian approximation.
- `mldqB` (sunindextype) lower half-bandwidth to be used in the difference-quotient Jacobian approximation.
- `mukeepB` (sunindextype) upper half-bandwidth of the retained banded approximate Jacobian block.
- `mlkeepB` (sunindextype) lower half-bandwidth of the retained banded approximate Jacobian block.
- `dqrelyB` (realtype) the relative increment in components of \( y_B \) used in the difference quotient approximations. The default is \( dqrelyB = \sqrt{\text{unit roundoff}} \), which can be specified by passing \( dqrely = 0.0 \).
- `glocB` (CVBBDLocalFnB) the function which computes the function \( g_B(t, y, y_B) \) approximating the right-hand side of the backward problem.
- `gcommB` (CVBBDCommFnB) the optional function which performs all interprocess communication required for the computation of \( g_B \).

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

- CVLS_SUCCESS The call to CVodeBBDPrecInitB was successful.
- CVLS_MEM_FAIL A memory allocation request has failed.
- CVLS_MEM_NULL The `cvode_mem` argument was NULL.
- CVLS_LMEM_NULL No linear solver has been attached.
- CVLS_ILL_INPUT An invalid parameter has been passed.

To reinitialize the CVBBDPRE preconditioner module for the backward problem, possibly with changes in `mudqB`, `mldqB`, or `dqrelyB`, call the following function:

```c
CVBBDPrecReInitB
```

Call `flag = CVBBDPrecReInitB(cvode_mem, which, mudqB, mldqB, dqrelyB);`
6.4 Using CVODES preconditioner modules for the backward problem

Description The function CVBBDPrecReInitB reinitializes the CVBBDPRE preconditioner for the backward problem.

Arguments

- cvode_mem (void *) pointer to the CVODES memory block returned by CVodeCreate.
- which (int) the identifier of the backward problem.
- mudqB (sunindextype) upper half-bandwidth to be used in the difference-quotient Jacobian approximation.
- mldqB (sunindextype) lower half-bandwidth to be used in the difference-quotient Jacobian approximation.
- dqrelyB (realtype) the relative increment in components of yB used in the difference quotient approximations.

Return value The return value flag (of type int) is one of:
- CVLS_SUCCESS The call to CVodeBBDPrecReInitB was successful.
- CVLS_MEM_FAIL A memory allocation request has failed.
- CVLS_MEM_NULL The cvode_mem argument was NULL.
- CVLS_PMEM_NULL The CVodeBBDPrecInitB has not been previously called.
- CVLS_LMEM_NULL No linear solver has been attached.
- CVLS_ILL_INPUT An invalid parameter has been passed.

For more details on cvbbdpre see §4.8.2.

6.4.2.2 User-supplied functions for CVBBDPRE

To use the cvbbdpre module, the user must supply one or two functions which the module calls to construct the preconditioner: a required function glocB (of type CVBBDLocalFnB) which approximates the right-hand side of the backward problem and which is computed locally, and an optional function gcommB (of type CVBBDCommFnB) which performs all interprocess communication necessary to evaluate this approximate right-hand side (see §4.8.2). The prototypes for these two functions are described below.

CVBBDLocalFnB

definition typedef int (*CVBBDLocalFnB)(sunindextype NlocalB, realtype t, N_Vector y, N_Vector yB, N_Vector gB, void *user_dataB);

Purpose This glocB function loads the vector gB, an approximation to the right-hand side f_B of the backward problem, as a function of t, y, and yB.

Arguments

- NlocalB is the local vector length for the backward problem.
- t is the value of the independent variable.
- y is the current value of the forward solution vector.
- yB is the current value of the backward dependent variable vector.
- gB is the output vector, gB(t, y, yB).
- user_dataB is a pointer to user data — the same as the user_dataB parameter passed to CVodeSetUserDataB.

Return value An CVBBDLocalFnB should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODES will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and CVodeB returns CV_LSETUP_FAIL).

Notes This routine must assume that all interprocess communication of data needed to calculate gB has already been done, and this data is accessible within user_dataB.

Before calling the user’s CVBBDLocalFnB, CVODES needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, CVODES triggers an unrecoverable failure in the preconditioner setup function which will halt the integration (CVodeB returns CV_LSETUP_FAIL).
CVBBDCommFnB

Definition

typedef int (*CVBBDCommFnB)(sunindextype NlocalB, realtype t, N_Vector y, N_Vector yB, void *user_dataB);

Purpose

This gcommB function must perform all interprocess communications necessary for the execution of the glocB function above, using the input vectors \( y \) and \( yB \).

Arguments

- \( NlocalB \) is the local vector length.
- \( t \) is the value of the independent variable.
- \( y \) is the current value of the forward solution vector.
- \( yB \) is the current value of the backward dependent variable vector.
- \( user\_dataB \) is a pointer to user data — the same as the \( user\_dataB \) parameter passed to CVodeSetUserDataB.

Return value

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

Notes

The gcommB function is expected to save communicated data in space defined within the structure user_dataB.

Each call to the gcommB function is preceded by a call to the function that evaluates the right-hand side of the backward problem with the same \( t, y, \) and \( yB \), arguments. If there is no additional communication needed, then pass gcommB = NULL to CVBBDPrecInitB.
Chapter 7

Description of the NVECTOR module

The Sundials solvers are written in a data-independent manner. They all operate on generic vectors (of type N_Vector) through a set of operations defined by the particular NVECTOR implementation. Users can provide their own specific implementation of the NVECTOR module, or use one of the implementations provided with Sundials. The generic operations are described below and the implementations provided with Sundials are described in the following sections.

The generic N_Vector type is a pointer to a structure that has an implementation-dependent content field containing the description and actual data of the vector, and an ops field pointing to a structure with generic vector operations. The type N_Vector is defined as

typedef struct _generic_N_Vector *N_Vector;

struct _generic_N_Vector {
  void *content;
  struct _generic_N_Vector_Ops *ops;
};

The _generic_N_Vector_Ops structure is essentially a list of pointers to the various actual vector operations, and is defined as

struct _generic_N_Vector_Ops {
  N_Vector_ID (*nvgetvectorid)(N_Vector);
  N_Vector (*nvclone)(N_Vector);
  N_Vector (*nvclonempty)(N_Vector);
  void (*nvdestroy)(N_Vector);
  void (*nvspace)(N_Vector, sunindextype *, sunindextype *);
  realtype* (*nvgetarraypointer)(N_Vector);
  void (*nvsetarraypointer)(realtype *, N_Vector);
  void (*nvinolinearsum)(realtype, N_Vector, realtype, N_Vector, N_Vector);
  void (*nvlinear)(realtype, N_Vector, N_Vector);
  void (*nvdiv)(N_Vector, 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 (*nvdotprod)(N_Vector, N_Vector);
  realtype (*nvmaxnorm)(N_Vector);
  realtype (*nvwrmsnorm)(N_Vector, N_Vector);
The generic nvector module defines and implements the vector operations acting on an NVector. 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 NVector structure. To illustrate this point we show below the implementation of a typical vector operation from the generic nvector module, namely NVScale, 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 7.2 contains a complete list of all standard vector operations defined by the generic nvector module. Tables 7.3 and 7.4 list optional fused and vector array 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. Currently, all fused and vector array operations are disabled by default however, SUNDIALS provided nvector implementations define 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.

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 NVector, each of the same type as an existing NVector. Their prototypes are

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

and their definitions are based on the implementation-specific N_VClone and N_VCloneEmpty operations, respectively.

An array of variables of type NVector can be destroyed by calling N_VDestroyVectorArray, whose prototype is

```c
void N_VDestroyVectorArray(N_Vector x);
```
Table 7.1: 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_OPENMPDEV</td>
<td>OpenMP shared memory parallel with device offloading</td>
<td>6</td>
</tr>
<tr>
<td>SUNDIALS_NVEC_TRILINOS</td>
<td>Trilinos Tpetra vector</td>
<td>7</td>
</tr>
<tr>
<td>SUNDIALS_NVEC_CUSTOM</td>
<td>User-provided custom vector</td>
<td>8</td>
</tr>
</tbody>
</table>

```c
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 7.1. It is recommended that a user-supplied `nvector` implementation use the `SUNDIALS_NVEC_CUSTOM` identifier.
Table 7.2: 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 7.1.</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_VDestroy(v); Destroys the N_Vector v and frees memory allocated for its internal data.</td>
</tr>
<tr>
<td>N_VSpace</td>
<td>N_VSpace(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>N_VGetArrayPointer</td>
<td>vdata = N_VGetArrayPointer(v); Returns a pointer to a realtype array from the N_Vector v. Note that this assumes that the internal data in N_Vector is a contiguous array of realtype. This routine is only used in the solver-specific interfaces to the dense and banded (serial) linear solvers, the sparse linear solvers (serial and threaded), and in the interfaces to the banded (serial) and band-block-diagonal (parallel) preconditioner modules provided with SUNDIALS.</td>
</tr>
<tr>
<td>N_VSetArrayPointer</td>
<td>N_VSetArrayPointer(vdata, v); Overwrites the data in an N_Vector with a given array of realtype. Note that this assumes that the internal data in N_Vector is a contiguous array of realtype. This routine is only used in the interfaces to the dense (serial) linear solver, hence need not exist in a user-supplied NVECTOR module for a parallel environment.</td>
</tr>
</tbody>
</table>

*continued on next page*
<table>
<thead>
<tr>
<th>Name</th>
<th>Usage and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N_VLinearSum</td>
<td><code>N_VLinearSum(a, x, b, y, z);</code> Performs the operation ( z = ax + by ), where ( a ) and ( b ) are realtype scalars and ( x ) and ( y ) are of type N_Vector: ( z_i = ax_i + by_i, \ i = 0,\ldots, n - 1 ).</td>
</tr>
<tr>
<td>N_VConst</td>
<td><code>N_VConst(c, z);</code> Sets all components of the N_Vector ( z ) to realtype ( c ): ( z_i = c, i = 0,\ldots, n - 1 ).</td>
</tr>
<tr>
<td>N_VProd</td>
<td><code>N_VProd(x, y, z);</code> Sets the N_Vector ( z ) to be the component-wise product of the N_Vector inputs ( x ) and ( y ): ( z_i = x_i y_i, \ i = 0,\ldots, n - 1 ).</td>
</tr>
<tr>
<td>N_VDiv</td>
<td><code>N_VDiv(x, y, z);</code> Sets the N_Vector ( z ) to be the component-wise ratio of the N_Vector inputs ( x ) and ( y ): ( z_i = x_i / y_i, \ i = 0,\ldots, n - 1 ). The ( y_i ) may not be tested for 0 values. It should only be called with a ( y ) that is guaranteed to have all nonzero components.</td>
</tr>
<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 ): ( z_i = cx_i, i = 0,\ldots, n - 1 ).</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 ): ( z_i = 1.0 / x_i, \ i = 0,\ldots, n - 1 ). This routine may not check for division by 0. It should be called only with an ( x ) which is guaranteed to have all nonzero components.</td>
</tr>
<tr>
<td>N_VAddConst</td>
<td><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 ): ( z_i = x_i + b, i = 0,\ldots, n - 1 ).</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 ): ( d = \sum_{i=0}^{n-1} x_i y_i ).</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>
</tbody>
</table>

*continued on next page*
### Description of the NVECTOR module

**Name** | **Usage and Description**  
---|---  
N_VWrmsNorm | \( m = \text{N_VWrmsNorm}(x, w) \)  
Returns the weighted root-mean-square norm of the N_Vector \( x \) with **realtype** weight vector \( w \): \[ m = \sqrt{\frac{\sum_{i=0}^{n-1} (x_i w_i)^2}{n}}. \]  
N_VWrmsNormMask | \( m = \text{N_VWrmsNormMask}(x, w, \text{id}) \)  
Returns the weighted root mean square norm of the N_Vector \( x \) with **realtype** weight vector \( w \) built using only the elements of \( x \) corresponding to positive elements of the N_Vector \( \text{id} \): \[ m = \sqrt{\frac{\sum_{i=0}^{n-1} (x_i w_i H(\text{id})_i)^2}{n}}, \text{ where } H(\alpha) = \begin{cases} 1 & \alpha > 0 \\ 0 & \alpha \leq 0 \end{cases} \]  
N_VMin | \( m = \text{N_VMin}(x) \)  
Returns the smallest element of the N_Vector \( x \): \( m = \min_i x_i \).  
N_VL2Norm | \( m = \text{N_VL2Norm}(x, w) \)  
Returns the weighted Euclidean \( \ell_2 \) norm of the N_Vector \( x \) with **realtype** weight vector \( w \): \[ m = \sqrt{\sum_{i=0}^{n-1} (x_i w_i)^2}. \]  
N_VL1Norm | \( m = \text{N_VL1Norm}(x) \)  
Returns the \( \ell_1 \) norm of the N_Vector \( x \): \( m = \sum_{i=0}^{n-1} |x_i| \).  
N_VCompare | \( \text{N_VCompare}(c, x, z) \)  
Compares 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 | \( \text{t} = \text{N_VInvTest}(x, z) \)  
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 | \( \text{t} = \text{N_VConstrMask}(c, x, m) \)  
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.
<table>
<thead>
<tr>
<th>Name</th>
<th>Usage and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N_VMinQuotient</td>
<td>minq = N_VMinQuotient(num, denom); This routine returns the minimum of the quotients obtained by term-wise dividing num, by denom. A zero element in denom will be skipped. If no such quotients are found, then the large value BIG_REAL (defined in the header file sundials_types.h) is returned.</td>
</tr>
</tbody>
</table>

Table 7.3: Description of the NVECTOR fused operations

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

continued on next page
### Table 7.4: Description of the NVECTOR vector array operations

<table>
<thead>
<tr>
<th>Name</th>
<th>Usage and Description</th>
</tr>
</thead>
</table>
| N_VDotProdMulti          | `ier = N_VDotProdMulti(nv, x, Y, d);`<br>This routine computes the dot product of a vector with `nv` other vectors:<br>
\[
d_j = \sum_{i=0}^{n-1} x_i y_{j,i}, \quad j = 0, \ldots, nv - 1,
\]
where `d` (type `realtype*`) is an array of `nv` scalars containing the dot products of the vector `x` (type `N_Vector`) with each of the `nv` vectors in the vector array `Y` (type `N_Vector*`). The operation returns 0 for success and a non-zero value otherwise. |
| N_VLinearSumVectorArray  | `ier = N_VLinearSumVectorArray(nv, a, X, b, Y, Z);`<br>This routine computes the linear sum of two vector arrays containing `nv` vectors of `n` elements:<br>
\[
z_{j,i} = ax_{j,i} + by_{j,i}, \quad i = 0, \ldots, n - 1 \quad j = 0, \ldots, nv - 1,
\]
where `a` and `b` are `realtype` scalars and `X`, `Y`, and `Z` are arrays of `nv` vectors (type `N_Vector*`). The operation returns 0 for success and a non-zero value otherwise. |
| N_VScaleVectorArray      | `ier = N_VScaleVectorArray(nv, c, X, Z);`<br>This routine scales each vector of `n` elements in a vector array of `nv` vectors by a potentially different constant:<br>
\[
z_{j,i} = c_j x_{j,i}, \quad i = 0, \ldots, n - 1 \quad j = 0, \ldots, nv - 1,
\]
where `c` is an array of `nv` scalars (type `realtype*`) and `X` and `Z` are arrays of `nv` vectors (type `N_Vector*`). The operation returns 0 for success and a non-zero value otherwise. |
### Name | Usage and Description
--- | ---
N_VConstVectorArray | `ier = N_VConstVectorArray(nv, c, X);`
This routine sets each element in a vector of `n` elements in a vector array of `nv` vectors to the same value:

\[ z_{j,i} = c, \quad i = 0, \ldots, n - 1 \quad j = 0, \ldots, nv - 1, \]

where `c` is a `realtype` scalar and `X` is an array of `nv` vectors (type `N_Vector`). The operation returns 0 for success and a non-zero value otherwise.

N_VWrmsNormVectorArray | `ier = N_VWrmsNormVectorArray(nv, X, W, m);`
This routine computes the weighted root mean square norm of `nv` vectors with `n` elements:

\[ m_j = \left( \frac{1}{n} \sum_{i=0}^{n-1} (x_{j,i} w_{j,i})^2 \right)^{1/2}, \quad j = 0, \ldots, nv - 1, \]

where `m` (type `realtype`) contains the `nv` 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.

N_VWrmsNormMaskVectorArray | `ier = N_VWrmsNormMaskVectorArray(nv, X, W, id, m);`
This routine computes the masked weighted root mean square norm of `nv` vectors with `n` elements:

\[ m_j = \left( \frac{1}{n} \sum_{i=0}^{n-1} (x_{j,i} w_{j,i} H(id_i))^2 \right)^{1/2}, \quad j = 0, \ldots, nv - 1, \]

\[ H(id_i) = 1 \text{ for } id_i > 0 \text{ and is zero otherwise}, \]

`m` (type `realtype`) contains the `nv` 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.
170 Description of the NVECTOR module

Name | Usage and Description
--- | ---
N_VScaleAddMultiVectorArray | `ier = N_VScaleAddMultiVectorArray(nv, ns, c, X, YY, ZZ);`
This routine scales and adds a vector in a vector array of `nv` vectors to the corresponding vector in `ns` vector arrays:

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

where `c` is an array of `ns` scalars (type `realtype`), `X` is a vector array of `nv` vectors (type `idN_Vector`*) to be scaled and added to the corresponding vector in each of the `ns` 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_VLinearCombinationVectorArray | `ier = N_VLinearCombinationVectorArray(nv, ns, c, XX, Z);`
This routine computes the linear combination of `ns` vector arrays containing `nv` vectors with `n` elements:

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

where `c` is an array of `ns` scalars (type `realtype`), `XX` (type `N_Vector**) is an array of `ns` vector arrays each containing `nv` vectors to be summed into the output vector array of `nv` vectors `Z` (type `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.

7.1 NVECTOR functions used by CVODES

In Table 7.5 below, we list the vector functions in the NVECTOR module used within the CVODES package. The table also shows, for each function, which of the code modules uses the function. The CVODES column shows function usage within the main integrator module, while the remaining columns show function usage within each of the CVODES linear solver interfaces, the CVBANDPRE and CVBBDPRE preconditioner modules, and the CVODES adjoint sensitivity module (denoted here by CVODEA). Here CVLS stands for the generic linear solver interface in CVODES, and CVDIAG stands for the diagonal linear solver interface in CVODES.

At this point, we should emphasize that the CVODES user does not need to know anything about the usage of vector functions by the CVODES code modules in order to use CVODES. 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 CVODES modules for user feedback.
3. The optional function \texttt{N_VDotProdMulti} is only used in the \texttt{SUNNONLINSOL\_FIXEDPOINT} module, or when Classical Gram-Schmidt is enabled with \texttt{SPGMR} or \texttt{SPFGMR}.

Each \texttt{SUNLINSOL} object may require additional \texttt{NVECTOR} routines not listed in the table above. Please see the relevant descriptions of these modules in Sections 9.5-9.15 for additional detail on their \texttt{NVECTOR} requirements.

The remaining operations from Tables 7.3 and 7.4 not listed above are unused and a user-supplied \texttt{NVECTOR} module for \texttt{CVODE} could omit these operations. The functions \texttt{N\_MinQuotient}, \texttt{N\_VConstrMask}, and \texttt{N\_VCompare} are only used when constraint checking is enabled and may be omitted if this feature is not used.

### 7.2 The \texttt{NVECTOR\_SERIAL} implementation

The serial implementation of the \texttt{NVECTOR} module provided with \texttt{SUNDIALS}, \texttt{NVECTOR\_SERIAL}, defines the \texttt{content} field of \texttt{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 \texttt{own\_data} which specifies the ownership of \texttt{data}.

```c
struct _N\_VectorContent\_Serial {
    sunindextype length;
    booleantype own\_data;
    realtype *data;
};
```

The header file to include when using this module is \texttt{nvector\_serial.h}. The installed module library to link to is \texttt{libsundials\_nvecserial.lib} where \texttt{.lib} is typically \texttt{.so} for shared libraries and \texttt{.a} for static libraries.

### 7.2.1 \texttt{NVECTOR\_SERIAL} accessor macros

The following macros are provided to access the content of an \texttt{NVECTOR\_SERIAL} vector. The suffix \texttt{\_S} in the names denotes the serial version.

- \texttt{NV\_CONTENT\_S}
  This routine gives access to the contents of the serial vector \texttt{N\_Vector}.
  The assignment \texttt{v\_cont} = \texttt{NV\_CONTENT\_S(v)} sets \texttt{v\_cont} to be a pointer to the serial \texttt{N\_Vector} content structure.
  Implementation:
  ```c
  #define NV\_CONTENT\_S(v) ( (N\_VectorContent\_Serial)(v->content) )
  ```

- \texttt{NV\_OWN\_DATA\_S, NV\_DATA\_S, NV\_LENGTH\_S}
  These macros give individual access to the parts of the content of a serial \texttt{N\_Vector}.
  The assignment \texttt{v\_data} = \texttt{NV\_DATA\_S(v)} sets \texttt{v\_data} to be a pointer to the first component of the data for the \texttt{N\_Vector} \texttt{v}. The assignment \texttt{NV\_DATA\_S(v) = v\_data} sets the component array of \texttt{v} to be \texttt{v\_data} by storing the pointer \texttt{v\_data}.
  The assignment \texttt{v\_len} = \texttt{NV\_LENGTH\_S(v)} sets \texttt{v\_len} to be the length of \texttt{v}. On the other hand, the call \texttt{NV\_LENGTH\_S(v) = len\_v} sets the length of \texttt{v} to be \texttt{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:

\[
define NV_Ith_S(v,i) ( NV_DATA_S(v)[i] )
\]

### 7.2.2 NVECTOR_SERIAL functions

The NVECTOR_SERIAL module defines serial implementations of all vector operations listed in Tables 7.2, 7.3, and 7.4. Their names are obtained from those in Tables 7.2, 7.3, and 7.4 by appending the suffix _Serial (e.g. N_VDestroy_Serial). All the standard vector operations listed in 7.2 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.
### 7.2 The NVECTOR_SERIAL implementation

#### N_VDestroyVectorArray_Serial

**Prototype**

```c
void N_VDestroyVectorArray_Serial(N_Vector *vs, int count);
```

**Description**

This function frees memory allocated for the array of `count` variables of type `N_Vector` created with `N_VCloneVectorArray_Serial` or with `N_VCloneVectorArrayEmpty_Serial`.

#### N_VGetLength_Serial

**Prototype**

```c
sunindextype N_VGetLength_Serial(N_Vector v);
```

**Description**

This function returns the number of vector elements.

**F2003 Name**

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

#### N_VPrint_Serial

**Prototype**

```c
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**

```c
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**

```c
int N_VEnableFusedOps_Serial(N_Vector v, booleantype 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**

```c
int N_VEnableLinearCombination_Serial(N_Vector v, booleantype 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, booleantype 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, booleantype 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, booleantype 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.

**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.
7.2 The NVECTOR_SERIAL implementation

\textbf{N\_VEnableScaleAddMultiVectorArray\_Serial}

Prototype: \texttt{int N\_VEnableScaleAddMultiVectorArray\_Serial(N\_Vector v, boolean\_type tf);}  

Description: This function enables (\texttt{SUNTRUE}) or disables (\texttt{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 \texttt{ops} structure are NULL.

\textbf{N\_VEnableLinearCombinationVectorArray\_Serial}

Prototype: \texttt{int N\_VEnableLinearCombinationVectorArray\_Serial(N\_Vector v, boolean\_type tf);}  

Description: This function enables (\texttt{SUNTRUE}) or disables (\texttt{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 \texttt{ops} structure are NULL.

Notes

- When looping over the components of an \texttt{N\_Vector v}, it is more efficient to first obtain the component array via \texttt{v\_data = NV\_DATA\_S(v)} and then access \texttt{v\_data[i]} within the loop than it is to use \texttt{NV\_Ith\_S(v,i)} within the loop.

- \texttt{N\_VNewEmpty\_Serial}, \texttt{N\_VMake\_Serial}, and \texttt{N\_VC1oneVectorArray\_Empty\_Serial} set the field \texttt{own\_data = SUNFALSE}. \texttt{N\_VDestroy\_Serial} and \texttt{N\_VDestroyVectorArray\_Serial} 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 NVECTOR_SERIAL 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.

7.2.3 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.

\textbf{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 \texttt{N\_VNew\_Serial} is interfaced as \texttt{FN\_VNew\_Serial}.

The FORTRAN 2003 NVECTOR_SERIAL interface module can be accessed with the use statement, i.e. use fnvector_serial_mod, and linking to the library 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.

\textbf{FORTRAN 77 interface functions}

For solvers that include a FORTRAN 77 interface module, the NVECTOR_SERIAL module also includes a FORTRAN-callable function \texttt{FNVINITs(code, NEQ, IER)}, to initialize this NVECTOR_SERIAL 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 long int); and \texttt{IER} is an error return flag equal 0 for success and -1 for failure.
7.3 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;
    booleanown_data;
    realtyt*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.

7.3.1 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:
  ```c
  #define NV_CONTENT_P(v) ( (N_VectorContent_Parallel)(v->content) )
  ```

  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) = v_llen_v sets the local length of v to be v_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) = v_glen_v sets the global length of v to be v_glen_v.
  Implementation:
  ```c
  #define NV_OWN_DATA_P(v) ( NV_CONTENT_P(v)->own_data )
  #define NV_DATA_P(v) ( NVCONTENT_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:
  ```c
  #define NV_COMM_P(v) ( NV_CONTENT_P(v)->comm )
  ```
7.3 The NVVECTOR_PARALLEL implementation

- NV_Ith_P
  This macro gives access to the individual components of the local data array of an N_vector.
  The assignment \( r = NV_{\text{Ith}}(v,i) \) sets \( r \) to be the value of the \( i \)-th component of the local part of \( v \). The assignment \( NV_{\text{Ith}}(v,i) = r \) 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:
  ```c
  #define NV_Ith_P(v,i) ( NV_DATA_P(v)[i] )
  ```

7.3.2 NVVECTOR_PARALLEL functions

The NVVECTOR_PARALLEL module defines parallel implementations of all vector operations listed in Tables 7.2, 7.3, and 7.4. Their names are obtained from those in Tables 7.2, 7.3, and 7.4 by appending the suffix _Parallel (e.g. N_VDestroy_Parallel). The module NVVECTOR_PARALLEL provides the following additional user-callable routines:

- N_VNew_Parallel
  Prototype: \( \text{N\_Vector N\_VNew\_Parallel(MPI\_Comm comm, sunindextype local\_length, sunindextype global\_length);} \)
  Description: This function creates and allocates memory for a parallel vector.

- N_VNewEmpty_Parallel
  Prototype: \( \text{N\_Vector N\_VNewEmpty\_Parallel(MPI\_Comm comm, sunindextype local\_length, sunindextype global\_length);} \)
  Description: This function creates a new parallel N\_Vector with an empty (NULL) data array.

- N_VMake_Parallel
  Prototype: \( \text{N\_Vector N\_VMake\_Parallel(MPI\_Comm comm, sunindextype local\_length, sunindextype global\_length, realtype *v\_data);} \)
  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: \( \text{N\_Vector *N\_VCloneVectorArray\_Parallel(int count, N\_Vector w);} \)
  Description: This function creates (by cloning) an array of count parallel vectors.

- N_VCloneVectorArrayEmpty_Parallel
  Prototype: \( \text{N\_Vector *N\_VCloneVectorArrayEmpty\_Parallel(int count, N\_Vector w);} \)
  Description: This function creates (by cloning) an array of count parallel vectors, each with an empty (NULL) data array.

- N_VDestroyVectorArray_Parallel
  Prototype: \( \text{void N\_VDestroyVectorArray\_Parallel(N\_Vector *vs, int count);} \)
  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.
Description of the NVECTOR module

**N_VGetLength_Parallel**

Prototype: `sunindextype N_VGetLength_Parallel(NVector v);`

Description: This function returns the number of vector elements (global vector length).

**N_VGetLocalLength_Parallel**

Prototype: `sunindextype N_VGetLocalLength_Parallel(NVector v);`

Description: This function returns the local vector length.

**N_VPrint_Parallel**

Prototype: `void N_VPrint_Parallel(NVector v);`

Description: This function prints the local content of a parallel vector to `stdout`.

**N_VPrintFile_Parallel**

Prototype: `void N_VPrintFile_Parallel(NVector 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 NVector_Parallel module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with 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 NVector_Parallel module.

**N_VEnableFusedOps_Parallel**

Prototype: `int N_VEnableFusedOps_Parallel(NVector v, booleantype tf);`

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

**N_VEnableLinearCombination_Parallel**

Prototype: `int N_VEnableLinearCombination_Parallel(NVector v, booleantype tf);`

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

**N_VEnableScaleAddMulti_Parallel**

Prototype: `int N_VEnableScaleAddMulti_Parallel(NVector v, booleantype tf);`

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

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

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

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

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

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

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

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

Prototype: `int N_VEnableLinearCombinationVectorArray_Parallel(N_Vector v, boolantype tf);`

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

Notes:

- When looping over the components of an N_Vector v, it is more efficient to first obtain the local component array via `v_data = NV_DATA_P(v)` and then access `v_data[i]` within the loop than it is to use `NV_Ith_P(v,i)` within the loop.

- `N_VNewEmpty_Parallel`, `N_VMake_Parallel`, and `N_VCloneVectorArrayEmpty_Parallel` set the field `own_data` = SUNFALSE. `N_VDestroy_Parallel` and `N_VDestroyVectorArray_Parallel` will not attempt to free the pointer `data` for any N_Vector with `own_data` set to SUNFALSE. In such a case, it is the user’s responsibility to deallocate the `data` pointer.

- To maximize efficiency, vector operations in the NVECTOR_PARALLEL implementation that have more than one N_Vector argument do not check for consistent internal representation of these vectors. It is the user’s responsibility to ensure that such routines are called with N_Vector arguments that were all created with the same internal representations.

### 7.3.3 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 forida, 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.

### 7.4 The NVECTOR_OPENMP implementation

In situations where a user has a multi-core processing unit capable of running multiple parallel threads with shared memory, SUNDIALS provides an implementation of NVECTOR using OpenMP, called NVECTOR_OPENMP, and an implementation using Pthreads, called NVECTOR_PTHREADS. Testing has shown that vectors should be of length at least 100,000 before the overhead associated with creating and using the threads is made up by the parallelism in the vector calculations.

The OpenMP NVECTOR implementation provided with SUNDIALS, NVECTOR_OPENMP, defines the `content` field of N_Vector to be a structure containing the length of the vector, a pointer to the beginning of a contiguous data array, a boolean flag `own_data` which specifies the ownership of `data`, and the number of threads. Operations on the vector are threaded using OpenMP.

```c
struct _N_VectorContent_OpenMP {
    sunindextype length;
    boolantype own_data;
    realtype *data;
    int num_threads;
};
```
7.4 The NVECTOR_OPENMP implementation

The header file to include when using this module is nvector_openmp.h. The installed module library to link to is lib sundials_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.

7.4.1 NVECTOR_OPENMP accessor macros

The following macros are provided to access the content of an NVECTOR_OPENMP vector. The suffix _OMP in the names denotes the OpenMP version.

- **NV_CONTENT_OMP**
  
  This routine gives access to the contents of the OpenMP vector N_Vector.
  
  The assignment \( v\_\text{cont} = \text{NV\_CONTENT\_OMP}(v) \) sets \( v\_\text{cont} \) 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 \( v\_\text{data} = \text{NV\_DATA\_OMP}(v) \) sets \( v\_\text{data} \) to be a pointer to the first component of the data for the N_Vector v. The assignment \( \text{NV\_DATA\_OMP}(v) = v\_\text{data} \) sets the component array of v to be \( v\_\text{data} \) by storing the pointer \( v\_\text{data} \).
  
  The assignment \( v\_\text{len} = \text{NV\_LENGTH\_OMP}(v) \) sets \( v\_\text{len} \) to be the length of v. On the other hand, the call \( \text{NV\_LENGTH\_OMP}(v) = v\_\text{len} \) sets the length of v to be \( v\_\text{len} \).
  
  The assignment \( v\_\text{num\_threads} = \text{NV\_NUM\_THREADS\_OMP}(v) \) sets \( v\_\text{num\_threads} \) to be the number of threads from v. On the other hand, the call \( \text{NV\_NUM\_THREADS\_OMP}(v) = v\_\text{num\_threads} \) sets the number of threads for v to be \( v\_\text{num\_threads} \).
  
  Implementation:
  
  ```c
  #define NV_OWN_DATA_OMP(v) ( NV_CONTENT_OMP(v)->own_data )
  #define NV_DATA_OMP(v) ( NV_CONTENT_OMP(v)->data )
  #define NV_LENGTH_OMP(v) ( NV_CONTENT_OMP(v)->length )
  #define NV_NUM_THREADS_OMP(v) ( NV_CONTENT_OMP(v)->num_threads )
  ```

- **NV_Ith_OMP**
  
  This macro gives access to the individual components of the data array of an N_Vector.
  
  The assignment \( r = \text{NV\_Ith\_OMP}(v,i) \) sets \( r \) to be the value of the i-th component of v. The assignment \( \text{NV\_Ith\_OMP}(v,i) = r \) 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] )
  ```

7.4.2 NVECTOR_OPENMP functions

The nvector_openmp module defines OpenMP implementations of all vector operations listed in Tables 7.2, 7.3, and 7.4. Their names are obtained from those in Tables 7.2, 7.3, and 7.4 by appending the suffix _OpenMP (e.g. N_VDestroy_OpenMP). All the standard vector operations listed in 7.2 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:
Description of the NVeCToR module

**N.New.OpenMP**

Prototype: `N_Vector N.New_OpenMP(sunindextype vec_length, int num_threads)`

Description: This function creates and allocates memory for an OpenMP `N_Vector`. Arguments are the vector length and number of threads.

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

**N.New.Empty.OpenMP**

Prototype: `N_Vector N.NewEmpty_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_NewEmpty_OpenMP` when using the Fortran 2003 interface module.

**N.Make.OpenMP**

Prototype: `N_Vector N.Make_OpenMP(sunindextype vec_length, realtype *v_data, int num_threads)`

Description: This function creates and allocates memory for an OpenMP vector with user-provided data array. This function does not allocate memory for `v_data` itself.

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

**N.CloneVectorArray.OpenMP**

Prototype: `N_Vector *N.CloneVectorArray_OpenMP(int count, N_Vector w)`

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

**N.CloneVectorArrayEmpty.OpenMP**

Prototype: `N_Vector *N.CloneVectorArrayEmpty_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.DestroyVectorArray.OpenMP**

Prototype: `void N.DestroyVectorArray_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.CloneVectorArray_OpenMP` or with `N.CloneVectorArrayEmpty_OpenMP`.

**N.GetLength.OpenMP**

Prototype: `sunindextype N.GetLength_OpenMP(N_Vector v)`

Description: This function returns number of vector elements.

F2003 Name: This function is callable as `FN.GetLength_OpenMP` when using the Fortran 2003 interface module.
7.4 The NVECTOR_OPENMP implementation

The NVECTOR_OPENMP implementation provides functions for printing the content of an OpenMP vector and enabling or disabling fused and vector array operations for a specific vector. The following routines are provided to enable or disable these operations for a specific vector.

### N_VPrint_OpenMP

**Prototype**

```c
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**

```c
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**

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

**Description**

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

### N_VEnableLinearCombination_OpenMP

**Prototype**

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

**Description**

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

### N_VEnableScaleAddMulti_OpenMP

**Prototype**

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

**Description**

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

### N_VEnableDotProdMulti_OpenMP

**Prototype**

```c
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:  

```c
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:  

```c
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:  

```c
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:  

```c
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:  

```c
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:  

```c
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.

**N_VEnableLinearCombinationVectorArray_OpenMP**

Prototype:  

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

Description:  

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

- When looping over the components of an `N_Vector v`, it is more efficient to first obtain the component array via `v.data = NV_DATA_OMP(v)` and then access `v.data[i]` within the loop than it is to use `NV_Ith_OMP(v, i)` within the loop.

- `N_VNewEmpty_OpenMP`, `N_VMake_OpenMP`, and `N_VCloneVectorArrayEmpty_OpenMP` set the field `own_data = SUNFALSE`. `N_VDestroy_OpenMP` and `N_VDestroyVectorArray_OpenMP` will not attempt to free the pointer `data` for any `N_Vector` with `own_data` set to `SUNFALSE`. In such a case, it is the user’s responsibility to deallocate the `data` pointer.

- To maximize efficiency, vector operations in the `NVECTOR_OPENMP` implementation that have more than one `N_Vector` argument do not check for consistent internal representation of these vectors. It is the user’s responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the same internal representations.

### 7.4.3 NVECTOR_OPENMP Fortran interfaces

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

**FORTRAN 2003 interface module**

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

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

**FORTRAN 77 interface functions**

For solvers that include a FORTRAN 77 interface module, the NVECTOR_OPENMP module also includes a FORTRAN-callable function `FNINITOMP(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.

### 7.5 The NVECTOR_PTHREADS implementation

In situations where a user has a multi-core processing unit capable of running multiple parallel threads with shared memory, SUNDIALS provides an implementation of NVECTOR using OpenMP, called `NVECTOR_OPENMP`, and an implementation using Pthreads, called `NVECTOR_PTHREADS`. Testing has shown that vectors should be of length at least 100,000 before the overhead associated with creating and using the threads is made up by the parallelism in the vector calculations.

The Pthreads NVECTOR implementation provided with SUNDIALS, denoted `NVECTOR_PTHREADS`, defines the `content` field of `N_Vector` to be a structure containing the length of the vector, a pointer to the beginning of a contiguous data array, a boolean flag `own_data` which specifies the ownership of `data`, and the number of threads. Operations on the vector are threaded using POSIX threads (Pthreads).
struct _N_VectorContent_Pthreads {
    sunindextype length;
    booleantype own_data;
    realtype *data;
    int num_threads;
};

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

7.5.1 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] )
  ```
7.5.2 NVIRTUAL_PTHREADS functions

The NVECTOR_PTHREADS module defines Pthreads implementations of all vector operations listed in Tables 7.2, 7.3, and 7.4. Their names are obtained from those in Tables 7.2, 7.3, and 7.4 by appending the suffix _Pthreads (e.g. N_VDestroy_Pthreads). All the standard vector operations listed in 7.2 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:

\textbf{N_VNew_Pthreads}

Prototype: \texttt{N_Vector N_VNew_Pthreads(sunindextype vec\_length, int num\_threads)}

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

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

\textbf{N_VNewEmpty_Pthreads}

Prototype: \texttt{N_Vector N_VNewEmpty_Pthreads(sunindextype vec\_length, int num\_threads)}

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

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

\textbf{N_VMake_Pthreads}

Prototype: \texttt{N_Vector N_VMake_Pthreads(sunindextype vec\_length, realtype *v\_data, int num\_threads)}

Description: This function creates and allocates memory for a Pthreads vector with user-provided data array. This function does \textit{not} allocate memory for \texttt{v\_data} itself.

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

\textbf{N_VCloneVectorArray_Pthreads}

Prototype: \texttt{N_Vector *N_VCloneVectorArray_Pthreads(int count, N_Vector w)}

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

\textbf{N_VCloneVectorArrayEmpty_Pthreads}

Prototype: \texttt{N_Vector *N_VCloneVectorArrayEmpty_Pthreads(int count, N_Vector w)}

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

\textbf{N_VDestroyVectorArray_Pthreads}

Prototype: \texttt{void N_VDestroyVectorArray_Pthreads(N_Vector *vs, int count)}

Description: This function frees memory allocated for the array of \textit{count} variables of type N_Vector created with \texttt{N_VCloneVectorArray_Pthreads} or with \texttt{N_VCloneVectorArrayEmpty_Pthreads}.
Description of the NVECTOR module

**N_VGetLength_Pthreads**

Prototype: `sunindextype N_VGetLength_Pthreads(N_Vector v)`

Description: This function returns the number of vector elements.

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

**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 `NVECTOR_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 `NVECTOR_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**

```c
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**

```c
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**

```c
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**

```c
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**

```c
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**

```c
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**

```c
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.

7.5.3 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.

7.6 The NVECTOR_PARHYHP implementation

The NVECTOR_PARHYHP 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.
7.6 The NVECTOR_PARHYP implementation

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

The header file to include when using this module is `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.

7.6.1 NVECTOR_PARHYP functions

The NVECTOR_PARHYP module defines implementations of all vector operations listed in Tables 7.2, 7.3, and 7.4, 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 [39].

The names of parhyp methods are obtained from those in Tables 7.2, 7.3, and 7.4 by appending the suffix _ParHyp (e.g. N_VDestroy_ParHyp). The module NVECTOR_PARHYP provides the following additional user-callable routines:

**N_VNewEmpty_ParHyp**

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.

**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.
7.6 The NVECTOR_PARHYP implementation 193

**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.

**N_VEnableConstVectorArray_ParHyp**

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.

**N_VEnableWrmsNormVectorArray_ParHyp**

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.

**N_VEnableWrmsNormMaskVectorArray_ParHyp**

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.

**N_VEnableScaleAddMultiVectorArray_ParHyp**

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.
**N_VEnableLinearCombinationVectorArray_ParHyp**

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 `x_vec = N_VGetVector_ParHyp(v)` and then access components using appropriate hypre functions.
- `N_VNewEmpty_ParHyp`, `N_VMake_ParHyp`, and `N_VCloneVectorArrayEmpty_ParHyp` set the field `own_parvector` to SUNFALSE. `N_Destroy_ParHyp` and `N_DestroyVectorArray_ParHyp` will not attempt to delete an underlying hypre vector for any N_Vector with `own_parvector` set to SUNFALSE. In such a case, it is the user's responsibility to delete the underlying vector.
- To maximize efficiency, vector operations in the NVECTOR_PARHYP implementation that have more than one N_Vector argument do not check for consistent internal representations of these vectors. It is the user’s responsibility to ensure that such routines are called with N_Vector arguments that were all created with the same internal representations.

### 7.7 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.

### 7.7.1 NVECTOR_PETSC functions

The NVECTOR_PETSC module defines implementations of all vector operations listed in Tables 7.2, 7.3, and 7.4, except for `N_GetArrayPointer` and `N_SetArrayPointer`. 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 7.2, 7.3, and 7.4 by appending the suffix _Petsc (e.g. N_Destroy_Petsc). The module NVECTOR_PETSC provides the following additional user-callable routines:
### 7.7 The NVECTOR_PETSC implementation

**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.

**N_VCloneVectorArray_Petsc**

**Prototype**

```c
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**

```c
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**

```c
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**

```c
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**

```c
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.

**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.
7.8 The NVECTOR_CUDA implementation

\begin{verbatim}
N_VEnableWrmsNormVectorArray_Petsc
Prototype int N_VEnableWrmsNormVectorArray_Petsc(N_Vector v, booleantype tf)
Description This function enables \texttt{(SUNTRUE)} or disables \texttt{(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 \texttt{ops} structure are NULL.

N_VEnableWrmsNormMaskVectorArray_Petsc
Prototype int N_VEnableWrmsNormMaskVectorArray_Petsc(N_Vector v, booleantype tf)
Description This function enables \texttt{(SUNTRUE)} or disables \texttt{(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 \texttt{ops} structure are NULL.

N_VEnableScaleAddMultiVectorArray_Petsc
Prototype int N_VEnableScaleAddMultiVectorArray_Petsc(N_Vector v, booleantype tf)
Description This function enables \texttt{(SUNTRUE)} or disables \texttt{(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 \texttt{ops} structure are NULL.

N_VEnableLinearCombinationVectorArray_Petsc
Prototype int N_VEnableLinearCombinationVectorArray_Petsc(N_Vector v, booleantype tf)
Description This function enables \texttt{(SUNTRUE)} or disables \texttt{(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 \texttt{ops} structure are NULL.

Notes

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

- The functions \texttt{N_VNewEmpty_Petsc}, \texttt{N_VMake_Petsc}, and \texttt{N_VCloneVectorArrayEmpty_Petsc} set the field \texttt{own_data} to \texttt{SUNFALSE}. \texttt{N_VDestroy_Petsc} and \texttt{N_VDestroyVectorArray_Petsc} will not attempt to free the pointer \texttt{pvec} 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{pvec} pointer.

- To maximize efficiency, vector operations in the NVVECTOR_PETSC implementation that have more than one \texttt{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 \texttt{N_Vector} arguments that were all created with the same internal representations.

7.8 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 \texttt{Vector} in the namespace \texttt{suncudavec} manages the vector data layout:
\end{verbatim}
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_;  
    ...
};

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 \texttt{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 \texttt{Vector} inherits from the empty structure

\texttt{struct \_N\_VectorContent\_Cuda \{};

\texttt{}};

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

When instantiated with \texttt{N\_VNew\_Cuda}, the class \texttt{Vector} will allocate memory on both the host and the device. Alternatively, a user can provide host and device data arrays by using the \texttt{N\_VMake\_Cuda} constructor. To use CUDA managed memory, the constructors \texttt{N\_VNewManaged\_Cuda} and \texttt{N\_VMakeManaged\_Cuda} are provided. Details on each of these constructors are provided below.

The \texttt{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 \texttt{nvector\_cuda.h} and the library to link to is \texttt{libsundials\_nveccuda.lib}. In the a distributed setting the header file to include is \texttt{nvector\_mpicuda.h} and the library to link to is \texttt{libsundials\_nvecmpicuda.lib}. The extension, \texttt{.lib}, is typically \texttt{.so} for shared libraries and \texttt{.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.

### 7.8.1 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\_VGetLength\_Cuda**
  - Prototype: `sunindextype N\_VGetLength\_Cuda(N\_Vector v)`
  - Description: This function returns the global length of the vector.

- **N\_VGetLocalLength\_Cuda**
  - Prototype: `sunindextype N\_VGetLocalLength\_Cuda(N\_Vector v)`
7.8 The NV_VECTOR_CUDA implementation

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
nvectormpicuda.h and the library to link to is lib sundials_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
nvectormpicuda.h and the library to link to is lib sundials_nvecmpicuda.lib.

**N_VIsManagedMemory_Cuda**

Prototype  `booltype *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 NV_VECTOR_CUDA module defines implementations of all vector operations listed in Tables 7.2,
7.3, and 7.4, except for N_VGetArrayPointer and N_VSetArrayPointer. As such, this vector cannot be
used with the SUNDIALS Fortran interfaces, nor with the SUNDIALS direct solvers and preconditioners.
Instead, the NV_VECTOR_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 NV_VECTOR_CUDA are provided in some example programs for CVODE [29].

The names of vector operations are obtained from those in Tables 7.2, 7.3, and 7.4 by appending the
suffix _Cuda (e.g. N_VDestroy_Cuda). The module NV_VECTOR_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 nvectormcuda.h and the
library to link to is lib sundials_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 nvectormpicuda.h and the
library to link to is lib sundials_nvecmpicuda.lib.
Description of the NVECTOR module

**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`.

**N_VNewEmpty_Cuda**

Prototype: `N_Vector N_VNewEmpty_Cuda()`

Description: This function creates a new `NVECTOR` wrapper with the pointer to the wrapped CUDA vector set to NULL. It is used by the `N_New_Cuda`, `N_Make_Cuda`, and `N_Clon_Cuda` implementations.

**N_VMake_Cuda**

**Single-node usage**

Prototype: `N_Vector N_VMake_Cuda(sunindextype length, realtype *h_vdata, realtype *d_vdata)`

Description: This function creates an `NVECTOR_CUDA` with user-supplied vector data arrays `h_vdata` and `d_vdata`. This function does not allocate memory for data itself. In the single-node setting, the inputs are the vector length, the host data array, and the device data. This constructor is defined in the header `nvector_cuda.h` and the library to link to is `libsundials_nveccuda.lib`.

**Distributed-memory parallel usage**

Prototype: `N_Vector N_VMake_Cuda(MPI_Comm comm, sunindextype local_length, sunindextype global_length, realtype *h_vdata, realtype *d_vdata)`

Description: This function creates an `NVECTOR_CUDA` with user-supplied vector data arrays `h_vdata` and `d_vdata`. This function does not allocate memory for data itself. When used in a distributed context with MPI, the arguments are the MPI communicator, the local vector length, the global vector length, the host data array, and the device data array. This constructor is defined in the header `nvector_mpicuda.h` and the library to link to is `libsundials_nvecmpicuda.lib`.

**N_VMakeManaged_Cuda**

**Single-node usage**

Prototype: `N_Vector N_VMakeManaged_Cuda(sunindextype length, realtype *vdata)`
Description This function creates an 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 nvctor_cuda.h and the library to link to is libsundials_nvvecuda.lib.

Distributed-memory parallel usage

Prototype

```c
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 nvctor_mpivecuda.h and the library to link to is libsundials_nvmpivecuda.lib.

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

### N_VSetCudaStream_Cuda

Prototype

```c
void N_VSetCudaStream_Cuda(N_Vector v, cudaStream_t *stream)
```

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

```c
void N_VCopyToDevice_Cuda(N_Vector v)
```

Description This function copies host vector data to the device.

### N_VCopyFromDevice_Cuda

Prototype

```c
void N_VCopyFromDevice_Cuda(N_Vector v)
```

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

### N_VPrint_Cuda

Prototype

```c
void N_VPrint_Cuda(N_Vector v)
```

Description This function prints the content of a CUDA vector to stdout.

### N_VPrintFile_Cuda

Prototype

```c
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, booleantype tf)

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

**N_VEnableLinearCombination_Cuda**

Prototype: int N_VEnableLinearCombination_Cuda(N_Vector v, booleantype tf)

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

**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.
7.9 The NVECTOR RAJA implementation

The nvector raja module is an experimental NVECTOR implementation using the RAJA hardware abstraction layer. In this implementation, RAJA allows for SUNDIALS vector kernels to run on GPU devices. The module is intended for users who are already familiar with RAJA and GPU programming. Building this vector module requires a C++11 compliant compiler and a CUDA software development toolkit. Besides the CUDA backend, RAJA has other backends such as serial, OpenMP, and OpenACC. These backends are not used in this SUNDIALS release. Class Vector in namespace sunrajavec manages the vector data layout:

```cpp
template <class T, class I>
class Vector {
    I size_;  
    I mem_size_;  
    I global_size_;  
};
```
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

```c
struct _N_VectorContent_Raja {
};
```

to interface the C++ class with the NVVECTOR 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 `sunrajayvec::Vector` class will change frequently in future SUNDIALS releases. The code is structured so that it can tolerate significant changes in the `sunrajayvec::Vector` class without requiring changes to the user API.

The `NVVECTOR_RAJA` module can be utilized for single-node parallelism or in a distributed context with MPI. The header file to include when using this module for single-node parallelism is `nvvector_raja.h`. The header file to include when using this module in the distributed case is `nvvector_mpiraja.h`. The installed module libraries to link to are `libsundials_nvecraja.lib` in the single-node case, or `libsundials_nvecmpicudaraja.lib` in the distributed case. Only one of these libraries may be linked to when creating an executable or library. SUNDIALS must be built with MPI support if the distributed library is desired.

### 7.9.1 NVVECTOR_RAJA functions

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

**N_VGetLength_Raja**

Prototype: `sunindextype N_VGetLength_Raja(N_Vector v)`

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

**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 `nvvector_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.
The \texttt{NVECTOR\_RAJA} implementation defines the implementations of all vector operations listed in Tables 7.2, 7.3, and 7.4, except for \texttt{N\_VDotProdMulti}, \texttt{N\_VWrmsNormVectorArray}, and \texttt{N\_VWrmsNormMaskVectorArray} as support for arrays of reduction vectors is not yet supported in RAJA. These function will be added to the \texttt{NVECTOR\_RAJA} implementation in the future. Additionally the vector operations \texttt{N\_VGetArrayPointer} and \texttt{N\_VSetArrayPointer} are not implemented by the RAJA vector. As such, this vector cannot be used with the SUNDIALS Fortran interfaces, nor with the SUNDIALS direct solvers and preconditioners. The \texttt{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 \texttt{NVECTOR\_RAJA} are provided in some example programs for CVODE [29].

The names of vector operations are obtained from those in Tables 7.2, 7.3, and 7.4, by appending the suffix \_Raja (e.g. \texttt{N\_VDestroy\_Raja}). The module \texttt{NVECTOR\_RAJA} provides the following additional user-callable routines:

**N\_VNew\_Raja**

\textit{Single-node usage}

\textbf{Prototype} \quad \texttt{N\_Vector N\_VNew\_Raja(sunindextype length)}

\textbf{Description} This function creates and allocates memory for a CUDA \texttt{N\_Vector}. The vector data array is allocated on both the host and device. In the \textit{single-node} setting, the only input is the vector length. This constructor is defined in the header \texttt{nvector\_raja.h} and the library to link to is \texttt{libsundials\_nvec\_cuda\_raja.lib}.

\textbf{Distributed-memory parallel usage}

\textbf{Prototype} \quad \texttt{N\_Vector N\_VNew\_Raja(MPI\_Comm comm, sunindextype local\_length, sunindextype global\_length)}

\textbf{Description} This function creates and allocates memory for a CUDA \texttt{N\_Vector}. The vector data array is allocated on both the host and device. When used in a \textit{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 \texttt{nvector\_mpi\_raja.h} and the library to link to is \texttt{libsundials\_nvec\_mpicuda\_raja.lib}.

**N\_VNewEmpty\_Raja**

\textbf{Prototype} \quad \texttt{N\_Vector N\_VNewEmpty\_Raja()}\newline

\textbf{Description} This function creates a new \texttt{NVECTOR} wrapper with the pointer to the wrapped RAJA vector set to NULL. It is used by the \texttt{N\_VNew\_Raja}, \texttt{N\_VMake\_Raja}, and \texttt{N\_VClone\_Raja} implementations.

**N\_VMake\_Raja**

\textbf{Prototype} \quad \texttt{N\_Vector N\_VMake\_Raja(N\_VectorContent\_Raja c)}\newline

\textbf{Description} This function creates and allocates memory for an \texttt{NVECTOR\_RAJA} wrapper around a user-provided \texttt{sunrajavec\_:Vector} class. Its only argument is of type \texttt{N\_VectorContent\_Raja}, which is the pointer to the class.
Description of the NVVECTOR module

**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.

**N_VPrintFile_Raja**

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

Description: This function prints the content of a RAJA vector to outfile.

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

**N_VEnableFusedOps_Raja**

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

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

**N_VEnableLinearCombination_Raja**

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

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

**N_VEnableScaleAddMulti_Raja**

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

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

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

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

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

**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.

**7.10 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.

7.10.1 NVECTOR_OPENMPDEV accessor macros

The following macros are provided to access the content of an NVECTOR_OPENMPDEV vector.

- **NV_CONTENT_OMPDEV**
  This routine gives access to the contents of the NVECTOR_OPENMPDEV vector N_Vector.
  The assignment \( v\text{\_cont} = \text{NV\_CONTENT\_OMPDEV}(v) \) sets \( v\text{\_cont} \) to be a pointer to the NVECTOR_OPENMPDEV N_Vector content structure.
  Implementation:
  ```c
  #define NV_CONTENT_OMPDEV(v) ( (N_VectorContent_OpenMPDEV)(v->content) )
  ```

- **NV_OWN_DATA_OMPDEV, NV_DATA_HOST_OMPDEV, NV_DATA_DEV_OMPDEV, NV_LENGTH_OMPDEV**
  These macros give individual access to the parts of the content of an NVECTOR_OPENMPDEV N_Vector.
  The assignment \( v\text{\_data} = \text{NV\_DATA\_HOST\_OMPDEV}(v) \) 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 \( \text{NV\_DATA\_HOST\_OMPDEV}(v) = v\text{\_data} \) sets the host component array of \( v \) to be \( v\text{\_data} \) by storing the pointer \( v\text{\_data} \).
  The assignment \( v\text{\_dev\_data} = \text{NV\_DATA\_DEV\_OMPDEV}(v) \) sets \( v\text{\_dev\_data} \) to be a pointer to the first component of the data on the device for the N_Vector \( v \). The assignment \( \text{NV\_DATA\_DEV\_OMPDEV}(v) = v\text{\_dev\_data} \) sets the device component array of \( v \) to be \( v\text{\_dev\_data} \) by storing the pointer \( v\text{\_dev\_data} \).
  The assignment \( v\text{\_len} = \text{NV\_LENGTH\_OMPDEV}(v) \) sets \( v\text{\_len} \) to be the length of \( v \). On the other hand, the call \( \text{NV\_LENGTH\_OMPDEV}(v) = \text{\_len\_v} \) sets the length of \( v \) to be \( \text{len\_v} \).
  Implementation:
  ```c
  #define NV_OWN_DATA_OMPDEV(v) ( NV_CONTENT_OMPDEV(v)->own_data )
  #define NV_DATA_HOST_OMPDEV(v) ( NV_CONTENT_OMPDEV(v)->host_data )
  #define NV_DATA_DEV_OMPDEV(v) ( NV_CONTENT_OMPDEV(v)->dev_data )
  #define NV_LENGTH_OMPDEV(v) ( NV_CONTENT_OMPDEV(v)->length )
  ```

7.10.2 NVECTOR_OPENMPDEV functions

The NVECTOR_OPENMPDEV module defines OpenMP device offloading implementations of all vector operations listed in Tables 7.2, 7.3, and 7.4, except for N_VGetArrayPointer and N_VSetArrayPointer. As such, this vector cannot be used with the SUNDIALS Fortran interfaces, nor with the SUNDIALS direct solvers and preconditioners. 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 7.2, 7.3, and 7.4 by appending the suffix _OpenMPDEV (e.g. N_VDestroy_OpenMPDEV). The module NVECTOR_OPENMPDEV provides the following additional user-callable routines:
7.10 The NVECTOR_OPENMPDEV implementation

**N_VNew_OpenMPDEV**
Prototype: `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: `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.

**N_VMake_OpenMPDEV**
Prototype: `N_Vector N_VMake_OpenMPDEV(sunindextype vec_length, realtype *h_vdata, realtype *d_vdata)`
Description: This function creates an NVECTOR_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 NVECTOR_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 NVECTOR_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_VGetLength_OpenMPDEV**
Prototype: `sunindextype N_VGetLength_OpenMPDEV(N_Vector v)`
Description: This function returns the number of vector elements.

**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.
Description of the NVECTOR module

N_VPrint_OpenMPDEV
Prototype: void N_VPrint_OpenMPDEV(N_Vector v)
Description: This function prints the content of an NVECTOR_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 NVECTOR_OPENMPDEV vector to outfile.

N_VCopyToDevice_OpenMPDEV
Prototype: void N_VCopyToDevice_OpenMPDEV(N_Vector v)
Description: This function copies the content of an NVECTOR_OPENMPDEV vector's host data array to the device data array.

N_VCopyFromDevice_OpenMPDEV
Prototype: void N_VCopyFromDevice_OpenMPDEV(N_Vector v)
Description: This function copies the content of an NVECTOR_OPENMPDEV vector's device data array to the host data array.

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

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

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

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

**N_VEnableDotProdMulti_OpenMPDEV**

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

**N_VEnableLinearSumVectorArray_OpenMPDEV**

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

**N_VEnableScaleVectorArray_OpenMPDEV**

Prototype: `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: `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: `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: `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: `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.
Prototype

```c
int N_VEnableLinearCombinationVectorArray_OpenMPDEV(N_Vector v,
    boolanetype 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_OMPDEV(v)` for the host array or `d_data = NV_DATA_DEV_OMPDEV(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.

7.11 The NVETER_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

```c
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:

```c
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_nvectortrilinos.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

The NVECTOR_TRILINOS module defines implementations of all vector operations listed in Table 7.2, 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 Table 7.2 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_VMakne_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_VMakne_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.

7.12 NVECTOR Examples

There are NVector examples that may be installed for the implementations provided with SUNDIALS. Each implementation makes use of the functions in test_nvector.c. These example functions show simple usage of the NVector family of functions. The input to the examples are the vector length, number of threads (if threaded implementation), and a print timing flag. The following is a list of the example functions in test_nvector.c:

- Test_N_VClone: Creates clone of vector and checks validity of clone.
- Test_N_VCloneEmpty: Creates clone of empty vector and checks validity of clone.
- Test_N_VCloneVectorArray: Creates clone of vector array and checks validity of cloned array.
- Test_N_VCloneVectorArray: Creates clone of empty vector array and checks validity of cloned array.
- Test_N_VGetArrayPointer: Get array pointer.
- Test_N_VSetArrayPointer: Allocate new vector, set pointer to new vector array, and check values.
• `Test_N_VLinearSum` Case 1a: Test $y = x + y$
• `Test_N_VLinearSum` Case 1b: Test $y = -x + y$
• `Test_N_VLinearSum` Case 1c: Test $y = ax + y$
• `Test_N_VLinearSum` Case 2a: Test $x = x + y$
• `Test_N_VLinearSum` Case 2b: Test $x = x - y$
• `Test_N_VLinearSum` Case 2c: Test $x = x + by$
• `Test_N_VLinearSum` Case 3: Test $z = x + y$
• `Test_N_VLinearSum` Case 4a: Test $z = x - y$
• `Test_N_VLinearSum` Case 4b: Test $z = -x + y$
• `Test_N_VLinearSum` Case 5a: Test $z = x + by$
• `Test_N_VLinearSum` Case 5b: Test $z = ax + y$
• `Test_N_VLinearSum` Case 6a: Test $z = -x + by$
• `Test_N_VLinearSum` Case 6b: Test $z = ax - by$
• `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: $z = x \times y$
• `Test_N_VDiv`: Test vector division: $z = x \div y$
• `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_VRmsNorm`: Create vector of known values, find and validate the weighted root mean square.
• `Test_N_VRmsNormMask`: 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_VL2Norm`: Create vector, find and validate the weighted Euclidean L2 norm.
• `Test_N_VL1Norm`: Create vector, find and validate the L1 norm.
• **Test_NVCompare**: Compare vector with constant returning and validating comparison vector.

• **Test_NVInvTest**: Test $z[i] = 1 / x[i]$

• **Test_NVConstrMask**: Test mask of vector $x$ with vector $c$.

• **Test_NVMinQuotient**: Fill two vectors with known values. Calculate and validate minimum quotient.

• **Test_NVLinearCombination** Case 1a: Test $x = a x$

• **Test_NVLinearCombination** Case 1b: Test $z = a x$

• **Test_NVLinearCombination** Case 2a: Test $x = a x + b y$

• **Test_NVLinearCombination** Case 2b: Test $z = a x + b y$

• **Test_NVLinearCombination** Case 3a: Test $x = x + a y + b z$

• **Test_NVLinearCombination** Case 3b: Test $x = a x + b y + c z$

• **Test_NVLinearCombination** Case 3c: Test $w = a x + b y + c z$

• **Test_NVScaleAddMulti** Case 1a: $y = a x + y$

• **Test_NVScaleAddMulti** Case 1b: $z = a x + y$

• **Test_NVScaleAddMulti** Case 2a: $Y[i] = c[i] x + Y[i]$, $i = 1,2,3$

• **Test_NVScaleAddMulti** Case 2b: $Z[i] = c[i] x + Y[i]$, $i = 1,2,3$

• **Test_NVDotProdMulti** Case 1: Calculate the dot product of two vectors

• **Test_NVDotProdMulti** Case 2: Calculate the dot product of one vector with three other vectors in a vector array.

• **Test_NVLinearSumVectorArray** Case 1: $z = a x + b y$

• **Test_NVLinearSumVectorArray** Case 2a: $Z[i] = a X[i] + b Y[i]$

• **Test_NVLinearSumVectorArray** Case 2b: $X[i] = a X[i] + b Y[i]$

• **Test_NVLinearSumVectorArray** Case 2c: $Y[i] = a X[i] + b Y[i]$

• **Test_NVScaleVectorArray** Case 1a: $y = c y$

• **Test_NVScaleVectorArray** Case 1b: $z = c y$

• **Test_NVScaleVectorArray** Case 2a: $Y[i] = c[i] Y[i]$

• **Test_NVScaleVectorArray** Case 2b: $Z[i] = c[i] Y[i]$

• **Test_NVScaleVectorArray** Case 1a: $z = c$

• **Test_NVScaleVectorArray** Case 1b: $Z[i] = c$

• **Test_NVWrmsNormVectorArray** Case 1a: Create a vector of known values, find and validate the weighted root mean square norm.

• **Test_NVWrmsNormVectorArray** Case 1b: Create a vector array of three vectors of known values, find and validate the weighted root mean square norm of each.

• **Test_NVWrmsNormMaskVectorArray** Case 1a: Create a vector of known values, find and validate the weighted root mean square norm using all elements except one.
• Test_N_VWrmsNormMaskVectorArray Case 1b: Create a vector array of three vectors of known values, find and validate the weighted root mean square norm of each using all elements except one.

• Test_N_VScaleAddMultiVectorArray Case 1a: \(y = a \times x + y\)
• Test_N_VScaleAddMultiVectorArray Case 1b: \(z = a \times x + y\)

• Test_N_VScaleAddMultiVectorArray Case 2a: \(Y[j][0] = a[j] \times X[0] + Y[j][0]\)
• Test_N_VScaleAddMultiVectorArray Case 2b: \(Z[j][0] = a[j] \times X[0] + Y[j][0]\)

• Test_N_VScaleAddMultiVectorArray Case 3a: \(Y[0][i] = a[0] \times X[i] + Y[0][i]\)
• Test_N_VScaleAddMultiVectorArray Case 3b: \(Z[0][i] = a[0] \times X[i] + Y[0][i]\)

• Test_N_VScaleAddMultiVectorArray Case 4a: \(Y[j][i] = a[j] \times X[i] + Y[j][i]\)
• Test_N_VScaleAddMultiVectorArray Case 4b: \(Z[j][i] = a[j] \times X[i] + Y[j][i]\)

• Test_N_VLinearCombinationVectorArray Case 1a: \(x = a \times\)
• Test_N_VLinearCombinationVectorArray Case 1b: \(z = a \times\)

• Test_N_VLinearCombinationVectorArray Case 2a: \(x = a \times + b \times y\)
• Test_N_VLinearCombinationVectorArray Case 2b: \(z = a \times + b \times y\)

• Test_N_VLinearCombinationVectorArray Case 3a: \(x = a \times + b \times y + c \times z\)
• Test_N_VLinearCombinationVectorArray Case 3b: \(w = a \times + b \times y + c \times z\)

• Test_N_VLinearCombinationVectorArray Case 4a: \(X[0][i] = c[0] \times X[0][i]\)
• Test_N_VLinearCombinationVectorArray Case 4b: \(Z[i] = c[0] \times X[0][i]\)

• Test_N_VLinearCombinationVectorArray Case 5a: \(X[0][i] = c[0] \times X[0][i] + c[1] \times X[1][i]\)
• Test_N_VLinearCombinationVectorArray Case 5b: \(Z[i] = c[0] \times X[0][i] + c[1] \times X[1][i]\)

• Test_N_VLinearCombinationVectorArray Case 6a: \(X[0][i] = X[0][i] + c[1] \times X[1][i] + c[2] \times X[2][i]\)
• Test_N_VLinearCombinationVectorArray Case 6b: \(X[0][i] = c[0] \times X[0][i] + c[1] \times X[1][i] + c[2] \times X[2][i]\)

• Test_N_VLinearCombinationVectorArray Case 6c: \(Z[i] = c[0] \times X[0][i] + c[1] \times X[1][i] + c[2] \times X[2][i]\)
### Table 7.5: List of vector functions usage by CVODES code modules

<table>
<thead>
<tr>
<th>Function</th>
<th>CVODES</th>
<th>CVLS</th>
<th>CVDIAG</th>
<th>CYBANDPRE</th>
<th>CYBBPRE</th>
<th>CVODEA</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></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_VDestroyVectorArray</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VSpace</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VGetArrayPointer</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VSetArrayPointer</td>
<td>✓</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_VConst</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_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_MinQuotient</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>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VLinearCombinationVectorArray</td>
<td>✓</td>
<td></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>
<tr>
<td>N_VConstVectorArray</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VWrmsNormVectorArray</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VScaleAddMultiVectorArray</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VLinearCombinationVectorArray</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 8

Description of the SUNMatrix module

For problems that involve direct methods for solving linear systems, the SUNDIALS solvers not only operate on generic vectors, but also on generic matrices (of type SUNMatrix), through a set of operations defined by the particular SUNMATRIX implementation. Users can provide their own specific implementation of the SUNMATRIX module, particularly in cases where they provide their own NVECTOR and/or linear solver modules, and require matrices that are compatible with those implementations. Alternately, we provide three SUNMATRIX implementations: dense, banded, and sparse. The generic operations are described below, and descriptions of the implementations provided with SUNDIALS follow.

The generic SUNMatrix type has been modeled after the object-oriented style of the generic N_Vector type. Specifically, a generic SUNMatrix is a pointer to a structure that has an implementation-dependent content field containing the description and actual data of the matrix, and an ops field pointing to a structure with generic matrix operations. The type SUNMatrix is defined as

typedef struct _generic_SUNMatrix *SUNMatrix;

struct _generic_SUNMatrix {
  void *content;
  struct _generic_SUNMatrix_Ops *ops;
};

The _generic_SUNMatrix_Ops structure is essentially a list of pointers to the various actual matrix operations, and is defined as

struct _generic_SUNMatrix_Ops {
  SUNMatrix_ID (*getid)(SUNMatrix);
  SUNMatrix (*clone)(SUNMatrix);
  void (*destroy)(SUNMatrix);
  int (*zero)(SUNMatrix);
  int (*copy)(SUNMatrix, SUNMatrix);
  int (*scaleadd)(realtype, SUNMatrix, SUNMatrix);
  int (*scaleaddi)(realtype, SUNMatrix);
  int (*matvec)(SUNMatrix, N_Vector, N_Vector);
  int (*space)(SUNMatrix, long int*, long int*);
};

The generic SUNMATRIX module defines and implements the matrix operations acting on SUNMatrix objects. These routines are nothing but wrappers for the matrix operations defined by a particular SUNMATRIX implementation, which are accessed through the ops field of the SUNMatrix structure. To
Table 8.1: 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_CUSTOM</td>
<td>User-provided custom matrix</td>
<td>3</td>
</tr>
</tbody>
</table>

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 8.2 contains a complete list of all matrix operations defined by the generic SUNMATRIX module. A particular implementation of the SUNMATRIX module must:

- Specify the `content` field of the SUNMatrix object.
- Define and implement a minimal subset of the matrix operations. See the documentation for each SUNDIALS solver to determine which SUNMATRIX operations they require.
  
  Note that the names of these routines should be unique to that implementation in order to permit using more than one SUNMATRIX module (each with different SUNMatrix internal data representations) in the same code.
- Define and implement user-callable constructor and destructor routines to create and free a SUNMatrix with the new `content` field and with `ops` pointing to the new matrix operations.
- Optionally, define and implement additional user-callable routines acting on the newly defined SUNMatrix (e.g., a routine to print the content for debugging purposes).
- Optionally, provide accessor macros or functions as needed for that particular implementation to access different parts of the `content` field of the newly defined SUNMatrix.

Each SUNMATRIX implementation included in SUNDIALS has a unique identifier specified in enumeration and shown in Table 8.1. It is recommended that a user-supplied SUNMATRIX implementation use the SUNMATRIX_CUSTOM identifier.

Table 8.2: 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 8.1.</td>
</tr>
</tbody>
</table>

*continued on next page*
<table>
<thead>
<tr>
<th>Name</th>
<th>Usage and Description</th>
</tr>
</thead>
<tbody>
<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>
<tr>
<td>SUNMatDestroy</td>
<td>SUNMatDestroy(A); Destroys the SUNMatrix A and frees memory allocated for its internal data.</td>
</tr>
<tr>
<td>SUNMatSpace</td>
<td>ier = SUNMatSpace(A, &amp;lrw, &amp;liw); Returns the storage requirements for the matrix A. lrw is a long int containing the number of realtype words and liw is a long int containing the number of integer words. The return value is an integer flag denoting success/failure of the operation. This function is advisory only, for use in determining a user’s total space requirements; it could be a dummy function in a user-supplied SUNMATRIX module if that information is not of interest.</td>
</tr>
<tr>
<td>SUNMatZero</td>
<td>ier = SUNMatZero(A); Performs the operation $A_{ij} = 0$ for all entries of the matrix A. The return value is an integer flag denoting success/failure of the operation.</td>
</tr>
<tr>
<td>SUNMatCopy</td>
<td>ier = SUNMatCopy(A,B); Performs the operation $B_{ij} = A_{i,j}$ for all entries of the matrices A and B. The return value is an integer flag denoting success/failure of the operation.</td>
</tr>
<tr>
<td>SUNMatScaleAdd</td>
<td>ier = SUNMatScaleAdd(c, A, B); Performs the operation $A = cA + B$. The return value is an integer flag denoting success/failure of the operation.</td>
</tr>
<tr>
<td>SUNMatScaleAddI</td>
<td>ier = SUNMatScaleAddI(c, A); Performs the operation $A = cA + I$. The return value is an integer flag denoting success/failure of the operation.</td>
</tr>
<tr>
<td>SUNMatMatvec</td>
<td>ier = SUNMatMatvec(A, x, y); Performs the matrix-vector product operation, $y = Ax$. It should only be called with vectors x and y that are compatible with the matrix A – both in storage type and dimensions. The return value is an integer flag denoting success/failure of the operation.</td>
</tr>
</tbody>
</table>

We note that not all SUNMATRIX types are compatible with all NVECTOR types provided with SUNDIALS. This is primarily due to the need for compatibility within the SUNMatMatvec routine; however, compatibility between SUNMATRIX and NVECTOR implementations is more crucial when considering their interaction within SUNLINSOL objects, as will be described in more detail in Chapter 9. More specifically, in Table 8.3 we show the matrix interfaces available as SUNMATRIX modules, and the compatible vector implementations.

Table 8.3: 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>
</tbody>
</table>

continued on next page
8.1 SUNMatrix functions used by CVODES

In Table 8.4, we list the matrix functions in the SUNMATRIX module used within the CVODES package. The table also shows, for each function, which of the code modules uses the function. The main CVODES 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 CVODES user does not need to know anything about the usage of matrix functions by the CVODES code modules in order to use CVODES. The information is presented as an implementation detail for the interested reader.

8.2 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;
};
```
8.2 The SUNMatrix_Dense implementation

sunindextype ldata;
realtype **cols;
};

These entries of the `content` field contain the following information:

- `M` - number of rows
- `N` - number of columns
- `data` - pointer to a contiguous block of `realtype` variables. The elements of the dense matrix are stored columnwise, i.e. the \((i,j)\)-th element of a dense SUNMATRIX \(A\) (with \(0 \leq i < M\) and \(0 \leq j < N\)) may be accessed via `data[j*M+i]`.
- `ldata` - length of the data array (= \(M \cdot N\)).
- `cols` - array of pointers. `cols[j]` points to the first element of the \(j\)-th column of the matrix in the array `data`. The \((i,j)\)-th element of a dense SUNMATRIX \(A\) (with \(0 \leq i < M\) and \(0 \leq j < N\)) may be accessed via `cols[j][i]`.

The header file to include when using this module is `sunmatrix/sunmatrix_dense.h`. The SUNMATRIX_DENSE module is accessible from all SUNDIALS solvers without linking to the `libsundials_sunmatrixdense` module library.

8.2.1 SUNMatrix_Dense accessor macros

The following macros are provided to access the content of a SUNMATRIX_DENSE matrix. The prefix `SM` in the names denotes that these macros are for SUNMatrix implementations, and the suffix `_D` denotes that these are specific to the dense version.

- **SM_CONTENT_D**
  This macro gives access to the contents of the dense SUNMatrix.
  The assignment `A_cont = SM_CONTENT_D(A)` sets `A_cont` to be a pointer to the dense SUNMatrix content structure.
  Implementation:
  ```
  #define SM_CONTENT_D(A) ( (SUNMatrixContent_Dense)(A->content) )
  ```

- **SM_ROWS_D, SM_COLUMNS_D, and SM_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_rows = SM_ROWS_D(A)` sets `A_rows` to be the number of rows in the matrix \(A\). Similarly, the assignment `SM_COLUMNS_D(A) = A_cols` sets the number of columns in \(A\) to equal \(A_cols\).
  Implementation:
  ```
  #define SM_ROWS_D(A) ( SM_CONTENT_D(A)->M )
  #define SM_COLUMNS_D(A) ( SM_CONTENT_D(A)->N )
  #define SM_LDATA_D(A) ( SM_CONTENT_D(A)->ldata )
  ```

- **SM_DATA_D and SM_COLS_D**
  These macros give access to the `data` and `cols` pointers for the matrix entries.
  The assignment `A_data = SM_DATA_D(A)` sets `A_data` to be a pointer to the first component of the data array for the dense SUNMatrix \(A\). The assignment `SM_DATA_D(A) = A_data` sets the data array of \(A\) to be `A_data` by storing the pointer `A_data`.
  Similarly, the assignment `A_cols = SM_COLS_D(A)` sets `A_cols` to be a pointer to the array of column pointers for the dense SUNMatrix \(A\). The assignment `SM_COLS_D(A) = A_cols` sets the column pointer array of \(A\) to be `A_cols` by storing the pointer `A_cols`.
Implementation:
#define SM_DATA_D(A) (SM_CONTENT_D(A)->data)
#define SM_COLS_D(A) (SM_CONTENT_D(A)->cols)

- SM_COLUMN_D and SM_ELEMENT_D

These macros give access to the individual columns and entries of the data array of a dense SUNMatrix.

The assignment \( \text{col}_j = \text{SM_COLUMN}_D(A,j) \) 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:
#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] )

8.2.2 SUNMatrix_Dense functions

The \text{sunmatrix_dense} module defines dense implementations of all matrix operations listed in Table 8.2. Their names are obtained from those in Table 8.2 by appending the suffix \_Dense (e.g. \text{SUNMatCopy_Dense}). All the standard matrix operations listed in 8.2 with the suffix \_Dense appended are callable via the \text{FORTRAN 2003} interface by prepending an `'F'` (e.g. \text{FSUNMatCopy_Dense}).

The module \text{sunmatrix_dense} provides the following additional user-callable routines:

- \text{SUNDenseMatrix}Prototype SUNMatrix SUNDenseMatrix(sunindextype M, sunindextype N)

Description This constructor function creates and allocates memory for a dense SUNMatrix. Its arguments are the number of rows, \( M \), and columns, \( N \), for the dense matrix.

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

- \text{SUNDenseMatrix_Print}Prototype void SUNDenseMatrix_Print(SUNMatrix A, FILE* outfile)

Description This function prints the content of a dense SUNMatrix to the output stream specified by \text{outfile}. Note: \text{stdout} or \text{stderr} may be used as arguments for \text{outfile} to print directly to standard output or standard error, respectively.

- \text{SUNDenseMatrix_Rows}Prototype sunindextype SUNDenseMatrix_Rows(SUNMatrix A)

Description This function returns the number of rows in the dense SUNMatrix.

F2003 Name This function is callable as \text{FSUNDenseMatrix_Rows} when using the Fortran 2003 interface module.
8.2 The SUNMatrix_Dense implementation

**SUNDenseMatrix_Columns**

Prototype: `sunindextype SUNDenseMatrix_Columns(SUNMatrix A)`
Description: This function returns the number of columns in the dense SUNMatrix.
F2003 Name: This function is callable as `FSUNDenseMatrix_Columns` when using the Fortran 2003 interface module.

**SUNDenseMatrix_LData**

Prototype: `sunindextype SUNDenseMatrix_LData(SUNMatrix A)`
Description: This function returns the length of the data array for the dense SUNMatrix.
F2003 Name: This function is callable as `FSUNDenseMatrix_LData` when using the Fortran 2003 interface module.

**SUNDenseMatrix_Data**

Prototype: `realttype* 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: `realttype** SUNDenseMatrix_Cols(SUNMatrix A)`
Description: This function returns a pointer to the cols array for the dense SUNMatrix.

**SUNDenseMatrix_Column**

Prototype: `realttype* SUNDenseMatrix_Column(SUNMatrix A, sunindextype j)`
Description: This function returns a pointer to the first entry of the jth column of the dense SUNMatrix. The resulting pointer should be indexed over the range 0 to M - 1.
F2003 Name: This function is callable as `FSUNDenseMatrix_Column` when using the Fortran 2003 interface module.

**Notes**

- When looping over the components of a dense SUNMatrix A, the most efficient approaches are to:
  - First obtain the component array via `A.data = SM_DATA_D(A)` or `A.data = SUNDenseMatrix_Data(A)` and then access `A.data[i]` within the loop.
  - First obtain the array of column pointers via `A.cols = SM_COLS_D(A)` or `A.cols = SUNDenseMatrix_Cols(A)`, and then access `A.cols[j][i]` within the loop.
  - Within a loop over the columns, access the column pointer via `A.colj = SUNDenseMatrix_Column(A,j)` and then to access the entries within that column using `A.colj[i]` within the loop.

  All three of these are more efficient than using `SM_ELEMENT_D(A,i,j)` within a double loop.

- Within the SUNMatMatvec_Dense routine, internal consistency checks are performed to ensure that the matrix is called with consistent NVECTOR implementations. These are currently limited to: NVECTOR_SERIAL, NVECTOR_OPENMP, and NVECTOR_PTHREADS. As additional compatible vector implementations are added to SUNDIALS, these will be included within this compatibility check.
8.2.3 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 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.

8.3 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 8.1. 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
8.3 The SUNMatrix Band implementation

\( \text{s\_mu} \) - storage upper bandwidth, \( \mu \leq s\_mu < N \). The LU decomposition routines in the associated SUNLINSOL_BAND and SUNLINSOL_LAPACK_BAND modules write the LU factors into the storage for \( A \). The upper triangular factor \( U \), however, may have an upper bandwidth as big as \( \min(N-1, \mu+ml) \) because of partial pivoting. The \( s\_mu \) field holds the upper half-bandwidth allocated for \( A \).

\( \text{ldim} \) - leading dimension (\( \text{ldim} \geq s\_mu + ml + 1 \))

\( \text{data} \) - pointer to a contiguous block of \text{realttype} 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. \( \text{data} \) is a pointer to \( \text{ldata} \) contiguous locations which hold the elements within the band of \( A \).

\( \text{ldata} \) - length of the data array (= \( \text{ldim} \cdot N \))

\( \text{cols} \) - array of pointers. \( \text{cols}[j] \) is a pointer to the uppermost element within the band in the \( j \)-th column. This pointer may be treated as an array indexed from \( s\_mu - \mu \) to \( s\_mu + ml \) (to access the lowest element within the band in the \( j \)-th column). Indices from 0 to \( s\_mu - \mu - 1 \) give access to extra storage elements required by the LU decomposition function. Finally, \( \text{cols}[j][i-j+s\_mu] \) is the \((i,j)\)-th element with \( j-\mu \leq i \leq j+ml \).

The header file to include when using this module is \text{sunmatrix/sunmatrix\_band.h}. The \text{SUNMatrix\_Band} module is accessible from all \text{SUNDIALS} solvers \emph{without} linking to the \text{libsundials/sunmatrix\_band} module library.

8.3.1 SUNMatrix Band accessor macros

The following macros are provided to access the content of a SUNMATRIX_BAND matrix. The prefix \( \text{SM} \) in the names denotes that these macros are for \text{SUNMatrix} implementations, and the suffix \_B denotes that these are specific to the \emph{banded} version.

- \text{SM\_CONTENT\_B}

  This routine gives access to the contents of the banded \text{SUNMatrix}.

  The assignment \( A\_\text{cont} = \text{SM\_CONTENT\_B}(A) \) sets \( A\_\text{cont} \) to be a pointer to the banded \text{SUNMatrix} content structure.

  Implementation:

  \[
  \text{#define SM\_CONTENT\_B}(A) \quad \left( \text{SUNMatrixContent\_Band}(A\rightarrow\text{content}) \right)
  \]

- \text{SM\_ROWS\_B}, \text{SM\_COLUMNS\_B}, \text{SM\_UBAND\_B}, \text{SM\_LBAND\_B}, \text{SM\_SUBAND\_B}, \text{SM\_LDIM\_B}, and \text{SM\_LDATA\_B}

  These macros give individual access to various lengths relevant to the content of a banded \text{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:

  \[
  \text{#define SM\_ROWS\_B}(A) \quad \left( \text{SM\_CONTENT\_B}(A)\rightarrow M \right)
  \]

  \[
  \text{#define SM\_COLUMNS\_B}(A) \quad \left( \text{SM\_CONTENT\_B}(A)\rightarrow N \right)
  \]

  \[
  \text{#define SM\_UBAND\_B}(A) \quad \left( \text{SM\_CONTENT\_B}(A)\rightarrow \mu \right)
  \]

  \[
  \text{#define SM\_LBAND\_B}(A) \quad \left( \text{SM\_CONTENT\_B}(A)\rightarrow ml \right)
  \]

  \[
  \text{#define SM\_SUBAND\_B}(A) \quad \left( \text{SM\_CONTENT\_B}(A)\rightarrow s\_mu \right)
  \]

  \[
  \text{#define SM\_LDIM\_B}(A) \quad \left( \text{SM\_CONTENT\_B}(A)\rightarrow \text{ldim} \right)
  \]

  \[
  \text{#define SM\_LDATA\_B}(A) \quad \left( \text{SM\_CONTENT\_B}(A)\rightarrow \text{ldata} \right)
  \]
Figure 8.1: 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.
8.3 The SUNMatrix Band implementation

- **SM_DATA_B** and **SM_COLS_B**
  These macros give access to the `data` and `cols` pointers for the matrix entries.
  
  The assignment `A_data = SM_DATA_B(A)` sets `A_data` to be a pointer to the first component of the data array for the banded SUNMatrix `A`. The assignment `SM_DATA_B(A) = A_data` sets the data array of `A` to be `A_data` by storing the pointer `A_data`.
  
  Similarly, the assignment `A_cols = SM_COLS_B(A)` sets `A_cols` to be a pointer to the array of column pointers for the banded SUNMatrix `A`. The assignment `SM_COLS_B(A) = A_cols` sets the column pointer array of `A` to be `A_cols` by storing the pointer `A_cols`.

  Implementation:
  ```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 `SM_ELEMENT_B(A,i,j) = a_{ij}` and `a_{ij} = SM_ELEMENT_B(A,i,j)` reference the \((i,j)\)-th element of the \(N \times N\) band matrix `A`, where \(0 \leq i, j \leq N - 1\). The location \((i,j)\) should further satisfy \(j - mu \leq i \leq j + ml\).
  
  The assignment `col_j = SM_COLUMN_B(A,j)` sets `col_j` to be a pointer to the diagonal element of the \(j\)-th column of the \(N \times N\) 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 \(ml\).
  
  The assignments `SM_COLUMN_ELEMENT_B(col_j,i,j) = a_{ij}` and `a_{ij} = SM_COLUMN_ELEMENT_B(col_j,i,j)` reference the \((i,j)\)-th entry of the band matrix `A` when used in conjunction with `SM_COLUMN_B` to reference the \(j\)-th column through `col_j`. The index \((i,j)\) should satisfy \(j - mu \leq i \leq j + ml\).

  Implementation:
  ```c
  #define SM_COLUMN_B(A,j) ( ((SM_CONTENT_B(A)->cols)[j])+SM_SUBAND_B(A) )
  #define SM_COLUMN_ELEMENT_B(col_j,i,j) (col_j[(i)-(j)])
  #define SM_ELEMENT_B(A,i,j) ( (SM_CONTENT_B(A)->cols)[j][(i)-(j)]+SM_SUBAND_B(A))
  ```

8.3.2 SUNMatrix Band functions

The sunmatrix_band module defines banded implementations of all matrix operations listed in Table 8.2. Their names are obtained from those in Table 8.2 by appending the suffix _Band (e.g. SUNMatCopy_Band). All the standard matrix operations listed in 8.2 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:

```c
SUNBandMatrix
Prototype SUNMatrix SUNBandMatrix(sunindextype N, sunindextype mu, sunindextype ml)
Description This constructor function creates and allocates memory for a banded SUNMatrix. Its arguments are the matrix size, `N`, and the upper and lower half-bandwidths of the matrix, `mu` and `ml`. The stored upper bandwidth is set to `mu+ml` to accommodate subsequent factorization in the sunlinsol_band and sunlinsol_lapackband modules.
F2003 Name This function is callable as FSUNBandMatrix when using the Fortran 2003 interface module.
```
Description of the SUNMatrix module

**SUNBandMatrixStorage**

Prototype: `SUNMatrix SUNBandMatrixStorage(sunindextype N, sunindextype mu, 
sunindextype ml, sunindextype smu)`

Description: This constructor function creates and allocates memory for a banded SUNMatrix. Its arguments are the matrix size, N, the upper and lower half-bandwidths of the matrix, mu and ml, and the stored upper bandwidth, smu. When creating a band SUNMatrix, this value should be

- at least min(N-1, mu+ml) if the matrix will be used by the SUNLINSOL_BAND module;
- exactly equal to mu+ml if the matrix will be used by the SUNLINSOL_LAPACKBAND module;
- at least mu if used in some other manner.

*Note: it is strongly recommended that users call the default constructor, SUNBandMatrix, in all standard use cases. This advanced constructor is used internally within SUNDIALS solvers, and is provided to users who require banded matrices for non-default purposes.*

**SUNBandMatrix_Print**

Prototype: `void SUNBandMatrix_Print(SUNMatrix A, FILE* outfile)`

Description: This function prints the content of a banded SUNMatrix to the output stream specified by outfile. Note: stdout or stderr may be used as arguments for outfile to print directly to standard output or standard error, respectively.

**SUNBandMatrix_Rows**

Prototype: `sunindextype SUNBandMatrix_Rows(SUNMatrix A)`

Description: This function returns the number of rows in the banded SUNMatrix.

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

**SUNBandMatrix_Columns**

Prototype: `sunindextype SUNBandMatrix_Columns(SUNMatrix A)`

Description: This function returns the number of columns in the banded SUNMatrix.

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

**SUNBandMatrix_LowerBandwidth**

Prototype: `sunindextype SUNBandMatrix_LowerBandwidth(SUNMatrix A)`

Description: This function returns the lower half-bandwidth of the banded SUNMatrix.

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

**SUNBandMatrix_UpperBandwidth**

Prototype: `sunindextype SUNBandMatrix_UpperBandwidth(SUNMatrix A)`

Description: This function returns the upper half-bandwidth of the banded SUNMatrix.

F2003 Name: This function is callable as FSUNBandMatrix_UpperBandwidth when using the Fortran 2003 interface module.
8.3 The SUNMatrix_Band implementation

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: realtime* 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: realtime** SUNBandMatrix_Cols(SUNMatrix A)
Description: This function returns a pointer to the cols array for the banded SUNMatrix.

SUNBandMatrix_Column

Prototype: realtime* 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 $\mu$.
F2003 Name: This function is callable as FSUNBandMatrix_Column when using the Fortran 2003 interface module.

Notes

- When looping over the components of a banded SUNMatrix $A$, the most efficient approaches are to:
  - First obtain the component array via $A\_data = SM\_DATA\_B(A)$ or $A\_data = SUNBandMatrix\_Data(A)$ and then access $A\_data[i]$ within the loop.
  - First obtain the array of column pointers via $A\_cols = SM\_COLS\_B(A)$ or $A\_cols = SUNBandMatrix\_Cols(A)$, and then access $A\_cols[j][i]$ within the loop.
  - Within a loop over the columns, access the column pointer via $A\_colj = SUNBandMatrix\_Column(A,j)$ and then to access the entries within that column using $SM\_COLUMN\_ELEMENT\_B(A\_colj,i,j)$.

All three of these are more efficient than using $SM\_ELEMENT\_B(A,i,j)$ within a double loop.

- Within the SUNMatMatvec_Band routine, internal consistency checks are performed to ensure that the matrix is called with consistent NVECTOR implementations. These are currently limited to NVECTOR_SERIAL, NVECTOR_OPENMP, and NVECTOR_PTHREADS. As additional compatible vector implementations are added to SUNDIALS, these will be included within this compatibility check.
8.3.3 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.

8.4 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 8.2 (the CSR format is similar). A more complete description of the parts of this content field is given below:
8.4 The SUNMatrix_Sparse implementation

- number of rows
- number of columns
- maximum number of nonzero entries in the matrix (allocated length of data and indexvals arrays)
- 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.
- pointer to a contiguous block of realtype variables (of length NNZ), containing the values of the nonzero entries in the matrix
- type of the sparse matrix (CSC_MAT or CSR_MAT)
- 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
- pointer to a contiguous block of int variables (of length NP+1). For CSC matrices each entry provides the index of the first column entry into the data and indexvals arrays, e.g. if indexptr[3]=7, then the first nonzero entry in the fourth column of the matrix is located in data[7], and is located in row indexvals[7] of the matrix. The last entry contains the total number of nonzero values in the matrix and hence points one past the end of the active data in the data and indexvals arrays. For CSR matrices, each entry provides the index of the first row entry into the data and indexvals arrays.

The following pointers are added to the SlsMat type for user convenience, to provide a more intuitive interface to the CSC and CSR sparse matrix data structures. They are set automatically when creating a sparse SUNMATRIX, based on the sparse matrix storage type.
- pointer to indexvals when sparsetype is CSC_MAT, otherwise set to NULL.
- pointer to indexptrs when sparsetype is CSC_MAT, otherwise set to NULL.
- pointer to indexvals when sparsetype is CSR_MAT, otherwise set to NULL.
- pointer to indexptrs when sparsetype is CSR_MAT, otherwise set to NULL.

For example, the 5 × 4 CSC matrix

\[
\begin{bmatrix}
0 & 3 & 1 & 0 \\
3 & 0 & 0 & 2 \\
0 & 7 & 0 & 0 \\
1 & 0 & 0 & 9 \\
0 & 0 & 0 & 5
\end{bmatrix}
\]

could be stored in this structure as either

```c
M = 5;
N = 4;
NNZ = 8;
NP = N;
data = {3.0, 1.0, 3.0, 7.0, 1.0, 2.0, 9.0, 5.0};
sparsetype = CSC_MAT;
indexvals = {1, 3, 0, 2, 0, 1, 3, 4};
indexptrs = {0, 2, 4, 5, 8};
```
or

```c
M = 5;
N = 4;
NNZ = 10;
NP = N;
data = {3.0, 1.0, 3.0, 7.0, 1.0, 2.0, 9.0, 5.0, *, *};
sparsetype = CSC_MAT;
indexvals = {1, 3, 0, 2, 0, 1, 3, 4, *, *};
indexptrs = {0, 2, 4, 5, 8};
```
where the first has no unused space, and the second has additional storage (the entries marked with * may contain any values). Note in both cases that the final value in indexptrs is 8, indicating the total number of nonzero entries in the matrix.

Similarly, in CSR format, the same matrix could be stored as

```c
M = 5;
N = 4;
NNZ = 8;
NP = N;
data = {3.0, 1.0, 3.0, 2.0, 7.0, 1.0, 9.0, 5.0};
sparsetype = CSR_MAT;
indexvals = {1, 2, 0, 3, 1, 0, 3, 3};
indexptrs = {0, 2, 4, 5, 7, 8};
```

The header file to include when using this module is `sunmatrix/sunmatrix_sparse.h`. The SUNMATRIX_SPARSE module is accessible from all SUNDIALS solvers without linking to the `libsundials_sunmatrix_sparse` module library.

### 8.4.1 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:
  ```c
  #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:
  ```c
  #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
8.4 The SUNMatrix_Sparse implementation

Figure 8.2: 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 data and 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.
matrix) for the sparse SUNMatrix A. The assignment \( A_{\text{indexptrs}} = \text{SM\_INDEXPTRS\_S}(A) \) sets \( A_{\text{indexptrs}} \) 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:

```c
#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 )
```

### 8.4.2 SUNMatrix_Sparse functions

The SUNMATRIX\_SPARSE module defines sparse implementations of all matrix operations listed in Table 8.2. Their names are obtained from those in Table 8.2 by appending the suffix \_Sparse (e.g. SUNMatCopy\_Sparse). All the standard matrix operations listed in 8.2 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**

```c
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 \( \text{sparsetype} \) 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**

```c
SUNMatrix SUNSparseFromDenseMatrix(SUNMatrix A, realtype droptol, int sparsetype);
```

**Description**

This function creates a new sparse matrix from an existing dense matrix by copying all values with magnitude larger than \( \text{droptol} \) into the sparse matrix structure.

**Requirements:**

- \( A \) must have type SUNMATRIX\_DENSE;
- \( \text{droptol} \) must be non-negative;
- \( \text{sparsetype} \) must be either CSC\_MAT or CSR\_MAT.

The function returns NULL if any requirements are violated, or if the matrix storage request cannot be satisfied.

**F2003 Name**

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

#### SUNSparseFromBandMatrix

**Prototype**

```c
SUNMatrix SUNSparseFromBandMatrix(SUNMatrix A, realtype droptol, int sparsetype);
```

**Description**

This function creates a new sparse matrix from an existing band matrix by copying all values with magnitude larger than \( \text{droptol} \) into the sparse matrix structure.

**Requirements:**

- \( A \) must have type SUNMATRIX\_BAND;
- \( \text{droptol} \) must be non-negative;
- \( \text{sparsetype} \) 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.
8.4 The SUNMatrix_Sparse implementation

- A must have type SUNMATRIX_BAND;
- droptol must be non-negative;
- sparsetype must be either CSC_MAT or CSR_MAT.

The function returns NULL if any requirements are violated, or if the matrix storage request cannot be satisfied.

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

---

SUNSparseMatrix_Realloc

Prototype 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 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 void SUNSparseMatrix_Print(SUNMatrix A, FILE* outfile)

Description This function prints the content of a sparse SUNMatrix to the output stream specified by outfile. Note: stdout or stderr may be used as arguments for outfile to print directly to standard output or standard error, respectively.

---

SUNSparseMatrix_Rows

Prototype sunindextype SUNSparseMatrix_Rows(SUNMatrix A)

Description This function returns the number of rows in the sparse SUNMatrix.

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

---

SUNSparseMatrix_Columns

Prototype sunindextype SUNSparseMatrix_Columns(SUNMatrix A)

Description This function returns the number of columns in the sparse SUNMatrix.

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

Prototype  
sunindextype SUNSparseMatrix_NNZ(SUNMatrix A)

Description This function returns the number of entries allocated for nonzero storage for the sparse matrix SUNMatrix.

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

SUNSparseMatrix_NP

Prototype  
sunindextype SUNSparseMatrix_NP(SUNMatrix A)

Description This function returns the number of columns/rows for the sparse SUNMatrix, depending on whether the matrix uses CSC/CSR format, respectively. The indexptrs array has NP+1 entries.

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

SUNSparseMatrix_SparseType

Prototype  
int SUNSparseMatrix_SparseType(SUNMatrix A)

Description This function returns the storage type (CSR_MAT or CSC_MAT) for the sparse SUNMatrix.

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

SUNSparseMatrix_Data

Prototype  
realtype* SUNSparseMatrix_Data(SUNMatrix A)

Description This function returns a pointer to the data array for the sparse SUNMatrix.

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

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 NVVECTOR implementations. These are currently limited to: NVVECTOR_SERIAL, NVVECTOR_OPENMP, and NVVECTOR_PTHREADS. As additional compatible vector implementations are added to SUNDIALS, these will be included within this compatibility check.
8.4 The SUNMatrix_Sparse implementation

8.4.3 SUNMatrix_Sparse Fortran interfaces

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

FORTRAN 2003 interface module

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

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

FORTRAN 77 interface functions

For solvers that include a Fortran interface module, the SUNMATRIX_SPARSE module also includes the Fortran-callable function FSUNSparseMatInit(code, M, N, NNZ, sparsetype, ier) to initialize this sunmatrix_sparse module for a given SUNDIALS solver. Here code is an integer input for the solver id (1 for cvode, 2 for ida, 3 for kinsol, 4 for arkode); M, N and NNZ are the corresponding sparse matrix construction arguments (declared to match C type long int); sparsetype is an integer flag indicating the sparse storage type (0 for CSC, 1 for CSR); and ier is an error return flag equal to 0 for success and -1 for failure. Each of code, sparsetype and ier are declared so as to match C type int. Additionally, when using ARKODE with a non-identity mass matrix, the Fortran-callable function FSUNSparseMassMatInit(M, N, NNZ, sparsetype, ier) initializes this SUNMATRIX_SPARSE module for storing the mass matrix.
Chapter 9

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

\[
\tilde{A}\tilde{x} = \tilde{b}
\]

where

\[
\tilde{A} = S_1P_1^{-1}AP_2^{-1}S_2^{-1},
\]
\[
\tilde{b} = S_1P_1^{-1}b,
\]
\[
\tilde{x} = S_2P_2x,
\]

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 $S_1P_1^{-1}b$ and $S_2P_2x$ have dimensionless components. If preconditioning is done on the left only ($P_2 = I$), by a matrix $P$, then $S_2$ must be a scaling for $x$, while $S_1$ is a scaling for $P^{-1}b$, and so may also be taken as a scaling for $x$. Similarly, if preconditioning is done on the right only ($P_1 = I$ and $P_2 = P$), then $S_1$ must be a scaling for $b$, while $S_2$ is a scaling for $Px$, and may also be taken as a scaling for $b$.

SUNDIALS packages request that iterative linear solvers stop based on the 2-norm of the scaled preconditioned residual meeting a prescribed tolerance

$$\|\tilde{b} - \tilde{A}\tilde{x}\|_2 < \text{tol}.$$  

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 §9.4.2 for more details). In this case, they instead request that iterative linear solvers stop based on the criteria

$$\|P_1^{-1}b - P_1^{-1}Ax\|_2 < \text{tol}.$$  

We note that the corresponding adjustments to tol in this case are non-optimal, in that they cannot balance error between specific entries of the solution $x$, only the aggregate error in the overall solution vector.

We further note that not all of the SUNDIALS-provided iterative linear solvers support the full range of the above options (e.g., separate left/right preconditioning), and that some of the SUNDIALS packages only utilize a subset of these options. Further details on these exceptions are described in the documentation for each SUNLINSOL implementation, or for each SUNDIALS package.

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 9.1.1 – 9.1.3. This is followed by the definition of functions supplied to a linear solver implementation in section 9.1.4. A table of linear solver return codes is given in section 9.1.5. The SUNLinearSolver type and the generic SUNLINSOL module are defined in section 9.1.6. The section 9.2 discusses compatibility between the SUNDIALS-provided SUNLINSOL modules and SUNMATRIX modules. Section 9.3 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 9.4. The remaining sections of this chapter present the SUNLINSOL modules provided with SUNDIALS.

9.1 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.

9.1.1 SUNLinearSolver core functions

The core linear solver functions consist of four required routines to get the linear solver type (SUNLinSolGetType), initialize the linear solver object once all solver-specific options have been set (SUNLinSolInitialize), set up the linear solver object to utilize an updated matrix $A$ (SUNLinSolSetup), and solve the linear system $Ax = b$ (SUNLinSolSolve). The remaining routine for destruction of the linear solver object (SUNLinSolFree) is optional.
9.1 The SUNLinearSolver API

**SUNLinSolGetType**

Call       type = SUNLinSolGetType(LS);

Description The required function SUNLinSolGetType returns the type identifier for the linear solver LS. It is used to determine the solver type (direct, iterative, or matrix-iterative) from the abstract SUNLinearSolver interface.

Arguments   LS (SUNLinearSolver) a SUNLINSOL object.

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

- **SUNLINEARSOLVER DIRECT** − 0, the SUNLINSOL module requires a matrix, and computes an ‘exact’ solution to the linear system defined by that matrix.

- **SUNLINEARSOLVER ITERATIVE** − 1, the SUNLINSOL module does not require a matrix (though one may be provided), and computes an inexact solution to the linear system using a matrix-free iterative algorithm. That is it solves the linear system defined by the package-supplied ATimes routine (see SUNLinSolSetATimes below), even if that linear system differs from the one encoded in the matrix object (if one is provided). As the solver computes the solution only 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 9.3.1 for more information on intended use cases corresponding to the linear solver type.

**SUNLinSolInitialize**

Call       retval = SUNLinSolInitialize(LS);

Description The required function SUNLinSolInitialize performs linear solver initialization (assuming that all solver-specific options have been set).

Arguments   LS (SUNLinearSolver) a SUNLINSOL object.

Return value This should return zero for a successful call, and a negative value for a failure, ideally returning one of the generic error codes listed in Table 9.1.

**SUNLinSolSetup**

Call       retval = SUNLinSolSetup(LS, A);

Description The required function SUNLinSolSetup performs any linear solver setup needed, based on an updated system SUNMATRIX A. This may be called frequently (e.g., with a full Newton method) or infrequently (for a modified Newton method), based on the type of integrator and/or nonlinear solver requesting the solves.

Arguments   LS (SUNLinearSolver) a SUNLINSOL object.

A (SUNMatrix) a SUNMATRIX object.

Return value This should return zero for a successful call, a positive value for a recoverable failure and a negative value for an unrecoverable failure, ideally returning one of the generic error codes listed in Table 9.1.
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 9.1.

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 $\text{tol}$ 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.

9.1.2 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 9.1.

**SUNLinSolSetPreconditioner**

Call \( \text{retval} = \text{SUNLinSolSetPreconditioner}(\text{LS}, \text{Pdata}, \text{Pset}, \text{Psol}); \)

Description The *optional* function \( \text{SUNLinSolSetPreconditioner} \) provides \( \text{PSetupFn} \) and \( \text{PSolveFn} \) function pointers that implement the preconditioner solves \( P_1^{-1} \) and \( P_2^{-1} \) from equations (9.1)-(9.2). This routine will be called by a SUNDIALS package, which will provide translation between the generic \( \text{Pset} \) and \( \text{Psol} \) calls and the package- or user-supplied routines.

Arguments \( \text{LS} \) (SUNLinearSolver) a SUNLINSOL object.
\( \text{Pdata} \) (void*) data structure passed to both \( \text{Pset} \) and \( \text{Psol} \).
\( \text{Pset} \) (PSetupFn) function pointer implementing the preconditioner setup.
\( \text{Psol} \) (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 9.1.

**SUNLinSolSetScalingVectors**

Call \( \text{retval} = \text{SUNLinSolSetScalingVectors}(\text{LS}, s1, s2); \)

Description The *optional* function \( \text{SUNLinSolSetScalingVectors} \) provides left/right scaling vectors for the linear system solve. Here, \( s1 \) and \( s2 \) are NVECTOR of positive scale factors containing the diagonal of the matrices \( S_1 \) and \( S_2 \) from equations (9.1)-(9.2), 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 \( \text{LS} \) (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 9.1.

### 9.1.3 SUNLinearSolver get functions

The following get functions allow SUNDIALS packages to retrieve results from a linear solve. All routines are optional.

**SUNLinSolNumIters**

Call \( \text{its} = \text{SUNLinSolNumIters}(\text{LS}); \)

Description The *optional* function \( \text{SUNLinSolNumIters} \) should return the number of linear iterations performed in the last ‘solve’ call.

Arguments \( \text{LS} \) (SUNLinearSolver) a SUNLINSOL object.

Return value int containing the number of iterations

**SUNLinSolResNorm**

Call \( \text{rnorm} = \text{SUNLinSolResNorm}(\text{LS}); \)

Description The *optional* function \( \text{SUNLinSolResNorm} \) should return the final residual norm from the last ‘solve’ call.

Arguments \( \text{LS} \) (SUNLinearSolver) a SUNLINSOL object.

Return value realtype containing the final residual norm
Description of the SUNLinearSolver module

**SUNLinSolResid**

**Call**

```c
rvec = SUNLinSolResid(LS);
```

**Description**

If an iterative method computes the preconditioned initial residual and returns with a successful solve without performing any iterations (i.e., either the initial guess or the preconditioner is sufficiently accurate), then this *optional* routine may be called by the SUNDIALS package. This routine should return the NVECTOR containing the preconditioned initial residual vector.

**Arguments**

- `LS` (SUNLinearSolver) a SUNLINSOL object.

**Return value**

N_Vector containing the final residual vector

**Notes**

Since N_Vector is actually a pointer, and the results are not modified, this routine should *not* require additional memory allocation. If the SUNLINSOL object does not retain a vector for this purpose, then this function pointer should be set to NULL in the implementation.

**SUNLinSolLastFlag**

**Call**

```c
lflag = SUNLinSolLastFlag(LS);
```

**Description**

The *optional* function SUNLinSolLastFlag should return the last error flag encountered within the linear solver. This is not called by the SUNDIALS packages directly; it allows the user to investigate linear solver issues after a failed solve.

**Arguments**

- `LS` (SUNLinearSolver) a SUNLINSOL object.

**Return value**

long int containing the most recent error flag

**SUNLinSolSpace**

**Call**

```c
retval = SUNLinSolSpace(LS, &lrw, &liw);
```

**Description**

The *optional* function SUNLinSolSpace should return the storage requirements for the linear solver LS.

**Arguments**

- `LS` (SUNLinearSolver) a SUNLINSOL object.
- `lrw` (long int*) the number of realtype words stored by the linear solver.
- `liw` (long int*) the number of integer words stored by the linear solver.

**Return value**

This should return zero for a successful call, and a negative value for a failure, ideally returning one of the generic error codes listed in Table 9.1.

**Notes**

This function is advisory only, for use in determining a user’s total space requirements.

### 9.1.4 Functions provided by SUNDIALS packages

To interface with the SUNLINSOL modules, the SUNDIALS packages supply a variety of routines for evaluating the matrix-vector product, and setting up and applying the preconditioner. These package-provided routines translate between the user-supplied ODE, DAE, or nonlinear systems and the generic interfaces to the linear systems of equations that result in their solution. The types for functions provided to a SUNLINSOL module are defined in the header file `sundials/sundials_iterative.h`, and are described below.

**ATimesFn**

**Definition**

```c
typedef int (*ATimesFn)(void *A_data, N_Vector v, N_Vector z);
```

**Purpose**

These functions compute the action of a matrix on a vector, performing the operation \( z = Av \). Memory for \( z \) should already be allocated prior to calling this function. The vector \( v \) should be left unchanged.
9.1 The SUNLinearSolver API 247

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.

\[ \texttt{PSetupFn} \]

Definition

\[
\text{typedef int (}\ast\text{PSetupFn})(\text{void } \ast\text{P}\text{\_data})
\]

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.

\[ \texttt{PSolveFn} \]

Definition

\[
\text{typedef int (}\ast\text{PSolveFn})(\text{void } \ast\text{P}\text{\_data}, \text{N\_Vector r, N\_Vector z, realtype tol, int lr})
\]

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.

9.1.5 SUNLinearSolver return codes

The functions provided to SUNLINSOL modules by each SUNDIALS package, and functions within the SUNDIALS-provided SUNLINSOL implementations utilize a common set of return codes, shown in Table 9.1. 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.
### 9.1.6 The generic SUNLinearSolver module

SUNDIALS packages interact with specific SUNLINSOL implementations through the generic SUNLINSOL module on which all other SUNLINSOL implementations are built. The SUNLinearSolver type is a pointer to a structure containing an implementation-dependent content field, and an ops field. The type SUNLinearSolver is defined as

```c
typedef struct _generic_SUNLinearSolver *SUNLinearSolver;

struct _generic_SUNLinearSolver {
    void *content;
    struct _generic_SUNLinearSolver_Ops *ops;
};
```

where the _generic_SUNLinearSolver_Ops structure is a list of pointers to the various actual linear solver operations provided by a specific implementation. The _generic_SUNLinearSolver_Ops structure is defined as

```c
struct _generic_SUNLinearSolver_Ops {
    SUNLinearSolver_Type (*gettype)(SUNLinearSolver);
};
```
9.2 Compatibility of SUNLinearSolver modules

The generic SUNLINSOL module defines and implements the linear solver operations defined in Sections 9.1.1-9.1.3. 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));
}
```

9.2 Compatibility of SUNLinearSolver modules

We note that not all SUNLINSOL types are compatible with all SUNMATRIX and NVECTOR types provided with SUNDIALS. In Table 9.2 we show the matrix-based linear solvers available as SUNLINSOL modules, and the compatible matrix implementations. Recall that Table 4.1 shows the compatibility between all SUNLINSOL modules and vector implementations.

<table>
<thead>
<tr>
<th>Linear Solver Interface</th>
<th>Dense Matrix</th>
<th>Banded Matrix</th>
<th>Sparse Matrix</th>
<th>User Supplied</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dense</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Band</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>LapackDense</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>LapackBand</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>KLU</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUPERLUMT</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>User supplied</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

9.3 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 9.4 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.

### 9.3.1 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 8.1 and 9.4 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 8 and 9. 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 *A* times 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 9.4.2 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 9. 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_SPFQMR, SUNLINSOL_SPBCGS, SUNLINSOL_SPTQMR, or SUNLINSOL_PCGB) 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.

9.4 CVODES SUNLinearSolver interface

Table 9.3 below lists the SUNLINSOL module linear solver functions used within the CVLS interface. As with the SUNMATRIX module, we emphasize that the CVODES user does not need to know detailed usage of linear solver functions by the CVODES code modules in order to use CVODES. 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 \textsc{cvls} does not call \texttt{SUNLinSolFree} directly, this routine should be available for users to call when cleaning up from a simulation.

Table 9.3: List of linear solver function usage in the \textsc{cvls} interface

<table>
<thead>
<tr>
<th>Function</th>
<th>DIRECT</th>
<th>ITERATIVE</th>
<th>MATRIX-ITERATIVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{SUNLinSolGetType}</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>\texttt{SUNLinSolSetATimes}</td>
<td>†</td>
<td>✓</td>
<td>†</td>
</tr>
<tr>
<td>\texttt{SUNLinSolSetPreconditioner}</td>
<td>†</td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td>\texttt{SUNLinSolSetScalingVectors}</td>
<td>†</td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td>\texttt{SUNLinSolInitialize}</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>\texttt{SUNLinSolSetup}</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>\texttt{SUNLinSolSolve}</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>\texttt{SUNLinSolNumIters}</td>
<td>†</td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td>\texttt{SUNLinSolLastFlag}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\texttt{SUNLinSolFree}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\texttt{SUNLinSolSpace}</td>
<td>†</td>
<td>†</td>
<td>†</td>
</tr>
</tbody>
</table>

Since there are a wide range of potential \textsc{sunlinsol} use cases, the following subsections describe some details of the \textsc{cvls} interface, in the case that interested users wish to develop custom \textsc{sunlinsol} modules.

9.4.1 Lagged matrix information

If the \textsc{sunlinsol} object self-identifies as having type \texttt{SUNLINEARSOLVER_DIRECT} or \texttt{SUNLINEARSOLVER_MATRIX_ITERATIVE}, then the \textsc{sunlinsol} object solves a linear system defined by a \textsc{sunmatrix} object. \textsc{cvls} will update the matrix information infrequently according to the strategies outlined in §2.1. When solving a linear system

\[ M\bar{x} = b \iff (I - \bar{\gamma}J)\bar{x} = b \]

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 \textsc{sunlinsol}-provided \texttt{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

\[ x = \frac{2}{1 + \gamma/\bar{\gamma}} \bar{x}. \]  

(9.3)

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 (9.3) 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
\[
2b = (I - \gamma J)x + (I - \gamma J)
= x - \gamma Jx + \bar{x} - J(\bar{\gamma} \bar{x})
= \frac{3}{2}(I - \gamma J)x + \frac{1}{2} \left( \frac{\gamma}{\bar{\gamma}} I - \bar{\gamma} J \right)x
= \frac{3}{2}b + \frac{1}{2} \left( \frac{\gamma}{\bar{\gamma}} I - \bar{\gamma} J \right)x.
\]

When \(\gamma/\bar{\gamma} \approx 1\), this latter term is approximately equal to \(\frac{1}{2}b\).

### 9.4.2 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 \(\text{delta}\) 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 (9.1)-(9.2)):
\[
\left\| \tilde{b} - \tilde{A}\tilde{x} \right\|_2 < \text{tol}
\iff \left\| SP_1^{-1} b - SP_1^{-1} Ax \right\|_2 < \text{tol}
\iff \sum_{i=0}^{n-1} W_i (P_1^{-1} (b - Ax))_i^2 < \text{tol}^2
\iff W_{\text{mean}}^2 \sum_{i=0}^{n-1} [(P_1^{-1} (b - Ax))_i]^2 < \text{tol}^2
\iff \sum_{i=0}^{n-1} [(P_1^{-1} (b - Ax))_i]^2 < \left( \frac{\text{tol}}{W_{\text{mean}}} \right)^2
\iff \left\| P_1^{-1} (b - Ax) \right\|_2 < \text{tol} \frac{1}{W_{\text{mean}}}
\]

Therefore the tolerance scaling factor
\[W_{\text{mean}} = \|W\|_2/\sqrt{n}\]

is computed and the scaled tolerance \(\text{delta} = \text{tol}/W_{\text{mean}}\) is supplied to the SUNLINSOL object.

### 9.5 The SUNLinearSolver_Dense implementation

This section describes the SUNLINSOL implementation for solving dense linear systems. The SUNLINEARSOLVER_DENSE module is designed to be used with the corresponding SUNMATRIX_DENSE matrix type, and
one of the serial or shared-memory 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.

### 9.5.1 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).

### 9.5.2 SUNLinearSolver_Dense functions

The SUNLINSOL_DENSE module provides the following user-callable constructor for creating a SUNLinearSolver object.

```c
SUNLinSol_Dense
```

**Call**

```c
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 9.1.1 – 9.1.3:

- SUNLinSolGetType_Dense
- SUNLinSolInitialize_Dense – this does nothing, since all consistency checks are performed at solver creation.
- SUNLinSolSetup_Dense – this performs the $LU$ factorization.
- SUNLinSolSolve_Dense – this uses the $LU$ factors and pivots array to perform the solve.
9.5 The SUNLinearSolver_Dense implementation

- 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.

9.5.3 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 lib sundials fsunlinsoldense_mod.lib in addition to the C library. For details on where the library and module file fsunlinsol_dense_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 fsunlinsoldense_mod library.

FORTRAN 77 interface functions

For solvers that include a FORTRAN 77 interface module, the SUNLINSOL_DENSE module also includes a Fortran-callable function for creating a SUNLinearSolver object.

**FSUNDENSELINSOLINIT**

Call: FSUNDENSELINSOLINIT(code, ier)

Description: The function FSUNDENSELINSOLINIT can be called for Fortran programs to create a dense SUNLinearSolver object.

Arguments: code (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).

Return value: ier is a return completion flag equal to 0 for a success return and ~1 otherwise. See printed message for details in case of failure.

Notes: This routine must be called after both the NVVECTOR and SUNMATRIX objects have been initialized.

Additionally, when using ARKODE with a non-identity mass matrix, the SUNLINSOL_DENSE module includes a Fortran-callable function for creating a SUNLinearSolver mass matrix solver object.

**FSUNMASSDENSELINSOLINIT**

Call: FSUNMASSDENSELINSOLINIT(ier)

Description: The function FSUNMASSDENSELINSOLINIT can be called for Fortran programs to create a dense SUNLinearSolver object for mass matrix linear systems.

Arguments: None

Return value: ier is an int return completion flag equal to 0 for a success return and ~1 otherwise. See printed message for details in case of failure.
Notes This routine must be called after both the NVECTOR and SUNMATRIX mass-matrix objects have been initialized.

9.5.4 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.

9.6 The SUNLinearSolver_Band implementation

This section describes the SUNLINSOL implementation for solving banded linear systems. The SUNLINSOL_BAND module is designed to be used with the corresponding SUNMATRIX_BAND matrix type, and one of the serial or shared-memory 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.

9.6.1 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 \( \mu \), then the upper triangular factor \( U \) can have upper bandwidth as big as \( smu = \text{MIN}(N-1,\mu+\mu) \). The lower triangular factor \( L \) has lower bandwidth \( \mu \).

9.6.2 SUNLinearSolver_Band functions

The SUNLINSOL_BAND module provides the following user-callable constructor for creating a SUNLinearSolver object.
The SUNLinearSolver_Band implementation

Call

\[ \text{LS} = \text{SUNLinSol\_Band}(y, A); \]

Description

The function \text{SUNLinSol\_Band} creates and allocates memory for a band SUNLinearSolver object.

Arguments

- \( y \) (\text{N\_Vector}) a template for cloning vectors needed within the solver
- \( A \) (\text{SUNMatrix}) a \text{SUNMATRIX\_BAND} matrix template for cloning matrices needed within the solver

Return value

This returns a \text{SUNLinearSolver} object. If either \( A \) or \( y \) are incompatible then this routine will return \text{NULL}.

Notes

This routine will perform consistency checks to ensure that it is called with consistent \text{NVVECTOR} and \text{SUNMATRIX} implementations. These are currently limited to the \text{SUNMATRIX\_BAND} matrix type and the \text{NVVECTOR\_SERIAL}, \text{NVVECTOR\_OPENMP}, and \text{NVVECTOR\_PTHREADS} vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.

Additionally, this routine will verify that the input matrix \( A \) is allocated with appropriate upper bandwidth storage for the \text{LU} factorization.

Deprecated Name

For backward compatibility, the wrapper function \text{SUNBandLinearSolver} with identical input and output arguments is also provided.

F2003 Name

This function is callable as \text{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 9.1.1 – 9.1.3:

- \text{SUNLinSolGetType\_Band}
- \text{SUNLinSolInitialize\_Band} – this does nothing, since all consistency checks are performed at solver creation.
- \text{SUNLinSolSetup\_Band} – this performs the \text{LU} factorization.
- \text{SUNLinSolSolve\_Band} – this uses the \text{LU} factors and \text{pivots} array to perform the solve.
- \text{SUNLinSolLastFlag\_Band}
- \text{SUNLinSolSpace\_Band} – this only returns information for the storage within the solver object, i.e. storage for \( N \), \text{last\_flag}, and \text{pivots}.
- \text{SUNLinSolFree\_Band}

All of the listed operations are callable via the FORTRAN 2003 interface module by prepending an ‘F’ to the function name.

9.6.3 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 \text{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 \text{SUNLinSol\_Band} is interfaced as \text{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 are installed see Appendix A. We note that the module is accessible from the Fortran 2003 Sundials integrators without separately linking to the libsundials_fsunlinsolband_mod library.

**FORTRAN 77 interface functions**

For solvers that include a Fortran 77 interface module, the sunlinsol_band module also includes a Fortran-callable function for creating a SUNLinearSolver object.

**FSUNBANDLINSOLINIT**

Call                      FSUNBANDLINSOLINIT(code, ier)
Description              The function FSUNBANDLINSOLINIT can be called for Fortran programs to create a band SUNLinearSolver object.
Arguments                code (int*) is an integer input specifying the solver id (1 for cvode, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).
Return value             ier is a return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes                    This routine must be called after both the NVECTOR and SUNMATRIX objects have been initialized.

Additionally, when using ARKODE with a non-identity mass matrix, the sunlinsol_band module includes a Fortran-callable function for creating a SUNLinearSolver mass matrix solver object.

**FSUNMASSBANDLINSOLINIT**

Call                      FSUNMASSBANDLINSOLINIT(ier)
Description              The function FSUNMASSBANDLINSOLINIT can be called for Fortran programs to create a band SUNLinearSolver object for mass matrix linear systems.
Arguments                None
Return value             ier is an int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes                    This routine must be called after both the NVECTOR and SUNMATRIX mass-matrix objects have been initialized.

**9.6.4 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.
9.7 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 NVVECTOR implementations (NVVECTOR_SERIAL, NVVECTOR_OPENMP, or NVVECTOR_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.

9.7.1 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.

9.7.2 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 consistent NVVECTOR and SUNMATRIX implementations. These are currently limited to the SUNMATRIX_DENSE matrix type and the NVVECTOR_SERIAL, NVVECTOR_OPENMP, and NVVECTOR_PTHREADS vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.
```
Description of the SUNLinearSolver module

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 9.1.1 – 9.1.3:

- SUNLinSolGetType_LapackDense
- SUNLinSolInitialize_LapackDense – this does nothing, since all consistency checks are performed at solver creation.
- SUNLinSolSetup_LapackDense – this calls either DGETRF or SGETRF to perform the LU factorization.
- SUNLinSolSolve_LapackDense – this calls either DGETRS or SGETRS to use the LU factors and pivots array to perform the solve.
- SUNLinSolLastFlag_LapackDense
- SUNLinSolSpace_LapackDense – this only returns information for the storage within the solver object, i.e. storage for \( N \), last_flag, and pivots.
- SUNLinSolFree_LapackDense

9.7.3 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 NVECTOR and SUNMATRIX objects have been initialized.

Additionally, when using ARKODE with a non-identity mass matrix, the SUNLINSOL_LAPACKDENSE module includes a Fortran-callable function for creating a SUNLinearSolver mass matrix solver object.

**FSUNMASSLAPACKDENSEINIT**

Call

```
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

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.
9.7.4 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.

9.8 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_sunlinsol_lapackband.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.

9.8.1 SUNLinearSolver_LapackBand description

This solver is constructed to perform the following operations:

- The “setup” call performs a LU factorization with partial (row) pivoting, \( PA = LU \), where \( P \) is a permutation matrix, \( L \) is a lower triangular matrix with 1’s on the diagonal, and \( U \) is an upper triangular matrix. This factorization is stored in-place on the input SUNMATRIX_BAND object \( A \), with pivoting information encoding \( P \) stored in the pivots array.

- The “solve” call performs pivoting and forward and backward substitution using the stored pivots array and the LU factors held in the SUNMATRIX_BAND object.

- \( A \) must be allocated to accommodate the increase in upper bandwidth that occurs during factorization. More precisely, if \( A \) is a band matrix with upper bandwidth \( \mu \) and lower bandwidth \( m \), then the upper triangular factor \( U \) can have upper bandwidth as big as \( \text{smu} = \text{MIN}(N-1, \mu + m) \). The lower triangular factor \( L \) has lower bandwidth \( m \).
9.8.2 SUNLinearSolver_LapackBand functions

The SUNLINSOL_LAPACKBAND module provides the following user-callable constructor for creating a SUNLinearSolver object.

```
SUNLinSol_LapackBand
Call LS = SUNLinSol_LapackBand(y, A);
Description The function SUNLinSol_LapackBand creates and allocates memory for a LAPACK-based, band SUNLinearSolver object.
Arguments y (N_Vector) a template for cloning vectors needed within the solver
A (SUNMatrix) a SUNMATRIX_BAND matrix template for cloning matrices needed within the solver
Return value This returns a SUNLinearSolver object. If either A or y are incompatible then this routine will return NULL.
Notes This routine will perform consistency checks to ensure that it is called with consistent NVECTOR and SUNMATRIX implementations. These are currently limited to the SUNMATRIX_BAND matrix type and the NVECTOR_SERIAL, NVECTOR_OPENMP, and NVECTOR_PTHREADS vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.
Additionally, this routine will verify that the input matrix A is allocated with appropriate upper bandwidth storage for the LU factorization.
Deprecated Name For backward compatibility, the wrapper function SUNLapackBand with identical input and output arguments is also provided.
```

The SUNLINSOL_LAPACKBAND module defines band implementations of all “direct” linear solver operations listed in Sections 9.1.1 – 9.1.3:

- 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

9.8.3 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.
9.9 The SUNLinearSolver_KLU implementation

**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.

9.8.4 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.

9.9 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, 14]. In order to use the SUNLINSOL_KLU interface to KLU, it is assumed that KLU has been installed on the system prior to installation of SUNDIALS, and that SUNDIALS has been configured appropriately to link with KLU (see Appendix A for details). Additionally, this wrapper only supports double-precision calculations, and therefore cannot be compiled if SUNDIALS is configured to have realtype set to either extended or single (see Section 4.2). Since the KLU library supports both 32-bit and 64-bit integers, this interface will be compiled for either of the available sunindextype options.

9.9.1 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.

9.9.2 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
```
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 9.1.1 – 9.1.3:

• 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.

```c
SUNLinSol_KLUreInit
```

Call

```c
retval = SUNLinSol_KLUreInit(LS, A, nnz, reinit_type);
```

Description

The function SUNLinSol_KLUreInit reinitializes memory and flags for a new factorization (symbolic and numeric) to be conducted at the next solver setup call. This routine is useful in the cases where the number of nonzeroes has changed or if the structure of the linear system has changed which would require a new symbolic (and numeric factorization).

Arguments

- **LS** (SUNLinearSolver) a template for cloning vectors needed within the solver
- **A** (SUNMatrix) a SUNMATRIX_SPARSE matrix template for cloning matrices needed within the solver
- **nnz** (sunindextype) the new number of nonzeros in the matrix
- **reinit_type** (int) flag governing the level of reinitialization. The allowed values are:
Description of the SUNLinearSolver module

- **SUNKLU_REINIT_FULL** – The Jacobian matrix will be destroyed and a new one will be allocated based on the \( \text{nnz} \) 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 \( \text{nnz} \) given in the sparse matrix provided to the original constructor routine (or the previous SUNLinSol_KLUReInit call).

Return value

The return values from this function are **SUNLS_MEM_NULL** (either \( S \) or \( A \) are NULL), **SUNLS_ILL_INPUT** (\( A \) does not have type **SUNMATRIX_SPARSE** or \( \text{reinit.type} \) 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.

---

**SUNKLU_ReInit**

Call

\[
\text{retval} = \text{SUNLinSol_KLUReInit}(\text{LS}, \text{ordering});
\]

Description

This function sets the ordering used by **KLU** for reducing fill in the linear solve.

Arguments

\( \text{LS} \) (SUNLinearSolver) the SUNLINSOL_KLU object

ordering (int) flag indicating the reordering algorithm to use, the options are:

- 0 AMD,
- 1 COLAMD, and
- 2 the natural ordering.

The default is 1 for COLAMD.

Return value

The return values from this function are **SUNLS_MEM_NULL** (\( S \) is NULL), **SUNLS_ILL_INPUT** (invalid ordering choice), or **SUNLS_SUCCESS**.

Deprecated Name

For backward compatibility, the wrapper function **SUNKLUSetOrdering** with identical input and output arguments is also provided.

F2003 Name

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

---

### 9.9.3 SUNLinearSolver_KLU Fortran interfaces

The SUNLINSOL_KLU module provides a **FORTAN** 2003 module as well as **FORTRAN 77** style interface functions for use from **FORTRAN** applications.
9.9 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 lib sundials_fsunlinsolklu_mod.lib in addition to the C library. For details on where the library and module file fsunlinsol_klu_mod.mod are installed see Appendix A.

FORTRAN 77 interface functions

For solvers that include a FORTRAN 77 interface module, the SUNLINSOL_KLU module also includes a Fortran-callable function for creating a SUNLinearSolver object.

FSUNKLUINIT

Call
FSUNKLUINIT(code, ier)

Description The function FSUNKLUINIT can be called for Fortran programs to create a SUNLINSOL_KLU object.

Arguments code (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).

Return value 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
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.

FSUNMASSKLUSETORDERING

Call FSUNMASSKLUSETORDERING(ier)

Description The function FSUNMASSKLUSETORDERING can be called for Fortran programs to change the reordering algorithm used by KLU for mass matrix linear systems.

Arguments The arguments are identical to FSUNKLUSETORDERING above, except that code is not needed since mass matrix linear systems only arise in ARKODE.

Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes See SUNLinSol_KLUSetOrdering for complete further documentation of this routine.
9.10 The SUNLinearSolver_SuperLUMT implementation

9.9.4 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).

9.10 The SUNLinearSolver_SuperLUMT implementation

This section describes the SUNLINSOL implementation for solving sparse linear systems with SuperLU_MT. The superlumt module is designed to be used with the corresponding SUNMATRIXSparse matrix type, and one of the serial or shared-memory NVECTOR implementations (NVECTOR_SERIAL, NVECTOR_OPENMP, or NVECTOR_PTHREADS). While these are compatible, it is not recommended to use a threaded vector module with SUNLINSOL_SUPERLUMT unless it is the NVECTOR_OPENMP module and the superlumt library has also been compiled with OpenMP.

The header file to include when using this module is sunlinsol/sunlinsol_superlumt.h. The installed module library to link to is libsundials.sunlinsolsuperlumt.lib where .lib is typically .so for shared libraries and .a for static libraries.

The SUNLINSOL_SUPERLUMT module is a SUNLINSOL wrapper for the SUPERLUMT sparse matrix factorization and solver library written by X. Sherry Li [2, 34, 16]. The package performs matrix factorization using threads to enhance efficiency in shared memory parallel environments. It should be noted that threads are only used in the factorization step. In order to use the SUNLINSOL_SUPERLUMT interface to SUPERLUMT, it is assumed that SUPERLUMT has been installed on the system prior to installation of SUNDIALS, and that SUNDIALS has been configured appropriately to link with SUPERLUMT (see Appendix A for details). Additionally, this wrapper only supports single- and double-precision calculations, and therefore cannot be compiled if SUNDIALS is configured to have realltype set to extended (see Section 4.2). Moreover, since the SUPERLUMT library may be installed to support either 32-bit or 64-bit integers, it is assumed that the SUPERLUMT library is installed using the same integer precision as the SUNDIALS sunindextype option.

9.10.1 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 SUNLINSOL_SUPERLUMT module is the COLAMD ordering.
Since the linear systems that arise within the context of SUNDIALS calculations will typically have identical sparsity patterns, the SUNLINSOL_SUPERLUMT module is constructed to perform the following operations:

- The first time that the “setup” routine is called, it performs the symbolic factorization, followed by an initial numerical factorization.
- On subsequent calls to the “setup” routine, it skips the symbolic factorization, and only refactorizes 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.

### 9.10.2 SUNLinearSolver_SuperLUMT functions

The module SUNLINSOL_SUPERLUMT provides the following user-callable constructor for creating a SUNLinearSolver object.

```
SUNLinSol_SuperLUMT
```

**Call**

```c
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 9.1.1 – 9.1.3:

- SUNLinSolGetType_SuperLUMT
- SUNLinSolInitialize_SuperLUMT — this sets the first_factorize flag to 1 and resets the internal SUPERLUMT statistics variables.
- SUNLinSolSetup_SuperLUMT — this performs either a LU factorization or refactorization of the input matrix.
• **SUNLinSolSolve_SuperLUMT** – this calls the appropriate SUPERLUMT solve routine to utilize the LU factors to solve the linear system.

• **SUNLinSolLastFlag_SuperLUMT**

• **SUNLinSolSpace_SuperLUMT** – this only returns information for the storage within the solver interface, i.e. storage for the integers last_flag and first_factorize. For additional space requirements, see the SUPERLUMT documentation.

• **SUNLinSolFree_SuperLUMT**

The SUNLINSOL_SUPERLUMT module also defines the following additional user-callable function.

```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_I LL_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.

### 9.10.3 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.

```fortran
FSUNSUPERLUMTINIT
```

Call

```fortran
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.
**FSUNMASSUPERLUMTINIT**

**Description**  
The function **FSUNMASSUPERLUMTINIT** 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**

**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.

**FSUNMASSUPERLUMTSETORDERING**

**Description**  
The function **FSUNMASSUPERLUMTSETORDERING** can be called for Fortran programs to update the ordering algorithm in a **sunlinsol_superlumt** object for mass matrix linear systems.

**Arguments**  
- `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.
9.10.4 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.

9.11 The SUNLinearSolver_SPGMR implementation

This section describes the SUNLinearSolver implementation of the SPGMR (Scaled, Preconditioned, Generalized Minimum Residual [41]) iterative linear solver. The sunlinsol_spgmr module is designed to be compatible with any NVVECTOR implementation that supports a minimal subset of operations (N_VClone, N_VDotProd, N_VScale, N_VLinearSum, N_VProd, N_VConst, N_VDiv, and N_VDestroy). When using Classical Gram-Schmidt, the optional function N_VDotProdMulti may be supplied for increased efficiency.

To access the sunlinsol_spgmr module, include the header file sunlinsol/sunlinsol_spgmr.h. We note that the sunlinsol_spgmr module is accessible from Sundials packages without separately linking to the libsundials_sunlinsolspgmr module library.

9.11.1 SUNLinearSolver_SPGMR description

This solver is constructed to perform the following operations:

- During construction, the xcor and vtemp arrays are cloned from a template NVVECTOR that is input, and default solver parameters are set.
- User-facing “set” routines may be called to modify default solver parameters.
- Additional “set” routines are called by the Sundials solver that interfaces with SUNLINSOL_SPGMR to supply the ATimes, PSetup, and Psolve function pointers and s1 and s2 scaling vectors.
• In the “initialize” call, the remaining solver data is allocated (V, Hes, givens, and yg)

• In the “setup” call, any non-NULL PSetup function is called. Typically, this is provided by the SUNDIALS solver itself, that translates between the generic PSetup function and the solver-specific routine (solver-supplied or user-supplied).

• In the “solve” call, the GMRES iteration is performed. This will include scaling, preconditioning, and restarts if those options have been supplied.

9.11.2 SUNLinearSolver_SPGMR functions

The SUNLINSOL_SPGMR module provides the following user-callable constructor for creating a SUNLinearSolver object.

```c
SUNLinSol_SPGMR
```

Call

\[
LS = \text{SUNLinSol}_\text{SPGMR}(y, \text{pretype}, \text{maxl});
\]

Description

The function SUNLinSol_SPGMR creates and allocates memory for a SPGMR SUNLinearSolver object.

Arguments

- **y** (N_Vector) a template for cloning vectors needed within the solver
- **pretype** (int) flag indicating the desired type of preconditioning, allowed values are:
  - PREC_NONE (0)
  - PREC_LEFT (1)
  - PREC_RIGHT (2)
  - PREC_BOTH (3)

Any other integer input will result in the default (no preconditioning).

- **maxl** (int) the number of Krylov basis vectors to use. Values \( \leq 0 \) 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_SPGMR object to use any of the preconditioning options with these solvers, this use mode is not supported and may result in inferior performance.

Deprecated Name

For backward compatibility, the wrapper function SUNSPGMR with identical input and output arguments is also provided.

F2003 Name

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

The SUNLINSOL_SPGMR module defines implementations of all “iterative” linear solver operations listed in Sections 9.1.1 – 9.1.3:

- SUNLinSolGetType_SPGMR
- SUNLinSolInitialize_SPGMR
- SUNLinSolSetATimes_SPGMR
- SUNLinSolSetPreconditioner_SPGMR
9.11 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**

```c
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_IILL_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**

```c
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_IILL_INPUT`.

**Return value**

This routine will return with one of the error codes `SUNLS_IILL_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

SUNLinSol_SPGMRSetMaxRestarts

Call

\texttt{retval = SUNLinSol\_SPGMRSetMaxRestarts(LS, maxrs);}  

Description

The function \texttt{SUNLinSol\_SPGMRSetMaxRestarts} sets the number of GMRES restarts to allow in the \texttt{SUNLIN\_SOL\_SPGMR} object.

Arguments

\begin{itemize}
  \item \texttt{LS} (\texttt{SUNLinearSolver}) the \texttt{SUNLIN\_SOL\_SPGMR} object to update
  \item \texttt{maxrs} (\texttt{int}) integer indicating number of restarts to allow. A negative input will result in the default of 0.
\end{itemize}

Return value

This routine will return with one of the error codes \texttt{SUNLS\_MEM\_NULL} (\texttt{S} is \texttt{NULL}) or \texttt{SUNLS\_SUCCESS}.

Deprecated Name

For backward compatibility, the wrapper function \texttt{SUNSPGMRSetMaxRestarts} with identical input and output arguments is also provided.

F2003 Name

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

9.11.3 SUNLinearSolver\_SPGMR Fortran interfaces

The \texttt{SUNLIN\_SOL\_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 \texttt{fsunlinsol\_spgmr\_mod} FORTRAN module defines interfaces to all \texttt{SUNLIN\_SOL\_SPGMR} 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{SUNLinSol\_SPGMR} is interfaced as \texttt{FSUNLinSol\_SPGMR}.

The FORTRAN 2003 \texttt{SUNLIN\_SOL\_SPGMR} interface module can be accessed with the \texttt{use} statement, i.e. \texttt{use fsunlinsol\_spgmr\_mod}, and linking to the library \texttt{libsundials\_fsunlinsol\_spgmr\_mod.lib} in addition to the C library. For details on where the library and module file \texttt{fsunlinsol\_spgmr\_mod.mod} are installed see Appendix A. We note that the module is accessible from the FORTRAN 2003 SUNDIALS integrators \texttt{without} separately linking to the \texttt{libsundials\_fsunlinsol\_spgmr\_mod.lib} library.

FORTRAN 77 interface functions

For solvers that include a FORTRAN 77 interface module, the \texttt{SUNLIN\_SOL\_SPGMR} module also includes a Fortran-callable function for creating a \texttt{SUNLinearSolver} object.

FSUNSPGMRINIT

Call

\texttt{FSUNSPGMRINIT(code, pretype, maxl, ier)}

Description

The function \texttt{FSUNSPGMRINIT} can be called for Fortran programs to create a \texttt{SUNLIN\_SOL\_SPGMR} object.

Arguments

\begin{itemize}
  \item \texttt{code} (\texttt{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}).
  \item \texttt{pretype} (\texttt{int*}) flag indicating desired preconditioning type
  \item \texttt{maxl} (\texttt{int*}) flag indicating Krylov subspace size
\end{itemize}

Return value

\texttt{ier} is a return completion flag equal to 0 for a success return and \texttt{-1} otherwise. See printed message for details in case of failure.

Notes

This routine must be called \textit{after} the \texttt{NVECTOR} object has been initialized.

Allowable values for \texttt{pretype} and \texttt{maxl} are the same as for the C function \texttt{SUNLinSol\_SPGMR}.
Additionally, when using ARKODE with a non-identity mass matrix, the SUNLINSOL_SPGMR module includes a Fortran-callable function for creating a SUNLinearSolver mass matrix solver object.

**FSUNMASSSPGMRINIT**

Call: `FSUNMASSSPGMRINIT(pretype, maxl, ier)`

Description: The function `FSUNMASSSPGMRINIT` can be called for Fortran programs to create a SUNLINSOL_SPGMR object for mass matrix linear systems.

Arguments:
- `pretype` (int*) flag indicating desired preconditioning type
- `maxl` (int*) flag indicating Krylov subspace size

Return value: `ier` is an int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes: This routine must be called after the NVECTOR object has been initialized. Allowable values for `pretype` and `maxl` are the same as for the C function `SUNLinSol_SPGMR`.

The `SUNLinSol_SPGMRSetPrecType`, `SUNLinSol_SPGMRSetGSType` and `SUNLinSol_SPGMRSetMaxRestarts` routines also support Fortran interfaces for the system and mass matrix solvers.

**FSUNSPGMRSETGSTYPE**

Call: `FSUNSPGMRSETGSTYPE(code, gstype, ier)`

Description: The function `FSUNSPGMRSETGSTYPE` can be called for Fortran programs to change the Gram-Schmidt orthogonalization algorithm.

Arguments:
- `code` (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).
- `gstype` (int*) flag indicating the desired orthogonalization algorithm.

Return value: `ier` is an int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes: See `SUNLinSol_SPGMRSetGSType` for complete further documentation of this routine.

**FSUNMASSSPGMRSETGSTYPE**

Call: `FSUNMASSSPGMRSETGSTYPE(gstype, ier)`

Description: The function `FSUNMASSSPGMRSETGSTYPE` can be called for Fortran programs to change the Gram-Schmidt orthogonalization algorithm for mass matrix linear systems.

Arguments: The arguments are identical to `FSUNSPGMRSETGSTYPE` above, except that `code` is not needed since mass matrix linear systems only arise in ARKODE.

Return value: `ier` is an int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes: See `SUNLinSol_SPGMRSetGSType` for complete further documentation of this routine.

**FSUNSPGMRSETPRECTYPE**

Call: `FSUNSPGMRSETPRECTYPE(code, pretype, ier)`

Description: The function `FSUNSPGMRSETPRECTYPE` can be called for Fortran programs to change the type of preconditioning to use.

Arguments:
- `code` (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).
- `pretype` (int*) flag indicating the type of preconditioning to use.
Return value \textit{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\_SPGMRSetPrecType} for complete further documentation of this routine.

\textbf{FSUNMASSSPGMRSETPRECTYPE}  
\textbf{Call} \texttt{FSUNMASSSPGMRSETPRECTYPE(prtype, ier)}  
\textbf{Description} The function \texttt{FSUNMASSSPGMRSETPRECTYPE} can be called for Fortran programs to change the type of preconditioning for mass matrix linear systems.
\textbf{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.
\textbf{Return value} \textit{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.
\textbf{Notes} See \texttt{SUNLinSol\_SPGMRSetPrecType} for complete further documentation of this routine.

\textbf{FSUNSPGMRSETMAXRS}  
\textbf{Call} \texttt{FSUNSPGMRSETMAXRS(code, maxrs, ier)}  
\textbf{Description} The function \texttt{FSUNSPGMRSETMAXRS} can be called for Fortran programs to change the maximum number of restarts allowed for \texttt{spgmr}.
\textbf{Arguments} \texttt{code} (\texttt{int*}) is an integer input specifying the solver id (1 for \texttt{cvoce}, 2 for \texttt{ida}, 3 for \texttt{kinsol}, and 4 for \texttt{arkode}). \texttt{maxrs} (\texttt{int*}) maximum allowed number of restarts.
\textbf{Return value} \textit{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.
\textbf{Notes} See \texttt{SUNLinSol\_SPGMRSetMaxRestarts} for complete further documentation of this routine.

\textbf{FSUNMASSSPGMRSETMAXRS}  
\textbf{Call} \texttt{FSUNMASSSPGMRSETMAXRS(maxrs, ier)}  
\textbf{Description} The function \texttt{FSUNMASSSPGMRSETMAXRS} can be called for Fortran programs to change the maximum number of restarts allowed for \texttt{spgmr} for mass matrix linear systems.
\textbf{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.
\textbf{Return value} \textit{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.
\textbf{Notes} See \texttt{SUNLinSol\_SPGMRSetMaxRestarts} for complete further documentation of this routine.

\subsection{SUNLinearSolver\_SPGMR content}

The \texttt{SUNLINSOL\_SPGMR} module defines the \textit{content} field of a \texttt{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 $A v$ 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

\[
  F_i = \begin{bmatrix}
    1 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & 1 \\
    \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\
    c_i & -s_i & 1 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
    s_i & c_i & & 1 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
    & & & & & & & & & & & \\
    & & & & & & & & & & & \\
    & & & & & & & & & & & \\
    & & & & & & & & & & & \\
    & & & & & & & & & & & \\
    & & & & & & & & & & & \\
    \end{bmatrix},
\]

xcor - a vector which holds the scaled, preconditioned correction to the initial guess,
yg - a length (maxl+1) array of realtype values used to hold “short” vectors (e.g. y and g),
vtemp - temporary vector storage.

9.12 The SUNLinearSolver_SPFGMR implementation

This section describes the SUNLINSOL implementation of the SPFGMR (Scaled, Preconditioned, Flexible, Generalized Minimum Residual [40]) 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, 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 sunlinsol/sunlinsol_spfgmr.h. We note that the SUNLINSOL_SPFGMR module is accessible from SUNDIALS packages without separately linking to the libsundials_sunlinsolspfgmr module library.

9.12.1 SUNLinearSolver_SPFGMR description

This solver is constructed to perform the following operations:

- During construction, the xcor and vtemp arrays are cloned from a template nvector that is input, and default solver parameters are set.
- User-facing “set” routines may be called to modify default solver parameters.
- Additional “set” routines are called by the SUNDIALS solver that interfaces with SUNLINSOL_SPFGMR to supply the ATimes, PSetup, and Psolve function pointers and s1 and s2 scaling vectors.
- In the “initialize” call, the remaining solver data is allocated (V, Hes, givens, and yg)
- In the “setup” call, any non-NULL PSetup function is called. Typically, this is provided by the SUNDIALS solver itself, that translates between the generic PSetup function and the solver-specific routine (solver-supplied or user-supplied).
- In the “solve” call, the FGMRES iteration is performed. This will include scaling, preconditioning, and restarts if those options have been supplied.

9.12.2 SUNLinearSolver_SPFGMR functions

The SUNLINSOL_SPFGMR module provides the following user-callable constructor for creating a SUNLinearSolver object.

```
SUNLinSol_SPFGMR
Call LS = SUNLinSol_SPFGMR(y, pretype, maxl);
Description The function SUNLinSol_SPFGMR creates and allocates memory for a SPFGMR SUNLinearSolver object.
Arguments y (N_Vector) a template for cloning vectors needed within the solver
pretype (int) flag indicating the desired type of preconditioning, allowed values are:
  • PREC_NONE (0)
  • PREC_LEFT (1)
```
### 9.12 The SUNLinearSolver_SPFGMR implementation

#### SUNSPFGMR

The SUNSPFGMR module defines implementations of all “iterative” linear solver operations listed in Sections 9.1.1 – 9.1.3:

- 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 SUNSPFGMR module also defines the following additional user-callable functions.

#### SUNLinSol_SPFGMRSSetPrecType

**Call**

```c
retval = SUNLinSol_SPFGMRSSetPrecType(LS, pretype);
```

**Description**

The function SUNLinSol_SPFGMRSSetPrecType updates the type of preconditioning to use in the SUNSPFGMR object.

**Arguments**

- **LS** (SUNLinearSolver) the SUNSPFGMR object to update
- **pretype** (int) flag indicating the desired type of preconditioning, allowed values match those discussed in SUNLinSol_SPFGMR.
Return value  This routine will return with one of the error codes SUNLS_Ill_INPUT (illegal pretype), SUNLS_MEM_NULL (S is NULL) or SUNLS_SUCCESS.

Deprecated Name  For backward compatibility, the wrapper function SUNSPFGMRSetPrecType with identical input and output arguments is also provided.

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

**SUNLinSol_SPFGMRSetGSType**

Call  

```c
retval = SUNLinSol_SPFGMRSetGSType(LS, gstype);
```

Description  The function SUNLinSol_SPFGMRSetGSType sets the type of Gram-Schmidt orthogonalization to use in the SUNLINSOL_SPFGMR object.

Arguments  

- **LS** (SUNLinearSolver) the SUNLINSOL_SPFGMR object to update
- **gstype** (int) flag indicating the desired orthogonalization algorithm; allowed values are:
  - MODIFIED_GS (1)
  - CLASSICAL_GS (2)

Any other integer input will result in a failure, returning error code SUNLS_Ill_INPUT.

Return value  This routine will return with one of the error codes SUNLS_Ill_INPUT (illegal pretype), SUNLS_MEM_NULL (S is NULL) or SUNLS_SUCCESS.

Deprecated Name  For backward compatibility, the wrapper function SUNSPFGMRSetGSType with identical input and output arguments is also provided.

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

**SUNLinSol_SPFGMRSetMaxRestarts**

Call  

```c
retval = SUNLinSol_SPFGMRSetMaxRestarts(LS, maxrs);
```

Description  The function SUNLinSol_SPFGMRSetMaxRestarts sets the number of GMRES restarts to allow in the SUNLINSOL_SPFGMR object.

Arguments  

- **LS** (SUNLinearSolver) the SUNLINSOL_SPFGMR object to update
- **maxrs** (int) integer indicating number of restarts to allow. A negative input will result in the default of 0.

Return value  This routine will return with one of the error codes SUNLS_MEM_NULL (S is NULL) or SUNLS_SUCCESS.

Deprecated Name  For backward compatibility, the wrapper function SUNSPFGMRSetMaxRestarts with identical input and output arguments is also provided.

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

### 9.12.3 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.
9.12 The SUNLinearSolver_SPFGMR 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 NVECTOR object has been initialized. Allowable values for pretype and maxl are the same as for the C function SUNLinSol_SPFGMR.

Additionally, when using ARKODE with a non-identity mass matrix, the SUNLINSOL_SPFGMR module includes a Fortran-callable function for creating a SUNLinearSolver mass matrix solver object.

FSUNMASSSPFGMRINIT

Call

FSUNMASSSPFGMRINIT(pretype, maxl, ier)

Description

The function FSUNMASSSPFGMRINIT can be called for Fortran programs to create a SUNLINSOL_SPFGMR object for mass matrix linear systems.

Arguments

- pretype (int*) flag indicating desired preconditioning type
- maxl (int*) flag indicating Krylov subspace size

Return value

ier is a int return completion flag equal to 0 for a success return and ~1 otherwise. See printed message for details in case of failure.

Notes

This routine must be called after the NVECTOR object has been initialized. Allowable values for pretype and maxl are the same as for the C function SUNLinSol_SPFGMR.

The SUNLinSol_SPFGMRSetPrecType, SUNLinSol_SPFGMRSetGSType and SUNLinSol_SPFGMRSetMaxRestarts routines also support Fortran interfaces for the system and mass matrix solvers.
# Description of the SUNLinearSolver module

**FSUNSPFGMRSETGSTYPE**

**Call**

FSUNSPFGMRSETGSTYPE(code, gstype, ier)

**Description**
The function FSUNSPFGMRSETGSTYPE can be called for Fortran programs to change the Gram-Schmidt orthogonalization algorithm.

**Arguments**
- **code** (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).
- **gstype** (int*) flag indicating the desired orthogonalization algorithm.

**Return value**
- **ier** is an int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

**Notes**
- See SUNLinSol_SPFGMRSetGSType for complete further documentation of this routine.

**FSUNMASSSPFGMRSETGSTYPE**

**Call**

FSUNMASSSPFGMRSETGSTYPE(gstype, ier)

**Description**
The function FSUNMASSSPFGMRSETGSTYPE can be called for Fortran programs to change the Gram-Schmidt orthogonalization algorithm for mass matrix linear systems.

**Arguments**
The arguments are identical to FSUNSPFGMRSETGSTYPE above, except that code is not needed since mass matrix linear systems only arise in ARKODE.

**Return value**
- **ier** is an int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

**Notes**
- See SUNLinSol_SPFGMRSetGSType for complete further documentation of this routine.

**FSUNSPFGMRSETPRECTYPE**

**Call**

FSUNSPFGMRSETPRECTYPE(code, pretype, ier)

**Description**
The function FSUNSPFGMRSETPRECTYPE can be called for Fortran programs to change the type of preconditioning to use.

**Arguments**
- **code** (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).
- **pretype** (int*) flag indicating the type of preconditioning to use.

**Return value**
- **ier** is 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**

FSUNMASSSPFGMRSETPRECTYPE(pretype, ier)

**Description**
The function FSUNMASSSPFGMRSETPRECTYPE can be called for Fortran programs to change the type of preconditioning for mass matrix linear systems.

**Arguments**
The arguments are identical to FSUNSPFGMRSETPRECTYPE above, except that code is not needed since mass matrix linear systems only arise in ARKODE.

**Return value**
- **ier** is 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.
9.12 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.

9.12.4 SUNLinearSolver_SPFGMR content

The SUNLINSOL_SPFGMR module defines the content field of a SUNLinearSolver as the following structure:

```
struct _SUNLinearSolverContent_SPFGMR {
    int maxl;
    int pretype;
    int gstype;
    int max_restarts;
    int numiters;
    realtype resnorm;
    long int last_flag;
    ATimesFn ATimes;
    void* ATData;
    PSetupFn Psetup;
    PSolveFn Psolve;
    void* PData;
    N_Vector s1;
    N_Vector s2;
    N_Vector *V;
    N_Vector *Z;
    realtype **Hes;
    realtype *givens;
    N_Vector xcor;
    realtype *yg;
    N_Vector vtemp;
};
```
These entries of the content field contain the following information:

- **maxl** - number of FGMRES basis vectors to use (default is 5),
- **pretype** - flag for 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 $V[0], \ldots, V[\text{maxl}]$. 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 $Z[0], \ldots, Z[\text{maxl}]$. Each $z_i$ is a vector of type nvector.,
- **Hes** - the $(\text{maxl} + 1) \times \text{maxl}$ Hessenberg matrix. It is stored row-wise so that the $(i,j)$th element is given by $Hes[i][j]$,
- **givens** - a length $2 \times \text{maxl}$ array which represents the Givens rotation matrices that arise in the FGMRES algorithm. These matrices are $F_0, F_1, \ldots, F_j$, where

$$F_i = \begin{bmatrix}
1 & & \\
\vdots & \ddots & \\
& & 1 \\
& & c_i & -s_i \\
& & s_i & c_i \\
& & & \ddots & \ddots & \\
& & & & 1 & \\
& & & & & 1
\end{bmatrix},$$

are represented in the givens vector as $\text{givens}[0] = c_0, \text{givens}[1] = s_0, \text{givens}[2] = c_1, \text{givens}[3] = s_1, \ldots \text{givens}[2j] = c_j, \text{givens}[2j+1] = s_j$,
- **xcor** - a vector which holds the scaled, preconditioned correction to the initial guess,
- **yg** - a length $(\text{maxl}+1)$ array of realtype values used to hold "short" vectors (e.g. y and $g$),
- **vtemp** - temporary vector storage.

### 9.13 The SUNLinearSolver SPBCGS implementation

This section describes the SUNLINSOL implementation of the SPBCGS (Scaled, Preconditioned, Bi-Conjugate Gradient, Stabilized [44]) iterative linear solver. The SUNLINSOL_SPBCGS 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_VDiv, and N_VDestroy). Unlike the SPGMR and SPFGMR algorithms, SPBCGS requires a fixed amount of memory that does not increase with the number of allowed iterations.
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_sunlinsolspbgs` module library.

### 9.13.1 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.

### 9.13.2 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 9.1.1 – 9.1.3:

- 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  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.
9.13 The SUNLinearSolver_SPBCGS implementation

```c
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.

### 9.13.3 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.lib` 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.

```fortran
FSUNSPBCGSINIT
```

Call: `FSUNSPBCGSINIT(code, pretype, maxl, ier)`

Description: The function `FSUNSPBCGSINIT` can be called for Fortran programs to create a `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).
- `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.
9.13 The SUNLinearSolver_SPBCGS implementation

Return value \texttt{ier} is a \texttt{int} return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes See SUNLinSol_SPBCGSSetMaxl for complete further documentation of this routine.

FSUNMASSSPBCGSSETMAXL

Call FSUNMASSSPBCGSSETMAXL(maxl, ier)

Description The function FSUNMASSSPBCGSSETMAXL can be called for Fortran programs to change the type of preconditioning for mass matrix linear systems.

Arguments The arguments are identical to FSUNSPBCGSSETMAXL above, except that \texttt{code} is not needed since mass matrix linear systems only arise in ARKODE.

Return value \texttt{ier} is a \texttt{int} return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes See SUNLinSol_SPBCGSSetMaxl for complete further documentation of this routine.

9.13.4 SUNLinearSolver_SPBCGS content

The sunlinsol_spbcgs module defines the content field of a SUNLinearSolver as the following structure:

```c
struct _SUNLinearSolverContent_SPBCGS {
    int maxl;
    int pretype;
    int numiters;
    realtype resnorm;
    long int last_flag;
    ATimesFn ATimes;
    void* ATData;
    PSetupFn Psetup;
    PSolveFn Psolve;
    void* PData;
    N_Vector s1;
    N_Vector s2;
    N_Vector r;
    N_Vector r_star;
    N_Vector p;
    N_Vector q;
    N_Vector u;
    N_Vector Ap;
    N_Vector vtemp;
};
```

These entries of the content field contain the following information:

- \texttt{maxl} - number of 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.

9.14 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.

9.14.1 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.

9.14.2 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:
```
9.14 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 \( \leq 0 \) 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_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 9.1.1 – 9.1.3:

- 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 ≤ 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.

9.14.3 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 inter-operating 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_fsunlinsolspfqmrmr_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_fsunlinsolspfqmrmr_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.
9.14 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 NVVECTOR object has been initialized.
Allowable values for pretype and maxl are the same as for the C function
SUNLinSol_SPTFQMR.
Additionally, when using ARKODE with a non-identity mass matrix, the SUNINSOL_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
SUNINSOL_SPTFQMR object for mass matrix linear systems.
Arguments pretype (int*) flag indicating desired preconditioning type
maxl (int*) flag indicating number of iterations to allow
Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise.
See printed message for details in case of failure.
Notes This routine must be called after the NVVECTOR object has been initialized.
Allowable values for pretype and maxl are the same as for the C function
SUNLinSol_SPTFQMR.
The SUNLinSol_SPTFQMRSetPrecType and SUNLinSol_SPTFQMRSetMaxl routines also support Fortran
interfaces for the system and mass matrix solvers.

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 rou-
tine.

FSUNMASSSPTFQMRSETPRECTYPE
Call FSUNMASSSPTFQMRSETPRECTYPE(pretype, ier)
Description The function FSUNMASSSPTFQMRSETPRECTYPE can be called for Fortran programs to
change the type of preconditioning for mass matrix linear systems.
Arguments
The arguments are identical to `FSUNSPTFQMRSETPRECTYPE` above, except that `code` is not needed since mass matrix linear systems only arise in `ARKODE`.

Return value
`ier` is an `int` return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes
See `SUNLinSol_SPTFQMRSetPrecType` for complete further documentation of this routine.

---

### `FSUNSPTFQMRSETMAXL`

**Call**
`FSUNSPTFQMRSETMAXL(code, maxl, ier)`

**Description**
The function `FSUNSPTFQMRSETMAXL` can be called for Fortran programs to change the maximum number of iterations to allow.

**Arguments**
- `code` (`int*`) is an integer input specifying the solver id (1 for `CVODE`, 2 for `IDA`, 3 for `KINSOL`, and 4 for `ARKODE`).
- `maxl` (`int*`) the number of iterations to allow.

**Return value**
`ier` is an `int` return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

**Notes**
See `SUNLinSol_SPTFQMRSetMaxl` for complete further documentation of this routine.

---

### `FSUNMASSSPTFQMRSETMAXL`

**Call**
`FSUNMASSSPTFQMRSETMAXL(maxl, ier)`

**Description**
The function `FSUNMASSSPTFQMRSETMAXL` can be called for Fortran programs to change the type of preconditioning for mass matrix linear systems.

**Arguments**
The arguments are identical to `FSUNSPTFQMRSETMAXL` above, except that `code` is not needed since mass matrix linear systems only arise in `ARKODE`.

**Return value**
`ier` is an `int` return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

**Notes**
See `SUNLinSol_SPTFQMRSetMaxl` for complete further documentation of this routine.

---

### 9.14.4 SUNLinearSolver_SPTFQMR content

The `SUNLINSOL_SPTFQMR` module defines the `content` field of a `SUNLinearSolver` as the following structure:

```c
struct _SUNLinearSolverContent_SPTFQMR {
    int maxl;
    int pretype;
    int numiters;
    realtype resnorm;
    long int last_flag;
    ATimesFn ATimes;
    void* ATData;
    PSetupFn Psetup;
    PSolveFn Psolve;
    void* PData;
    N_Vector s1;
    N_Vector s2;
    N_Vector r_star;
    N_Vector q;
    N_Vector d;
    N_Vector v;
};
```
9.15 The SUNLinearSolver_PCG implementation

This section describes the SUNLINSOL implementation of the PCG (Preconditioned Conjugate Gradient [21]) 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.

9.15.1 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

$$\tilde{A}\tilde{x} = \tilde{b}$$

where

$$\tilde{A} = SP^{-1}AP^{-1}S,$$

$$\tilde{b} = SP^{-1}b,$$

$$\tilde{x} = S^{-1}Px.$$ (9.5)

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 $\|v\|_S = \sqrt{v^T S T S v}$, 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.

### 9.15.2 SUNLinearSolver_PCG functions

The SUNLINSOL_PCG module provides the following user-callable constructor for creating a SUNLinearSolver object.

```c
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
```
pretype (int) flag indicating whether to use preconditioning. Since the PCG algorithm is designed to only support symmetric preconditioning, then any of the pretype inputs PREC_LEFT (1), PREC_RIGHT (2), or PREC_BOTH (3) will result in use of the symmetric preconditioner; any other integer input will result in the default (no preconditioning).

maxl (int) the number of linear iterations to allow; values \( \leq 0 \) 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.

Although some SUNDIALS solvers are designed to only work with left preconditioning (IDA and IDAS) and others with only right preconditioning (KINSOL), PCG should only be used with these packages when the linear systems are known to be symmetric. Since the scaling of matrix rows and columns must be identical in a symmetric matrix, symmetric preconditioning should work appropriately even for packages designed with one-sided preconditioning in mind.

Deprecated Name For backward compatibility, the wrapper function SUNPCG with identical input and output arguments is also provided.

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

The SUNLINSOL_PCG module defines implementations of all “iterative” linear solver operations listed in Sections 9.1.1 – 9.1.3:

- SUNLinSolGetType_PCG
- SUNLinSolInitialize_PCG
- SUNLinSolSetATimes_PCG
- SUNLinSolSetPreconditioner_PCG
- SUNLinSolSetScalingVectors_PCG – since PCG only supports symmetric scaling, the second NVECTOR argument to this function is ignored
- SUNLinSolSetup_PCG
- SUNLinSolSolve_PCG
- SUNLinSolNumIters_PCG
- SUNLinSolResNorm_PCG
- SUNLinSolResid_PCG
- SUNLinSolLastFlag_PCG
- SUNLinSolSpace_PCG
- SUNLinSolFree_PCG

All of the listed operations are callable via the FORTRAN 2003 interface module by prepending an ‘F’ to the function name.

The SUNLINSOL_PCG module also defines the following additional user-callable functions.
**SUNLinSol_PCGSetPrecType**

Call

\[
\text{retval} = \text{SUNLinSol_PCGSetPrecType}(\text{LS, pretype});
\]

Description The function SUNLinSol_PCGSetPrecType updates the flag indicating use of preconditioning in the SUNLINSOL_PCG object.

Arguments

- **LS** (SUNLinearSolver) the SUNLINSOL_PCG object to update
- **pretype** (int) flag indicating use of preconditioning, allowed values match those discussed in SUNLinSol_PCG.

Return value This routine will return with one of the error codes SUNLS_ILL_INPUT (illegal pretype), SUNLS_MEM_NULL (S is NULL) or SUNLS_SUCCESS.

Deprecated Name For backward compatibility, the wrapper function SUNPCGSetPrecType with identical input and output arguments is also provided.

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

**SUNLinSol_PCGSetMaxl**

Call

\[
\text{retval} = \text{SUNLinSol_PCGSetMaxl}(\text{LS, maxl});
\]

Description The function SUNLinSol_PCGSetMaxl updates the number of linear solver iterations to allow.

Arguments

- **LS** (SUNLinearSolver) the SUNLINSOL_PCG object to update
- **maxl** (int) flag indicating the number of iterations to allow; values ≤ 0 will result in the default value (5)

Return value This routine will return with one of the error codes SUNLS_MEM_NULL (S is NULL) or SUNLS_SUCCESS.

Deprecated Name For backward compatibility, the wrapper function SUNPCGSetMaxl with identical input and output arguments is also provided.

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

### 9.15.3 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.
9.15 The SUNLinearSolver_PCG implementation

**FSUNPCGINIT**

Call

FSUNPCGINIT(code, pretype, maxl, ier)

Description

The function FSUNPCGINIT can be called for Fortran programs to create a SUNLIN-
sol_PCG object.

Arguments

- code (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).
- pretype (int*) flag indicating desired preconditioning type
- maxl (int*) flag indicating number of iterations to allow

Return value

ier is a return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes

This routine must be called after the NVECTOR object has been initialized.

Allowable values for pretype and maxl are the same as for the C function SUNLinSol_PCG.

Additionally, when using ARKODE with a non-identity mass matrix, the SUNLINSOL_PCG module includes a Fortran-callable function for creating a SUNLinearSolver mass matrix solver object.

**FSUNMASSPCGINIT**

Call

FSUNMASSPCGINIT(pretype, maxl, ier)

Description

The function FSUNMASSPCGINIT can be called for Fortran programs to create a SUNLIN-
sol_PCG object for mass matrix linear systems.

Arguments

- pretype (int*) flag indicating desired preconditioning type
- maxl (int*) flag indicating number of iterations to allow

Return value

ier is an int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes

This routine must be called after the NVECTOR object has been initialized.

Allowable values for pretype and maxl are the same as for the C function SUNLinSol_PCG.

The SUNLinSol_PCGSetPrecType and SUNLinSol_PCGSetMax1 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_PCGSetPrecType for complete further documentation of this routine.

**FSUNMASSPCGSETPRECTYPE**

Call

FSUNMASSPCGSETPRECTYPE(pretype, ier)

Description

The function FSUNMASSPCGSETPRECTYPE can be called for Fortran programs to change the type of preconditioning for mass matrix linear systems.

Arguments

The arguments are identical to FSUNPCGSETPRECTYPE above, except that code is not needed since mass matrix linear systems only arise in ARKODE.
Return value $\text{ier}$ is an integer return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes See SUNLinSol_PCGSetPrecType for complete further documentation of this routine.

### FSUNPCGSETMAXL

**Call**
FSUNPCGSETMAXL (code, maxl, ier)

**Description**
The function FSUNPCGSETMAXL can be called for Fortran programs to change the maximum number of iterations to allow.

**Arguments**
- **code** (int*) is an integer input specifying the solver id (1 for cvode, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).
- **maxl** (int*) the number of iterations to allow.

**Return value** $\text{ier}$ is an integer return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

**Notes**
See SUNLinSol_PCGSetMaxl for complete further documentation of this routine.

### FSUNMASSPCGSETMAXL

**Call**
FSUNMASSPCGSETMAXL (maxl, ier)

**Description**
The function FSUNMASSPCGSETMAXL can be called for Fortran programs to change the type of preconditioning for mass matrix linear systems.

**Arguments**
The arguments are identical to FSUNPCGSETMAXL above, except that code is not needed since mass matrix linear systems only arise in ARKODE.

**Return value** $\text{ier}$ is an integer return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

**Notes**
See SUNLinSol_PCGSetMaxl for complete further documentation of this routine.

### 9.15.4 SUNLinearSolver_PCG content

The SUNLINSOL_PCG module defines the content field of a SUNLinearSolver as the following structure:

```c
struct _SUNLinearSolverContent_PCG {
  int maxl;
  int pretype;
  int numiters;
  realtype resnorm;
  long int last_flag;
  ATimesFn ATimes;
  void* ATData;
  PSetupFn Psetup;
  PSolveFn Psolve;
  void* PData;
  N_Vector s;
  N_Vector r;
  N_Vector p;
  N_Vector z;
  N_Vector Ap;
};
```

These entries of the content field contain the following information:
- **maxl** - number of PCG iterations to allow (default is 5),
- **pretype** - flag for use of preconditioning (default is none),
numiters - number of iterations from the most-recent solve,
resnorm - final linear residual norm from the most-recent solve,
last_flag - last error return flag from an internal function,
ATimes - function pointer to perform $Av$ product,
ATData - pointer to structure for ATimes,
Psetup - function pointer to preconditioner setup routine,
Psolve - function pointer to preconditioner solve routine,
PData - pointer to structure for Psetup and Psolve,
s - vector pointer for supplied scaling matrix (default is NULL),
r - a NVECTOR which holds the preconditioned linear system residual,
p, z, Ap - NVECTORS used for workspace by the PCG algorithm.

9.16 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 10

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 10.1 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 10.1.1 – 10.1.3. This is followed by the definition of functions supplied to a nonlinear solver implementation in section 10.1.4. A table of nonlinear solver return codes is given in section 10.1.5. The SUNNonlinearSolver type and the generic SUNNONLINSOL module are defined in section 10.1.6. Section 10.1.7 describes how SUNNONLINSOL models interface with SUNDIALS integrators providing sensitivity analysis capabilities (CVODES and IDAS). Finally, section 10.1.8 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.

10.1 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.

10.1.1 SUNNonlinearSolver core functions

The core nonlinear solver functions consist of two required functions to get the nonlinear solver type (SUNNonlinsSolGetType) and solve the nonlinear system (SUNNonlinSolSolve). The remaining three functions for nonlinear solver initialization (SUNNonlinsSolInitialization), setup (SUNNonlinSolSetup), and destruction (SUNNonlinSolFree) are optional.
### SUNNonlinSolGetType

**Call**

```c
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**

```c
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**

```c
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**

```c
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` (*boolean*) a flag indicating that the integrator recommends for the linear solver setup function to be called.
10.1 The SUNNonlinearSolver API

The SUNNonlinearSolver API is 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.

10.1.2 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.

SUNNonlinSolFree

Call

retval = SUNNonlinSolFree(NLS);

Description The optional function SUNNonlinSolFree frees any memory allocated by the nonlinear
solver.

Arguments NLS (SUNNonlinearSolver) a SUNNONLINSOL object.

Return value The return value retval (of type int) should be zero for a successful call, and a negative
value for a failure. SUNNONLINSOL implementations that do not allocate data may set
this operation to NULL.

SUNNonlinSolSetSysFn

Call

retval = SUNNonlinSolSetSysFn(NLS, SysFn);

Description The required function SUNNonlinSolSetSysFn is used to provide the nonlinear solver
with the function defining the nonlinear system. This is the function $F(y)$ in $F(y) = 0$
for SUNNONLINEARSOLVER_ROOTFIND modules or $G(y)$ in $G(y) = y$ for
SUNNONLINEARSOLVER_FIXEDPOINT modules.

Arguments NLS (SUNNonlinearSolver) a SUNNONLINSOL object.
SysFn (SUNNonlinSolSysFn) the function defining the nonlinear system. See section 10.1.4 for the definition of
SUNNonlinSolSysFn.

Return value The return value retval (of type int) should be zero for a successful call, and a negative
value for a failure.

SUNNonlinSolSetLSetupFn

Call

retval = SUNNonlinSolSetLSetupFn(NLS, LSetupFn);

Description The optional function SUNNonlinSolSetLSetupFn is called by sundials integrators to
provide the nonlinear solver with access to its linear solver setup function.

Arguments NLS (SUNNonlinearSolver) a SUNNONLINSOL object.
LSetupFn (SUNNonlinSolLSetupFn) a wrapper function to the sundials integrator’s
linear solver setup function. See section 10.1.4 for the definition of
SUNNonlinSolLSetupFn.

Return value The return value retval (of type int) should be zero for a successful call, and a negative
value for a failure.

Notes The SUNNonlinLSetupFn function sets up the linear system $Ax = b$ where $A = \frac{\partial F}{\partial y}$ is
the linearization of the nonlinear residual function $F(y) = 0$ (when using SUNLINSOL
direct linear solvers) or calls the user-defined preconditioner setup function (when using
SUNLINSOL iterative linear solvers). SUNNONLINSOL implementations that do not require
solving this system, do not utilize SUNLINSOL linear solvers, or use SUNLINSOL linear
solvers that do not require setup may set this operation to NULL.
Description of the SUNNonlinearSolver module

SUNNonlinSolSetLSolveFn

Call

\[
\text{retval} = \text{SUNNonlinSolSetLSolveFn}(\text{NLS}, \text{LSolveFn});
\]

Description

The \textit{optional} function \texttt{SUNNonlinSolSetLSolveFn} is called by \textsc{sundials} integrators to provide the nonlinear solver with access to its linear solver solve function.

Arguments

\begin{itemize}
\item \texttt{NLS} (\texttt{SUNNonlinearSolver}) a \texttt{SUNNONLINSL} object
\item \texttt{LSolveFn} (\texttt{SUNNonlinSolLSolveFn}) a wrapper function to the \textsc{sundials} integrator’s linear solver solve function. See section 10.1.4 for the definition of \texttt{SUNNonlinSolLSolveFn}.
\end{itemize}

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 \(F(y) = 0\). \texttt{SUNNONLINSL} implementations that do not require solving this system or do not use \texttt{SUNLINSOL} linear solvers may set this operation to \texttt{NULL}.

SUNNonlinSolSetConvTestFn

Call

\[
\text{retval} = \text{SUNNonlinSolSetConvTestFn}(\text{NLS}, \text{CTestFn});
\]

Description

The \textit{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 \textsc{sundials} integrators to define their nonlinear convergence criteria, but may be replaced by the user.

Arguments

\begin{itemize}
\item \texttt{NLS} (\texttt{SUNNonlinearSolver}) a \texttt{SUNNONLINSL} object.
\item \texttt{CTestFn} (\texttt{SUNNonlineSolConvTestFn}) a \textsc{sundials} integrator’s nonlinear solver convergence test function. See section 10.1.4 for the definition of \texttt{SUNNonlinSolConvTestFn}.
\end{itemize}

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

\texttt{SUNNONLINSL} implementations utilizing their own convergence test criteria may set this function to \texttt{NULL}.

SUNNonlinSolSetMaxIters

Call

\[
\text{retval} = \text{SUNNonlinSolSetMaxIters}(\text{NLS}, \text{maxiters});
\]

Description

The \textit{optional} function \texttt{SUNNonlinSolSetMaxIters} sets the maximum number of nonlinear solver iterations. This is typically called by \textsc{sundials} integrators to define their default iteration limit, but may be adjusted by the user.

Arguments

\begin{itemize}
\item \texttt{NLS} (\texttt{SUNNonlinearSolver}) a \texttt{SUNNONLINSL} object.
\item \texttt{maxiters} (\texttt{int}) the maximum number of nonlinear iterations.
\end{itemize}

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).

10.1.3 SUNNonlinearSolver get functions

The following get functions allow \textsc{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 \textsc{sundials} integrator or by the \texttt{ARKODE} and \texttt{CVODE} linear solver interfaces. Otherwise, \texttt{SUNNonlinSolGetCurIter} is optional.
10.1 The SUNNonlinearSolver API

SUNNonlinSolGetNumIters

Call

\texttt{retval = SUNNonlinSolGetNumIters(NLS, numiters);} \\

Description

The \textit{optional} function \texttt{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

\- \texttt{NLS} (SUNNonlinearSolver) a SUNNONLINSOL object
\- \texttt{numiters} (long int*) the total number of nonlinear solver 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.

SUNNonlinSolGetCurIter

Call

\texttt{retval = SUNNonlinSolGetCurIter(NLS, iter);} \\

Description

The function \texttt{SUNNonlinSolGetCurIter} returns the iteration index of the current nonlinear solve. This function is \textit{required} when using SUNDIALS integrator-provided convergence tests or when using a SUNLINSOL spils linear solver; otherwise it is \textit{optional}.

Arguments

\- \texttt{NLS} (SUNNonlinearSolver) a SUNNONLINSOL object
\- \texttt{iter} (int*) the nonlinear solver iteration in the current solve starting from zero.

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.

SUNNonlinSolGetNumConvFails

Call

\texttt{retval = SUNNonlinSolGetNumConvFails(NLS, nconvfails);} \\

Description

The \textit{optional} function \texttt{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

\- \texttt{NLS} (SUNNonlinearSolver) a SUNNONLINSOL object
\- \texttt{nconvfails} (long int*) the total number of nonlinear solver convergence failures.

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.

10.1.4 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 \texttt{sundials/sundials_nonlinearsolver.h}, and are described below.

SUNNonlinSolSysFn

Definition

\texttt{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 \( y \) \textit{must} be left unchanged.

Arguments

\- \texttt{y} is the state vector at which the nonlinear system should be evaluated.
\- \texttt{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 SUNNonlinSolLSetupFn 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). SUNNONLINBSOL 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 SUNNonlinSolLSolveFn function solves the linear system **Ax** = **b** where **A** = \( \frac{\partial F}{\partial y} \) is the linearization of the nonlinear residual function **F**(**y**) = 0. SUNNONLINBSOL 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.
10.1 The SUNNonlinearSolver API

Arguments

- NLS is the SUNNONLINSOL object.
- y is the current nonlinear iterate.
- del is the difference between the current and prior nonlinear iterates.
- tol is the nonlinear solver tolerance.
- ewt is the weight vector used in computing weighted norms.
- mem is the SUNDIALS integrator memory structure.

Return value

The return value of this routine will be a negative value if an unrecoverable error occurred or one of the following:

- SUN_NLS_SUCCESS the iteration is converged.
- SUN_NLS_CONTINUE the iteration has not converged, keep iterating.
- SUN_NLS_CONV_RECVR the iteration appears to be diverging, try to recover.

Notes

The tolerance passed to this routine by SUNDIALS integrators is the tolerance in a weighted root-mean-squared norm with error weight vector ewt. SUNNONLINSOL modules utilizing their own convergence criteria may ignore these functions.

10.1.5 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 10.1. 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_RECVR</td>
<td>2</td>
<td>the nonlinear solver appears to be diverging, try to recover</td>
</tr>
<tr>
<td>SUN_NLS_MEM_NULL</td>
<td>-1</td>
<td>a memory argument is NULL</td>
</tr>
<tr>
<td>SUN_NLS_MEM_FAIL</td>
<td>-2</td>
<td>a memory access or allocation failed</td>
</tr>
<tr>
<td>SUN_NLS_Ill_INPUT</td>
<td>-3</td>
<td>an illegal input option was provided</td>
</tr>
</tbody>
</table>

10.1.6 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 10.1.1 – 10.1.3. 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));
}
```

### 10.1.7 Usage with sensitivity enabled integrators

When used with SUNDIALS packages that support sensitivity analysis capabilities (e.g., CVODES and IDAS) a special NVECTOR module is used to interface with SUNNONLINSOL modules for solves involving sensitivity vectors stored in an NVECTOR array. As described below, the NVECTOR_SENSWRAPPER module is an NVECTOR implementation where the vector content is an NVECTOR array. This wrapper vector allows SUNNONLINSOL modules to operate on data stored as a collection of vectors.

For all SUNDIALS-provided SUNNONLINSOL modules a special constructor wrapper is provided so users do not need to interact directly with the NVECTOR_SENSWRAPPER module. These constructors follow the naming convention SUNNonlinSol_***Sens(count,...) where *** is the name of the SUNNONLINSOL module, count is the size of the vector wrapper, and ... are the module-specific constructor arguments.

### The NVECTOR_SENSWRAPPER module

This section describes the NVECTOR_SENSWRAPPER implementation of an NVECTOR. To access the NVECTOR_SENSWRAPPER module, include the header file sundials/sundials_nvector_senswrapper.h.

The NVECTOR_SENSWRAPPER module defines an N_Vector implementing all of the standard vectors operations defined in Table 7.2 but with some changes to how operations are computed in order to accommodate operating on a collection of vectors.
1. Element-wise vector operations are computed on a vector-by-vector basis. For example, the linear sum of two wrappers containing \( n_v \) vectors of length \( n \), \( N_{\text{VLinearSum}}(a,x,b,y,z) \), is computed as
\[
z_{j,i} = ax_{j,i} + by_{j,i}, \quad i = 0, \ldots, n - 1, \quad j = 0, \ldots, n_v - 1.
\]

2. The dot product of two wrappers containing \( n_v \) vectors of length \( n \) is computed as if it were the dot product of two vectors of length \( nn_v \). Thus \( d = N_{\text{VDotProd}}(x,y) \) is
\[
d = \sum_{j=0}^{n_v-1} \sum_{i=0}^{n-1} x_{j,i}y_{j,i}.
\]

3. All norms are computed as the maximum of the individual norms of the \( n_v \) vectors in the wrapper. For example, the weighted root mean square norm \( m = N_{\text{VWrmsNorm}}(x, w) \) is
\[
m = \max_j \left( \frac{1}{n} \sum_{i=0}^{n-1} (x_{j,i}w_{j,i})^2 \right)^{\frac{1}{2}}
\]

To enable usage alongside other \text{nvector} modules the \text{nvector_senswrapper} functions implementing vector operations have \_SensWrapper appended to the generic vector operation name.

The \text{nvector_senswrapper} module provides the following constructors for creating an \text{nvector_senswrapper}:

**\text{N\_VNewEmpty\_SensWrapper}**

Call
\[
w = \text{N\_VNewEmpty\_SensWrapper}(\text{count});
\]

Description The function \text{N\_VNewEmpty\_SensWrapper} creates an empty \text{nvector_senswrapper} wrapper with space for \text{count} vectors.

Arguments
\text{count} (int) the number of vectors the wrapper will contain.

Return value The return value \( w \) (of type \text{N\_Vector}) will be a \text{nvector} object if the constructor exits successfully, otherwise \( w \) will be \text{NULL}.

**\text{N\_VNew\_SensWrapper}**

Call
\[
w = \text{N\_VNew\_SensWrapper}(\text{count}, y);
\]

Description The function \text{N\_VNew\_SensWrapper} creates an \text{nvector_senswrapper} wrapper containing \text{count} vectors cloned from \( y \).

Arguments
\text{count} (int) the number of vectors the wrapper will contain.
\( y \) (\text{N\_Vector}) the template vectors to use in creating the vector wrapper.

Return value The return value \( w \) (of type \text{N\_Vector}) will be a \text{nvector} object if the constructor exits successfully, otherwise \( w \) will be \text{NULL}.

The \text{nvector_senswrapper} implementation of the \text{nvector} module defines the \text{content} field of the \text{N\_Vector} to be a structure containing an \text{N\_Vector} array, the number of vectors in the vector array, and a boolean flag indicating ownership of the vectors in the vector array.

\textbf{struct } _\text{N\_VectorContent\_SensWrapper} \{ \n\text{N\_Vector}\* vecs;
int nvecs;
bool own_vecs;
\};

The following macros are provided to access the content of an \text{nvector_senswrapper} vector.
• NV_CONTENT_SW(v) - provides access to the content structure
• NV_VECS_SW(v) - provides access to the vector array
• NV_NVECS_SW(v) - provides access to the number of vectors
• NV_OWN_VECS_SW(v) - provides access to the ownership flag
• NV_VEC_SW(v,i) - provides access to the i-th vector in the vector array

10.1.8 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 10.1.1 – 10.1.3. 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.

10.2 The SUNNonlinearSolver_Newton implementation

This section describes the SUNNONLINSOL implementation of Newton’s method. To access the SUNNONLINSOL_NEWTON module, include the header file sunnonlinsol/sunnonlinsol_newton.h. We note that the SUNNONLINSOL_NEWTON module is accessible from SUNDIALS integrators without separately linking to the lib sundials_sunnnonlinsolnewton module library.

10.2.1 SUNNonlinearSolver_Newton description

To find the solution to
\[ F(y) = 0 \]  
(10.1)
given an initial guess \( y^{(0)} \), Newton’s method computes a series of approximate solutions
\[ y^{(m+1)} = y^{(m)} + \delta^{(m+1)} \]  
(10.2)
where \( m \) is the Newton iteration index, and the Newton update \( \delta^{(m+1)} \) is the solution of the linear system
\[ A(y^{(m)})\delta^{(m+1)} = -F(y^{(m)}), \]  
(10.3)
in which \( A \) is the Jacobian matrix
\[ A \equiv \partial F / \partial y. \]  
(10.4)
Depending on the linear solver used, the SUNNONLINSOL_NEWTON module will employ either a Modified Newton method, or an Inexact Newton method [4, 7, 15, 17, 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 SUNNonlinSolLSetupFn function are made infrequently to amortize the increased cost of matrix operations (updating $A$ and its factorization within direct linear solvers, or updating the preconditioner within iterative linear solvers). Specifically, SUNNONLINсол Newton will call the SUNNonlinSolLSetupFn function in two instances:

(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, SUNNONLINсол Newton will set $jbad$ to SUNTRUE before calling the SUNNonlinSolLSetupFn 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 SUNNONLINсол 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 SUNNONLINсол Newton object to the integrator.

### 10.2.2 SUNNonlinearSolver_Newton functions

The SUNNONLINсол Newton module provides the following constructors for creating a SUNNonlinearSolver object.

```c
SUNNonlinSol Newton
```

Call

```c
NLS = SUNNonlinSol Newton(y);
```

Description

The function SUNNonlinSol Newton creates a SUNNonlinearSolver object for use with SUNDIALS integrators to solve nonlinear systems of the form $F(y) = 0$ using Newton’s method.

Arguments

- `y` (N_Vector) a template for cloning vectors needed within the solver.

Return value

The return value NLS (of type SUNNonlinearSolver) will be a SUNNONLINсол 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.

```c
SUNNonlinSol NewtonSens
```

Call

```c
NLS = SUNNonlinSol NewtonSens(count, y);
```

Description

The function SUNNonlinSol NewtonSens creates a SUNNonlinearSolver object for use with SUNDIALS sensitivity enabled integrators (CVODES and IDAS) to solve nonlinear systems of the form $F(y) = 0$ using Newton’s method.

Arguments

- `count` (int) the number of vectors in the nonlinear solve. When integrating a system containing $Ns$ sensitivities the value of `count` is:
  - $Ns+1$ if using a simultaneous corrector approach.
  - $Ns$ if using a staggered corrector approach.
- `y` (N_Vector) a template for cloning vectors needed within the solver.

Return value

The return value NLS (of type SUNNonlinearSolver) will be a SUNNONLINсол 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 10.1.1 – 10.1.3 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 10.1.1 – 10.1.3 should be called in favor of the SUNNONLINSOL\_NEWTON-specific implementations.

The SUNNONLINSOL\_NEWTON module also defines the following additional user-callable function.

\textbf{SUNNonlinSolGetSysFn\_Newton}

\begin{verbatim}
call retval = SUNNonlinSolGetSysFn\_Newton(NLS, SysFn);
\end{verbatim}

\textbf{Description} The function SUNNonlinSolGetSysFn\_Newton returns the residual function that defines the nonlinear system.

\textbf{Arguments} 
- \textbf{NLS} (SUNNonlinearSolver) a SUNNONLINSOL object
- \textbf{SysFn} (SUNNonlinSolSysFn*) the function defining the nonlinear system.

\textbf{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.

\textbf{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 \texttt{SysFn}.

\textbf{F2003 Name} This function is callable as \texttt{FSUNNonlinSolGetSysFn\_Newton} when using the Fortran 2003 interface module.

10.2.3 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.

\textbf{FORTRAN 2003 interface module}

The \texttt{fsunnonlinsol\_newton\_mod} FORTRAN module defines interfaces to all SUNNONLINSOL\_NEWTON C functions using the intrinsic iso\_c\_binding module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading ‘F’. For example, the function \texttt{SUNNonlinSol\_Newton} is interfaced as \texttt{FSUNNonlinSol\_Newton}.

The FORTRAN 2003 SUNNONLINSOL\_NEWTON interface module can be accessed with the \texttt{use} statement, i.e. \texttt{use fsunnonlinsol\_newton\_mod}, and linking to the library \texttt{libsundials\_fsunnonlinsol\_newton\_mod\_lib} in addition to the C library. For details on where the library and module file \texttt{fsunnonlinsol\_newton\_mod\_mod} are installed see Appendix A. We note that the module is accessible from the FORTRAN 2003 SUNDIALS integrators \textit{without} separately linking to the \texttt{libsundials\_fsunnonlinsol\_newton\_mod\_library}.

\textbf{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.

\textbf{FSUNNEWTONINIT}

\begin{verbatim}
call FSUNNEWTONINIT(code, ier);
\end{verbatim}

\textbf{Description} The function FSUNNEWTONINIT can be called for Fortran programs to create a SUNNonlinearSolver object for use with SUNDIALS integrators to solve nonlinear systems of the form $F(y) = 0$ with Newton’s method.
10.3 The SUNNonlinearSolver_FixedPoint implementation

Arguments  code (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, and 4 for ARKODE).

Return value  ier is a return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

10.2.4 SUNNonlinearSolver_Newton content

The SUNNONLINSOL_NEWTON module defines the content field of a SUNNonlinearSolver as the following structure:

```
struct _SUNNonlinearSolverContent_Newton {
    SUNNonlinSolSysFn Sys;
    SUNNonlinSolLSetupFn LSetup;
    SUNNonlinSolLSolveFn LSolve;
    SUNNonlinSolConvTestFn CTest;
    N_Vector delta;
    booleantype jcur;
    int curiter;
    int maxiters;
    long int niters;
    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
- **niters** - the total number of nonlinear iterations across all solves.
- **nconvfails** - the total number of nonlinear convergence failures across all solves.

10.3 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 lib sundials_sunnonsolfixedpoint module library.

10.3.1 SUNNonlinearSolver_FixedPoint description

To find the solution to

\[ G(y) = y \]  

(10.5)

given an initial guess \( y^{(0)} \), the fixed point iteration computes a series of approximate solutions

\[ y^{(n+1)} = G(y^{(n)}) \]  

(10.6)
where \( n \) is the iteration index. The convergence of this iteration may be accelerated using Anderson’s method [3, 45, 18, 35]. With Anderson acceleration using subspace size \( m \), the series of approximate solutions can be formulated as the linear combination

\[
y(n+1) = \sum_{i=0}^{m_n} \alpha_i^{(n)} G(y(n-m_n+i))
\]

(10.7)

where \( m_n = \min\{m, n\} \) and the factors

\[
\alpha^{(n)} = (\alpha_0^{(n)}, \ldots, \alpha_{m_n}^{(n)})
\]

(10.8)
solve the minimization problem \( \min_{\alpha} \| F_n \alpha^T \|_2 \) under the constraint that \( \sum_{i=0}^{m_n} \alpha_i = 1 \) where

\[
F_n = (f_{n-m_n}, \ldots, f_n)
\]

(10.9)

with \( f_i = G(y^{(i)}) - y^{(i)} \). Due to this constraint, in the limit of \( m = 0 \) the accelerated fixed point iteration formula (10.7) simplifies to the standard fixed point iteration (10.6).

Following the recommendations made in [45], the SUNNONLINSOL_FIXEDPOINT implementation computes the series of approximate solutions as

\[
y(n+1) = G(y^{(n)}) - \sum_{i=0}^{m_n-1} \gamma_i^{(n)} \Delta g_{n-m_n+i}
\]

(10.10)

with \( \Delta g_i = G(y^{(i+1)}) - G(y^{(i)}) \) and where the factors

\[
\gamma^{(n)} = (\gamma_0^{(n)}, \ldots, \gamma_{m_n-1}^{(n)})
\]

(10.11)
solve the unconstrained minimization problem \( \min_{\gamma} \| f_n - \Delta F_n \gamma^T \|_2 \) where

\[
\Delta F_n = (\Delta f_{n-m_n}, \ldots, \Delta f_{n-1})
\]

(10.12)

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.

### 10.3.2 SUNNonlinearSolver_FixedPoint functions

The SUNNONLINSOL_FIXEDPOINT module provides the following constructors for creating a SUNNonlinearSolver object.

**SUNNonlinSol_FixedPoint**

Call

```c
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.
10.3 The SUNNonlinearSolver_FixedPoint implementation

**SUNNonlinSol_FixedPointSens**

**Call**

\[ \text{NLS} = \text{SUNNonlinSol_FixedPointSens}(\text{count}, y, m); \]

**Description**

The function SUNNonlinSol_FixedPointSens creates a SUNNonlinearSolver object for use with SUNDIALS sensitivity enabled integrators (CVODES and IDAS) to solve nonlinear systems of the form \( G(y) = y \).

**Arguments**

- `count` (**int**) the number of vectors in the nonlinear solve. When integrating a system containing \( N_s \) sensitivities the value of `count` is:
  - \( N_s + 1 \) if using a *simultaneous* corrector approach.
  - \( N_s \) if using a *staggered* corrector approach.
- `y` (**N_Vector**) a template for cloning vectors needed within the solver.
- `m` (**int**) the number of acceleration vectors to use.

**Return value**

The return value `NLS` (of type **SUNNonlinearSolver**) will be a SUNNONLINSOL object if the constructor exits successfully, otherwise `NLS` will be **NULL**.

**F2003 Name**

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

Since the accelerated fixed point iteration (10.6) does not require the setup or solution of any linear systems, the SUNNONLINSOL_FIXEDPOINT module implements all of the functions defined in sections 10.1.1 – 10.1.3 except for the SUNNonlinSolSetup, SUNNonlinSolSetLSetupFn, and SUNNonlinSolSetLSolveFn functions, that are set to **NULL**. The SUNNONLINSOL_FIXEDPOINT functions have the same names as those defined by the generic SUNNONLINSOL API with _FixedPoint appended to the function name. Unless using the SUNNONLINSOL_FIXEDPOINT module as a standalone nonlinear solver the generic functions defined in sections 10.1.1 – 10.1.3 should be called in favor of the SUNNONLINSOL_FIXEDPOINT-specific implementations.

The SUNNONLINSOL_FIXEDPOINT module also defines the following additional user-callable function.

**SUNNonlinSolGetSysFn_FixedPoint**

**Call**

\[ \text{retval} = \text{SUNNonlinSolGetSysFn_FixedPoint}(\text{NLS}, \text{SysFn}); \]

**Description**

The function SUNNonlinSolGetSysFn_FixedPoint returns the fixed-point function that defines the nonlinear system.

**Arguments**

- `NLS` (**SUNNonlinearSolver**) a SUNNONLINSOL object
- `SysFn` (**SUNNonlinSolSysFn**) the function defining the nonlinear system.

**Return value**

The return value `retval` (of type **int**) should be zero for a successful call, and a negative value for a failure.

**Notes**

This function is intended for users that wish to evaluate the fixed-point function in a custom convergence test function for the SUNNONLINSOL_FIXEDPOINT module. We note that SUNNONLINSOL_FIXEDPOINT will not leverage the results from any user calls to `SysFn`.

**F2003 Name**

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

10.3.3 SUNNonlinearSolver_FixedPoint Fortran interfaces

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

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

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

10.3.4 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;
```

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
- **niter** - the total number of nonlinear iterations across all solves.
- **nconvfails** - the total number of nonlinear convergence failures across all solves.
- **m** - number of acceleration vectors.

If Anderson acceleration is requested (i.e., $m > 0$ in the call to SUNNonlinSol_FixedPoint), then the following items are also allocated within the content field:

- **imap** - index array used in acceleration algorithm (length $m$)
- **R** - small matrix used in acceleration algorithm (length $m \times m$)
- **gamma** - small vector used in acceleration algorithm (length $m$)
- **cvals** - small vector used in acceleration algorithm (length $m+1$)
- **df** - array of N_Vectors used in acceleration algorithm (length $m$)
- **dg** - array of N_Vectors used in acceleration algorithm (length $m$)
- **q** - array of N_Vectors used in acceleration algorithm (length $m$)
- **Xvecs** - N_Vector pointer array used in acceleration algorithm (length $m+1$)
- **fold** - N_Vector used in acceleration algorithm
- **gold** - N_Vector used in acceleration algorithm
Appendix A

SUNDIALS Package Installation Procedure

The installation of any SUNDIALS package is accomplished by installing the SUNDIALS suite as a whole, according to the instructions that follow. The same procedure applies whether or not the downloaded file contains one or all solvers in SUNDIALS.

The SUNDIALS suite (or individual solvers) are distributed as compressed archives (.tar.gz). The name of the distribution archive is of the form solver-x.y.z.tar.gz, where solver is one of: sundials, cvode, cvodes, arkode, ida, idas, or kinsol, and x.y.z represents the version number (of the SUNDIALS suite or of the individual solver). To begin the installation, first uncompress and expand the sources, by issuing

% tar xzf solver-x.y.z.tar.gz

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

Starting with version 2.6.0 of SUNDIALS, CMake is the only supported method of installation. The explanations of the installation procedure begins with a few common observations:

- The remainder of this chapter will follow these conventions:
  - solverdir is the directory solver-x.y.z created above; i.e., the directory containing the SUNDIALS sources.
  - builddir is the (temporary) directory under which SUNDIALS is built.
  - instdir is the directory under which the SUNDIALS exported header files and libraries will be installed. Typically, header files are exported under a directory instdir/include while libraries are installed under instdir/CMAKE_INSTALL_LIBDIR, with instdir and CMAKE_INSTALL_LIBDIR specified at configuration time.

- For SUNDIALS CMake-based installation, in-source builds are prohibited; in other words, the build directory builddir can not be the same as solverdir and such an attempt will lead to an error. This prevents “polluting” the source tree and allows efficient builds for different configurations and/or options.

- The installation directory instdir can not be the same as the source directory solverdir.

- By default, only the libraries and header files are exported to the installation directory instdir. If enabled by the user (with the appropriate toggle for CMake), the examples distributed with SUNDIALS will be built together with the solver libraries but the installation step will result in exporting (by default in a subdirectory of the installation directory) the example sources and sample outputs together with automatically generated configuration files that reference the installed SUNDIALS headers and libraries. As such, these configuration files for the SUNDIALS examples can be used as “templates” for your own problems. CMake installs CMakeLists.txt files
and also (as an option available only under Unix/Linux) Makefile files. Note this installation approach also allows the option of building the SUNDIALS examples without having to install them. (This can be used as a sanity check for the freshly built libraries.)

- Even if generation of shared libraries is enabled, only static libraries are created for the FCMIX modules. (Because of the use of fixed names for the Fortran user-provided subroutines, FCMIX shared libraries would result in "undefined symbol" errors at link time.)

A.1 CMake-based installation

CMake-based installation provides a platform-independent build system. CMake can generate Unix and Linux Makefiles, as well as KDevelop, Visual Studio, and (Apple) XCode project files from the same configuration file. In addition, CMake also provides a GUI front end and which allows an interactive build and installation process.

The SUNDIALS build process requires CMake version 3.1.3 or higher and a working C compiler. On Unix-like operating systems, it also requires Make (and curses, including its development libraries, for the GUI front end to CMake, ccmake), while on Windows it requires Visual Studio. CMake is continually adding new features, and the latest version can be downloaded from http://www.cmake.org. Build instructions for CMake (only necessary for Unix-like systems) can be found on the CMake website. Once CMake is installed, Linux/Unix users will be able to use ccmake, while Windows users will be able to use CMakeSetup.

As previously noted, when using CMake to configure, build and install SUNDIALS, it is always required to use a separate build directory. While in-source builds are possible, they are explicitly prohibited by the SUNDIALS CMake scripts (one of the reasons being that, unlike autotools, CMake does not provide a make distclean procedure and it is therefore difficult to clean-up the source tree after an in-source build). By ensuring a separate build directory, it is an easy task for the user to clean-up all traces of the build by simply removing the build directory. CMake does generate a make clean which will remove files generated by the compiler and linker.

A.1.1 Configuring, building, and installing on Unix-like systems

The default CMake configuration will build all included solvers and associated examples and will build static and shared libraries. The instdir defaults to /usr/local and can be changed by setting the CMAKE_INSTALL_PREFIX variable. Support for FORTRAN and all other options are disabled.

CMake can be used from the command line with the cmake command, or from a curses-based GUI by using the ccmake command. Examples for using both methods will be presented. For the examples shown it is assumed that there is a top level SUNDIALS directory with appropriate source, build and install directories:

```
% mkdir (...)sundials/instdir
% mkdir (...)sundials/builddir
% cd (...)sundials/builddir
```

Building with the GUI

Using CMake with the GUI follows this general process:

- Select and modify values, run configure (c key)
- New values are denoted with an asterisk
- To set a variable, move the cursor to the variable and press enter
  - If it is a boolean (ON/OFF) it will toggle the value
  - If it is string or file, it will allow editing of the string
For file and directories, the <tab> key can be used to complete

- Repeat until all values are set as desired and the generate option is available (g key)
- Some variables (advanced variables) are not visible right away
- To see advanced variables, toggle to advanced mode (t key)
- To search for a variable press / key, and to repeat the search, press the n key

To build the default configuration using the GUI, from the builddir enter the ccmake command and point to the solverdir:

```bash
% 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
- EXAMPLES_INSTALL_PATH

as shown in figure A.2.

Pressing the (g key) will generate makefiles including all dependencies and all rules to build sundials on this system. Back at the command prompt, you can now run:
% make

To install SUNDIALS in the installation directory specified in the configuration, simply run:

% make install

Building from the command line

Using CMake from the command line is simply a matter of specifying CMake variable settings with the `cmake` command. The following will build the default configuration:

```
% cmake -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir "
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples 
> ../solverdir
% make
% make install
```

A.1.2 Configuration options (Unix/Linux)

A complete list of all available options for a CMake-based SUNDIALS configuration is provided below. Note that the default values shown are for a typical configuration on a Linux system and are provided as illustration only.
A.1 CMake-based installation

**BLAS_ENABLE** - Enable BLAS support
    Default: OFF
    Note: Setting this option to ON will trigger additional CMake options. See additional information on building with BLAS enabled in A.1.4.

**BLAS_LIBRARIES** - BLAS library
    Default: `/usr/lib/libblas.so`
    Note: CMake will search for libraries in your `LD_LIBRARY_PATH` prior to searching default system paths.

**BUILD_ARKODE** - Build the ARKODE library
    Default: ON

**BUILD_CVODE** - Build the CVODE library
    Default: ON

**BUILD_CVODES** - Build the CVODES library
    Default: ON

**BUILD_IDA** - Build the IDA library
    Default: ON

**BUILD_IDAS** - Build the IDAS library
    Default: ON

**BUILD_KINSOL** - Build the KINSOL library
    Default: ON

**BUILD_SHARED_LIBS** - Build shared libraries
    Default: ON

**BUILD_STATIC_LIBS** - Build static libraries
    Default: ON

**CMAKE_BUILD_TYPE** - Choose the type of build, options are: None (CMAKE_C_FLAGS used), Debug, Release, RelWithDebInfo, and MinSizeRel
    Default:
    Note: Specifying a build type will trigger the corresponding build type specific compiler flag options below which will be appended to the flags set by CMAKE_<language>_FLAGS.

**CMAKE_C_COMPILER** - C compiler
    Default: `/usr/bin/cc`

**CMAKE_C_FLAGS** - Flags for C compiler
    Default:

**CMAKE_C_FLAGS_DEBUG** - Flags used by the C compiler during debug builds
    Default: `-g`

**CMAKE_C_FLAGS_MINSIZEREL** - Flags used by the C compiler during release minsize builds
    Default: `-Os -DNDEBUG`

**CMAKE_C_FLAGS_RELEASE** - Flags used by the C compiler during release builds
    Default: `-O3 -DNDEBUG`

**CMAKE_CXX_COMPILER** - C++ compiler
    Default: `/usr/bin/c++`
    Note: A C++ compiler (and all related options) are only triggered if C++ examples are enabled (EXAMPLES_ENABLE_CXX is ON). All Sundials solvers can be used from C++ applications by default without setting any additional configuration options.
CMAKE_CXX_FLAGS - Flags for C++ compiler
Default:

CMAKE_CXX_FLAGS_DEBUG - Flags used by the C++ compiler during debug builds
Default: -g

CMAKE_CXX_FLAGS_MINSIZEREL - Flags used by the C++ compiler during release minsize builds
Default: -Os -DNDEBUG

CMAKE_CXX_FLAGS_RELEASE - Flags used by the C++ compiler during release builds
Default: -O3 -DNDEBUG

CMAKE_Fortran_COMPILER - Fortran compiler
Default: /usr/bin/gfortran
Note: Fortran support (and all related options) are triggered only if either Fortran-C support is enabled (FCMIX_ENABLE is ON) or BLAS/LAPACK support is enabled (BLAS_ENABLE or LAPACK_ENABLE is ON).

CMAKE_Fortran_FLAGS - Flags for Fortran compiler
Default:

CMAKE_Fortran_FLAGS_DEBUG - Flags used by the Fortran compiler during debug builds
Default: -g

CMAKE_Fortran_FLAGS_MINSIZEREL - Flags used by the Fortran compiler during release minsize builds
Default: -Os

CMAKE_Fortran_FLAGS_RELEASE - Flags used by the Fortran compiler during release builds
Default: -O3

CMAKE_INSTALL_PREFIX - Install path prefix, prepended onto install directories
Default: /usr/local
Note: The user must have write access to the location specified through this option. Exported SUNDIALS header files and libraries will be installed under subdirectories include and CMAKE_INSTALL_LIBDIR of CMAKE_INSTALL_PREFIX, respectively.

CMAKE_INSTALL_LIBDIR - Library installation directory
Default:
Note: This is the directory within CMAKE_INSTALL_PREFIX that the SUNDIALS libraries will be installed under. The default is automatically set based on the operating system using the GNUInstallDirs CMake module.

Fortran_INSTALL_MODDIR - Fortran module installation directory
Default: fortran

CUDA_ENABLE - Build the SUNDIALS CUDA vector module.
Default: OFF

EXAMPLES_ENABLE_C - Build the SUNDIALS C examples
Default: ON

EXAMPLES_ENABLE_CUDA - Build the SUNDIALS CUDA examples
Default: OFF
Note: You need to enable CUDA support to build these examples.

EXAMPLES_ENABLE_CXX - Build the SUNDIALS C++ examples
Default: OFF unless Trilinos_ENABLE is ON.

EXAMPLES_ENABLE_F77 - Build the SUNDIALS Fortran77 examples
Default: ON (if F77_INTERFACE_ENABLE is ON)
A.1 CMake-based installation

**EXAMPLES**

- **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 information
  on building with LAPACK enabled in A.1.4.

- **LAPACK_LIBRARIES** - LAPACK (and BLAS) libraries
  Default: /usr/lib/liblapack.so;/usr/lib/libblas.so
  Note: CMake will search for libraries in your LD_LIBRARY_PATH prior to searching default system
  paths.

- **MPI_ENABLE** - Enable MPI support (build the parallel nvector).
  Default: OFF
  Note: Setting this option to ON will trigger several additional options related to MPI.

- **MPI_C_COMPILER** - mpicc program
  Default:
MPI_CXX_COMPILER - mpicxx program
   Default:
   Note: This option is triggered only if MPI is enabled (MPI_ENABLE is ON) and C++ examples are enabled (EXAMPLES_ENABLE_CXX is ON). All SUNDIALS solvers can be used from C++ MPI applications by default without setting any additional configuration options other than MPI_ENABLE.

MPI_Fortran_COMPILER - mpif77 or mpif90 program
   Default:
   Note: This option is triggered only if MPI is enabled (MPI_ENABLE is ON) 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.
A.1 CMake-based installation

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

SUPERLUMT_ENABLE - Enable SuperLU\_MT support
Default: OFF
Note: See additional information on building with SuperLU\_MT enabled in A.1.4.

SUPERLUMT_INCLUDE_DIR - Path to SuperLU\_MT header files (typically SRC directory)

SUPERLUMT_LIBRARY_DIR - Path to SuperLU\_MT installed library files

SUPERLUMT_THREAD_TYPE - Must be set to Pthread or OpenMP
Default: Pthread

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

TPL_ENABLE_LAPACK - Enable LAPACK support
   Default: OFF
   SUNDIALS equivalent: LAPACK_ENABLE

TPL_ENABLE_SUPERLUMT - Enable SuperLU_MT support
   Default: OFF
   SUNDIALS equivalent: SUPERLUMT_ENABLE

TPL_HYPRE_INCLUDE_DIRS - Path to hYPRE header files
   SUNDIALS equivalent: HYPRE_INCLUDE_DIR

TPL_HYPRE_LIBRARIES - hYPRE library
   SUNDIALS equivalent: N/A

TPL_KLU_INCLUDE_DIRS - Path to KLU header files
   SUNDIALS equivalent: KLU_INCLUDE_DIR

TPL_KLU_LIBRARIES - KLU library
   SUNDIALS equivalent: N/A
A.1 CMake-based installation

TPL_LAPACK_LIBRARIES - LAPACK (and BLAS) libraries
Default: /usr/lib/liblapack.so;/usr/lib/libblas.so
SUNDIALS equivalent: LAPACK_LIBRARIES
Note: CMake will search for libraries in your LD_LIBRARY_PATH prior to searching default system paths.

TPL_PETSC_INCLUDE_DIRS - Path to PETSc header files
SUNDIALS equivalent: PETSC_INCLUDE_DIR

TPL_PETSC_LIBRARIES - PETSc library
SUNDIALS equivalent: N/A

TPL_SUPERLUMT_INCLUDE_DIRS - Path to SuperLU_MT header files
SUNDIALS equivalent: SUPERLUMT_INCLUDE_DIR

TPL_SUPERLUMT_LIBRARIES - SuperLU_MT library
SUNDIALS equivalent: N/A

TPL_SUPERLUMT_THREAD_TYPE - SuperLU_MT library thread type
SUNDIALS equivalent: SUPERLUMT_THREAD_TYPE

USE_XSDK_DEFAULTS - Enable xSDK default configuration settings
Default: OFF
SUNDIALS equivalent: N/A
Note: Enabling xSDK defaults also sets CMAKE_BUILD_TYPE to Debug

XSDK_ENABLE_FORTRAN - Enable SUNDIALS Fortran 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:

```bash
% cmake \\
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \\
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \\
> -DMPI_ENABLE=ON \\
> -DFCMIX_ENABLE=ON \\
> /home/myname/sundials/solverdir \\
% 
% make install 
%
```

To disable installation of the examples, use:
A.1.4 Working with external Libraries

The sundials suite contains many options to enable implementation flexibility when developing solutions. The following are some notes addressing specific configurations when using the supported third party libraries. When building sundials as a shared library external libraries any used with sundials must also be build as a shared library or as a static library compiled with the -fPIC flag.

Building with BLAS

Sundials does not utilize BLAS directly but it may be needed by other external libraries that sundials can be built with (e.g. LAPACK, PETSC, SuperLU_MT, etc.). To enable BLAS, set the BLAS_ENABLE option to ON. If the directory containing the BLAS library is in the LD_LIBRARY_PATH environment variable, CMake will set the BLAS_LIBRARIES variable accordingly, otherwise CMake will attempt to find the BLAS library in standard system locations. To explicitly tell CMake what libraries to use, the BLAS_LIBRARIES variable can be set to the desired library. Example:

```
% cmake \
  -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
  -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
  -DMPI_ENABLE=ON \
  -DFCMIX_ENABLE=ON \
  -DEXAMPLES_INSTALL=OFF \
  /home/myname/sundials/solverdir
%
% make install
%
```

When allowing CMake to automatically locate the LAPACK library, CMake may also locate the corresponding BLAS library.

If a working Fortran compiler is not available to infer the Fortran name-mangling scheme, the options SUNDIALS_F77_FUNC_CASE and SUNDIALS_F77_FUNC_UNDERSCORES must be set in order to bypass the check for a Fortran compiler and define the name-mangling scheme. The defaults for these options in earlier versions of sundials were lower and one respectively.

Building with LAPACK

To enable LAPACK, set the LAPACK_ENABLE option to ON. If the directory containing the LAPACK library is in the LD_LIBRARY_PATH environment variable, CMake will set the LAPACK_LIBRARIES variable accordingly, otherwise CMake will attempt to find the LAPACK library in standard system locations. To explicitly tell CMake what library to use, the LAPACK_LIBRARIES variable can be set to the desired libraries. When setting the LAPACK location explicitly the location of the corresponding BLAS library will also need to be set. Example:
A.1 CMake-based installation

% cmake \
> -DCMAKE_INSTALL_PREFIX=/home/mynname/sundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/mynname/sundials/instdir/examples \
> -DBLAS_ENABLE=ON \
> -DBLAS_LIBRARIES=/mylapackpath/lib/libblas.so \
> -DLAPACK_ENABLE=ON \
> -DLAPACK_LIBRARIES=/mylapackpath/lib/liblapack.so \
> /home/mynname/sundials/solverdir \
% \
% make install \
%

When allowing CMake to automatically locate the LAPACK library, CMake may also locate the corresponding BLAS library.

If a working Fortran compiler is not available to infer the Fortran name-mangling scheme, the options SUNDIALS_F77_FUNC_CASE and SUNDIALS_F77_FUNC_UNDERSCORES must be set in order to bypass the check for a Fortran compiler and define the name-mangling scheme. The defaults for these options in earlier versions of SUNDIALS were lower and one respectively.

Building with KLU

The KLU libraries are part of SuiteSparse, a suite of sparse matrix software, available from the Texas A&M University website: http://faculty.cse.tamu.edu/davis/suitesparse.html. SUNDIALS has been tested with SuiteSparse version 4.5.3. To enable KLU, set KLU_ENABLE to ON, set KLU_INCLUDE_DIR to the include path of the KLU installation and set KLU_LIBRARY_DIR to the lib path of the KLU installation. The CMake configure will result in populating the following variables: AMD_LIBRARY, AMD_LIBRARY_DIR, BTF_LIBRARY, BTF_LIBRARY_DIR, COLAMD_LIBRARY, COLAMD_LIBRARY_DIR, and KLU_LIBRARY.

Building with SuperLU_MT

The SuperLU_MT libraries are available for download from the Lawrence Berkeley National Laboratory website: http://crd-legacy.lbl.gov/~xiaoye/SuperLU/#superlu_mt. SUNDIALS has been tested with SuperLU_MT version 3.1. To enable SuperLU_MT, set SUPERLUMT_ENABLE to ON, set SUPERLUMT_INCLUDE_DIR to the SRC path of the SuperLU_MT installation, and set the variable SUPERLUMT_LIBRARY_DIR to the lib path of the SuperLU_MT installation. At the same time, the variable SUPERLUMT_THREAD_TYPE must be set to either Pthread or OpenMP. Do not mix thread types when building SUNDIALS solvers. If threading is enabled for SUNDIALS by having either OPENMP_ENABLE or PTHREAD_ENABLE set to ON then SuperLU_MT should be set to use the same threading type.

Building with PETSc

The PETSc libraries are available for download from the Argonne National Laboratory website: http://www.mcs.anl.gov/petsc. SUNDIALS has been tested with PETSc version 3.7.2. To enable PETSc, set PETSC_ENABLE to ON, set PETSC_INCLUDE_DIR to the include path of the PETsc installation, and set the variable PETSC_LIBRARY_DIR to the lib path of the PETsc installation.

Building with hypre

The hypre libraries are available for download from the Lawrence Livermore National Laboratory website: http://computation.llnl.gov/projects/hypre. SUNDIALS has been tested with hypre version 2.11.1. To enable hypre, set HYPRE_ENABLE to ON, set HYPRE_INCLUDE_DIR to the include path of the hypre installation, and set the variable HYPRE_LIBRARY_DIR to the lib path of the hypre installation.
Building with CUDA

SUNDIALS CUDA modules and examples have been tested with version 8.0 of the CUDA toolkit. To build them, you need to install the Toolkit and compatible NVIDIA drivers. Both are available for download from the NVIDIA website: https://developer.nvidia.com/cuda-downloads. To enable CUDA, set CUDA_ENABLE to ON. If CUDA is installed in a nonstandard location, you may be prompted to set the variable CUDA_TOOLKIT_ROOT_DIR with your CUDA Toolkit installation path. To enable CUDA examples, set EXAMPLES_ENABLE_CUDA to ON.

Building with RAJA

RAJA is a performance portability layer developed by Lawrence Livermore National Laboratory and can be obtained from https://github.com/LLNL/RAJA. SUNDIALS RAJA modules and examples have been tested with RAJA version 0.3. Building SUNDIALS RAJA modules requires a CUDA-enabled RAJA installation. To enable RAJA, set CUDA_ENABLE and RAJA_ENABLE to ON. If RAJA is installed in a nonstandard location you will be prompted to set the variable RAJA_DIR with the path to the RAJA CMake configuration file. To enable building the RAJA examples set EXAMPLES_ENABLE_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 desireable to build the Trilinos vector interface with same compiler and options that were used to build Trilinos. CMake will try to find the correct compiler settings automatically from the Trilinos configuration file. If that is not successful, the compilers and options can be manually set with the following CMake variables:

- TrilinosINTERFACE_C_COMPILER
- TrilinosINTERFACE_C_COMPILER_FLAGS
- TrilinosINTERFACE_CXX_COMPILER
- TrilinosINTERFACE_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.
A.3 Configuring, building, and installing on Windows

Either the CMakeLists.txt file or the traditional Makefile may be used to build the examples as well as serve as a template for creating user developed solutions. To use the supplied Makefile simply run make to compile and generate the executables. To use CMake from within the installed example directory, run cmake (or ccmake to use the GUI) followed by make to compile the example code. Note that if CMake is used, it will overwrite the traditional Makefile with a new CMake-generated Makefile. The resulting output from running the examples can be compared with example output bundled in the SUNDIALS distribution.

NOTE: There will potentially be differences in the output due to machine architecture, compiler versions, use of third party libraries etc.

A.3 Configuring, building, and installing on Windows

CMake can also be used to build SUNDIALS on Windows. To build SUNDIALS for use with Visual Studio the following steps should be performed:

1. Unzip the downloaded tar file(s) into a directory. This will be the solverdir
2. Create a separate builddir
3. Open a Visual Studio Command Prompt and cd to builddir
4. Run cmake-gui ../solverdir
   (a) Hit Configure
   (b) Check/Uncheck solvers to be built
   (c) Change CMAKE_INSTALL_PREFIX to instdir
   (d) Set other options as desired
   (e) Hit Generate
5. Back in the VS Command Window:
   (a) Run msbuild ALL_BUILD.vcxproj
   (b) Run msbuild INSTALL.vcxproj

The resulting libraries will be in the instdir. The SUNDIALS project can also now be opened in Visual Studio. Double click on the ALL_BUILD.vcxproj file to open the project. Build the whole solution to create the SUNDIALS libraries. To use the SUNDIALS libraries in your own projects, you must set the include directories for your project, add the SUNDIALS libraries to your project solution, and set the SUNDIALS libraries as dependencies for your project.

A.4 Installed libraries and exported header files

Using the CMake SUNDIALS build system, the command

% make install

will install the libraries under libdir and the public header files under includedir. The values for these directories are instdir/CMAKE_INSTALL_LIBDIR and instdir/include, respectively. The location can be changed by setting the CMake variable CMAKE_INSTALL_PREFIX. Although all installed libraries reside under libdir/CMAKE_INSTALL_LIBDIR, the public header files are further organized into subdirectories under includedir/include.

The installed libraries and exported header files are listed for reference in Table A.1. The file extension .lib is typically .so for shared libraries and .a for static libraries. Note that, in the Tables, names are relative to libdir for libraries and to includedir for header files.
A typical user program need not explicitly include any of the shared SUNDIALS header files from under the `includedir/include/sundials` directory since they are explicitly included by the appropriate solver header files (e.g., `cvode_dense.h` includes `sundials_dense.h`). However, it is both legal and safe to do so, and would be useful, for example, if the functions declared in `sundials_dense.h` are to be used in building a preconditioner.
<table>
<thead>
<tr>
<th><strong>SHARED</strong></th>
<th><strong>Libraries</strong></th>
<th><strong>Header files</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Libraries</strong></td>
<td>n/a</td>
<td>sundials/sundials_config.h</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sundials/sundials_fconfig.h</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sundials/sundials_types.h</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sundials/sundials_math.h</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sundials/sundials_nvector.h</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sundials/sundials_fnvector.h</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sundials/sundials_matrix.h</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sundials/sundials_linear_solver.h</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sundials/sundials_iterative.h</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sundials/sundials_direct.h</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sundials/sundials_dense.h</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sundials/sundials_band.h</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sundials/sundials_nonlinear_solver.h</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sundials/sundials_version.h</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sundials/sundials_mpi_types.h</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>NVVECTOR_SERIAL</strong></th>
<th><strong>Libraries</strong></th>
<th><strong>Header files</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Libraries</strong></td>
<td></td>
<td>libsundials_nvecserial.lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td>libsundials_fnvecserial_mod.lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td>libsundials_fnvecserial.a</td>
</tr>
<tr>
<td><strong>Header files</strong></td>
<td>nvector/nvector_serial.h</td>
<td></td>
</tr>
<tr>
<td><strong>Module files</strong></td>
<td>fnvector_serial_mod.mod</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>NVVECTOR_PARALLEL</strong></th>
<th><strong>Libraries</strong></th>
<th><strong>Header files</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Libraries</strong></td>
<td></td>
<td>libsundials_nvecparallel.lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td>libsundials_fnvecparallel_mod.lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td>libsundials_fnvecparallel.a</td>
</tr>
<tr>
<td><strong>Header files</strong></td>
<td>nvector/nvector_parallel.h</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>NVVECTOR_OPENMP</strong></th>
<th><strong>Libraries</strong></th>
<th><strong>Header files</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Libraries</strong></td>
<td></td>
<td>libsundials_nvecopenmp.lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td>libsundials_fnvecopenmp_mod.lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td>libsundials_fnvecopenmp.a</td>
</tr>
<tr>
<td><strong>Header files</strong></td>
<td>nvector/nvector_openmp.h</td>
<td></td>
</tr>
<tr>
<td><strong>Module files</strong></td>
<td>fnvector_openmp_mod.mod</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>NVVECTOR_OPENMPDEV</strong></th>
<th><strong>Libraries</strong></th>
<th><strong>Header files</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Libraries</strong></td>
<td></td>
<td>libsundials_nvecopenmpdev.lib</td>
</tr>
<tr>
<td><strong>Header files</strong></td>
<td>nvector/nvector_openmpdev.h</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>NVVECTOR_PTHREADS</strong></th>
<th><strong>Libraries</strong></th>
<th><strong>Header files</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Libraries</strong></td>
<td></td>
<td>libsundials_nvecpthreads.lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td>libsundials_fnvecpthreads_mod.lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td>libsundials_fnvecpthreads.a</td>
</tr>
<tr>
<td><strong>Header files</strong></td>
<td>nvector/nvector_pthreads.h</td>
<td></td>
</tr>
<tr>
<td><strong>Module files</strong></td>
<td>fnvector_pthreads_mod.mod</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>NVVECTOR_PARHYP</strong></th>
<th><strong>Libraries</strong></th>
<th><strong>Header files</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Libraries</strong></td>
<td></td>
<td>libsundials_nvecparhyp.lib</td>
</tr>
<tr>
<td><strong>Header files</strong></td>
<td>nvector/nvector_parhyp.h</td>
<td></td>
</tr>
</tbody>
</table>

*continued on next page*
<table>
<thead>
<tr>
<th>Libraries</th>
<th>Header files</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVECTORM PETSC</td>
<td>nvector/nvecpetsc.lib</td>
</tr>
<tr>
<td>NVECTORM CUDA</td>
<td>nvector/nvec_cuda.h</td>
</tr>
<tr>
<td>NVECTORM RAJA</td>
<td>nvector/nvec_cuda.h</td>
</tr>
<tr>
<td>NVECTORM TRILINOS</td>
<td>nvector/nvec_trilinos.lib</td>
</tr>
<tr>
<td>SUNMATRIX BAND</td>
<td>nvector/trilinos/SundialsTpetraVectorInterface.h</td>
</tr>
<tr>
<td>SUNMATRIX DENSE</td>
<td>nvector/trilinos/SundialsTpetraVectorKernels.h</td>
</tr>
<tr>
<td>SUNMATRIX SPARSE</td>
<td>nvector/trilinos/SundialsTpetraVectorInterface.h</td>
</tr>
<tr>
<td>SUNLINSOL BAND</td>
<td>nvector/trilinos/SundialsTpetraVectorInterface.h</td>
</tr>
</tbody>
</table>

continued on next page
### A.4 Installed libraries and exported header files

<table>
<thead>
<tr>
<th>SUNLINSOL_DENSE</th>
<th>Libraries</th>
<th>Header files</th>
<th>Module files</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>sunlinsol/sunlinsol.dense.h</td>
<td>fsunlinsol_dense_mod.mod</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SUNLINSOL_KLU</th>
<th>Libraries</th>
<th>Header files</th>
<th>Module files</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>sunlinsol/sunlinsol_klu.h</td>
<td>fsunlinsol_klu_mod.mod</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SUNLINSOL_LAPACKBAND</th>
<th>Libraries</th>
<th>Header files</th>
<th>Module files</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>sunlinsol/sunlinsol_lapackband.h</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SUNLINSOL_LAPACKDENSE</th>
<th>Libraries</th>
<th>Header files</th>
<th>Module files</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>sunlinsol/sunlinsol_lapackdense.lib</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SUNLINSOL_PCG</th>
<th>Libraries</th>
<th>Header files</th>
<th>Module files</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>sunlinsol/sunlinsol_pcg.h</td>
<td>fsunlinsol_pcg_mod.mod</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SUNLINSOL_SPBCGS</th>
<th>Libraries</th>
<th>Header files</th>
<th>Module files</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>sunlinsol/sunlinsol_spbcgs.h</td>
<td>fsunlinsol_spbcgs_mod.mod</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SUNLINSOL_SPFGMR</th>
<th>Libraries</th>
<th>Header files</th>
<th>Module files</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>sunlinsol/sunlinsol_spfgmr.h</td>
<td>fsunlinsol_spfgmr_mod.mod</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SUNLINSOL_SPGMR</th>
<th>Libraries</th>
<th>Header files</th>
<th>Module files</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>sunlinsol/sunlinsol_spgmr.h</td>
<td>fsunlinsol_spgmr_mod.mod</td>
</tr>
</tbody>
</table>

*continued on next page*
| SUNLINSOL_SPTFQMR         | Libraries       | libsundials_sunlinsolsptfqmr.lib  |
|                         |                | libsundials_fsunlinsolsptfqmr_mod.lib  |
|                         |                | libsundials_fsunlinsolsptfqmr.a  |
| Header files           | sunlinsol/sunlinsol_sptfqmr.h  |
| Module files           | fsunlinsol_sptfqmr_mod.mod  |
| SUNLINSOL_SUPERLUMT     | Libraries       | libsundials_sunlinsolsuperlumt.lib  |
|                         |                | libsundials_fsunlinsolsuperlumt.a  |
| Header files           | sunlinsol/sunlinsol_superlumt.h  |
| SUNNONLINSOL_NEWTON     | Libraries       | libsundials_sunnonlinsolnewton.lib  |
|                         |                | libsundials_fsunnonlinsolnewton_mod.lib  |
|                         |                | libsundials_fsunnonlinsolnewton.a  |
| Header files           | sunnonlinsol/sunnonlinsol_newton.h  |
| Module files           | fsunnonlinsol_newton_mod.mod  |
| SUNNONLINSOL_FIXEDPOINT | Libraries       | libsundials_sunnonlinsolfixedpoint.lib  |
|                         |                | libsundials_fsunnonlinsolfixedpoint.a  |
|                         |                | libsundials_fsunnonlinsolfixedpoint_mod.lib  |
| Header files           | sunnonlinsol/sunnonlinsol_fixedpoint.h  |
| Module files           | fsunnonlinsol_fixedpoint_mod.mod  |
| CVODE                  | Libraries       | libsundials_cvode.lib  |
|                         |                | libsundials_fcvode.a  |
| Header files           | cvode/cvode.h  |
|                         | cvode/cvode_direct.h  |
|                         | cvode/cvode_spils.h  |
|                         | cvode/cvode_bbddpre.h  |
| Module files           | fcvode_mod.mod  |
| CVODES                 | Libraries       | libsundials_cvodes.lib  |
| Header files           | cvodes/cvodes.h  |
|                         | cvodes/cvodes_direct.h  |
|                         | cvodes/cvodes_spils.h  |
|                         | cvodes/cvodes_bbddpre.h  |
| ARKODE                 | Libraries       | libsundials_arkode.lib  |
|                         |                | libsundials_farkode.a  |
| Header files           | arkode/arkode.h  |
|                         | arkode/arkode_ls.h  |
|                         | arkode/arkode_bbddpre.h  |
| IDA                    | Libraries       | libsundials_ida.lib  |
|                         |                | libsundials_fida.a  |
| Header files           | ida/ida.h  |
|                         | ida/ida_direct.h  |
|                         | ida/ida_spils.h  |
|                         | ida/ida_bbddpre.h  |
| IDAS                   | Libraries       | libsundials_idas.lib  |
| continued on next page | continued on next page | continued on next page |
## Installed libraries and exported header files

### KINSOL

<table>
<thead>
<tr>
<th>Libraries</th>
<th>Header files</th>
</tr>
</thead>
<tbody>
<tr>
<td>libsundials</td>
<td>kinsol/kinsol.h</td>
</tr>
<tr>
<td></td>
<td>kinsol/kinsol_direct.h</td>
</tr>
<tr>
<td></td>
<td>kinsol/kinsol_spils.h</td>
</tr>
</tbody>
</table>

|                 | Idas/Idas.h                                       |
|                 | idas/idas_impl.h                                  |
|                 | idas/idas_direct.h                                |
|                 | idas/idas_ls.h                                    |
|                 | idas/idas_spils.h                                 |
|                 | idas/idas_bbdpre.h                                |

<table>
<thead>
<tr>
<th>Libraries</th>
<th>Header files</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>kinsol/kinsol.h</td>
</tr>
<tr>
<td></td>
<td>kinsol/kinsol_direct.h</td>
</tr>
<tr>
<td></td>
<td>kinsol/kinsol_spils.h</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>KINSOL</th>
<th>libsundials_kinsol.lib</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>libsundials_kinsol.a</td>
</tr>
</tbody>
</table>

### Header files

<table>
<thead>
<tr>
<th>Libraries</th>
<th>Header files</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>idas/idas.h</td>
</tr>
<tr>
<td></td>
<td>idas/idas_direct.h</td>
</tr>
<tr>
<td></td>
<td>idas/idas_spils.h</td>
</tr>
<tr>
<td></td>
<td>idas/idas_ls.h</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Libraries</th>
<th>Header files</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>idas/idas_impl.h</td>
</tr>
<tr>
<td></td>
<td>idas/idas_direct.h</td>
</tr>
<tr>
<td></td>
<td>idas/idas_ls.h</td>
</tr>
<tr>
<td></td>
<td>idas/idas_spils.h</td>
</tr>
</tbody>
</table>
# Appendix B

## CVODES 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 CVODES input constants

<table>
<thead>
<tr>
<th>CVODES main solver module</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CV_ADAMS</td>
<td>1</td>
<td>Adams-Moulton linear multistep method.</td>
</tr>
<tr>
<td>CV_BDF</td>
<td>2</td>
<td>BDF linear multistep method.</td>
</tr>
<tr>
<td>CV_NORMAL</td>
<td>1</td>
<td>Solver returns at specified output time.</td>
</tr>
<tr>
<td>CV_ONE_STEP</td>
<td>2</td>
<td>Solver returns after each successful step.</td>
</tr>
<tr>
<td>CV_SIMULTANEOUS</td>
<td>1</td>
<td>Simultaneous corrector forward sensitivity method.</td>
</tr>
<tr>
<td>CV_STAGGERED</td>
<td>2</td>
<td>Staggered corrector forward sensitivity method.</td>
</tr>
<tr>
<td>CV_STAGGERED1</td>
<td>3</td>
<td>Staggered (variant) corrector forward sensitivity method.</td>
</tr>
<tr>
<td>CV_CENTERED</td>
<td>1</td>
<td>Central difference quotient approximation ($2^{nd}$ order) of the sensitivity RHS.</td>
</tr>
<tr>
<td>CV_FORWARD</td>
<td>2</td>
<td>Forward difference quotient approximation ($1^{st}$ order) of the sensitivity RHS.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CVODES adjoint solver module</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CV_HERMITE</td>
<td>1</td>
<td>Use Hermite interpolation.</td>
</tr>
<tr>
<td>CV_POLYNOMIAL</td>
<td>2</td>
<td>Use variable-degree polynomial interpolation.</td>
</tr>
</tbody>
</table>

### Iterative linear solver modules

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No preconditioning</td>
</tr>
<tr>
<td>1</td>
<td>Preconditioning on the left only.</td>
</tr>
<tr>
<td>2</td>
<td>Preconditioning on the right only.</td>
</tr>
<tr>
<td>3</td>
<td>Preconditioning on both the left and the right.</td>
</tr>
<tr>
<td>1</td>
<td>Use modified Gram-Schmidt procedure.</td>
</tr>
<tr>
<td>2</td>
<td>Use classical Gram-Schmidt procedure.</td>
</tr>
</tbody>
</table>

### B.2 CVODES output constants
**CVODES main solver module**

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CV_SUCCESS</td>
<td>0</td>
<td>Successful function return.</td>
</tr>
<tr>
<td>CV_TSTOP_RETURN</td>
<td>1</td>
<td>CVode succeeded by reaching the specified stopping point.</td>
</tr>
<tr>
<td>CV_ROOT_RETURN</td>
<td>2</td>
<td>CVode succeeded and found one or more roots.</td>
</tr>
<tr>
<td>CV_WARNING</td>
<td>99</td>
<td>CVode succeeded but an unusual situation occurred.</td>
</tr>
<tr>
<td>CV_TOO_MUCH_WORK</td>
<td>-1</td>
<td>The solver took <code>mxstep</code> internal steps but could not reach <code>tout</code>.</td>
</tr>
<tr>
<td>CV_TOO_MUCH_ACC</td>
<td>-2</td>
<td>The solver could not satisfy the accuracy demanded by the user for some internal step.</td>
</tr>
<tr>
<td>CV_ERR_FAILURE</td>
<td>-3</td>
<td>Error test failures occurred too many times during one internal time step or minimum step size was reached.</td>
</tr>
<tr>
<td>CV_CONV_FAILURE</td>
<td>-4</td>
<td>Convergence test failures occurred too many times during one internal time step or minimum step size was reached.</td>
</tr>
<tr>
<td>CV_LIMIT_FAIL</td>
<td>-5</td>
<td>The linear solver’s initialization function failed.</td>
</tr>
<tr>
<td>CV_LSETUP_FAIL</td>
<td>-6</td>
<td>The linear solver’s setup function failed in an unrecoverable manner.</td>
</tr>
<tr>
<td>CV_LSOLVE_FAIL</td>
<td>-7</td>
<td>The linear solver’s solve function failed in an unrecoverable manner.</td>
</tr>
<tr>
<td>CV_RHSFUNC_FAIL</td>
<td>-8</td>
<td>The right-hand side function failed in an unrecoverable manner.</td>
</tr>
<tr>
<td>CV_FIRST_RHSFUNC_ERR</td>
<td>-9</td>
<td>The right-hand side function failed at the first call.</td>
</tr>
<tr>
<td>CV_REPTD_RHSFUNC_ERR</td>
<td>-10</td>
<td>The right-hand side function had repeated recoverable errors.</td>
</tr>
<tr>
<td>CV_UNREC_RHSFUNC_ERR</td>
<td>-11</td>
<td>The right-hand side function had a recoverable error, but no recovery is possible.</td>
</tr>
<tr>
<td>CV_RTFUNC_FAIL</td>
<td>-12</td>
<td>The rootfinding function failed in an unrecoverable manner.</td>
</tr>
<tr>
<td>CV-NLS_INIT_FAIL</td>
<td>-13</td>
<td>The nonlinear solver’s init routine failed.</td>
</tr>
<tr>
<td>CV-NLS_SETUP_FAIL</td>
<td>-14</td>
<td>The nonlinear solver’s setup routine failed.</td>
</tr>
<tr>
<td>CV_CONSTR_FAIL</td>
<td>-15</td>
<td>The inequality constraints were violated and the solver was unable to recover.</td>
</tr>
<tr>
<td>CV_MEM_FAIL</td>
<td>-20</td>
<td>A memory allocation failed.</td>
</tr>
<tr>
<td>CV_MEM_NULL</td>
<td>-21</td>
<td>The <code>cvode_mem</code> argument was <code>NULL</code>.</td>
</tr>
<tr>
<td>CV_JLL_INPUT</td>
<td>-22</td>
<td>One of the function inputs is illegal.</td>
</tr>
<tr>
<td>CV_NO_MALLOC</td>
<td>-23</td>
<td>The CVODE memory block was not allocated by a call to <code>CVodeMalloc</code>.</td>
</tr>
<tr>
<td>CV_BAD_K</td>
<td>-24</td>
<td>The derivative order <code>k</code> is larger than the order used.</td>
</tr>
<tr>
<td>CV_BAD_T</td>
<td>-25</td>
<td>The time <code>t</code> is outside the last step taken.</td>
</tr>
<tr>
<td>CV_BAD_DKY</td>
<td>-26</td>
<td>The output derivative vector is <code>NULL</code>.</td>
</tr>
<tr>
<td>CV_TOO_CLOSE</td>
<td>-27</td>
<td>The output and initial times are too close to each other.</td>
</tr>
<tr>
<td>CV_NO_QUAD</td>
<td>-30</td>
<td>Quadrature integration was not activated.</td>
</tr>
<tr>
<td>CV_QRHSFUNC_FAIL</td>
<td>-31</td>
<td>The quadrature right-hand side function failed in an unrecoverable manner.</td>
</tr>
<tr>
<td>CV_FIRST_QRHSFUNC_ERR</td>
<td>-32</td>
<td>The quadrature right-hand side function failed at the first call.</td>
</tr>
<tr>
<td>CV_REPTD_QRHSFUNC_ERR</td>
<td>-33</td>
<td>The quadrature right-hand side function had repeated recoverable errors.</td>
</tr>
</tbody>
</table>
### B.2 CVODES output constants

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CV_UNREC_QRHSFUNC_ERR</td>
<td>-34</td>
<td>The quadrature right-hand side function had a recoverable error, but no recovery is possible.</td>
</tr>
<tr>
<td>CV_NO_SENS</td>
<td>-40</td>
<td>Forward sensitivity integration was not activated.</td>
</tr>
<tr>
<td>CV_SRHSFUNC_FAIL</td>
<td>-41</td>
<td>The sensitivity right-hand side function failed in an unrecoverable manner.</td>
</tr>
<tr>
<td>CV_FIRST_SRHSFUNC_ERR</td>
<td>-42</td>
<td>The sensitivity right-hand side function failed at the first call.</td>
</tr>
<tr>
<td>CV_REPTD_SRHSFUNC_ERR</td>
<td>-43</td>
<td>The sensitivity ight-hand side function had repeated recoverable errors.</td>
</tr>
<tr>
<td>CV_UNREC_SRHSFUNC_ERR</td>
<td>-44</td>
<td>The sensitivity right-hand side function had a recoverable error, but no recovery is possible.</td>
</tr>
<tr>
<td>CV_BAD_IS</td>
<td>-45</td>
<td>The sensitivity index is larger than the number of sensitivities computed.</td>
</tr>
<tr>
<td>CV_NO_QUADSENS</td>
<td>-50</td>
<td>Forward sensitivity integration was not activated.</td>
</tr>
<tr>
<td>CV_QSRHSFUNC_FAIL</td>
<td>-51</td>
<td>The sensitivity right-hand side function failed in an unrecoverable manner.</td>
</tr>
<tr>
<td>CV_FIRST_QSRSRHFunc_ERR</td>
<td>-52</td>
<td>The sensitivity right-hand side function failed at the first call.</td>
</tr>
<tr>
<td>CV_REPTD_QSRSRHFunc_ERR</td>
<td>-53</td>
<td>The sensitivity right-hand side function had repeated recoverable errors.</td>
</tr>
<tr>
<td>CV_UNREC_QSRHSFUNC_ERR</td>
<td>-54</td>
<td>The sensitivity right-hand side function had a recoverable error, but no recovery is possible.</td>
</tr>
</tbody>
</table>

### CVODES adjoint solver module

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CV_NO_ADJ</td>
<td>-101</td>
<td>Adjoint module was not initialized.</td>
</tr>
<tr>
<td>CV_NO_FWD</td>
<td>-102</td>
<td>The forward integration was not yet performed.</td>
</tr>
<tr>
<td>CV_NO_BCK</td>
<td>-103</td>
<td>No backward problem was specified.</td>
</tr>
<tr>
<td>CV_BAD_TBO</td>
<td>-104</td>
<td>The final time for the adjoint problem is outside the interval over which the forward problem was solved.</td>
</tr>
<tr>
<td>CV_REIFWD_FAIL</td>
<td>-105</td>
<td>Reinitialization of the forward problem failed at the first checkpoint.</td>
</tr>
<tr>
<td>CV_FWD_FAIL</td>
<td>-106</td>
<td>An error occurred during the integration of the forward problem.</td>
</tr>
<tr>
<td>CV_GETY_BADT</td>
<td>-107</td>
<td>Wrong time in interpolation function.</td>
</tr>
</tbody>
</table>

### CVLS linear solver interface

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Code</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_ILMEM_NULL</td>
<td>-2</td>
<td>The CVLS linear solver has not been initialized.</td>
</tr>
<tr>
<td>CVLS_ILL_INPUT</td>
<td>-3</td>
<td>The CVLS solver is not compatible with the current NVECTOR module, or an input value was illegal.</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>Constant</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CVLS_SUNLS_FAIL</td>
<td>-9</td>
<td>An error occurred with the current sunlinsol module.</td>
</tr>
<tr>
<td>CVLS_NO_ADJ</td>
<td>-101</td>
<td>The combined forward-backward problem has not been initialized.</td>
</tr>
<tr>
<td>CVLS_LMEMB_NULL</td>
<td>-102</td>
<td>The linear solver was not initialized for the backward phase.</td>
</tr>
<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 nvector 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>
<tr>
<td>CVDIAG_NO_ADJ</td>
<td>-101</td>
<td>The combined forward-backward problem has not been initialized.</td>
</tr>
</tbody>
</table>

**CVDIAG linear solver module**

- **CVDIAG_SUCCESS**: 0 Successful function return.
- **CVDIAG_MEM_NULL**: -1 The cvode_mem argument was NULL.
- **CVDIAG_LMEM_NULL**: -2 The CVDIAG linear solver has not been initialized.
- **CVDIAG_ILL_INPUT**: -3 The CVDIAG solver is not compatible with the current NVECTOR module.
- **CVDIAG_MEM_FAIL**: -4 A memory allocation request failed.
- **CVDIAG_INV_FAIL**: -5 A diagonal element of the Jacobian was 0.
- **CVDIAG_RHSFUNC_UNRECVR**: -6 The right-hand side function failed in an unrecoverable manner.
- **CVDIAG_RHSFUNC_RECVR**: -7 The right-hand side function had a recoverable error.
- **CVDIAG_NO_ADJ**: -101 The combined forward-backward problem has not been initialized.
Bibliography


## Index

Adams method, 15
adjoint sensitivity analysis
  checkpointing, 27
  implementation in CVODES, 28, 32
  mathematical background, 25–28
quadrature evaluation, 148
right-hand side evaluation, 147
sensitivity-dependent quadrature evaluation, 149

BDF method, 15
BIG_REAL, 36, 167
booleantype, 36

CV_ADAMS, 42, 75, 134
CV_BAD_DKY, 60, 86, 107, 108, 120, 121
CV_BAD_IS, 108, 121
CV_BAD_ITASK, 138
CV_BAD_K, 60, 86, 107, 108, 120, 121
CV_BAD_T, 60, 86, 107, 108, 120, 121
CV_BAD_TBO, 134, 135
CV_BAD_TOUT, 138
CV_BCKMEM_NULL, 138
CV_BDF, 42, 75, 134
CV_CENTERED, 109
CV_CONSTR_FAIL, 49
CV_CONV_FAILURE, 48, 133, 138
CV_ERR_FAILURE, 48, 133, 138
CV_FIRST_QRHSFUNC_ERR, 89
CV_FIRST_QRHSFUNC_FAIL, 85
CV_FIRST_QRSRSFUNC_ERR, 119, 125
CV_FIRST_RHSFUNC_ERR, 76
CV_FIRST_RHSFUNC_FAIL, 49
CV_FIRST_SRHSFUNC_ERR, 106, 115, 116
CV_FORWARD, 109
CV_FUNCTIONAL, 55
CV_FWD_FAIL, 138
CV_GETY_BADT, 144
CV_HERMITE, 131
CV_ILL_INPUT, 42, 43, 47, 48, 52–56, 59, 75, 87,
CV_LINIT_FAIL, 48
CV_LSETUP_FAIL, 48, 78, 93, 133, 138, 150, 151,
  159, 160
CV_LSOLVE_FAIL, 48, 133
CV_MEM_FAIL, 42, 54, 68, 84, 101–103, 110, 113,
  131, 133, 134, 145, 146
CV_MEM_NULL, 42, 43, 47, 48, 51–56, 59, 60, 63–69,
CV_NEWTON, 55
CV_NO_ADJ, 132–136, 138–140, 145, 146
CV_NO_BCK, 138
CV_NO_FWD, 138
CV_NO_MALLOC, 43, 44, 48, 75, 133–136
CV_NO_QUAD, 84, 86–88, 122, 146
CV_NO_QUADSENS, 119–124
CV_NO_SENS, 103, 104, 107–109, 111–114, 118–121
CV_NORMAL, 48, 128, 132, 138
CV_ONE_STEP, 48, 128, 132, 138
CV_POLYNOMIAL, 131
CV_QRHS_FAIL, 124
CV_QRHSFUNC_FAIL, 85, 88, 148, 149
CV_QRSRSFUNC_ERR, 119
CV_REIFWD_FAIL, 138
CV_REPTD_QRHSFUNC_ERR, 85
CV_REPTD_QRSRSFUNC_ERR, 119
CV_REPTD_RHSFUNC_ERR, 49
CV_REPTD_SRHSFUNC_ERR, 106
CV_RHSFUNC_FAIL, 49, 76, 147, 148
CV_ROOT_RETURN, 48, 133
CV_RTFUNC_FAIL, 49, 77
CV_SIMULTANEOUS, 32, 101, 114
CV_SOLVE_FAIL, 138
CV_SRHSFUNC_FAIL, 106, 115, 116
CV_STAGGERED, 32, 101, 114
CV_STAGGERED1, 32, 102, 115
CV_SUCCESS, 42, 43, 47, 48, 51–56, 59, 60, 63–69,
CV_TOO_CLOSE, 48
CV_TOO_MUCH_ACC, 48, 133, 138
CV_TOO_MUCH_WORK, 48, 133, 138
CV_TSTOP_RETURN, 48, 133
CV_UNREC_QRHSFUNC_ERR, 89
CV_UNREC_QSRHSFUNC_ERR, 125
CV_UNREC_RHSFUNC_ERR, 49, 76, 85
CV_UNREC_SRHSFUNC_ERR, 106, 115, 116
CV_WARNING, 77
CvBandPrec preconditioner
  description, 89
  optional output, 90–91
  usage, 89–90
  usage with adjoint module, 157
  user-callable functions, 90, 157
CvBandPrecGetNumRhsEvals, 91
CvBandPrecGetWorkSpace, 91
CvBandPrecInit, 90
CvBandPrecInitB, 157
CvBBDPrec preconditioner
  description, 91–92
  optional output, 96
  usage, 93–94
  usage with adjoint module, 158–160
  user-callable functions, 94–96, 158–159
  user-supplied functions, 92–93, 159–160
CvBBDPrecGetNumGfnEvals, 96
CvBBDPrecGetWorkSpace, 96
CvBBDPrecInit, 94
CvBBDPrecInitB, 158
CvBBDPrecReInit, 95
CvBBDPrecReInitB, 159
CvdIag linear solver
  Jacobian approximation used by, 46
  selection of, 46
CvdIag linear solver interface
  memory requirements, 73
  optional output, 73–74
CvdIag, 40, 45, 46
CvdIag_Ill_INPUT, 46, 137
CvdIag_LMEM_NULL, 73, 74
CvdIag_MEM_FAIL, 46, 137
CvdIag_MEM_NULL, 46, 73, 74, 137
CvdIag_SUCCESS, 46, 73, 74, 137
CvdIagB, 130, 137
CvdIagGetLastFlag, 74
CvdIagGetNumRhsEvals, 74
CvdIagGetReturnFlagName, 74
CvdIagGetWorkSpace, 73
CvdIlsGetLastFlag, 73
CvdIlsGetNumJacEvals, 70
CvdIlsGetNumRhsEvals, 70
CvdIlsGetReturnFlagName, 73
CvdIlsGetWorkspace, 69
CvdIlsJacFn, 79
CvdIlsJacFnB, 150
CvdIlsJacFnBS, 151
CvdIlsSetJacFn, 57
CvdIlsSetJacFnB, 140
CVDlsSetJacFnBS, 141
CVDlsSetLinearSolver, 46
CVDlsSetLinearSolverB, 137
CVErHandlerFn, 76
CVEwtFn, 77
CvLs generic linear solver
  SUNLINSOL compatibility, 45
CvLS linear solver
  preconditioner setup function, 155
  preconditioner solve function, 154
CvLS linear solver interface
  convergence test, 58
  Jacobian approximation used by, 56
  Jacobian-vector product approximation used by, 57
  memory requirements, 69
  optional input, 56–143
  optional output, 69–73
  preconditioner setup function, 58, 81
  preconditioner solve function, 58, 81
CvLS_ILL_INPUT, 46, 59, 90, 95, 137, 140–143, 157–159
CvLS_JACFUNC_RECVR, 78, 150, 151
CvLS_JACFUNC_UN_RECVR, 78, 150, 151
CvLS_LMEM_NULL, 56–59, 69–72, 90, 95, 96, 140–143, 157–159
CvLS_MEM_FAIL, 46, 90, 95, 137, 157–159
CvLS_MEM_NULL, 46, 56–59, 69–72, 90, 95, 96, 137, 140–143, 157–159
CvLS_NO_ADJ, 137, 140–143
CvLS_PMEM_NULL, 91, 96, 159
CvLS_SUCCESS, 45, 56–59, 69–72, 90, 91, 95, 96, 137, 140–143, 157–159
CvLS_SUNLS_FAIL, 46, 57, 58
CvLSJacFn, 77
CvLSJacFnB, 150
CvLSJacFnBS, 150
CvLSJacTimesSetupFn, 80
CvLSJacTimesSetupFnB, 153
CvLSJacTimesSetupFnBS, 153
CvLSJacTimesVecFn, 79
CvLSJacTimesVecFnB, 151
CvLSJacTimesVecFnBS, 151
CvLSPrecSetupFn, 81
CvLSPrecSolveFn, 81
Cvode, 1
Cvode, 40, 48, 123
Cvode_MEM_FAIL, 118
Cvode_MEM_NULL, 118–124
CvodeAdjFree, 132
CvodeAdjInit, 128, 131
CvodeAdjReInit, 131
CvodeAdjSetNoSensi, 139
CvodeB, 130, 137, 138
CVodeCreate, 42
CVodeCreateB, 128, 133
CVodeF, 128, 132
CVodeFree, 41, 42
CVodeGetActualInitStep, 65
CVodeGetAdjCheckPointsInfo, 144
CVodeGetAdjCVodeBmem, 143
CVodeGetAdjY, 144
CVodeGetB, 139
CVodeGetCurrentOrder, 65
CVodeGetCurrentStep, 65
CVodeGetCurrentTime, 66
CVodeGetDky, 60
CVodeGetErrWeights, 66
CVodeGetEstLocalErrors, 67
CVodeGetIntegratorStats, 67
CVodeGetLastFlag, 72
CVodeGetLastOrder, 64
CVodeGetLastStep, 65
CVodeGetLinReturnFlagName, 73
CVodeGetLinWorkSpace, 69
CVodeGetNonlinSolvStats, 68
CVodeGetNumErrTestFails, 64
CVodeGetNumGEvals, 70
CVodeGetNumJacEvals, 70
CVodeGetNumJtimesEvals, 72
CVodeGetNumLinConvFails, 71
CVodeGetNumLinIters, 70
CVodeGetNumLinRhsEvals, 70
CVodeGetNumLinSolvSetups, 64
CVodeGetNumNonlinSolvConvFails, 68
CVodeGetNumNonlinSolvIters, 67
CVodeGetNumPrecEvals, 71
CVodeGetNumPrecSolves, 71
CVodeGetNumRhsEvals, 64
CVodeGetNumRhsEvalsSEns, 111
CVodeGetNumStabLimOrderReds, 66
CVodeGetNumSteps, 63
CVodeGetQuad, 85, 146
CVodeGetQuadB, 130, 146
CVodeGetQuadDky, 85, 86
CVodeGetQuadErrWeights, 88
CVodeGetQuadNumErrTestFails, 87
CVodeGetQuadNumRhsEvals, 87
CVodeGetQuadSens, 119
CVodeGetQuadSens1, 120
CVodeGetQuadSensDky, 120
CVodeGetQuadSensDky1, 121
CVodeGetQuadSensErrWeights, 123
CVodeGetQuadSensNumErrTestFails, 123
CVodeGetQuadSensNumRhsEvals, 123
CVodeGetQuadSensStats, 124
CVodeGetQuadStats, 88
CVodeGetReturnFlagName, 68
CVodeGetRootInfo, 68
CVodeGetSens, 100, 107
CVodeGetSens1, 100, 107
CVodeGetSensDky, 100, 107
CVodeGetSensDky1, 100, 108
CVodeGetSensErrWeights, 112
CVodeGetSensNonlinSolvStats, 113
CVodeGetSensNumErrTestFails, 111
CVodeGetSensNumLinSolvSetups, 111
CVodeGetSensNumNonlinSolvConvFails, 113
CVodeGetSensNumNonlinSolvIters, 112
CVodeGetSensNumRhsEvals, 110
CVodeGetSensStats, 112
CVodeGetStrgSensNumNonlinSolvConvFails, 114
CVodeGetStrgSensNumNonlinSolvIters, 113
CVodeGetToiScaleFactor, 66
CVodeGetWorkSpace, 63
CVodeInit, 42, 74
CVodeInitB, 129, 134
CVodeInitBS, 129, 134
CVodeQuadFree, 85
CVodeQuadInit, 84
CVodeQuadInitB, 145
CVodeQuadInitBS, 145
CVodeQuadReInit, 84
CVodeQuadReInitB, 145
CVodeQuadSensEEtolerances, 122, 123
CVodeQuadSensFree, 119
CVodeQuadSensInit, 118
CVodeQuadSensReInit, 118
CVodeQuadSensSStolerances, 122
CVodeQuadSStolerances, 122
CVodeQuadSVtolerances, 87
CVodeReInit, 74
CVodeReInitB, 135
CVodeRootInit, 47
CVODES
    brief description of, 1
    motivation for writing in C, 2
    package structure, 31
    relationship to CVODE, PVODE, 2
    relationship to VODE, VODPK, 1–2
    CVODES linear solver interface
        CVLS, 136
CVODES linear solver interfaces, 32
    CVDIAG, 46
CVLS, 45
    selecting one, 45
CVODES linear solvers
    header files, 37
    implementation details, 34
    NVECTOR compatibility, 35
selecting one, 45
usage with adjoint module, 136
cvodes/cvodes.h, 37
cvodes/cvodes_diag.h, 38
cvodes/cvodes_ls.h, 37
CVodeSensEEtolerances, 104
CVodeSensFree, 103
CVodeSensInit, 99, 101, 102
CVodeSensInit1, 99, 101, 102, 114
CVodeSensReInit, 102
CVodeSensSSStolerances, 104
CVodeSensSVtolerances, 104
CVodeSensToggleOff, 103
CVodeSetConstraints, 55
CVodeSetEpsLin, 59
CVodeSetEpsLinB, 143
CVodeSetErrFile, 49, 51
CVodeSetErrHandlerFn, 51
CVodeSetInitStep, 53
CVodeSetIterType, 55
CVodeSetJacFn, 57
CVodeSetJacFnB, 140
CVodeSetJacFnBS, 140
CVodeSetJacTimes, 57
CVodeSetJacTimesB, 141
CVodeSetJacTimesBS, 141
CVodeSetLinearSolver, 40, 45, 77, 222
CVodeSetLinearSolverB, 130, 136, 150, 157, 158
CVodeSetMaxConvFails, 54
CVodeSetMaxErrTestFails, 54
CVodeSetMaxHnilWarns, 52
CVodeSetMaxNonlinIters, 54
CVodeSetMaxNumSteps, 52
CVodeSetMaxOrd, 51
CVodeSetMaxStep, 53
CVodeSetMaxStepsBetweenJac, 56
CVodeSetMinStep, 53
CVodeSetNoInactiveRootWarn, 59
CVodeSetNonlinConvCoef, 55
CVodeSetNonlinearSolver, 47
CVodeSetNonlinearSolver, 40, 46, 105, 106
CVodeSetNonlinearSolverB, 130
CVodeSetNonlinearSolverSensSim, 105
CVodeSetNonlinearSolverSensSim, 105
CVodeSetNonlinearSolverSensStg, 105
CVodeSetNonlinearSolverSensStg, 105
CVodeSetNonlinearSolverSensStg1, 106
CVodeSetNonlinearSolverSensStg1, 105
CVodeSetPreconditioner, 58
CVodeSetPrecSolveFnB, 142
CVodeSetPrecSolveFnBS, 142
CVodeSetQuadErrCon, 86
CVodeSetQuadSensErrCon, 121
CVodeSetRootDirection, 59
CVodeSetSensDQMethod, 109
CVodeSetSensErrCon, 109
CVodeSetSensMaxNonlinIters, 110
CVodeSetSensParams, 109
CVodeSetStabLimDet, 52
CVodeSetStopTime, 54
CVodeSetUserData, 51
CVodeSSStolerances, 43
CVodeSSStolerancesB, 136
CVodeSVtolerances, 43
CVodeSVtolerancesB, 136
CVodeWFtolerances, 43
CVQuadRhsFn, 84, 88
CVQuadRhsFnB, 145, 148
CVQuadRhsFnBS, 145, 149
CVQuadSensRhsFn, 118, 124
CVRhsFn, 42, 75
CVRhsFnB, 134, 147
CVRhsFnBS, 135, 147
CVRootFn, 77
CVSensRhs1Fn, 102, 115
CVSensRhsFn, 101, 114
CVSpilsGetLastFlag, 73
CVSpilsGetNumConvFails, 71
CVSpilsGetNumJtimesEvals, 72
CVSpilsGetNumJtimesSetupEvals, 72
CVSpilsGetNumLinIters, 70
CVSpilsGetNumPrecEvals, 71
CVSpilsGetNumPrecSolves, 71
CVSpilsGetNumRhsEvals, 70
CVSpilsGetReturnFlagName, 73
CVSpilsGetWorkspace, 69
CVSpilsJacTimesSetupFn, 80
CVSpilsJacTimesSetupFnB, 153
CVSpilsJacTimesSetupFnBS, 154
CVSpilsJacTimesVecFn, 80
CVSpilsJacTimesVecFnB, 152
CVSpilsJacTimesVecFnBS, 153
CVSpilsPrecSetupFn, 82
CVSpilsPrecSetupFnB, 156
CVSpilsPrecSetupFnBS, 157
CVSpilsPrecSolveFn, 81
CVSpilsPrecSolveFnB, 155
CVSpilsPrecSolveFnBS, 155
CVSpilsSetEpsLin, 59
CVSpilsSetEpsLinB, 143
CVSpilsSetJtimesSetupFn, 80
CVSpilsSetJtimesSetupFnB, 153
CVSpilsSetJtimesSetupFnBS, 154
CVSpilsSetJacTimes, 58
CVSpilsSetJacTimesB, 141
CVSpilsSetJacTimesBS, 142
CVSpilsSetLinearSolver, 46
CVSpilsSetLinearSolverB, 137
CVSpilsSetPrecSolveFnB, 142
CVSpilsSetPrecSolveFnBS, 143
eh_data, 76
error control
   order selection, 18–19
   sensitivity variables, 24
   step size selection, 18
error messages, 49
   redirecting, 49
   user-defined handler, 51, 76
fnvector_serial_mod, 176
forward sensitivity analysis
   absolute tolerance selection, 24
   correction strategies, 23–24, 32, 101, 102
   mathematical background, 22–25
   right hand side evaluation, 25
   right-hand side evaluation, 24, 114–116
FSUNBANDLINSOLINIT, 258
FSUNDENSELINSOLINIT, 255
FSUNFIXEDPOINTINIT, 320
FSUNKLUIINIT, 267
FSUNKLUREINIT, 267
FSUNKLUSETORDERING, 268
FSUNLAPACKBANDINIT, 263
FSUNLAPACKDENSEINIT, 260
fsunlinsol_band_mod, 258
fsunlinsol_dense_mod, 255
fsunlinsol_klu_mod, 267
fsunlinsol_pcg_mod, 300
fsunlinsol_spbcgs_mod, 289
fsunlinsol_spfgmr_mod, 283
fsunlinsol_spgr_mod, 276
fsunlinsol_sptfqmr_mod, 294
FSUNMASSBANDLINSOLINIT, 258
FSUNMASSDENSELINSOLINIT, 256
FSUNMASSKLUIINIT, 267
FSUNMASSKLUREINIT, 268
FSUNMASSLUSETORDERING, 268
FSUNMASSLAPACKBANDINIT, 263
FSUNMASSLAPACKDENSEINIT, 260
FSUNMASSPCGINIT, 301
FSUNMASSPCGSETMAXL, 302
FSUNMASSPCGSETPRECTYPE, 301
FSUNSPBCGSINIT, 289
FSUNSPBCGSETMAXL, 290
FSUNSPBCGSETPRECTYPE, 290
FSUNSPBCGSSETMAXL, 290
FSUNSPBCGSSETPRECTYPE, 290
FSUNSPBFGMRSETGSTYPE, 284
FSUNSPBFGMRSETMAXXS, 285
FSUNSPBFGMRSETPRECTYPE, 284
FSUNSPGMRINIT, 276
FSUNSPGMRSETGSTYPE, 277
FSUNSPGMRSETMAXRS, 278
FSUNSPGMRSETPRECTYPE, 277
FSUNSPGMRSETGSTYPE, 277
FSUNSPGMRSETMAXRS, 278
FSUNSPGMRSETPRECTYPE, 277
FSUNSPGMRSETGSTYPE, 277
FSUNSPGMRSETMAXRS, 278
FSUNSPGMRSETPRECTYPE, 277
FSUNSPTFQMRINIT, 295
FSUNSPTFQMRSETMAXL, 296
FSUNSPTFQMRSETPRECTYPE, 295
FSUNSPTFQMRSETGSTYPE, 277
FSUNSPTFQMRSETMAXXS, 278
FSUNSPTFQMRSETPRECTYPE, 295
FSUNSUPERLUMTINIT, 271
FSUNSUPERLUMTSETORDERING, 272
half-bandwidths, 90, 94
header files, 37, 89, 93
itask, 40, 48, 132
iter, 55
Jacobian approximation function
diagonal
   difference quotient, 46
   difference quotient, 56
Jacobian times vector
difference quotient, 57
   user-supplied, 57
Jacobian-vector product
   user-supplied, 79–80
   user-supplied (backward), 141, 151
Jacobian-vector setup, 80
   user-supplied (backward), 153
   user-supplied, 56, 77–79
   user-supplied (backward), 140, 150
Jacobian update frequency
   optional input, 56
Jacobian-vector product
   setup and solve phases, 32
memory requirements
  CVBANDPRE preconditioner, 91
  CVBBDPRE preconditioner, 96
  CVDIAG linear solver interface, 73
  CVLS linear solver interface, 69
  CVODES solver, 84, 102, 118
  CVODES solver, 63

N_VCloneVectorArray, 162
N_VCloneVectorArray_OpenMP, 182
N_VCloneVectorArray_OpenMPDEV, 209
N_VCloneVectorArray_ParHyp, 192
N_VCloneVectorArray_Petsc, 195
N_VCloneVectorArray_Pthreads, 187
N_VCloneVectorArray_Serial, 173
N_VCloneVectorArrayEmpty, 162
N_VCloneVectorArrayEmpty_OpenMP, 183
N_VCloneVectorArrayEmpty_OpenMPDEV, 209
N_VCloneVectorArrayEmpty_ParHyp, 192
N_VCloneVectorArrayEmpty_Petsc, 195
N_VCloneVectorArrayEmpty_Pthreads, 187
N_VCloneVectorArrayEmpty_Serial, 173
N_VCopyFromDevice_Cuda, 201
N_VCopyFromDevice_OpenMPDEV, 210
N_VCopyFromDevice_Pthreads, 206
N_VCopyToDevice_Cuda, 201
N_VCopyToDevice_OpenMPDEV, 210
N_VCopyToDevice_Pthreads, 206
N_VDestroyVectorArray, 162
N_VDestroyVectorArray_OpenMP, 183
N_VDestroyVectorArray_OpenMPDEV, 209
N_VDestroyVectorArray_ParHyp, 192
N_VDestroyVectorArray_Petsc, 195
N_VDestroyVectorArray_Pthreads, 188
N_VDestroyVectorArray_Serial, 173
N_Vector, 37, 161
N_VEnableConstVectorArray_Cuda, 203
N_VEnableConstVectorArray_OpenMP, 184
N_VEnableConstVectorArray_OpenMPDEV, 211
N_VEnableConstVectorArray_ParHyp, 193
N_VEnableConstVectorArray_Petsc, 197
N_VEnableConstVectorArray_Pthreads, 189
N_VEnableConstVectorArray_Raja, 207
N_VEnableConstVectorArray_Serial, 175
N_VEnableDotProdMulti_Cuda, 202
N_VEnableDotProdMulti_OpenMP, 184
N_VEnableDotProdMulti_OpenMPDEV, 211
N_VEnableDotProdMulti_ParHyp, 193
N_VEnableDotProdMulti_Petsc, 196
N_VEnableDotProdMulti_Pthreads, 189
N_VEnableDotProdMulti_Serial, 174
N_VEnableFusedOps_Cuda, 202
N_VEnableFusedOps_OpenMP, 183
N_VEnableFusedOps_OpenMPDEV, 210
N_VEnableFusedOps_ParHyp, 192
N_VEnableFusedOps_Petsc, 196
N_VEnableFusedOps_Pthreads, 188
N_VEnableFusedOps_Serial, 174
N_VEnableLinearCombination_Cuda, 202
N_VEnableLinearCombination_OpenMP, 183
N_VEnableLinearCombination_OpenMPDEV, 210
N_VEnableLinearCombination_ParHyp, 192
N_VEnableLinearCombination_Petsc, 196
N_VEnableLinearCombination_Pthreads, 188
N_VEnableLinearCombination_Raja, 206
N_VEnableLinearCombination_Serial, 174
N_VEnableLinearCombinationVectorArray_Cuda, 203
N_VEnableLinearCombinationVectorArray_OpenMP, 185
N_VEnableLinearCombinationVectorArray_OpenMPDEV, 212
N_VEnableLinearCombinationVectorArray_ParHyp, 180
N_VEnableLinearCombinationVectorArray_Petsc, 197
N_VEnableLinearCombinationVectorArray_Pthreads, 190
N_VEnableLinearCombinationVectorArray_Raja, 207
N_VEnableLinearCombinationVectorArray_Serial, 175
N_VEnableLinearSumVectorArray_Cuda, 202
N_VEnableLinearSumVectorArray_OpenMP, 184
N_VEnableLinearSumVectorArray_OpenMPDEV, 211
N_VEnableLinearSumVectorArray_ParHyp, 193
N_VEnableLinearSumVectorArray_Petsc, 196
N_VEnableLinearSumVectorArray_Pthreads, 189
N_VEnableLinearSumVectorArray_Raja, 207
N_VEnableLinearSumVectorArray_Serial, 174
N_VEnableScaleAddMulti_Cuda, 202
N_VEnableScaleAddMulti_OpenMP, 184
N_VEnableScaleAddMulti_OpenMPDEV, 211
INDEX

359

N_VEnableScaleAddMulti_ParHyp, 193
N_VEnableScaleAddMulti_Petsc, 196
N_VEnableScaleAddMulti_Pthreads, 189
N_VEnableScaleAddMulti_Raja, 207
N_VEnableScaleAddMulti_Serial, 174
N_VEnableScaleAddMultiVectorArray_Cuda, 203
N_VEnableScaleAddMultiVectorArray_OpenMP, 187
N_VEnableScaleAddMultiVectorArray_OpenMPDEV, N_VGetVector_ParHyp, 191
N_VGetLength_Pthreads, 188
N_VGetLength_Raja, 204
N_VGetLength_Serial, 173
N_VGetLocalLength_Cuda, 199
N_VGetLocalLength_Parallel, 178
N_VGetLocalLength_Raja, 204
N_VGetMPIComm_Cuda, 199
N_VGetMPIComm_Raja, 205
N_VMakeManaged, N_VPrintFile_Cuda, 202
N_VPrintFile_OpenMP, 183
N_VPrintFile_OpenMPDEV, 210
N_VPrintFile_Parallel, 178
N_VPrintFile_ParHypl, 192
N_VPrintFile_Petsc, 196
N_VPrintFile_Pthreads, 188
N_VPrintFile_Raja, 206
N_VPrintFile_Serial, 174
N_VSetCudaStream_Cuda, 201
nonlinear system
  Convergence test, 17
  definition, 15–16
  Newton iteration, 17
N_VCOMM_P, 177
NVCONTENT_OMP, 181
NVCONTENT_OMPDEV, 208
NVCONTENT_P, 176
NVCONTENT_PT, 186
NVCONTENT_S, 172
NVDATA_DEV_OMPDEV, 208
NVDATA_HOST_OMPDEV, 208
NVDATA_OMP, 181
NVDATA_P, 177
NVDATA_PT, 186
NVDATA_S, 172
NVGLOBLENGTH_P, 177
NV_Ith_OMP, 182
NV_Ith_P, 177
NV_Ith_PT, 187
NV_Ith_S, 172
NVLENGTH_OMP, 181
NVLENGTH_OMPDEV, 208
NVLENGTH_PT, 186
NVLENGTH_S, 172
NVLOCLENGTH_P, 177
NVNUM_THREADS_OMP, 181
NVNUM_THREADS_PT, 186
NVOWN_DATA_OMP, 181
NVOWN_DATA_OMPDEV, 208
NVOWN_DATA_P, 177
NVOWN_DATA_PT, 186
NVOWN_DATA_S, 172
NVVECTOR module, 161
nvvector_openmp_mod, 185
nvvector_pthreads_mod, 190
sensitivity-dependent quadrature integration, 121–123
  solver, 49–56
optional output
  backward solver, 143–144
  band-block-diagonal preconditioner, 96
  banded preconditioner, 90–91
diagonal linear solver interface, 73–74
  forward sensitivity, 110–114
generic linear solver interface, 69–73
  interpolated quadratures, 85
interpolated sensitivities, 107
interpolated sensitivity-dep. quadratures, 120
interpolated solution, 60
quadrature integration, 87–88, 146
sensitivity-dependent quadrature integration, 123–124
  solver, 61–68
  version, 61
output mode, 19, 48, 132, 138
partial error control
  explanation of CVODES behavior, 125
portability, 36
Preconditioner update frequency
  optional input, 56
preconditioning
  advice on, 19–20, 34
  band-block diagonal, 91
  banded, 89
  setup and solve phases, 34
  user-supplied, 58, 81, 142–143, 154, 155
PVODE, 2
quadrature integration, 22
  forward sensitivity analysis, 25
RCONST, 36
realtype, 36
reinitialization, 74, 135
right-hand side function, 75
  backward problem, 147
  forward sensitivity, 114–116
  quadrature backward problem, 148
  quadrature equations, 88
  sensitivity-dep. quadrature backward problem, 149
  sensitivity-dependent quadrature equations, 124
Rootfinding, 21, 40, 47
second-order sensitivity analysis, 28
  support in CVODES, 29
SM_COLS_B, 229
SM_COLS_D, 223
SUNLinSolSolve, 242, 244
SUNLinSolSpace, 246
SUNMatDestroy, 41
SUNMatrix, 219
SUNMatrix module, 219
SUNNonlinearSolver, 37, 305
SUNNonlinearSolver module, 305
SUNNONLINEARSOLVER_FIXEDPOINT, 306
SUNNONLINEARSOLVER_ROOTFIND, 306
SUNNonlinSol_FixedPoint, 318, 321
SUNNonlinSol_FixedPointSens, 319
SUNNonlinSol_Newton, 315
SUNNonlinSol_NewtonSens, 315
SUNNonlinSolFree, 41, 307
SUNNonlinSolGetCurIter, 309
SUNNonlinSolGetNumConvFails, 309
SUNNonlinSolGetNumIters, 309
SUNNonlinSolGetSysFn_FixedPoint, 319
SUNNonlinSolGetSysFn_Newton, 316
SUNNonlinSolGetType, 306
SUNNonlinSolInitialize, 306
SUNNonlinSolSetupFn, 307
SUNNonlinSolSetConvTestFn, 308
SUNNonlinSolSetLSolveFn, 308
SUNNonlinSolSetMaxIters, 308
SUNNonlinSolSetSysFn, 307
SUNNonlinSolSetup, 306
SUNNonlinSolSolve, 306
SUNSparseFromBandMatrix, 236
SUNSparseFromDenseMatrix, 236
SUNSparseMatrix, 39, 129, 236
SUNSparseMatrix_Columns, 237
SUNSparseMatrix_Data, 238
SUNSparseMatrix_IndexPointers, 238
SUNSparseMatrix_IndexValues, 238
SUNSparseMatrix_NNZ, 79, 238
SUNSparseMatrix_NP, 238
SUNSparseMatrix_Print, 237
SUNSparseMatrix_Realloc, 237
SUNSparseMatrix_Reallocate, 237
SUNSparseMatrix_Rows, 237
SUNSparseMatrix_SparseType, 238

tolerances, 16, 44, 77, 87, 122

UNIT_ROUNDOFF, 36

User main program
- Adjoint sensitivity analysis, 127
- CVBANDBPRE usage, 89
- CVBBDBPRE usage, 93
- forward sensitivity analysis, 97
- integration of quadratures, 82
- integration of sensitivitiy-dependent quadratures, 116

IVP solution, 38
user data, 51, 76–78, 88, 93, 115, 124
user dataB, 159, 160

VODE, 1
VODPK, 1

weighted root-mean-square norm, 16–17