# Example Programs for IDA v2.7.0

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

April 11, 2012

UCRL-SM-208113

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

# 1 Introduction

This report is intended to serve as a companion document to the User Documentation of IDA [2]. It provides details, with listings, on the example programs supplied with the IDA distribution package.

The IDA distribution contains examples of four types: serial C examples, parallel C examples, and serial and parallel FORTRAN examples. With the exception of "demo"-type example files, the names of all the examples distributed with SUNDIALS are of the form `[slv][PbName]_[ls]_[prec]_[p]`, where

**[slv]** identifies the solver (for IDA examples this is `ida`, while for FIDA examples, this is `fida`);

**[PbName]** identifies the problem;

**[ls]** identifies the linear solver module used;

**[prec]** indicates the IDA preconditioner module used (if applicable — for examples using a Krylov linear solver and the IDABBDPRE module, this will be `bbd`);

**[p]** indicates an example using the parallel vector module NVECTOR_PARALLEL.

The following lists summarize all examples distributed with IDA.

The IDA distribution contains, in the *srcdir*`/examples/ida/serial` directory, the following six serial examples (using the NVECTOR_SERIAL module):

- `idaRoberts_dns` solves the Robertson chemical kinetics problem [3], which consists of two differential equations and one algebraic constraint. It also uses the rootfinding feature of IDA.

  The problem is solved with the IDADENSE linear solver using a user-supplied Jacobian.

- `idaSlCrank_dns` solves a system of index-2 DAEs, modeling a planar slider-crank mechanism.

  The problem is obtained through a stabilized index reduction (Gear-Gupta-Leimkuhler) starting from the index-3 DAE equations of motion derived using three generalized coordinates and two algebraic position constraints.

- `idaHeat2D_bnd` solves a 2-D heat equation, semidiscretized to a DAE on the unit square.

  This program solves the problem with the IDABAND linear solver and the default difference-quotient Jacobian approximation. For purposes of illustration, `IDACalcIC` is called to compute correct values at the boundary, given incorrect values as input initial guesses. The constraint $u > 0.0$ is imposed for all components.

- `idaHeat2D_kry` solves the same 2-D heat equation problem as `idaHeat2D_bnd`, with the Krylov linear solver IDASPGMR. The preconditioner uses only the diagonal elements of the Jacobian.

- `idaFoodWeb_bnd` solves a system of PDEs modelling a food web problem, with predator-prey interaction and diffusion, on the unit square in 2-D.

  The PDEs are discretized in space to a system of DAEs which are solved using the IDABAND linear solver with the default difference-quotient Jacobian approximation.

- idaKrylovDemo_ls solves the same problem as idaHeat2D_kry, with three Krylov linear solvers IDASPGMR, IDASPBCG, and IDASPTFQMR. The preconditioner uses only the diagonal elements of the Jacobian.

In the *srcdir*/examples/ida/parallel directory, the IDA distribution contains the following four parallel examples (using the NVECTOR_PARALLEL module):

- idaHeat2D_kry_p solves the same 2-D heat equation problem as idaHeat2D_kry, with IDASPGMR in parallel, and with a user-supplied diagonal preconditioner,
- idaHeat2D_kry_bbd_p solves the same problem as idaHeat2D_kry_p.

  This program uses the Krylov linear solver IDASPGMR in parallel, and the band-block-diagonal preconditioner IDABBDPRE with half-bandwidths equal to 1.
- idaFoodWeb_kry_p solves the same food web problem as idaFoodWeb_bnd, but with IDASPGMR and a user-supplied preconditioner.

  The preconditioner supplied to IDASPGMR is the block-diagonal part of the Jacobian with $n_s \times n_s$ blocks arising from the reaction terms only ($n_s$ = number of species).
- idaFoodWeb_kry_bbd_p solves the same food web problem as idaFoodWeb_kry_p.

  This program solves the problem using IDASPGMR in parallel and the IDABBDPRE preconditioner.

Within the FIDA module, in the two directories *srcdir*/examples/ida/fcmix_serial and *srcdir*/examples/ida/fcmix_parallel, are the following examples for the FORTRAN-C interface:

- fidaRoberts_dns is a serial chemical kinetics example (DENSE) with rootfinding, equivalent to idaRoberts_dns.
- fidaHeat2D_kry_bbd_p is a parallel example (SPGMR/IDABBDPRE) equivalent to the example idaHeat2D_kry_bbd_p.

In the following sections, we give detailed descriptions of some (but not all) of these examples. We also give our output files for each of these examples, but users should be cautioned that their results may differ slightly from these. Solution values may differ within tolerances, and differences in cumulative counters, such as numbers of steps or Newton iterations, may differ from one machine environment to another by as much as 10% to 20%.

In the descriptions below, we make frequent references to the IDA User Document [2]. All citations to specific sections (e.g. §4.2) are references to parts of that User Document, unless explicitly stated otherwise.

**Note**. The examples in the IDA distribution are written in such a way as to compile and run for any combination of configuration options during the installation of SUNDIALS (see Appendix A in the User Guide). As a consequence, they contain portions of code that will not be typically present in a user program. For example, all example programs make use of the variables SUNDIALS_EXTENDED_PRECISION and SUNDIALS_DOUBLE_PRECISION to test if the solver libraries were built in extended or double precision, and use the appropriate conversion specifiers in `printf` functions. Similarly, the FORTRAN examples in FIDA are automatically pre-processed to generate source code that corresponds to the manner in which the IDA libraries were built (see §4 in this document for more details).

## 2 Serial example problems

### 2.1 A dense example: idaRoberts_dns

This example, due to Robertson [3], is a model of a three-species chemical kinetics system written in DAE form. Differential equations are given for species $y_1$ and $y_2$ while an algebraic equation determines $y_3$. The equations for the species concentrations $y_i(t)$ are:

$$\begin{cases} y_1' &= -.04y_1 + 10^4 y_2 y_3 \\ y_2' &= +.04y_1 - 10^4 y_2 y_3 - 3 \cdot 10^7 y_2^2 \\ 0 &= y_1 + y_2 + y_3 - 1\,. \end{cases} \tag{1}$$

The initial values are taken as $y_1 = 1$, $y_2 = 0$, and $y_3 = 0$ This example computes the three concentration components on the interval from $t = 0$ through $t = 4 \cdot 10^{10}$. While integrating the system, the program also use the rootfinding feature to find the points at which $y_1 = 10^{-4}$ or at which $y_3 = 0.01$.

We give a rather detailed explanation of the parts of the program and their interaction with IDA.

Following the initial comment block, this program has a number of `#include` lines, which allow access to useful items in IDA header files. The `sundials_types.h` file provides the definition of the type `realtype` (see §4.2 in the user guide [2] for details). For now, it suffices to read `realtype` as `double`. The `ida.h` file provides prototypes for the IDA functions to be called (excluding the linear solver selection function), and also a number of constants that are to be used in setting input arguments and testing the return value of `IDASolve`. The `ida_dense.h` file provides the prototype for the `IDADense` function. The `nvector_serial.h` file is the header file for the serial implementation of the NVECTOR module and includes definitions of the `N_Vector` type, a macro to access vector components, and prototypes for the serial implementation specific machine environment memory allocation and freeing functions. Finally, note that `ida_dense.h` also includes the `sundials_dense.h` file, which provides the definition of the dense matrix type `DlsMat (type=1)` and a macro for accessing matrix elements.

This program includes the user-defined accessor macro `IJth` that is useful in writing the problem functions in a form closely matching the mathematical description of the DAE system, i.e. with components numbered from 1 instead of from 0. The `IJth` macro is used to access elements of a dense matrix of type `DlsMat`. It is defined using the DENSE accessor macro `DENSE_ELEM` which numbers matrix rows and columns starting with 0. The macro `DENSE_ELEM` is fully described in §4.6.5.

The program prologue ends with prototypes of the three user-supplied functions that are called by IDA and the prototypes of five private functions. Of the latter, the four `Print***` functions perform printing operations, and `check_flag` tests the return flag from the IDA user-callable functions.

After various declarations, the `main` program begins by allocating memory for the `yy`, `yp`, and `avtol` vectors using `N_VNew_Serial` with a length argument of `NEQ` ($= 3$). The lines following that load the initial values of the dependendent variable vectors into `yy` and `yp`, and set the relative tolerance `rtol` and absolute tolerance vector `avtol`. Serial `N_Vector` values are set by first accessing the pointer to their underlying data using the macro `NV_DATA_S` defined by NVECTOR_SERIAL in `nvector_serial.h`.

The calls to `N_VNew_Serial`, and also later calls to `IDA***` functions, make use of a private function, check_flag, which examines the return value and prints a message if there was a failure. This check_flag function was written to be used for any serial SUNDIALS application.

The call to IDACreate creates the IDA solver memory block. The return value of this function is a pointer to the memory block for this problem. In the case of failure, the return value is NULL. This pointer must be passed in the remaining calls to IDA functions.

The call to IDAInit allocates the solver memory block. Its arguments include the name of the C function resrob defining the residual function $F(t, y, y')$, and the initial values of $t$, $y$, and $y'$. The call to IDASVtolerances specifies a vector of absolute tolerances, and this call includes the relative tolerance rtol and the absolute tolerance vector avtol. See §4.5.1 and §4.5.2 for full details of these calls. (The avtol vector is then freed, because IDA keeps a separate copy of it.)

The call to IDARootInit specifies that a rootfinding problem is to be solved along with the integration of the DAE system, that the root functions are specified in the function grob, and that there are two such functions. Specifically, they are set to $y_1 - 0.0001$ and $y_3 - 0.01$, respectively. See §4.5.5 for a detailed description of this call.

The calls to IDADense (see §4.5.3) and IDADlsSetDenseJacFn (see §4.5.7.2) specify the IDADENSE linear solver with an analytic Jacobian supplied by the user-supplied function jacrob.

The actual solution of the DAE initial value problem is accomplished in the loop over values of the output time tout. In each pass of the loop, the program calls IDASolve in the IDA_NORMAL mode, meaning that the integrator is to take steps until it overshoots tout and then interpolate to $t =$tout, putting the computed value of $y($tout$)$ and $y'($tout$)$ into yy and yp, respectively, with tret = tout. The return value in this case is IDA_SUCCESS. However, if IDASolve finds a root before reaching the next value of tout, it returns IDA_ROOT_RETURN and stores the root location in tret and the solution there in yy and yp. In either case, the program prints $t$ (= tret) and yy, and also the cumulative number of steps taken so far, and the current method order and step size. In the case of a root, the program calls IDAGetRootInfo to get a length-2 array rootsfound of bits showing which root function was found to have a root. If IDASolve returned any negative value (indicating a failure), the program breaks out of the loop. In the case of a IDA_SUCCESS return, the value of tout is advanced (multiplied by 10) and a counter (iout) is advanced, so that the loop can be ended when that counter reaches the preset number of output times, NOUT = 12. See §4.5.6 for full details of the call to IDASolve.

Finally, the main program calls PrintFinalStats to extract and print several relevant statistical quantities, such as the total number of steps, the number of residual and Jacobian evaluations, and the number of error test and convergence test failures. It then calls IDAFree to free the IDA memory block and N_VDestroy_Serial to free the vectors yy and yp.

The function PrintFinalStats used here is actually suitable for general use in applications of IDA to any problem with a dense Jacobian. It calls various IDAGet*** and IDADenseGet*** functions to obtain the relevant counters, and then prints them. Specifically, these are: the cumulative number of steps (nst), the number of residual evaluations (nre) (excluding those for difference-quotient Jacobian evaluations), the number of residual evaluations for Jacobian evaluations (nreLS), the number of Jacobian evaluations (nje), the number of nonlinear (Newton) iterations (nni), the number of local error test failures (netf), the number of nonlinear convergence failures (ncfn), and the number of grob (root function) evaluations (nge). These optional outputs are described in §4.5.9.

The functions resrob (of type IDAResFn) and jacrob (of type IDADenseJacFn) are straightforward expressions of the DAE system (1) and its system Jacobian. The function jacrob makes use of the macro IJth discussed above. See §4.6.1 for detailed specifications of IDAResFn. Similarly, the function grob defines the two functions, $g_0$ and $g_1$, whose roots are to be found. See §4.6.4 for a detailed description of the grob function.

The output generated by idaRoberts_dns is shown below. It shows the output values at the 12 preset values of tout. It also shows the two root locations found, first at a root of $g_1$, and then at a root of $g_0$.

idaRoberts_dns sample output

```
idaRoberts_dns: Robertson kinetics DAE serial example problem for IDA
         Three equation chemical kinetics problem.

Linear solver: IDADENSE, with user-supplied Jacobian.
Tolerance parameters:  rtol = 0.0001   atol = 1e-08 1e-14 1e-06
Initial conditions y0 = (1 0 0)
Constraints and id not used.

-----------------------------------------------------------------------
  t             y1           y2           y3      | nst  k      h
-----------------------------------------------------------------------
2.6403e-01   9.8997e-01   3.4706e-05   1.0000e-02 |  85  2   6.4537e-02
    rootsfound[] =    0   1
4.0000e-01   9.8517e-01   3.3864e-05   1.4796e-02 |  88  2   6.4537e-02
4.0000e+00   9.0550e-01   2.2403e-05   9.4473e-02 | 102  4   4.1426e-01
4.0000e+01   7.1582e-01   9.1851e-06   2.8417e-01 | 136  2   1.3422e+00
4.0000e+02   4.5049e-01   3.2226e-06   5.4950e-01 | 190  4   3.3557e+01
4.0000e+03   1.8321e-01   8.9429e-07   8.1679e-01 | 239  4   3.4533e+02
4.0000e+04   3.8984e-02   1.6218e-07   9.6102e-01 | 287  5   2.0140e+03
4.0000e+05   4.9389e-03   1.9852e-08   9.9506e-01 | 339  3   1.6788e+04
4.0000e+06   5.1683e-04   2.0684e-09   9.9948e-01 | 444  4   2.1755e+05
2.0793e+07   1.0000e-04   4.0004e-10   9.9990e-01 | 495  4   1.0146e+06
    rootsfound[] =   -1   0
4.0000e+07   5.2036e-05   2.0816e-10   9.9995e-01 | 506  5   2.5503e+06
4.0000e+08   5.2103e-06   2.0841e-11   9.9999e-01 | 541  4   2.3847e+07
4.0000e+09   5.2125e-07   2.0850e-12   1.0000e-00 | 569  4   3.9351e+08
4.0000e+10   5.1091e-08   2.0437e-13   1.0000e-00 | 589  2   6.0246e+09

Final Run Statistics:

Number of steps                    = 589
Number of residual evaluations     = 832
Number of Jacobian evaluations     = 79
Number of nonlinear iterations     = 832
Number of error test failures      = 14
Number of nonlinear conv. failures = 0
Number of root fn. evaluations     = 624
```

## 2.2 A banded example: idaFoodWeb_bnd

This example is a model of a multi-species food web [1], in which predator-prey relationships with diffusion in a 2-D spatial domain are simulated. Here we consider a model with $s = 2p$ species: $p$ predators and $p$ prey. Species $1, \ldots, p$ (the prey) satisfy rate equations, while species $p + 1, \ldots, s$ (the predators) have infinitely fast reaction rates. The coupled PDEs for the species concentrations $c^i(x,y,t)$ are:

$$\begin{cases} \partial c^i/\partial t = R_i(x,y,c) + d_i(c^i_{xx} + c^i_{yy}) & i = 1,2,\ldots,p \\ \qquad\quad 0 = R_i(x,y,c) + d_i(c^i_{xx} + c^i_{yy}) & i = p+1,\ldots,s\ , \end{cases} \tag{2}$$

with

$$R_i(x,y,c) = c^i \left( b_i + \sum_{j=1}^{s} a_{ij} c^j \right) .$$

Here $c$ denotes the vector $\{c^i\}$. The interaction and diffusion coefficients $(a_{ij}, b_i, d_i)$ can be functions of $(x,y)$ in general. The choices made for this test problem are as follows:

$$a_{ij} = \begin{cases} -1 & i = j \\ -0.5 \cdot 10^{-6} & i \le p,\ j > p \\ 10^4 & i > p,\ j \le p \\ 0 & \text{all other } (i,j)\,, \end{cases}$$

$$b_i = b_i(x,y) = \begin{cases} (1 + \alpha xy + \beta \sin(4\pi x)\sin(4\pi y)) & i \le p \\ -(1 + \alpha xy + \beta \sin(4\pi x)\sin(4\pi y)) & i > p\,, \end{cases}$$

and

$$d_i = \begin{cases} 1 & i \le p \\ 0.5 & i > p\,. \end{cases}$$

The spatial domain is the unit square $0 \le x, y \le 1$, and the time interval is $0 \le t \le 1$. The boundary conditions are of homogeneous Neumann type (zero normal derivatives) everywhere. The coefficients are such that a unique stable equilibrium is guaranteed to exist when $\alpha = \beta = 0$ [1]. Empirically, a stable equilibrium appears to exist for (2) when $\alpha$ and $\beta$ are positive, although it may not be unique. In this problem we take $\alpha = 50$ and $\beta = 1000$. For the initial conditions, we set each prey concentration to a simple polynomial profile satisfying the boundary conditions, while the predator concentrations are all set to a flat value:

$$c^i(x,y,0) = \begin{cases} 10 + i[16x(1-x)y(1-y)]^2 & i \le p\,, \\ 10^5 & i > p\,. \end{cases}$$

We discretize this PDE system (2) (plus boundary conditions) with central differencing on an $L \times L$ mesh, so as to obtain a DAE system of size $N = sL^2$. The dependent variable vector $C$ consists of the values $c^i(x_j, y_k, t)$ grouped first by species index $i$, then by $x$, and lastly by $y$. At each spatial mesh point, the system has a block of $p$ ODE's followed by a block of $p$ algebraic equations, all coupled. For this example, we take $p = 1, s = 2$, and $L = 20$. The Jacobian is banded, with half-bandwidths `mu = ml` $= sL = 40$.

The `idaFoodWeb_bnd.c` program includes the file `ida_band.h` in order to use the IDABAND linear solver. This file contains the prototype for the `IDABand` routine, the definition for the band matrix type `DlsMat (type=2)`, and the `BAND_COL` and `BAND_COL_ELEM` macros for accessing matrix elements. See §8.1.4. The main IDA header file `ida.h` is included for the prototypes of the solver user-callable functions and IDA constants, while the file `nvector_serial.h` is included for the definition of the serial `N_Vector` type. The header file `sundials_dense.h` is included for the `newDenseMat` function used in allocating memory for the user data structure.

The include lines at the top of the file are followed by definitions of problem constants which include the $x$ and $y$ mesh dimensions, `MX` and `MY`, the number of equations `NEQ`, the scalar relative and absolute tolerances `RTOL` and `ATOL`, and various parameters for the food-web problem.

Spatial discretization of the PDE naturally produces a DAE system in which equations are numbered by mesh coordinates $(i, j)$. The user-defined macro `IJth_Vptr` isolates the translation for the mathematical two-dimensional index to the one-dimensional `N_Vector` index and allows the user to write clean, readable code to access components of the dependent variable. `IJ_Vptr(v,i,j)` returns a pointer to the location in `v` corresponding to the species with index `is` $= 0$, x-index `ix` $= i$, and y-index `jy` $= j$.

The type `UserData` is a pointer to a structure containing problem data used in the `resweb` function. This structure is allocated and initialized at the beginning of `main`. The pointer to it, called `webdata`, is then passed to `IDASetUserData` and as a result it will be passed back to the `resweb` function each time it is called.

The `main` program is straightforward and very similar to that for `idaRoberts_dns`. The differences come from the use of the IDABAND linear solver and from the use of the consistent initial conditions algorithm in IDA to correct the initial values. The call to `IDABand` includes the half-bandwidths `ml` and `mu`. `IDACalcIC` is called with the option `IDA_YA_YDP_INIT`, meaning that IDA is to compute the algebraic components of $y$ and differential components of $y'$, given the differential components of $y$. This option requires that the `N_Vector id` be set through a call to `IDASetId` specifying the differential and algebraic components. In this example, `id` has components equal to 1 for the prey (indicating differential variables) and 0 for the predators (algebraic variables).

Next, the `IDASolve` function is called in a loop over the output times, and the solution for the species concentrations at the bottom-left and top-right corners is printed, along with the cumulative number of time steps, current method order, and current step size.

Finally, the main program calls `PrintFinalStats` to get and print all of the relevant statistical quantities. It then calls `N_VDestroy_Serial` to free the vectors `cc`, `cp`, and `id`, and `IDAFree` to free the IDA memory block.

The function `PrintFinalStats` used here is actually suitable for general use in applications of IDA to any problem with a banded Jacobian. It calls various `IDAGet***` and `IDABandGet***` functions to obtain the relevant counters, and then prints them. Specifically, these are: the cumulative number of steps (`nst`), the number of residual evaluations (`nre`) (excluding those for difference-quotient Jacobian evaluations), the number of residual evaluations for Jacobian evaluations (`nreLS`), the number of Jacobian evaluations (`nje`), the number of nonlinear (Newton) iterations (`nni`), the number of local error test failures (`netf`), and the number of nonlinear convergence failures (`ncfn`). These optional outputs are described in §4.5.9.

The function `resweb` is a direct translation of the residual of (2). It first calls the private function `Fweb` to initialize the residual vector with the right-hand side of (2) and then it loops over all grid points, setting residual values appropriately for differential or algebraic components. The calculation of the interaction terms $R_i$ is done in the function `WebRates`.

Sample output from `idaFoodWeb_bnd` follows.

idaFoodWeb_bnd sample output

```

idaFoodWeb_bnd: Predator-prey DAE serial example problem for IDA

Number of species ns: 2     Mesh dimensions: 20 x 20     System size: 800
```

```
Tolerance parameters:  rtol = 1e-05   atol = 1e-05
Linear solver: IDABAND,  Band parameters mu = 40, ml = 40
CalcIC called to correct initial predator concentrations.

-----------------------------------------------------------
  t        bottom-left  top-right    | nst  k      h
-----------------------------------------------------------

0.00e+00   1.0000e+01   1.0000e+05   |   0  0   1.6310e-08
           1.0000e+05   1.0000e+05   |

1.00e-03   1.0318e+01   1.0822e+05   |  32  4   1.0823e-04
           1.0319e+05   1.0822e+05   |

1.00e-02   1.6188e+02   1.9734e+06   | 127  4   1.4203e-04
           1.6189e+06   1.9734e+06   |

1.00e-01   2.4019e+02   2.7072e+06   | 235  1   3.9160e-02
           2.4019e+06   2.7072e+06   |

4.00e-01   2.4019e+02   2.7072e+06   | 238  1   3.1328e-01
           2.4019e+06   2.7072e+06   |

7.00e-01   2.4019e+02   2.7072e+06   | 239  1   6.2657e-01
           2.4019e+06   2.7072e+06   |

1.00e+00   2.4019e+02   2.7072e+06   | 239  1   6.2657e-01
           2.4019e+06   2.7072e+06   |

-----------------------------------------------------------
Final run statistics:

Number of steps                    = 239
Number of residual evaluations     = 3339
Number of Jacobian evaluations     = 36
Number of nonlinear iterations     = 421
Number of error test failures      = 3
Number of nonlinear conv. failures = 0
```

### 2.3 A Krylov example: idaHeat2D_kry

This example solves a discretized 2D heat PDE problem. The DAE system arises from the Dirichlet boundary condition $u = 0$, along with the differential equations arising from the discretization of the interior of the region.

The domain is the unit square $\Omega = \{0 \le x, y \le 1\}$ and the equations solved are:

$$\begin{cases} \partial u/\partial t = u_{xx} + u_{yy} & (x,y) \in \Omega \\ u = 0 & (x,y) \in \partial\Omega\,. \end{cases} \tag{3}$$

The time interval is $0 \le t \le 10.24$, and the initial conditions are $u = 16x(1-x)y(1-y)$.

We discretize the PDE system (3) (plus boundary conditions) with central differencing on a $10 \times 10$ mesh, so as to obtain a DAE system of size $N = 100$. The dependent variable vector $u$ consists of the values $u(x_j, y_k, t)$ grouped first by $x$, and then by $y$. Each discrete boundary condition becomes an algebraic equation within the DAE system.

In this case, `ida_spgmr.h` is included for the definitions of constants and function prototypes associated with the SPGMR method.

After various initializations (including a vector of constraints with all components set to 1, imposing all solution components to be non-negative), the main program creates and initializes the IDA memory block and then attaches the IDASPGMR linear solver using the default `MODIFIED_GS` Gram-Scmidt orthogonalization algorithm.

The user-supplied preconditioner setup and solve functions, `PsetupHeat` and `PsolveHeat`, and the pointer to user data (`data`) are specified in a call to `IDASpilsSetPreconditioner`. In a loop over the desired output times, `IDASolve` is called in `IDA_NORMAL` mode and the maximum solution norm is printed. Following this, three more counters are printed.

The `main` program then re-initializes the IDA solver and the IDASPGMR linear solver and solves the problem again, this time using the `CLASSICAL_GS` Gramm-Schmidt orthogonalization algorithm. Finally, memory for the IDA solver and for the various vectors used is deallocated.

The user-supplied residual function `resHeat`, of type `IDAResFn`, loads the DAE residual with the value of $u$ on the boundary (representing the algebraic equations expressing the boundary conditions of (3)) and with the spatial discretization of the PDE (using central differences) in the rest of the domain.

The user-supplied functions `PsetupHeat` and `PsolveHeat` together define the left preconditoner matrix $P$ approximating the system Jacobian matrix $J = \partial F/\partial u + \alpha \partial F/\partial u'$ (where the DAE system is $F(t, u, u') = 0$), and solve the linear systems $Pz = r$. Preconditioning is done in this case by keeping only the diagonal elements of the $J$ matrix above, storing them as inverses in a vector `pp`, when computed in `PsetupHeat`, for subsequent use in `PsolveHeat`. In this instance, only `cj` $= \alpha$ and `data` (the user data structure) are used from the `PsetupHeat` argument list.

Sample output from `idaHeat2D_kry` follows.

idaHeat2D_kry sample output

```

idaHeat2D_kry: Heat equation, serial example problem for IDA
         Discretized heat equation on 2D unit square.
         Zero boundary conditions, polynomial initial conditions.
         Mesh dimensions: 10 x 10        Total system size: 100

Tolerance parameters:  rtol = 0   atol = 0.001
Constraints set to force all solution components >= 0.
Linear solver: IDASPGMR, preconditioner using diagonal elements.


Case 1: gsytpe = MODIFIED_GS

   Output Summary (umax = max-norm of solution)

  time     umax       k  nst  nni  nje   nre   nreLS    h      npe nps
-----------------------------------------------------------------------
  0.01   8.24106e-01  2   12   14    7    14     7   2.56e-03    8  21
  0.02   6.88134e-01  3   15   18   12    18    12   5.12e-03    8  30
  0.04   4.70711e-01  3   18   24   21    24    21   6.58e-03    9  45
  0.08   2.16509e-01  3   22   29   30    29    30   1.32e-02    9  59
  0.16   4.57687e-02  4   28   36   44    36    44   1.32e-02    9  80
  0.32   2.09938e-03  4   35   44   67    44    67   2.63e-02   10 111
  0.64   0.00000e+00  1   39   51   77    51    77   1.05e-01   12 128
  1.28   0.00000e+00  1   41   53   77    53    77   4.21e-01   14 130
```

```
  2.56   0.00000e+00  1   43   55   77    55    77   1.69e+00   16  132
  5.12   0.00000e+00  1   44   56   77    56    77   3.37e+00   17  133
 10.24   0.00000e+00  1   45   57   77    57    77   6.74e+00   18  134

Error test failures            = 1
Nonlinear convergence failures = 0
Linear convergence failures    = 0


Case 2: gstype = CLASSICAL_GS

   Output Summary (umax = max-norm of solution)

  time     umax       k  nst  nni  nje   nre   nreLS    h       npe nps
----------------------------------------------------------------------
  0.01   8.24106e-01  2   12   14    7    14     7   2.56e-03    8   21
  0.02   6.88134e-01  3   15   18   12    18    12   5.12e-03    8   30
  0.04   4.70711e-01  3   18   24   21    24    21   6.58e-03    9   45
  0.08   2.16509e-01  3   22   29   30    29    30   1.32e-02    9   59
  0.16   4.57687e-02  4   28   36   44    36    44   1.32e-02    9   80
  0.32   2.09938e-03  4   35   44   67    44    67   2.63e-02   10  111
  0.64   2.15648e-20  1   39   51   77    51    77   1.05e-01   12  128
  1.28   1.30250e-20  1   41   53   77    53    77   4.21e-01   14  130
  2.56   3.00951e-20  1   43   55   77    55    77   1.69e+00   16  132
  5.12   7.38674e-20  1   44   56   77    56    77   3.37e+00   17  133
 10.24   1.79685e-19  1   45   57   77    57    77   6.74e+00   18  134

Error test failures            = 1
Nonlinear convergence failures = 0
Linear convergence failures    = 0
```

# 3 Parallel example problems

## 3.1 A user preconditioner example: idaHeat2D_kry_p

As an example of using IDA with the parallel MPI NVECTOR_PARALLEL module and the Krylov linear solver IDASPGMR with user-defined preconditioner, we provide the example `idaHeat2D_kry_p` which solves the same 2-D heat PDE problem as `idaHeat2D_kry`.

In the parallel setting, we can think of the processors as being laid out in a grid of size `NPEX` × `NPEY`, with each processor computing a subset of the solution vector on a submesh of size `MXSUB` × `MYSUB`. As a consequence, the computation of the residual vector requires that each processor exchange boundary information (namely the components at all interior subgrid boundaries) with its neighboring processors. The message-passing (implemented in the function `rescomm`) uses blocking sends, non-blocking receives, and receive-waiting, in routines `BSend`, `BRecvPost`, and `BRecvWait`, respectively. The data received from each neighboring processor is then loaded into a work array, `uext`, which contains this ghost cell data along with the local portion of the solution.

The local portion of the residual vector is then computed in the routine `reslocal`, which assumes that all inter-processor communication of data needed to calculate `rr` has already been done. Components at interior subgrid boundaries are assumed to be in the work array `uext`. The local portion of the solution vector `uu` is first copied into `uext`. The diffusion terms are evaluated in terms of the `uext` array, and the residuals are formed. The zero Dirichlet boundary conditions are handled here by including the boundary components in the residual, giving algebraic equations for the discrete boundary conditions.

The preconditioner (implemented in `PsetupHeat` and `PsolveHeat`) uses the diagonal elements of the Jacobian only and therefore involves only local calculations.

The `idaHeat2D_kry_p` main program begins with MPI calls to initialize MPI and to set multi-processor environment parameters `npes` (number of processes) and `thispe` (local process index). Then the local and global vector lengths are set, the user-data structure `Userdata` is created and initialized, and `N_Vector` variables are created and initialized for the initial conditions (`uu` and `up`), for constraints, for the vector `id` specifying the differential and algebraic components of the solution vector, and for the preconditioner (`pp`). As in `idaHeat2D_kry`, constraints are passed to IDA through the `N_Vector constraints` and the function `IDASetConstraints`, with all components set to 1.0 to impose non-negativity on each solution component. A temporary `N_Vector res` is also created here, for use only in `SetInitialProfiles`. In addition, for illustration purposes, `idaHeat2D_kry_p` also excludes the algebraic components of the solution (specified through the `N_Vector id`) from the error test by calling `IDASetSuppressAlg` with a flag `TRUE`.

Sample output from `idaHeat2D_kry_p` follows.

idaHeat2D_kry_p sample output

```

idaHeat2D_kry_p: Heat equation, parallel example problem for IDA
             Discretized heat equation on 2D unit square.
             Zero boundary conditions, polynomial initial conditions.
             Mesh dimensions: 10 x 10        Total system size: 100

Subgrid dimensions: 5 x 5        Processor array: 2 x 2
Tolerance parameters:  rtol = 0   atol = 0.001
Constraints set to force all solution components >= 0.
SUPPRESSALG = TRUE to suppress local error testing on all boundary components.
Linear solver: IDASPGMR  Preconditioner: diagonal elements only.
```

```
    Output Summary (umax = max-norm of solution)

   time      umax       k  nst  nni  nli   nre   nreLS    h      npe nps
--------------------------------------------------------------------------
   0.00   9.75461e-01  0    0    0    0     0     0   0.00e+00    0   0
   0.01   8.24106e-01  2   12   14    7    14     7   2.56e-03    8  21
   0.02   6.88134e-01  3   15   18   12    18    12   5.12e-03    8  30
   0.04   4.70711e-01  3   18   24   21    24    21   6.58e-03    9  45
   0.08   2.16509e-01  3   22   29   30    29    30   1.32e-02    9  59
   0.16   4.57687e-02  4   28   36   44    36    44   1.32e-02    9  80
   0.32   2.09938e-03  4   35   44   67    44    67   2.63e-02   10 111
   0.64   5.54028e-21  1   39   51   77    51    77   1.05e-01   12 128
   1.28   3.85107e-20  1   41   53   77    53    77   4.21e-01   14 130
   2.56   5.00523e-20  1   43   55   77    55    77   1.69e+00   16 132
   5.12   1.50906e-19  1   44   56   77    56    77   3.37e+00   17 133
  10.24   4.63224e-19  1   45   57   77    57    77   6.74e+00   18 134

Error test failures            = 1
Nonlinear convergence failures = 0
Linear convergence failures    = 0
```

### 3.2 An IDABBDPRE preconditioner example: idaFoodWeb_kry_bbd_p

In this example, we solve the same food web problem as with `idaFoodWeb_bnd`, but in parallel and with the IDASPGMR linear solver and using the IDABBDPRE module, which generates and uses a band-block-diagonal preconditioner.

As with `idaHeat2D_kry_p`, we use a `NPEX` $\times$ `NPEY` processor grid, with an `MXSUB` $\times$ `MYSUB` submesh on each processor. Again, the residual evaluation begins with the communication of ghost data (in `rescomm`), followed by computation using an extended local array, `cext`, in the `reslocal` routine. The exterior Neumann boundary conditions are explicitly handled here by copying data from the first interior mesh line to the ghost cell locations in `cext`. Then the reaction and diffusion terms are evaluated in terms of the `cext` array, and the residuals are formed.

The Jacobian block on each processor is banded, and the half-bandwidths of that block are both equal to `NUM_SPECIES`$\cdot$`MXSUB`. This is the value supplied as `mudq` and `mldq` in the call to `IDABBDPrecInit`. But in order to reduce storage and computation costs for preconditioning, we supply the values `mukeep` = `mlkeep` = 2 (= `NUM_SPECIES`) as the half-bandwidths of the retained band matrix blocks. This means that the Jacobian elements are computed with a difference quotient scheme using the true bandwidth of the block, but only a narrow band matrix (bandwidth 5) is kept as the preconditioner.

The function `reslocal` is also passed to the IDABBDPRE preconditioner as the `Gres` argument, while a `NULL` pointer is passed for the `Gcomm` argument (since all required communication for the evaluation of `Gres` was already done for `resweb`).

In the `idaFoodWeb_kry_bbd_p` main program, following MPI initializations and creation of user data block `webdata` and `N_Vector` variables, the initial profiles are set, the IDA memory block is created and allocated, the IDABBDPRE preconditioner is initialized, and the IDASPGMR linear solver is attached to the IDA solver. The call to `IDACalcIC` corrects the initial values so that they are consistent with the DAE algebraic constraints.

In a loop over the desired output times, the main solver function `IDASolve` is called, and selected solution components (at the bottom-left and top-right corners of the computational domain) are collected on processor 0 and printed to `stdout`. The main program ends by printing final solver statistics, freeing memory, and finalizing MPI.

Sample output from `idaFoodWeb_kry_bbd_p` follows.

idaFoodWeb_kry_bbd_p sample output

```
idaFoodWeb_kry_bbd_p: Predator-prey DAE parallel example problem

Number of species ns: 2
Mesh dimensions:      20 x 20
Total system size:    800
Subgrid dimensions:   10 x 10
Processor array:      2 x 2
Tolerance parameters:
 relative tolerance = 1e-05
 absolute tolerance = 1e-05
Linear solver: scaled preconditioned GMRES (IDASPGMR)
 max. Krylov dimension: maxl = 16
Preconditioner: band-block-diagonal (IDABBDPRE)
 mudq = 20,  mldq = 20,  mukeep = 2,  mlkeep = 2
CalcIC called to correct initial predator concentrations

-----------------------------------------------------------
  t         bottom-left  top-right    | nst  k      h
-----------------------------------------------------------

0.00e+00   1.0000e+01   1.0000e+01   |   0  0   1.6310e-08
           1.0000e+05   1.0000e+05   |

1.00e-03   1.0318e+01   1.0827e+01   |  33  4   9.7404e-05
           1.0319e+05   1.0822e+05   |

1.00e-02   1.6189e+02   1.9735e+02   | 123  3   1.9481e-04
           1.6189e+06   1.9735e+06   |

1.00e-01   2.4019e+02   2.7072e+02   | 197  1   4.0396e-02
           2.4019e+06   2.7072e+06   |

4.00e-01   2.4019e+02   2.7072e+02   | 200  1   3.2316e-01
           2.4019e+06   2.7072e+06   |

7.00e-01   2.4019e+02   2.7072e+02   | 200  1   3.2316e-01
           2.4019e+06   2.7072e+06   |

1.00e+00   2.4019e+02   2.7072e+02   | 201  1   6.4633e-01
           2.4019e+06   2.7072e+06   |

-----------------------------------------------------------

Final statistics:

Number of steps                    = 201
Number of residual evaluations     = 1110
Number of nonlinear iterations     = 245
Number of error test failures      = 0
Number of nonlinear conv. failures = 0

Number of linear iterations        = 863
```

```
Number of linear conv. failures    = 0

Number of preconditioner setups    = 26
Number of preconditioner solves    = 1110
Number of local residual evals.    = 1092
```

# 4 Fortran example problems

The Fortran example problem programs supplied with the IDA package are all written in standard Fortran77 and use double precision arithmetic. However, when the Fortran examples are built, the source code is automatically modified according to the configure options supplied by the user and the system type. Integer variables are declared as `INTEGER*`$n$, where $n$ denotes the number of bytes in the corresponding C type (`long int` or `int`). Floating-point variable declarations remain unchanged if double precision is used, but are changed to `REAL*`$n$, where $n$ denotes the number of bytes in the SUNDIALS type `realtype`, if using single precision. Also, if using single precision, then declarations of floating-point constants are appropriately modified; e.g. `0.5D-4` is changed to `0.5E-4`.

## 4.1 A serial example: fidaRoberts_dns

The `fidaRoberts_dns` example is a Fortran equivalent of the `idaRoberts_dns` example.

The main program begins with declarations and initializations. It calls the routines `FNVINITS`, `FIDAMALLOC`, `FIDAROOTINIT`, `FIDADENSE`, and `FIDADENSESETJAC`, to initialize the NVECTOR_SERIAL module, the main solver memory, the rootfinding module, and the IDADENSE module, and to specify user-supplied Jacobian routine, respectively. It calls `FIDASOLVE` in a loop over `TOUT` values, with printing of the solution values and performance data (current order and step count from the `IOUT` array, and current step size from the `ROUT` array). In the case of a root return, an extra line is printed with the root information from `FIDAROOTINFO`. At the end, it prints a number of performance counters, and frees memory with calls to `FIDAROOTFREE` and `FIDAFREE`.

In `fidaRoberts_dns.f`, the `FIDARESFUN` routine is a straghtforward implementation of Eqns. (1). In `FIDADJAC`, the $3 \times 3$ system Jacobian is supplied. The `FIDAROOTFN` routine defines the two root functions, which are set to determine the points at which $y_1 = 10^{-4}$ or $y_3 = .01$. The final two routines are for printing a header and the final run statistics.

The following is sample output from `fidaRoberts_dns`. The performance of FIDA here is similar to that of IDA on the `idaRoberts_dns` problem, with somewhat lower cost counters owing to the larger absolute error tolerances.

fidaRoberts_dns sample output

```

fidaRoberts_dns: Robertson kinetics DAE serial exampleproblem for IDA
          Three equation chemicalkinetics problem.

Tolerance parameters:  rtol = 0.10E-03   atol =  0.10E-05 0.10E-09 0.10E-05
Initial conditions y0 = ( 0.10E+01 0.00E+00 0.00E+00)

  t             y1            y2            y3          nst  k    h
0.2640E+00   0.9900E+00   0.3471E-04   0.1000E-01   75  2  0.5716E-01
     Above is a root, INFO() =   0  1
0.4000E+00   0.9852E+00   0.3386E-04   0.1480E-01   77  3  0.1143E+00
0.4000E+01   0.9055E+00   0.2240E-04   0.9447E-01   91  4  0.3704E+00
0.4000E+02   0.7158E+00   0.9185E-05   0.2842E+00  127  4  0.2963E+01
0.4000E+03   0.4505E+00   0.3223E-05   0.5495E+00  177  3  0.1241E+02
0.4000E+04   0.1832E+00   0.8940E-06   0.8168E+00  228  3  0.2765E+03
0.4000E+05   0.3899E-01   0.1622E-06   0.9610E+00  278  5  0.2614E+04
0.4000E+06   0.4939E-02   0.1985E-07   0.9951E+00  324  5  0.2770E+05
0.4000E+07   0.5176E-03   0.2072E-08   0.9995E+00  355  4  0.3979E+06
0.2075E+08   0.1000E-03   0.4000E-09   0.9999E+00  374  4  0.1592E+07
```

```
     Above is a root, INFO() =  -1  0
0.4000E+08   0.5191E-04   0.2076E-09   0.9999E+00  380  3  0.6366E+07
0.4000E+09   0.5882E-05   0.2353E-10   0.1000E+01  394  1  0.9167E+08
0.4000E+10   0.7054E-06   0.2822E-11   0.1000E+01  402  1  0.1467E+10
0.4000E+11  -0.7300E-06  -0.2920E-11   0.1000E+01  407  1  0.2347E+11

Final Run Statistics:

Number of steps                    = 407
Number of residual evaluations     = 557
Number of Jacobian evaluations     =  65
Number of nonlinear iterations     = 557
Number of error test failures      =   6
Number of nonlinear conv. failures =   0
Number of root function evals.     = 437
```

### 4.2 A parallel example: fidaHeat2D_kry_bbd_p

This example, `fidaHeat2D_kry_bbd_p`, is the Fortran equivalent of `idaHeat2D_kry_bbd_p`. The heat equation problem is described under the `idaHeat2D_kry` example above, but here it is solved in parallel, using the IDABBDPRE (band-block-diagonal) preconditioner module. The decomposition of the problem onto a processor array is identical to that in the `idaHeat2D_kry_p` example above.

The problem is solved twice — once with half-bandwidths of 5 in the difference-quotient banded preconditioner blocks, and once with half-bandwidths of 1 (which results in lumping of Jacobian values). In both cases, the retained banded blocks are tridiagonal, even though the true Jacobian is not.

The main program begins with initializations, including MPI calls, a call to `FNVINITP` to initialize NVECTOR_PARALLEL, and a call to `SETINITPROFILE` to initialize the `UU`, `UP`, `ID`, and `CONSTR` arrays (containing the solution vector, solution derivative vector, the differential/algebraic bit vector, and the contraint specification vector, respectively). A call to `FIDASETIIN` and two calls to `FIDASETVIN` are made to suppress error control on the algebraic variables, and to supply the `ID` array and constraints array (making the computed solution non-negative). The call to `FIDAMALLOC` initializes the FIDA main memory, and the calls to `FIDASPGMR` and `FIDABBDINIT` and initialize the FIDABBD module.

In the first loop over `TOUT` values, the main program calls `FIDASOLVE` and prints the max-norm of the solution and selected counters. When finished, it calls `PRNTFINALSTATS` to print a few more counters.

The second solution is initialized by resetting `mudq` and `mldq`, followed by a second call to `SETINITPROFILE`, and by calls to `FIDAREINIT` and `FIDABBDREINIT`. After completing the second solution, the program frees memory and terminates MPI.

The `FIDARESFUN` routine simply calls two other routines: `FIDACOMMFN`, to communicate needed boundary data from `U` to an extension of it called `UEXT`; and `FIDAGLOCFN`, to compute the residuals in terms of `UEXT` and `UP`.

The following is a sample output from `fidaHeat2D_kry_bbd_p`, with a $10 \times 10$ mesh and `NPES` $= 4$ processors. The performance is similar for the two solutions. The second case requires more linear iterations, as expected, but their cost is offset by the much cheaper preconditioner evaluations.

fidaHeat2D_kry_bbd_p sample output

```
fidaHeat2D_kry_bbd_p: Heat equation, parallel example problem for FIDA
                Discretized heat equation on 2D unit square.
                Zero boundary conditions, polynomial conditions.
                Mesh dimensions: 10 x 10         Total system size: 100

Subgrid dimensions:  5 x  5           Processor array:  2 x  2
Tolerance parameters: rtol = 0.00E+00   atol = 0.10E-02
Constraints set to force all solution components >= 0.
SUPPRESSALG = TRUE to remove boundary components from the error test.
Linear solver: SPGMR.    Preconditioner: BBDPRE - Banded-block-diagonal.


Case  1
  Difference quotient half-bandwidths = 5
  Retained matrix half-bandwidths = 1

Output Summary
  umax = max-norm of solution
  nre = nre + nreLS (total number of RES evals.)

   time          umax       k nst  nni  nli   nre   nge      h     npe  nps
---------------------------------------------------------------------------
 0.1000E-01   0.82411E+00  2  12   14    7  14+ 7   96   0.26E-02   8   21
 0.2000E-01   0.68812E+00  3  15   18   12  18+12   96   0.51E-02   8   30
 0.4000E-01   0.47075E+00  3  18   24   22  24+22  108   0.66E-02   9   46
 0.8000E-01   0.21660E+00  3  22   29   30  29+30  108   0.13E-01   9   59
 0.1600E+00   0.45659E-01  4  28   37   43  37+43  120   0.26E-01  10   80
 0.3200E+00   0.21096E-02  4  35   45   59  45+59  120   0.24E-01  10  104
 0.6400E+00   0.55368E-04  1  40   54   71  54+71  156   0.19E+00  13  125
 0.1280E+01   0.15597E-18  1  42   56   71  56+71  180   0.76E+00  15  127
 0.2560E+01   0.33865E-20  1  43   57   71  57+71  192   0.15E+01  16  128
 0.5120E+01   0.86074E-20  1  44   58   71  58+71  204   0.30E+01  17  129
 0.1024E+02   0.16630E-19  1  45   59   71  59+71  216   0.61E+01  18  130

Error test failures            =  1
Nonlinear convergence failures =  0
Linear convergence failures    =  0


Case  2
  Difference quotient half-bandwidths = 1
  Retained matrix half-bandwidths = 1

Output Summary
  umax = max-norm of solution
  nre = nre + nreLS (total number of RES evals.)

   time          umax       k nst  nni  nli   nre   nge      h     npe  nps
---------------------------------------------------------------------------
 0.1000E-01   0.82411E+00  2  12   14    7  14+ 7   32   0.26E-02   8   21
 0.2000E-01   0.68812E+00  3  15   18   12  18+12   32   0.51E-02   8   30
 0.4000E-01   0.47093E+00  3  19   23   20  23+20   36   0.10E-01   9   43
 0.8000E-01   0.21655E+00  3  23   27   32  27+32   36   0.10E-01   9   59
 0.1600E+00   0.45225E-01  4  27   33   44  33+44   40   0.20E-01  10   77
 0.3200E+00   0.21868E-02  3  34   41   67  41+67   44   0.41E-01  11  108
 0.6400E+00   0.48847E-18  1  39   49   86  49+86   52   0.16E+00  13  135
```

```
 0.1280E+01   0.53982E-18  1  41   51   86  51+86   60   0.66E+00  15  137
 0.2560E+01   0.74194E-17  1  42   52   86  52+86   64   0.13E+01  16  138
 0.5120E+01   0.61081E-16  1  43   53   86  53+86   68   0.26E+01  17  139
 0.1024E+02   0.40536E-15  1  44   54   86  54+86   72   0.52E+01  18  140

Error test failures            =  0
Nonlinear convergence failures =  0
Linear convergence failures    =  0
```

# References

[1] Peter N. Brown. Decay to uniform states in food webs. *SIAM J. Appl. Math.*, 46:376–392, 1986.

[2] A. C. Hindmarsh, R. Serban, and A. Collier. User Documentation for IDA v2.7.0. Technical Report UCRL-SM-208112, LLNL, 2011.

[3] H. H. Robertson. The solution of a set of reaction rate equations. In J. Walsh, editor, *Numerical analysis: an introduction*, pages 178–182. Academ. Press, 1966.