FATODE-1.0 user guide

FATODE:a library for forward, adjoint, and tangent linear integration of ODEs

Hong Zhang* and Adrian Sandu†

November 12, 2012

*Email: zhang@vt.edu
†Email: sandu@cs.vt.edu

Contents

1 Introduction **3**

2 Integrators in FATODE **5**
2.1 Forward solution 5
2.2 Tangent linear model integration 7
2.3 Adjoint model integration 7

3 Code Organization **9**
3.1 Logical structure 9
3.2 Directory structure 10

4 How to use FATODE **14**
4.1 Prerequisites 14
4.2 Optional third party libraries for sparse linear algebra 15
4.3 Makefile 15
4.4 User supplied functions 18

5 User interface **24**
5.1 Options 24
5.2 Forward ODE integration 27
5.3 Tangent linear methods and forward sensitivity calculations 28
5.4 Adjoint methods and discrete adjoint sensitivity calculations 28

6 Example **30**
6.1 Description of the example problem 30
6.2 Using FATODE for IVP 30
6.3 Using FATODE for direct sensitivity analysis 31
6.4 Using FATODE for discrete adjoint sensitivity analysis 32

7 FATODE linear solvers **33**

8 Automatic Differentiation **34**

Bibliography **37**

1 Introduction

FATODE (forward, adjoint, and tangent linear integration of ODEs) is a library of explicit/implicit Runge-Kutta and Rosenbrock solvers for the simulation of nonstiff and stiff ODEs. The library performs forward simulations, and sensitivity analysis via the discrete adjoint and the tangent linear methods. FATODE is partially based on the numerical library implemented in the Kinetic PreProcessor KPP [1, 2] which is a widely used tool for the simulation of chemical kinetics.

FATODE provides a variety of integrators for solving the initial value problem:

$$y' = f\left(t, y; p\right), \quad t_0 \le t \le t_f, \quad y(t_0) = y_0, \tag{1.1}$$

where $y(t) \in \mathbb{R}^d$ is the solution vector, y_0 the initial condition, and $p \in \mathbb{R}^m$ a vector of model parameters.

Stiffness results from the existence of multiple dynamical scales, with the fastest characteristic times being much smaller than the time scales of interest in the simulation. It is well known that the numerical solution of stiff systems requires unconditionally stable discretizations which allow time steps that are not bounded by the fastest time scales in the system [3]. Here we assume that the system parameters p are independent of time. In the context of the ODE system (1.1), sensitivity analysis yields derivatives of the solution with respect to the initial conditions or system parameters, as follows

$$S_\ell(t) = \frac{\partial y(t)}{\partial p_\ell}, \; 1 \le \ell \le m. \tag{1.2}$$

Two main approaches are available in FATODE for computing the sensitivities (1.2). The direct (or tangent linear) method is efficient when the number of parameters is smaller than the dimension of the system ($m \ll d$), while the adjoint method is efficient when the number of parameters is larger than the dimension of the system ($m \gg d$).

This guide is organized as follows:

- Section 1 introduces FATODE library.
- Section 2 lists all the families of integrators in FATODE and describes their mathematical background briefly.
- Section 3 describes code organization including both logical structure and directory structure of the Code.
- Section 4 covers several topics on basic usage of FATODE, including prerequisites, third-parties libraries needed for sparse linear solvers, makefiles, and functions that user must provide when using FATODE.
- Section 5 describes the user interfaces that FATODE uses for ODE solution, tangent linear sensitivity analysis and adjoint sensitivity analysis respectively.

- Section 6 gives an example which shows how FATODE can be used in real application.
- Section 7 talks about how the linear solvers are implemented and how users can add their own linear solvers.
- Section 8 introduces an automatic differentiation tool TAMC, which can facilitate users to generate complex hessian related functions conveniently.

2

Integrators in FATODE

FATODE implements four families of methods: explicit Runge-Kutta, Rosenbrock, fully implicit Runge-Kutta, and singly diagonally implicit Runge-Kutta, as well as their tangent linear models and discrete adjoint models. Explicit Runge-Kutta methods [4] are well suited for solving nonstiff systems of ODEs. Implicit methods are preferred for solving stiff systems due to their better numerical stability properties.

2.1 Forward solution

Explicit Runge-Kutta methods

A general s-stage Runge-Kutta method reads [4]

$$T_i = t_n + c_j\, h\,, \quad Y_i = y_n + h \sum_{j=1}^{s} a_{i,j}\, f\left(T_j\,,\, Y_j\right)\,, \quad i = 1,\ldots,s\,, \tag{2.1a}$$

$$y_{n+1} = y_n + h \sum_{j=1}^{s} b_j\, f\left(T_j\,,\, Y_j\right)\,. \tag{2.1b}$$

where the coefficients

$$\boldsymbol{A} = \left[a_{i,j}\right]_{1\le i,j\le s}\,, \quad \boldsymbol{b} = \left[b_i\right]_{1\le i\le s}\,, \quad \boldsymbol{c} = \left[c_i\right]_{1\le i\le s} = \boldsymbol{A}\cdot \mathbf{1}_{(s,1)}\,, \tag{2.2}$$

define the method and determine its accuracy and stability properties. Explicit Runge-Kutta methods are characterized by the coefficients $a_{i,j} = 0$ for all i and $j \ge i$. Table 2.1 lists all the methods implemented in explicit Runge-Kutta methods family.

Table 2.1: Time stepping methods implemented in ERK family

Method	Stages	Order	Stability properties
RK2(3) [4]	3	2	conditionally stable
RK3(2) [4]	4	3	conditionally stable
RK4(3) [4]	5	4	conditionally stable
DOPRI-5 [4]	7	5	conditionally stable
Verner [4]	8	6	conditionally stable
DOPRI-853 [4]	12	8	conditionally stable

Singly Diagonally Implicit Runge-Kutta methods

Singly diagonal implicit Runge-Kutta methods are defined by (2.1) with the coefficients $a_{i,i} = \gamma$ and $a_{i,j} = 0$ for all i and $j > i$. See Table 2.2 for details.

Table 2.2: Time stepping methods implemented in SDIRK family

Method	Stages	Order	Stability properties	Stiffly-accurate
Sdirk-2a	2	2	L-stable	Y
Sdirk-2b	2	2	L-stable	Y
Sdirk-3a	3	2	L-stable	Y
Sdirk-4a [3]	5	4	L-stable	Y
Sdirk-4b [3]	5	4	L-stable	Y

Fully Implicit Runge-Kutta methods

Fully implicit methods have three stages and require a coupled solution of all of them.

Detailed information on available Runge-Kutta methods is given in Table 2.3.

Table 2.3: Time stepping methods implemented in FIRK family

Method	Stages	Order	Stability properties	Stiffly-accurate
Radau-1A [3]	3	5	L-stable	N
Radau-2A [3]	3	5	L-stable	Y
Lobatto-3C [3]	3	4	L-stable	Y
Gauss [3]	3	6	weakly L-stable	N

Rosenbrock methods

An s-stage Rosenbrock method [4] is given by the formulas

$$T_i = t_n + \alpha_i h, \quad Y_i = y_n + \sum_{j=1}^{i-1} \alpha_{i,j}\, k_j\,, \tag{2.3a}$$

$$k_i = h\, f\,(T_i, Y_i) + h\boldsymbol{f_y}(t_n, y_n) \cdot \sum_{j=1}^{i} \gamma_{i,j} k_j + \gamma_i h^2 f_t(t_n, y_n), \quad i = 1, \ldots, s\,, \tag{2.3b}$$

$$y_{n+1} = y_n + \sum_{j=1}^{s} b_j k_j\,, \tag{2.3c}$$

where particular methods are defined by their coefficients

$$\boldsymbol{\alpha} = \left[\alpha_{i,j}\right]_{1\le i,j\le s}\,, \quad \boldsymbol{b} = [b_i]_{1\le i\le s}\,, \quad \boldsymbol{\gamma} = \left[\gamma_{i,j}\right]_{1\le i,j\le s}\,, \tag{2.4}$$

and

$$\alpha_i = \sum_{j=1}^{s} \alpha_{i,j}\,,\ \gamma_i = \sum_{j=1}^{s} \gamma_{i,j}\,; \quad \gamma_{i,i} = \gamma\,,\ \alpha_{i,i} = 0\,; \quad \alpha_{i,j} = \gamma_{i,j} = 0\,,\ \forall\, i > j\,.$$

We have $\gamma_{i,i} = \gamma$ for all i for computational efficiency. Here $\boldsymbol{f_y} = \partial f / \partial y$ represents the Jacobian of the ODE function, as discussed in Appendix A. We will denote matrices and tensors by bold symbols, and vectors and scalar by regular symbols. Rosenbrock methods are attractive because of their outstanding stability properties and conservation of the linear invariants of

the system. They typically outperform backward differentiation formulas such as those implemented in SMVGEAR [5] for medium accuracy solutions.

Detailed information regarding these methods is given in Table 2.4. The fifth column shows the stability properties for each method. The method is A-stable if the region of the absolute stability contains the left half plane. The method is L-stable if it is A-stable and the stability function $\phi(z) \to 0$ as $|z| \to \infty$. The last column specifies wether the method is stiffly accurate. A multistage method is called stiffly accurate if the last stage solution the the final solution coincide. Stiff accuracy is an essential property in the solution of differential algebraic equations and stiff ODEs.

Table 2.4: Time stepping methods implemented in ROSENBROCK family

Method	Stages	Order	Stability properties	Stiffly-accurate
Ros-2 [6]	2	2	L-stable	N
Ros-3 [7]	3	3	L-stable	Y
Rodas-3 [7]	4	3	L-stable	Y
Ros-4 [3]	4	4	L-stable	N
Rodas-4 [3]	6	4	L-stable	Y

2.2 Tangent linear model integration

Small changes δy_0 in the initial conditions result in small perturbations $\delta y(t)$ of the solution of ODE system (1.1). Let $\dot{y} = \delta y / \|\delta y_0\|$ be the directions of solution change. These directions propagate forward in time according to the tangent linear ODE:

$$\dot{y}' = \boldsymbol{f_y}(t, y) \cdot \dot{y}, \quad t_0 \le t \le t_f, \quad \dot{y}(t_0) = \dot{y}_0, \quad \dot{y}(t) \in \mathbb{R}^d. \tag{2.5}$$

The sensitivity equations (2.5) are solved forward in time together with original ODE system (1.1). Tangent linear models are derived for direct sensitivity analysis with each of the families of methods in FATODE. Highly efficient implementations are obtained by re-using the LU decompositions from the forward solution on the sensitivity equations [8].

2.3 Adjoint model integration

Adjoint sensitivity analysis provides an efficient alternative to the direct method when gradients of a relatively few derived functionals with respect to many model parameters required. The continuous (differentiate then discretize) and the discrete (discretize-then-differentiate) adjoint approaches lead, in general, to different computational results [9]. The continuous adjoint approach requires interpolation to obtain intermediate state variables at the times required by the backward integration, which brings additional computational effort. The discrete adjoint approach follows exactly the same sequence of time steps as the forward integration, but in reverse order.

FATODE implements discrete adjoints of all the methods. Such discrete adjoints have good theoretical properties, in the sense that they are consistent discretizations of the adjoint ODE [10, 11]. For efficiency our implementation distinguishes between sensitivities with respect to

initial conditions and sensitivities with respect to parameters. We first discuss sensitivities with respect to initial conditions.

The goal is to evaluate the sensitivities of a scalar function of interest

$$\Psi = g(y(t_F)) \tag{2.6}$$

with respect to the initial conditions. The discrete adjoint model equations are obtained directly from the discrete forward model equations

$$y_{n+1} = \Phi^n(y_n), \;\; n = 0, \ldots, N-1 \tag{2.7}$$

where Φ^n represents the one-step numerical integration formula which advances the solution from t_n to t_{n+1}.

The discrete adjoint model equations propagate the adjoint variables λ_n backwards in time

$$\lambda_N = \boldsymbol{g}_{\boldsymbol{y}}^T(y_N); \quad \lambda_n = \boldsymbol{\Phi}_{\boldsymbol{y}}^n(y_n)^T \cdot \lambda_{n+1}, \;\; n = N-1, \ldots, 1. \tag{2.8}$$

The adjoint solution at the initial time represents the sensitivities

$$\left(\partial\Psi/\partial y_0\right)^T = \lambda_0. \tag{2.9}$$

For details on derivation see [1, 10].

A more general case is that the adjoint sensitivity is computed with respect to a time independent vector of parameters $p \in \mathbb{R}^m$ which appears in the right hand side of (1.1). The quantity of interest is a scalar derived function in the general form

$$\Psi = g\left(y(t_F), p\right) + \int_{t_0}^{t_F} r\left(t, y(t), p\right) dt. \tag{2.10}$$

To account for the evolution of the parameters we add the formal equations for the parameter evolution $p' = 0$. To compute the cost function (2.10) we add the quadrature variables $q \in \mathbb{R}$ whose evolution is defined by $q(t_0) = 0$ and $q' = r\left(y, p\right)$. We have that $\Psi = g\left(y(t_F), p\right) + q(t_F)$. The equation (1.1) becomes

$$\begin{bmatrix} y \\ p \\ q \end{bmatrix}' = \begin{bmatrix} f\left(t, y, p\right) \\ 0 \\ r\left(t, y, p\right) \end{bmatrix}, \quad t_0 \le t \le t_f; \quad \begin{bmatrix} y(t_0) \\ p(t_0) \\ q(t_0) \end{bmatrix} = \begin{bmatrix} y_0 \\ p \\ 0 \end{bmatrix}. \tag{2.11}$$

The numerical solution of (2.11) provides the discrete y_n and q_n. The discrete adjoint model equations calculate the adjoint variables λ_n and μ_n backward in time, such that

$$\lambda_N = \boldsymbol{g}_{\boldsymbol{y}}^T(y_N, p), \quad \mu_N = \boldsymbol{g}_{\boldsymbol{p}}^T(y_N, p); \quad \lambda_0 = \left(\partial\Psi/\partial y_0\right)^T, \quad \mu_0 = \left(\partial\Psi/\partial p\right)^T. \tag{2.12}$$

For further information see [12].

3

Code Organization

FATODE implements four types of methods: explicit, fully implicit, and singly diagonally implicit Runge-Kutta methods, and Rosenbrock methods. For each family of methods, a module is given for the main integrator, a module for linear system solver interface, and a set of modules for generic linear system solvers. They form the basic structure shown in Figure (3.1).

3.1 Logical structure

The main integration module provides the basic time stepping framework, and is independent of the linear system solver. The forward integrator calls the user-supplied right-hand side function and Jacobian and accesses the linear system solver, in order to compute the ODE solution. The tangent linear integrator and the adjoint integrator require users to also specify the parameters of interest as additional inputs. The tangent linear integrator, by default, considers the initial conditions of the ODE system as the parameters of interest and computes the sensitivity of the ODE solution with respect to them. The adjoint integrator, by default, computes the sensitivity of an objective function Ψ with respect to the initial conditions. The function and its derivatives are supplied by the user; function derivatives are used to define the adjoint initial conditions. FATODE also implements sensitivities of a general cost function (2.10) with respect to parameters other than the initial conditions (e.g., reaction coefficients in a chemical kinetic ODE system).

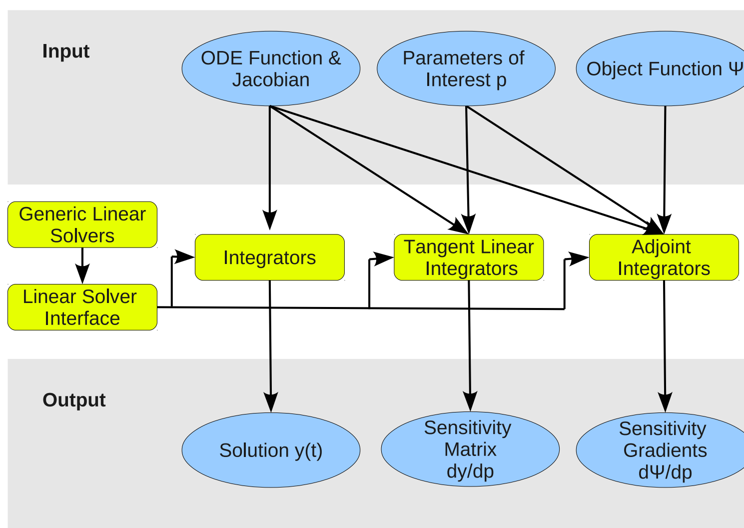

Figure 3.1: Logical structure of FATODE.

3.2 Directory structure

The root directory of FATODE contains the following items:

FATODE/README
general instructions (single file)

FATODE/GPL.txt
GPL license (single file)

FATODE/FWD/
forward model integrators (directory)

FATODE/ADJ/
adjoint model integrators (directory)

FATODE/TLM/
tangent linear model integrators (directory)

FATODE/LSS_LIBS/
linear solvers (directory)

FATODE/EXAMPLES/
example programs (directory)

FATODE/DOC/
documentation (directory)

The full listing of files in FATODE is shown as below.

FATODE directory structure

```
.
|-- FWD
|   |-- ERK
|   |   `-- ERK_f90_Integrator.F90
|   |-- RK
|   |   |-- LS_Solver.F90
|   |   `-- RK_f90_Integrator.F90
|   |-- ROS
|   |   |-- LS_Solver.F90
|   |   `-- ROS_f90_Integrator.F90
|   `-- SDIRK
|       |-- LS_Solver.F90
|       `-- SDIRK_f90_Integrator.F90
|-- ADJ
|   |-- ERK_ADJ
|   |   `-- ERK_ADJ_f90_Integrator.F90
|   |-- RK_ADJ
|   |   |-- LS_Solver.F90
|   |   `-- RK_ADJ_f90_Integrator.F90
|   |-- ROS_ADJ
```

```
|   |   |-- LS_Solver.F90
|   |   `-- ROS_ADJ_f90_Integrator.F90
|   `-- SDIRK_ADJ
|       |-- LS_Solver.F90
|       `-- SDIRK_ADJ_f90_Integrator.F90
|-- TLM
|   |-- ERK_TLM
|   |   `-- ERK_TLM_f90_Integrator.F90
|   |-- RK_TLM
|   |   |-- LS_Solver.F90
|   |   `-- RK_TLM_f90_Integrator.F90
|   |-- ROS_TLM
|   |   |-- LS_Solver.F90
|   |   `-- ROS_TLM_f90_Integrator.F90
|   `-- SDIRK_TLM
|       |-- LS_Solver.F90
|       `-- SDIRK_TLM_f90_Integrator.F90
|-- EXAMPLES
|   |-- roberts
|   |   |-- ROBERTS_RK_ADJ
|   |   |   |-- Makefile
|   |   |   |-- roberts_rk_adj
|   |   |   `-- roberts_rk_adj_dr.F90
|   |   |-- ROBERTS_ROS_ADJ
|   |   |   |-- Makefile
|   |   |   |-- roberts_ros_adj
|   |   |   `-- roberts_ros_adj_dr.F90
|   |   `-- ROBERTS_SDIRK_ADJ
|   |       |-- Makefile
|   |       |-- roberts_sdirk_adj
|   |       `-- roberts_sdirk_adj_dr.F90
|   |-- small_strato
|   |   |-- small_rk
|   |   |   |-- Makefile
|   |   |   |-- small_strato
|   |   |   |-- small_strato_dr.F90
|   |   |   `-- User_Parameters.F90
|   |   |-- small_ros
|   |   |   |-- Makefile
|   |   |   |-- small_strato
|   |   |   |-- small_strato_dr.F90
|   |   |   `-- User_Parameters.F90
|   |   |-- small_ros_adj
|   |   |   |-- Makefile
|   |   |   |-- small_strato
|   |   |   |-- small_strato_dr.F90
|   |   |   `-- User_Parameters.F90
|   |   `-- small_sdirk
```

```
|   |       |-- Makefile
|   |       |-- small_strato
|   |       |-- small_strato_dr.F90
|   |       `-- User_Parameters.F90
|   `-- swe
|       |-- SWE_ERK
|       |   |-- Makefile
|       |   |-- swe2D_erk
|       |   |-- swe2D_erk_dr.F90
|       |   |-- swe2D_upwind.F90
|       |   `-- swe_lsode_sol.txt
|       |-- SWE_ERK_ADJ
|       |   |-- Makefile
|       |   |-- swe2D_erk_adj
|       |   |-- swe2D_erk_adj_dr.F90
|       |   `-- swe2D_upwind.F90
|       |-- SWE_ERK_TLM
|       |   |-- Makefile
|       |   |-- swe2D_erk_tlm
|       |   |-- swe2D_erk_tlm_dr.F90
|       |   `-- swe2D_upwind.F90
|       |-- SWE_RK
|       |   |-- Makefile
|       |   |-- swe2D_rk
|       |   |-- swe2D_rk_dr.F90
|       |   |-- swe2D_upwind.F90
|       |   `-- swe_lsode_sol.txt
|       |-- SWE_RK_ADJ
|       |   |-- Makefile
|       |   |-- swe2D_rk_adj
|       |   |-- swe2D_rk_adj_dr.F90
|       |   `-- swe2D_upwind.F90
|       |-- SWE_RK_TLM
|       |   |-- Makefile
|       |   |-- swe2D_rk_tlm
|       |   |-- swe2D_rk_tlm_dr.F90
|       |   `-- swe2D_upwind.F90
|       |-- SWE_ROS
|       |   |-- Makefile
|       |   |-- swe2D_ros
|       |   |-- swe2D_ros_dr.F90
|       |   |-- swe2D_upwind.F90
|       |   `-- swe_lsode_sol.txt
|       |-- SWE_ROS_ADJ
|       |   |-- Makefile
|       |   |-- swe2D_ros_adj
|       |   |-- swe2D_ros_adj_dr.F90
|       |   `-- swe2D_upwind.F90
```

```
|       |-- SWE_ROS_TLM
|       |   |-- Makefile
|       |   |-- swe2D_ros_tlm
|       |   |-- swe2D_ros_tlm_dr.F90
|       |   `-- swe2D_upwind.F90
|       |-- SWE_SDIRK
|       |   |-- Makefile
|       |   |-- swe2D_sdirk
|       |   |-- swe2D_upwind.F90
|       |   `-- swe_lsode_sol.txt
|       |-- SWE_SDIRK_ADJ
|       |   |-- Makefile
|       |   |-- swe2D_sdirk_adj
|       |   |-- swe2D_sdirk_adj_dr.F90
|       |   `-- swe2D_upwind.F90
|       `-- SWE_SDIRK_TLM
|           |-- Makefile
|           |-- swe2D_sdirk_tlm
|           |-- swe2D_sdirk_tlm_dr.F90
|           `-- swe2D_upwind.F90
|-- LSS_LIBS
|   `-- x86_64_Linux
|       |-- SUPERLU
|       |   |-- c_fortran_dgssv.c
|       |   |-- c_fortran_dgssv.o
|       |   |-- c_fortran_zgssv.c
|       |   |-- c_fortran_zgssv.o
|       |   |-- libsuperlu_4.2.a
|       |   `-- Makefile
|       `-- UMFPACK
|           |-- libamd.a
|           |-- libumfpack.a
|           |-- Makefile
|           |-- umf4_f77wrapper.c
|           |-- umf4_f77wrapper.o
|           |-- umf4_f77zwrapper.c
|           `-- umf4_f77zwrapper.o
|-- LSS_LIBS
|   |-- FATODE_user_guide.pdf
|-- GPL.txt
`-- README
```

4 How to use FATODE

4.1 Prerequisites

The following resources need to be installed on your system before FATODE could be used.

A Fortran compiler
FATODE is written in Fortran 90, taking advantage of modules. Any Fortran 90/95 compiler should work with FATODE. All the code has been tested under the following compilers: Portland group's pgf90, Lahey's lf95, Sun's sunf90, gfortran, g95, Absoft.

A Unix-like environment
In principle you can compile your program incorporating FATODE under any operating system, but the instruction and support provided in this guide is only for a Unix-like environment.

GNU make GNU make is a free and widely used version of make. Our makefiles are tested with GNU make. They are likely to also work with other versions of make, but that has not yet been tested. Use: `make -version` to check whether you have GNU make. If not, you may get it from `http://www.gnu.org`.

BLAS, LAPACK
FATODE relies on BLAS and LAPACK for dense linear algebra operations. You will therefore need a working installation of them.

The BLAS (Basic Linear ALgebra Subprograms) is a collection of routines that provide standard building blocks for performing basic vector and matrix operations. A Fortran reference implementation of BLAS as well as installation information may be found at `http://www.netlib.org/blas/index.html`, but it is recommended to use a machine-optimized library if one is available.

LAPACK (Linear ALgebra Package) is a public domain software and can be downloaded from `http://www.netlib.org/lapack`. If you are using Linux, note that LAPACK comes along with many Linux distributions.

If these two libraries are installed successfully, you may find the library files , e.g. `libblas.a` and `liblapack.a` (names may differ), on your system. Their path needs to be used in makefiles. Alternatively, you configure the system environment variables to point to the location of these libraries; e.g., for `bash` shell use:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/some/directory/xxx.a
```

Another option is to copy the appropriate lib files manually to a location you like (typically your working directory) and link them when compiling.

Remark 1 *Some compilers, such as* `Absoft`*, Sun's* `sunf90`*, Portland Group's* `pgf90` *and LAHEY's* `lf95`*, come with their own optimized version of* BLAS *and* LAPACK*. Examples of linking these libraries can be found in makefiles under EXAMPLES directory.*

Remark 2 *For compilers which do not provide* BLAS *and* LAPACK*, we suggest the users to compile the sources of these libraries with the same compiler as for* FATODE *for compatibility considerations.*

4.2 Optional third party libraries for sparse linear algebra

For efficiently solving sparse linear systems, some widely used third-party linear solvers are incorporated in FATODE. Interfaces to the following libraries are provided. Installing these sparse solvers is optional, but they greatly enhance efficiency when dealing with large sparse linear systems which typically arise in semi-discretized PDE problems.

UMFPACK UMFPACK [13] is a set of routines for solving unsymmetric sparse linear systems using the Unsymmetric MultiFrontal method. The code is available from `http://www.cise.ufl.edu/research/sparse/umfpack`. It supports double precision real and complex data. We have tested version 5.1.

SuperLU SuperLU [14] is a general purpose library for the direct solution of large, sparse, nonsymmetric systems of linear equations on high performance machines, available from `http://crd.lbl.gov/~xiaoye/SuperLU`; provides serial factorization and triangular system solution for single and double precision, real and complex data. We have tested version 4.2.

Remark 3 *Both* UMFPACK *and* SuperLU *are written in C. Fortran wrappers for relevant C calls are provided in* FATODE*. We also include corresponding makefiles to help users generate object files from the libraries and wrapper functions. The object files will be linked if a sparse linear solver is specified to be used with* FATODE*. The* `libumfpack.a`*,* `libamd.a`*, and* `libsuperlu_4.2.a` *supplied with* FATODE *are built on an* `x86_64` *linux system. If you are using a different OS, make sure to replace these binary lib files with your own. Next, type* `make` *inside the corresponding directory to generate new object files.*

4.3 Makefile

To use FATODE it may be necessary to slightly adapt the Makefile to reflect the appropriate settings for your system. All the example programs in FATODE source directory come with Makefiles and these Makefiles can serve as templates for users.

Makefile

```
 ##########################################################################
#
```

```
#  Module:          Makefile
#
#  Purpose:         Top-level Makefile
#
#  Creation date:   Feb 27, 2011
#
#  Modified:
#
#  Send bug reports, comments or suggestions to zhang@vt.edu
#
############################################################################


export ARCH=ar -cr
export RANLIB=ranlib

#Configuration flags for linear solvers. Must choose one from the following three options.
#  -DFULL_ALGEBRA            use BLAS and LAPACK library (full algebra)
#  -DSPARSE_UMF              use UMFpack (sparse format)
#  -DSPARSE_LU               use SuperLU (sparse format)
#
LS_CONFIG  =  -DSPARSE_UMF

#~~~> NAG fortran
#export FC=nagfor
#export FFLAGS = -O3 $(LS_CONFIG)
#BLAS = ~/nagfor_libs/libblas.a
#LAPACK = ~/nagfor_libs/liblapack.a

#~~~> absoft fortran
#export FC=f90
#export FFLAGS = -O3 $(LS_CONFIG)
#BLAS = /Applications/Absoft10.1/lib/libblas.a
#LAPACK = /Applications/Absoft10.1/lib/liblapack.a

#~~~> intel fortran
#export FC=ifort
#export FFLAGS= -cpp -O3 -nogen-interface $(LS_CONFIG) -warn all
#BLAS=    /opt/ifort_libs/libblas.a
#LAPACK =  /opt/ifort_libs/liblapack.a

#~~~> sunf90
#export FC = /opt/oracle/solstudio12.2/bin/sunf90
#export FFLAGS = -fpp -O3 -free $(LS_CONFIG)
#BLAS =  -xlic_lib=sunperf
#LAPACK =

#~~~> gfortran (GNU FORTRAN Compiler)
```

```
export FC = gfortran
export FFLAGS = -cpp -O3 -ffree-line-length-none $(LS_CONFIG)
BLAS=/opt/gfortran_libs/libblas.a
LAPACK=/opt/gfortran_libs/liblapack.a


#~~~> PGF90 (Portland Group Compiler)
#export FC=pgf90
#export FFLAGS=-O3 -fastsse -Mcache_align -tp=penryn-64 -Mflushz -Minform=warn
               -Mpreprocess $(LS_CONFIG)
#BLAS=/opt/pgi/linux86-64/7.2-5/lib/libblas.a
#LAPACK=/opt/pgi/linux86-64/7.2-5/lib/liblapack.a
#other libraries
#XERBLA = -lpgftnrtl -pgf90libs


#~~~> LAHEY
#export FC  = lf95
#export FFLAGS = -O3 -Cpp $(LS_CONFIG)
#BLAS = -lssl2mt
#LAPACK =


FATDIR    = ../../..
LIBDIR    = $(FATDIR)/LSS_LIBS/x86_64_Linux
MODEL     = ADJ
FAMILY    = RK_ADJ

APP       = swe2D_rk_adj
PAR       = swe2D_upwind.o
LSSOLVER  = LS_Solver.o
FAMILY    = RK_ADJ

INTEGRATOR= $(FAMILY)_f90_Integrator.o

default: driver

swe2D_upwind.o: swe2D_upwind.F90
        $(FC) $(FFLAGS) -c $<

LS_Solver.o: $(FATDIR)/$(MODEL)/$(FAMILY)/LS_Solver.F90
        $(FC) $(FFLAGS) -c $<

$(FAMILY)_f90_Integrator.o: $(FATDIR)/$(MODEL)/$(FAMILY)/$(FAMILY)_f90_Integrator.F90
        $(FC) $(FFLAGS) -c $<

swe2D_rk_adj_dr.o: swe2D_rk_adj_dr.F90 $(PAR) $(LSSOLVER) $(INTEGRATOR)
        $(FC) $(FFLAGS) -c $<
```

```
LIB = -lm

LUPACK = $(LIBDIR)/SUPERLU/libsuperlu_4.2.a
LUWRAP = $(LIBDIR)/SUPERLU/c_fortran_dgssv.o $(LIBDIR)/SUPERLU/c_fortran_zgssv.o

UMFPACK = $(LIBDIR)/UMFPACK/libumfpack.a $(LIBDIR)/UMFPACK/libamd.a
UMFWRAP = $(LIBDIR)/UMFPACK/umf4_f77wrapper.o $(LIBDIR)/UMFPACK/umf4_f77zwrapper.o

default: driver

driver: swe2D_rk_adj_dr.o $(PAR) $(LSSOLVER) $(INTEGRATOR)
        $(FC) $(FFLAGS) -o $(APP) $< $(LUWRAP) $(LUPACK) $(UMFWRAP) $(UMFPACK) $(LSSOLVER)
        $(INTEGRATOR) $(PAR) $(LAPACK) $(BLAS) $(XERBLA) $(LIB)

purge:  clean

clean:
        rm -f *~ *.mod *.o

help:
        @$(ECHO) "usage: make ?"
```

4.4 User supplied functions

Two basic functions are required by the integrators in FATODE. One is the right-hand side function defining the ODE. The other is the Jacobian of the ODE function (needed by all integrators except the forward explicit Runge-Kutta). They must be defined with the following argument lists:

right hand side function

```
subroutine fun(n,t,y,f)
integer,intent(in)             :: n
double precision,intent(in)    :: t,y(n)
double precision,intent(inout) :: f(n)
...
end subroutine fun
```

Jacobian function

```
subroutine jac(n,t,y,fjac)
integer,intent(in)             :: n
double precision,intent(in)    :: t,y(n)
double precision,intent(inout) :: fjac(n,n)
...
end subroutine jac
```

n dimension of the ODE.

t current value of the independent variable (time).

y current value of the dependent variable $y(t)$ (state).

f output vector $f(t,y) = dy/dt$.

fjac output Jacobian matrix $\partial f/\partial y$

Adjoint integrators require an additional function `adjinit` to initialize the adjoint variable before the backward run.

adjoint variables initialization function

```
subroutine adjinit(n,np,nadj,t,y,lambda,mu)
integer,intent(in)          :: n,np,nadj,k
double precision,intent(in) :: t,y(n),lambda(n,nadj)
double precision, optional,intent(inout) :: mu(np,nadj)
...
end subroutine
```

n,t,y same as in functions fun and jac.

np number of input parameters of interest (with respect to which sensitivities are computed).

nadj number of output functionals of interest (whose sensitivities are computed).

lambda input and output adjoint variables denoting the sensitivities of given cost functional g w.r.t. initial conditions $\partial g/\partial y$.

mu input and output adjoint variables denoting the sensitivities of given cost functional g w.r.t. to parameters $\partial g/\partial p$.

For sensitivity analysis, users may need to provide some or all of these subroutines:

- `DRDP`
  computes the partial derivative of the function `r`, which is defined in (2.10), w.r.t. a set of parameters.

DRDP
```
subroutine DRDP(nadj,n,nr,t,y,rp)
integer, intent(in)              :: nadj,n,nr
double precision, intent(in)     :: t,y(n)
double precision, intent(inout) :: rp(nr,nadj)
...
end subroutine DRDP
```

- `DRDY`
  computes the partial derivative of the function `r`, which is defined in (2.10), w.r.t. the state vector `y`.

DRDY
```
subroutine DRDY(nadj,n,nr,t,y,ry)
integer, intent(in)              :: nadj,n,nry
double precision, intent(in)     :: t,y(n)
double precision, intent(inout) :: ry(nr,nadj)
...
end subroutine DRDY
```

- `JACP`
  computes the jacobian of the ODE function `f` w.r.t. a set of parameters.

JACP
```
subroutine JACP(n,np,t,y,fpjac)
integer, intent(in)              :: n,np
double precision, intent(in)     :: t,y(n)
double precision, intent(inout) :: fpjac(n,np)
...
end subroutine JACP
```

- QFUN
  computes the function `r`, which is defined in (2.10).

QFUN
```
subroutine QFUN(n,nr,t,y,r)
integer, intent(in)              :: n,nr
double precision, intent(in)     :: t,y(n)
double precision, intent(inout) :: r(nr)
...
end subroutine JACP
```

The variables used in the above subroutines are summerized below:

nr dimension of the quadrature function.

np number of parameters of interest.

r integrand in the given cost functional $r(t, y, p)$.

ry $\partial r / \partial y$, derivative of the integrand r w.r.t y.

rp $\partial r / \partial p$, derivative of the integrand r w.r.t p.

fpjac $\partial f / \partial p$, Jacobian matrix with respect to to p.

The behavior of the integrators will be determined by the presence of these subroutines together with the values of some parameters as shown in Figure (4.1). Note that JACP is needed if the sensitivities with respect to parameters are desired; QFUN, DRDP, DRDY are related to the quadrature term in the given cost functional.

✧ ✧ ✧

Sensitivity analysis with Rosenbrock methods require an additional function calculating the Hessian times a vector (in tangent linear model):

Hessian times vector

```
subroutine hess_vec(n,t,y,u,v,hv)
! hv = (f_yy x v) * u = (d(u*f_y)/dy) * v
integer, intent(in)               :: n
double precision, intent(in)      :: t,y(n),u(n),v(n)
double precision, intent(inout)   :: hv(n)
...
end subroutine hess_vec
```

or the Hessian transpose times a vector (in adjoint model):

Hessian transpose times vector

```
subroutine hesstr_vec(n,t,y,u,v,htv)
! htv = (f_yy x v)^T * u = (d(f_y^T * u)/dy) * v
integer, intent(in)               :: n
double precision, intent(in)      :: t,y(n),u(n),v(n)
double precision, intent(inout)   :: htv(n)
...
end subroutine hesstr_vec
```

The arguments are as follows:

u,v user defined input vectors of dimension n.

hv output vector of dimension n storing the result of Hessian times vector.

htv output vector of dimension n storing the result of Hessian transpose times vector.

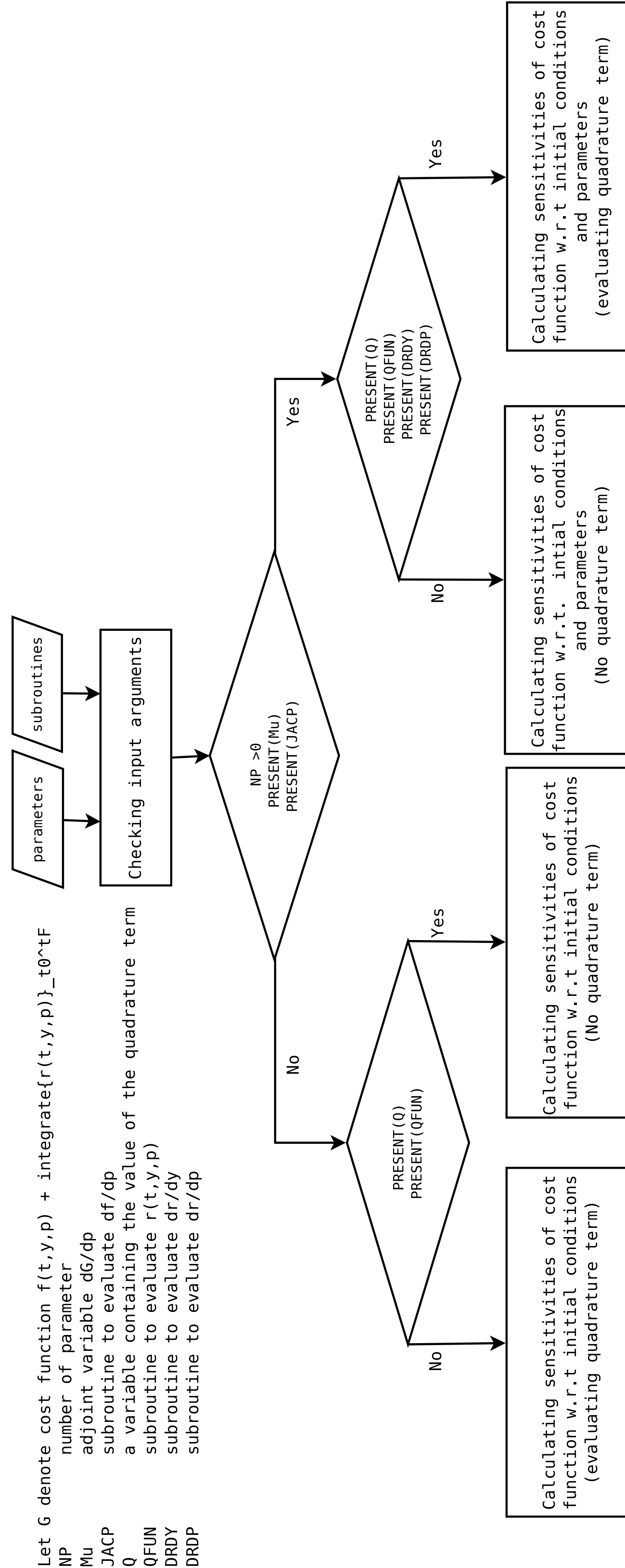

Figure 4.1: The behavior of adjoint integrators is driven by the user supplied functions.

The vectors hv and htv can be regarded as the derivatives of Jacobian-vector product times another vector, and the derivative of Jacobian-transposed-vector product times another vector, respectively:

$$hv := \frac{\partial\big(J(t,y,p)\cdot v\big)}{\partial y}\cdot u = \big(f_{yy}\cdot u\big)\cdot v, \tag{4.1}$$

$$htv := \frac{\partial\big(J(t,y,p)^T\cdot v\big)}{\partial y}\cdot u = \big(u\cdot f_{yy}\big)\cdot v\,. \tag{4.2}$$

✧ ✧ ✧

To perform adjoint sensitivity analysis with respect to parameters using a Rosenbrock method the function HESSTR_VEC_F_PY is needed. In addition, the function HESSTR_VEC_F_PY is also needed when the cost functional is defined using a quadrature term (time integration).

Hessian (F_PY)transpose times vector

```
subroutine HESSTR_VEC_F_PY(ny,np,t,y,u,k,htvg)
! htvg = (f_py x k)^T * u = (d(f_p^T * u)/dy) * k
integer :: ny,np
double precision :: t,y(ny),u(ny),k(ny),htvg(np)
```

Hessian (R_PY) transpose times vector

```
subroutine HESSTR_VEC_R_PY(ny,np,t,y,u,k,htvr)
! htvr = (f_py x k)^T * u = (d(f_p^T * u)/dy) * k
integer :: ny,np
double precision :: t,y(ny),u(ny),k(ny),htvr(np)
```

The output vectors htvg and htvr stand for

$$htvg := \frac{\partial\big(J(t,y,p)^T\cdot k\big)}{\partial y}\cdot u = (u\cdot f_{py})\cdot k, \tag{4.3}$$

$$htvr := \frac{\partial\big(JR(t,y,p)^T\cdot k\big)}{\partial y}\cdot u = (u\cdot r_{py})\cdot k, \tag{4.4}$$

where JR is the Jacobian of function R. The Hessian related functions may be too complex to be obtained analytically. In this case, we suggest users to take advantage of automatic differential techniques. Section 8 gives instructions on how to use TAMC [17] to generate these functions. Users can of course choose other automatic differential tools.

5 User interface

This section describes the interfaces used to call FATODE routines for forward ODE integration, direct sensitivity analysis via tangent linear models, and computing sensitivities of a cost function with respect to initial conditions and specified parameters via adjoint models.

FATODE provides standard interfaces which are very similar to those of classic ODE solvers such as LSODE, VODE, RADAU5. Users who are comfortable with these classic tools should also feel comfortable when use FATODE. For each specific purpose, a uniform interface is provided for all four solver families in FATODE. This interface allows users to control nearly every aspect of the solution process, such as method selection, step size adjustment, error estimation, convergence of simplified Newton iterations, and so on.

5.1 Options

An integer array `ICNTRL` and a real array `RCNTRL` (both of length 20), as optional arguments in all the solver interfaces, are used to control parameters. Table 5.1 and 5.2 describes the full set of options available in each family, as well as default settings. Default values are assigned to these parameters after careful experimenting and using [4]. One can simply take advantage of default settings either by assigning 0 to the array elements or not providing the arrays as input arguments to the interfaces.

Table 5.1: Summary of options tunable through the RCNTRL vector

Index	Description	Where used	Default Setting
1	lower bound on the step size	ERK,FIRK,SDIRK,ROS	0
2	upper bound on the step size	ERK,FIRK,SDIRK,ROS	|Tfinal-Tinitial|
3	starting step size	ERK,FIRK,SDIRK,ROS	max(RCNTRL(1),Roundoff)
4	lower bound on step size change ratio	ERK,FIRK,SDIRK,ROS	0.2
5	upper bound on step size change ratio	ERK,FIRK,SDIRK,ROS	10
6	factor to decrease step after 2 successive rejections	ERK,FIRK,SDIRK,ROS	0.1
7	factor by which the new step is slightly smaller than the predicted value	ERK,FIRK,SDIRK,ROS	0.9
8	factor deciding whether the Jacobian should be recomputed	FIRK,SDIRK	0.001
9	stopping criterion for Newton's method	FIRK,SDIRK	0.03
10	lower bound on the ratio of predicted step size to current step size which decides not to change the step size	ERK, FIRK, SDIRK	1.0
11	upper bound on the ratio of predicted step size to current step size which decides not to change the step size	ERK, FIRK, SDIRK	1.2

Table 5.2: Summary of options tunable through the ICNTRL vector

Index	Description	Where used	Default Setting
1	indicates whether the system is autonomous (autonomous if the value is 0 otherwise non-autonomous)	ROS	no
2	indicates whether tolerances ATOL and RTOL are vectors or scalars (vectors if the value is 0 otherwise scalars)	ERK,FIRK,SDIRK,ROS	vectors
3	method selection within each family	ERK,FIRK,SDIRK,ROS	
4	maximum number of integration steps before unsuccessful return	ERK,FIRK,SDIRK,ROS	200000
5	maximum number of Newton iterations	FIRK,SDIRK	8
6	indicates whether to use extrapolation for starting values of Newton iterations (yes if the value is 0 otherwise no)	FIRK,SDIRK	yes
11	indicates whether to use Gustafsson step size controller (yes if the value is 0 otherwise no)	FIRK	yes

5.2 Forward ODE integration

The call to the forward integrator routine is as follows:

```
CALL INTEGRATE(TIN,TOUT,NVAR,NNZERO,VAR,RTOL,ATOL,FUN,
               ICNTRL,RCNTRL,ISTATUS,RSTATUS,IERR)
```

All arguments except ISTATUS, RSTATUS, IERR are input arguments. ICNTRL, RCNTRL, ISTATUS, RSTATUS, IERR are optional. The arguments have the following meaning:

TIN start time

TOUT end time

NVAR number of ordinary differential equations to be solved

NNZERO number of non-zero elements in the Jacobian matrix

VAR vector of length NVAR containing the initial values of the dependent variables. Upon return, VAR is the numerical solution at the last step.

RTOL relative error tolerance

ATOL absolute error tolerance

FUN name of the user-supplied function that computes the f in the ODE

ICNTRL optional integer-valued array containing input parameters

RCNTRL optional real-valued array containing input parameters

ISTATUS optional integer-valued array containing output statistics

RSTATUS optional real-valued array containing output statistics

IERR job status upon return

The value of variable NNZERO is only needed in the case that a sparse linear solver is used. ICNTRL (an integer array) and RCNTRL (a real array) provide a wide range of options for users to tune the behavior of the integrator. Details can be found in Table 5.1 and 5.2. Note that not all elements in ICNTRL and RCNTRL are used, those not listed in the tables are reserved for future use.

Some options are common to all families while some others are only given to specific families of methods. The third columns in the tables show the name of methods in which the corresponding options can be used. For example, simplified Newton iterations are only used in fully implicit and singly diagonally implicit Runge-Kutta methods. For these two families, users can specify the maximum number of Newton iterations, stopping criterion, bounds on step decrease, increase, and on step rejection. Additional options regarding step size control and error estimation are provided for the fully implicit Runge-Kutta family. In this family two step-size strategies are implemented: the classical approach and the modified predictive controller [15]. There are also two strategies implemented for error estimation: the classical error estimation [4] using the embedded third order method based on an additional explicit stage, and an error estimator based on an additional SDIRK stage which re-uses the real LU decomposition from the main method.

5.3 Tangent linear methods and forward sensitivity calculations

The tangent linear model integrator can be called as follows:

```
CALL INTEGRATE_TLM(NVAR,NTLM,NNZERO,Y,Y_TLM,TIN,TOUT,ATOL_TLM,RTOL_TLM,ATOL,RTOL,
                   FUN,ICNTRL,RCNTRL,ISTATUS,RSTATUS,IERR)
```

There are four additional arguments compared to the forward integrator: `NTLM` specifies the number of sensitivity coefficients to be computed, `Y_TLM` contains sensitivities of `Y` with respect to the specified coefficients, and `ATOL_TLM` and `RTOL_TLM` are used to calculate error estimates for sensitivity coefficients if the switch `ICNTRL(9)` is set to 1. All the options of the forward code also apply to TLM code. Several TLM-specific options are as follows:

ICNTRL(7) solve TLM equations directly if its value is nonzero, or by Newton iterations if value is zero.

ICNTRL(9) switch for TLM Newton iteration error estimation strategies: same as forward integration if value equal to zero, otherwise use RTOL_TLM and ATOL_TLM to calculate an additional error estimate (for fully implicit Runge-Kutta and SDIRK families).

ICNTRL(12) use forward error estimation only if its value is zero, otherwise control the truncation errors for both the forward and the sensitivity solutions. (For both Newton iteration and truncation error FATODE takes the maximum between the forward error and the error of any column of the matrix of sensitivities).

The TLM integrators generate sensitivity results along with the numerical solution of forward ODE system. When sensitivities of $y(t)$ with respect to fixed model parameters are desired the augmenting technique described in [12] can be applied. Note that the argument list for the Rosenbrock method includes the additional function HESS_VEC for calculating the Hessian times vector term.

5.4 Adjoint methods and discrete adjoint sensitivity calculations

The adjoint model integrator is called as follows:

```
CALL INTEGRATE_ADJ(NVAR,NP,NADJ,NNZERO,Y,Lambda,Mu,TIN,TOUT,ATOL_adj,
                   RTOL_adj,ATOL,RTOL,FUN,JAC,ADJINIT,HESSTR_VEC,JACP,
                   DRDY,DRDP,HESSTR_VEC_F_PY,HESSTR_VEC_R_PY,HESSTR_VEC_R,
                   ICNTRL_U,RCNTRL_U,ISTATUS_U,RSTATUS_U,Q,QFUN)
```

The integer arguments NADJ and NP specify the number of cost functionals and the number of system parameters for which adjoint sensitivities are evaluated simultaneously. Lambda (of dimension NVAR × NADJ) and Mu (optional variable of dimension NP × NADJ) contain the sensitivities of the cost functional(s) with respect to the initial conditions and with respect to system parameters, respectively. ADJINIT is a user-provided routine for initializing the adjoint variables; it is called after the forward run ends, and before the backward run starts. The optional variable Q represents the quadrature term in the cost functional and the optional routine QFUN computes the integrand for the quadrature term. JACP, DRDY, DRDP are optional routines to calculate the derivatives $\boldsymbol{f_p}$, $\boldsymbol{r_y}$, and $\boldsymbol{r_p}$ discussed in section 4.4.

Several additional optional arguments (with names beginning with HESS), perform Hessian related operations and are only available in the interface of the Rosenbrock methods. The behavior of the program is controlled by the user through the following optional arguments.

- Mu and JACP are required when the sensitivities with respect to system parameters are desired.
- Q and QFUN are required when the cost functional(s) contain(s) a quadrature term. Upon completion Q stores the value of the quadrature term at the final step.
- DRDY (DRDP) are required when the cost functional(s) contain(s) a quadrature term and the sensitivities with respect to initial conditions (or system parameters, respectively) are desired.

The integrator performs a forward run from t_0 (specified by TIN) to t_F (specified by TOUT) followed by a backward, adjoint run from t_F to t_0. ATOL and RTOL define the tolerances for the forward run while ATOL_ADJ and RTOL_ADJ for the backward run. Tolerance settings are highly problem dependent. The value should never be smaller than the roundoff error of the machine (usually around $1.0E-15$). RTOL of the value $1.0E-3$ is often sufficient for getting a meaningful result. ATOL is a threshold below which the corresponding solution is unimportant. Note that ATOL_ADJ and RTOL_ADJ control the convergence of the iterations (only when Newton iterations are applied), not the time steps since the backward run follows exactly the same sequence of time steps as the forward run in reverse order. To be specific, adjoint RK integrator requires an appropriate setting of ATOL_ADJ and RTOL_ADJ. Adjoint SDIRK integrator need them if the option (ICNTRL(7)) which determines whether to solve the adjoint equations using Newton iterations is not of the value 1; Otherwise they take no effect. Adjoint Rosenbrock integrator does not rely on Newton iterations in the backward run thus ATOL_ADJ and RTOL_ADJ are useless. The definition of these variables is still kept in the code not only for the purpose of future expansion but also for consistence with the argument lists of other integrators.

All options available in forward integration are inherited by adjoint integrators. For fully implicit Runge-Kutta family, there is an additional option provided by ICNTRL(7) which determines how to the linear adjoint system is solved. If the value is 0 or 1, modified Newton re-using LU with a fixed number of iterations will be used; if 2, a direct linear algebra solution is employed; if 3,an adaptive strategy is applies which means that if the simplified Newton iterations for solving the adjoint stage variables do not converge, the code switches automatically to a direct solution method – at the expense of additional LU decompositions.

6

Example

To demonstrate the usage of FATODE, several examples are provided in the source package. Here we just show one of them covering all three different purposes.

6.1 Description of the example problem

We illustrate the capabilities of FATODE with the two-dimensional Saint-Venant system of shallow water equations

$$\begin{aligned}
&\frac{\partial}{\partial t} h + \frac{\partial}{\partial x}(uh) + \frac{\partial}{\partial y}(vh) = 0,\\
&\frac{\partial}{\partial t}(uh) + \frac{\partial}{\partial x}(u^2 + \frac{1}{2} g h^2) + \frac{\partial}{\partial y}(uvh) = 0,\\
&\frac{\partial}{\partial t}(vh) + \frac{\partial}{\partial t}(uvh) + \frac{\partial}{\partial y}(v^2 h + \frac{1}{2} g h^2) = 0,
\end{aligned} \tag{6.1}$$

on the spatial domain $\Omega = [-3, 3]^2$, where $u(t, x, y)$, $v(t, x, y)$ are the components of the velocity field, $h(t, x, y)$ is the fluid layer thickness, and g denotes the standard value of the gravitational acceleration.

The shallow water equations (6.1) are converted to a semi-discrete form using third order upwind finite differences. The spatial domain is covered by a grid of size 40×40, resulting in an ODE system of dimension $40 \times 40 \times 3 = 4800$ which is solved by FATODE.

6.2 Using FATODE for IVP

The velocity field and fluid layer thickness are stored in a two-dimensional matrix and updated by the subroutine compute_f. The standard right hand side function accepts only one-dimensional vector. So two subroutines vec2grid and grid2vec are used for transformation between matrices and vectors.

right hand side function

```
subroutine fun(n,t, y, p )
use swe2dxy_parameters
implicit none
integer, intent(in)              :: n
double precision, intent(in)     :: t,y(n)
double precision, intent(inout) :: p(n)
call vec2grid(y, temp_par)
call compute_f(temp_par,fz_par,feigvalues_par,geigvalues_par)
call grid2vec(fz_par, p)
end subroutine fun
```

The function computing Jacobian matrix compute_jac is generated by the automatic differential tool TAMC based on compute_f.

Jacobian function

```
subroutine jac(n,t,y,fjac)
use swe2dxy_parameters
implicit none
integer, intent(in)              :: n
double precision, intent(in)     :: t,y(n)
double precision, intent(inout) :: fjac(n,n)
call vec2grid(y,u_par)
call compute_f(u_par,f_par,feigvalues_par,geigvalues_par)
call compute_jacf(u_par,feigvalues_par,geigvalues_par)
fjac(:,:)=jf(:,:)
end subroutine jac
```

Since there might be optional variables in the middle of the argument list, we specify the names of the parameters in the procedure call. And this specification style must be followed by users to call FATODE integrators correctly.

calling the forward integrator

```
call integrate( tin=tstart, tout=tend, nvar=ndim, nnzero = nnz,var=var,&
                rtol=rtol, atol=atol,fun=fun,jac=jac,rstatus_u=rstate,&
                rcntrl_u=rcntrl, istatus_u=istate, icntrl_u=icntrl)
```

The meaning of each parameter has been elaborated in Section 5.

6.3 Using FATODE for direct sensitivity analysis

We are interested to compute the sensitivity of the final solution (at time $tend$) with respect to all initial conditions ($ntlm = nvar$). This simple case doesn't require any additional input beyond the basic right hand side and Jacobian functions. But the sensitivity variable y_tlm should be initialized as the derivative of $y(t)$ with respect to y_0 at start time $t = t_0$, yielding an identity matrix. We set the control parameters $icntrl$ and $rcntrl$ to zero so that the default setting will be used. The sensitivity results are stored in $y_t lm$ upon return from the subroutine.

calling the TLM integrator

```
call integrate_tlm(tin=tstart, tout=tend, n=nvar, nnzero=nnz, &
 y_tlm=y_tlm, y=var, rtol=rtol, atol=atol, ntlm=ntlm,&
 atol_tlm=atol_tlm, rtol_tlm=rtol_tlm, fun=fun, jac=jac,&
 rstatus_u=rstate,rcntrl_u=rcntrl, istatus_u=istate,&
  icntrl_u=icntrl)
```

6.4 Using FATODE for discrete adjoint sensitivity analysis

The adjoint is initialized at t_F. If we use nadj=nvar, and each variable at the final time is a functional, then we initialize with identity. The initialization function is given in the following:

Adjoint variable initialization

```
subroutine adjinit(n,np,nadj,t,y,lambda,mu)
integer, intent(in)                          :: n,np,nadj,k
double precision, intent(in)                 :: t,y(n)
double precision, intent(inout)              :: lambda(n,nadj)
double precision, optional, intent(inout) :: mu(np,nadj)
!~~~> if number of parameters is not zero, extra adjoint variable mu
!     should be defined
if(NP>0 .and. .not. present(mu)) stop 'undefined argument mu'
!~~~>  the adjoint values at the final time
 lambda(1:n,1:nadj) = 0.0d0
 do k=1,nadj
   lambda(k,k) = 1.0d0
 end do
end subroutine
```

Then we can call the adjoint integrator in the main procedure:

calling the ADJ integrator

```
 call integrate_adj(tin=tstart, tout=tend, np=np,nvar=ndim,nnzero=nnz, &
                    lambda=y_adj, y=var, rtol=rtol, atol=atol, nadj=nadj,&
                    atol_adj=atol_adj, rtol_adj=rtol_adj, fun=fun, jac=jac,&
                    adjinit=adjinit, rstatus_u=rstate,rcntrl_u=rcntrl, &
                    istatus_u=istate, icntrl_u=icntrl)
```

Remark 4 *If one desires the full sensitivity matrix, both TLM integrators and ADJ integrators can be used with settings $ntlm = nvar$ and $nadj = nvar$ respectively. Both the TLM variable y_tlm and the ADJ variable $lambda$ are initialized with identity matrices and generate the same sensitivity results after integration. If we set $ntlm = 1$ for TLM integrators and $nadj = 1$ for ADJ integrators, then the results differ. y_tlm gives the sensitivity of the final solution with respect to the first element of the initial vector (i.e., the first column of the sensitivity matrix), while $lambda$ corresponds to the sensitivity of the first element of the finial solution with respect to initial conditions (i.e., the first row of the sensitivity matrix).*

Remark 5 *In our example, Rosenbrock methods require Hessian times vector function hess_vec for tangent linear model integration and Hessian transpose times vector function hesstr_vec for adjoint integration.*

7 FATODE linear solvers

The linear system solver module provides interfaces to generic linear solvers, which are called transparently by the integration routines. Specifically, we provide interfaces to the following four generic routines: `LS_Init`, `LS_Decomp`, `LS_Solve`, and `LS_Free`.

- `LS_Init` deals with initialization and memory allocation required by the specific linear solver.
- `LS_Decomp` performs the LU decomposition.
- `LU_Solve` solves the triangular systems by substitution.
- `LS_Free` frees the memory allocated and clears the objects created during the initialization stage.

The main integrator makes calls to these functions without having to consider the implementation details of these routines; many linear algebra packages can be used without having to modify the time stepping code. The user can choose one of the linear solvers provided (LAPACK, UMFPACK, SUPERLU), or can add a new linear solver by providing their own implementation and adding it to this module. The linear system solver module also contains several routines related to computing the Jacobian and its transpose, and taking the product of the Jacobian (or its transpose) with a vector. The implementation of these operations depends critically on the data structures used to represent the Jacobian. For example, the Jacobian matrix could be stored in either dense format or in one of the many available sparse formats, and each representation leads to a different implementation. For computational efficiency users should take advantage of Jacobian matrix data structures that work well for their application. All the required code modifications are done within this single module.

Different families of methods require solving different types of linear systems. Fully implicit Runge-Kutta methods involve real and complex linear systems of dimension $d \times d$, or real linear systems of dimension $ds \times ds$. Singly diagonally implicit Runge-Kutta and Rosenbrock methods deal with only real-valued linear systems of dimension $d \times d$. Jacobian related operations also vary greatly between the three families of implicit methods. To manage this complexity we provide individual linear solver modules for each of the implicit time stepping families in FATODE.

The linear system modules in FATODE currently include three linear solvers: LAPACK [16], UMFPACK [13] and SUPERLU [14]. Each requires its own data format, e.g., a full matrix is used with the LAPACK version, while a compressed column sparse matrix is needed with UMFPACK or SUPERLU versions. The existing linear solvers can be used as templates for users who wish to add their own. A new solver requires the user to provide the four basic solution routines, as well as the Jacobian related routines if necessary. Though the framework is originally designed for direct solvers, it may apply to iterative solvers if one leaves the routine `LS_Decomp` blank and implements iterative solvers only in `LU_Solve`.

8 Automatic Differentiation

In the following we illustrate the use of theautomatic differentiation tool TAMC for code generation. Detailed information on TAMC is given in the user's manual [17].

Consider a given subroutine with the following parameter list:

```
subroutine fun(n, t, y, f)
! dimension of state vector y
integer :: n
double precision :: t
! y is the numerical solution at time t and p is the
! right hand side function at time t
double precision :: y(n), f(n)
```

The forward mode of TAMC can generate derivative code to compute the sensitivity of the dependent variable f with respect to the independent variable y by:

```
./stamc -reply <your_email> -toplevel fun -input y
   -output f -forward -pure -Jacobian <m> <source file name>
```

The generated code has the following parameter list:

```
subroutine g_fun(n, t, y, g_y, g_f)
! dimension of state vector y
integer :: n
double precision :: t
! y is the numerical solution at time t and p is the
! right hand side function at time t
double precision :: y(n), p(n)
double precision :: g_y(n,m), g_f(n,m)
```

If you specify the number after option '-Jacobian' with the number of dependent variables and initialize the input variable g_f with identity matrix, full Jacobian will be obtained and stored in g_y. Otherwise, the number after option '-Jacobian' defines the number of columns of a seed matrix C, denoted by m.

$$\frac{\partial f}{\partial y} \cdot C_{[n\times m]} \tag{8.1}$$

The backward mode of TAMC can generate derivative code to compute the sensitivity of the dependent variable with respect to the many independent variables by:

```
./stamc -reply <your_email> -toplevel fun -input y
      -output f -reverse -pure -Jacobian <m> <source file name>
```

The generated code has the following parameter list:

```
      subroutine adfun(n, t, y, ady, adf)
      ! dimension of state vector y
      integer :: n
      double precision :: t
      ! y is the numerical solution at time t and p is the
      ! right hand side function at time t
      double precision :: y(n), p(n)
      double precision :: ady(m,n), adf(m,n)
```

If you specify the number after option '-Jacobian' with the number of dependent variables and initialize the input variable adf with identity matrix, full Jacobian will be obtained and stored in ady. Otherwise, the number after option '-Jacobian' defines the number of rows of a seed matrix C, denoted by m.

$$C_{[m\times n]}\cdot\frac{\partial f}{\partial y} \tag{8.2}$$

Note that if you use the option '-Jacobian 1', product of Jacobian times a vector $\frac{\partial f}{\partial y}\cdot v$ is provided by the forward mode while product of Jacobian transpose times a vector $\left(\frac{\partial f}{\partial y}\right)^T\cdot v$ is provided by the backward mode.

To obtain the code calculating the Hessian times vector term $(u\cdot H)\cdot v$, we run TAMC in forward over reverse mode

```
./stamc -reply <your_email> -toplevel fun -input y
      -output f -reverse -pure -Jacobian 1 <source file name>
./stamc -reply <your_email> -toplevel fun -input y
      -output g_y -forward -pure -Jacobian 1 <source file name>
```

The generated code has the following parameter list:

```
      subroutine g_adfun(n, t, y, adf, g_y, g_ady)
      ! dimension of state vector y
      integer :: n
      double precision :: t
      ! y is the numerical solution at time t and p is the
      ! right hand side function at time t
      double precision :: y(n), p(n)
      double precision :: g_y(1,n), adf(1,n),g_ady(1,n)
```

where the variable adf corresponds to the vector v and g_y correspond to the vector u.

Similarly the Hessian transpose times vector term $(u \cdot H^T) \cdot v$ can be obtained by two consecutive forward runs:

```
./stamc -reply <your_email> -toplevel fun -input y
      -output f -forward -pure -Jacobian 1 <source file name>
./stamc -reply <your_email> -toplevel fun -input y
      -output g_y -forward -pure -Jacobian 1 <source file name>
```

The generated code has the following parameter list:

```
      subroutine g_g_fun(n, t, y, g_y, g_f, g_g_y)
      ! dimension of state vector y
      integer :: n
      double precision :: t
      ! y is the numerical solution at time t and p is the
      ! right hand side function at time t
      double precision :: y(n), p(n)
      double precision :: g_y(1,n), g_f(1,n),g_g_y(1,n)
```

where the variable g_f corresponds to the vector v and g_y correspond to the vector u.

Bibliography

[1] A. Sandu, D. Daescu, and G. Carmichael, "Direct and Adjoint Sensitivity Analysis of Chemical Kinetic Systems with KPP: I – Theory and Software Tools," Atmospheric Environment, vol. 37, pp. 5083–5096, 2003.

[2] A. Sandu and D. Daescu and G.R. Carmichael, "Direct and adjoint sensitivity analysis of chemical kinetic systems with KPP: II – Numerical validation and applications," Atmospheric Environment, vol. 37, pp. 5097–5114, 2003.

[3] E. Hairer and G. Wanner, Solving Ordinary Differential Equations II. Stiff and Differential-Algebraic Problems. Berlin: Springer Series in Computational Mathematics, 1991.

[4] E. Hairer, S. Norsett, and G. Wanner, Solving Ordinary Differential Equations I. Nonstiff Problems. Berlin: Springer-Verlag, 1993.

[5] P. Eller, K. Singh, A. Sandu, K. Bowman, D. K. Henze, and M. Lee, "Implementation and evaluation of an array of chemical solvers in the Global Chemical Transport Model GEOS-Chem," Geoscientific Model Development, vol. 2, p. 89â96, 2009.

[6] J. Verwer, E. Spee, J. G. Blom, and W. Hunsdorfer, "A second order Rosenbrock method applied to photochemical dispersion problems," SIAM Journal on Scientific Computing, vol. 20, pp. 1456–1480, 1999.

[7] A. Sandu, J. Verwer, J. Blom, E. Spee, G. Carmichael, and F. Potra, "Benchmarking stiff ODE solvers for atmospheric chemistry problems II: Rosenbrock methods," Atmospheric Environment, vol. 31, pp. 3459–3472, 1997.

[8] A. M. Dunker, "The decoupled direct method for calculating sensitivity coefficients in chemical kinetics," vol. 81, no. 5, pp. 2385–2393, 1984.

[9] Z. Sirkes and E. Tziperman, "Finite difference of adjoint or adjoint of finite difference," Monthly Weather Review, vol. 125, no. 12, pp. 3373–3378, 1997.

[10] A. Sandu, "On the properties of Runge Kutta discrete adjoints," in ICCS 2006, IV,LNCS 3994, (Berlin Heidelberg), pp. 550–557, Springer-Verlag, 2006.

[11] A. Sandu, "Solution of inverse ODE problems using discrete adjoints," Large Scale Inverse Problems and Quantification of Uncertainty, pp. 345–364, 2010.

[12] H. Zhang and A. Sandu, "FATODE: A Library for Forward, Adjoint, and Tangent Linear Integration of ODEs," Tech. Rep. TR-11-25, Computer Science, Virginia Tech, 2011.

[13] T. A. Davis, "Algorithm 832: UMFPACK V4.3—an unsymmetric-pattern multifrontal method," ACM Transactions on Mathematical Software, vol. 30, pp. 196–199, June 2004.

[14] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu, "A supernodal approach to sparse partial pivoting," SIAM Journal on Matrix Analysis and Applications, vol. 20, no. 3, pp. 720–755, 1999.

[15] K. Gustafsson, "Control-theoretic techniques for stepsize selection in implicit Runge-Kutta methods," ACM Transactions on Mathematical Software, vol. 20, no. 4, pp. 496–517, 1994.

[16] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen, LAPACK Users' guide (third ed.). Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1999.

[17] R. Giering, "Tangent linear and adjoint model compiler, users manual 1.4," 1999.