# PETSc/TS: A Modern Scalable DAE/ODE Solver Library

Shrirang Abhyankar, Jed Brown, Emil Constantinescu, Debojyoti Ghosh,
Barry F. Smith and Hong Zhang

High-quality ordinary differential equation (ODE) solver library packages have a long history, going back to the 1970s. Over the past several years we have implemented a new general-purpose, extensive, extensible ODE and differential algebraic equation (DAE) solver library on top of the PETSc linear and nonlinear solver package. This package includes support for both forward and adjoint sensitivities that can be easily utilized by the TAO optimization package, which is also part of PETSc. The ODE/DAE integrator library strives to be highly scalable but also deliver high efficiency for modest-sized problems. The library includes explicit solvers, implicit solvers, and a collection of implicit-explicit solvers all with a common user interface and runtime selection of solver types, adaptive error control, and monitoring of solution progress. The library also offers enormous flexibility in selection of nonlinear and linear solvers, including the entire suite of PETSc iterative solvers, as well as several parallel direct solvers.

## 1. INTRODUCTION

Sophisticated numerical algorithms for the integration of ordinary differential equations (ODEs) and differential algebraic equations (DAEs) have existed for well over one hundred years, while general-purpose software libraries for their solution have existed for at least forty years. With changes in the applications simulated and in the computer hardware, such libraries are constantly evolving. Perhaps the best-known such libraries for both ODEs and DAEs are those originating at Lawrence Livermore National Laboratory, including VODE, CVODE, DASSL, and SUNDIALS [Petzold 1992; Hindmarsh et al. 2005]. Other libraries with ODE/DAE support include Trilinos [Heroux et al. 2003], as well as commercial codes such as MATLAB [2014] and NAG [2018]. The theory of numerical algorithms for ODEs and DAEs is covered in several monographs, including [Hairer et al. 2008; Hairer and Wanner 2002; Ascher and Petzold 1998; Brenan et al. 1996; Butcher 2008].

The Portable, Extensible Toolkit for Scientific computation (Version 2.0), PETSc, was developed to provide scalable high-quality mathematical libraries for distributed memory parallel computing. Over the past several years we have implemented a general-

purpose, extensive, extensible ODE and DAE integrator package with local and global error control, support for computing sensitivities, and handling of events (discontinuities in solutions or parameters). In this paper we describe the design, properties, and usage of the ODE/DAE integrators in PETSc. The document is organized as follows. In Section 2 we introduce the organization of PETSc, followed in Section 3 by the PETSc time-stepping application programming interface (API) and in Section 4 by the time integration schemes in PETSc. In Section 5 we discuss adaptive time stepping and error control and in Section 6 the computation of sensitivities. In Section 7 we explain how events are handled and the tools for monitoring and visualizing solutions and the solution process. In Section 8 we discuss how PETSc handles discontinuities and events. We conclude in Section 9 with a brief summary of two high-level interfaces for accessing the ODE/DAE integrators: one for networks (for example the power grid) and one for chemical reactions.

## 2. BACKGROUND: PETSC LINEAR AND NONLINEAR SOLVERS

PETSc is a scalable, MPI-based, object-oriented numerical software library written in C and fully usable from C, C++, Fortran, and Python. See [Balay et al. 1997] for details on its design and the users manual [Balay et al. 2017] for how to use PETSC effectively. PETSc has several fundamental classes from which applications are composed.

— `IS` – index sets used to index into vectors and matrices, an abstraction of a list of integers
— `Vec` – vectors, abstract elements of $R^n$, used to contain the ODE solutions, function evaluations, and so forth
— `Mat` – matrices, representations of linear operations including matrix-free formulations and sparse and dense storage formats
— `PC` – preconditioners, single-step iterative solvers including domain decomposition and algebraic and geometric multigrid as well as direct solvers such as LU
— `KSP` – Krylov subspace solvers, multistep iterative solvers
— `SNES` – nonlinear solvers, including Newton's method, quasi-Newton methods, and nonlinear Krylov methods

In addition PETSc has a variety of helper classes that are useful for implicit ODE solvers, these include `MatColoring` and `MatFDColoring`, which are used to efficiently compute Jacobians via finite difference; see Section 3. Moreover, PETSc has an abstract class `DM` that serves as an adapter between meshes, discretizations, and other problem descriptors and the algebraic and time-stepper objects that are used to solve the discrete problem.

PETSc takes a minimalist approach to public APIs, attempting to keep them as small as possible and with as little overlap in functionalities as possible. In addition PETSc provides both a programmatic interface to almost all functionalities within PETSc as well as a simple string-based system, called the options database, that allows runtime control of almost all functionality in PETSc.

Because PETSc is written in C, which does not have native syntax to create classes and class hierarchies of object-oriented languages, the classes are managed "manually" by the use of C structs and function pointers. One feature of this approach, which can also be obtained in object-oriented languages through the use of delegators, is that any object can be changed at runtime, even multiple times during the run to different derived classes of the same base class. For example, a linear solver object that uses the GMRES method can later be changed to use Bi-CG-stab by simply resetting the solver type, without the need for factory classes. Many PETSc functions have optional arguments; since C does not support function overloading, one passes `PETSC_DEFAULT` (for optional scalar arguments) and `NULL` (for optional array arguments). In order to

allow changing basic type sizes at compile time PETSc has its own types: `PetscReal`, which can represent half-precision, single, double, or quad precision floating point; `PetscScalar` which represents complex values when PETSc is built with complex numbers; and `PetscInt` which represents either 32 or 64 bit integers. Since PETSc is written in C, we cannot utilize templates for this purpose as would be done in C++.

The usage of PETSc objects generally proceeds in the following order.

— `XXXCreate(MPI_Comm comm,XXX *xxx)` creates an object of type XXX, for example, KSPCreate(MPI_comm,KSP*) creates a Krylov solver object.
— `XXXSetYYY(XXX xxx,...)` sets options to the object, via the functional interface.
— `XXXSetType(XXX xxx,const char* typename)` sets the specific subclass of the object, for example, "gmres" is a subclass of KSP.
— `XXXSetFromOptions(XXX xxx)` allows setting the type and options of the object from the options database.
— `XXXSetYYY(XXX xxx,...)` sets additional options,
— `XXXSetUp(XXX xxx)` fully instantiates the object so that it is ready to be used
— `XXXDestroy(XXX *xxx)` frees all the space being used by the solver. PETSc uses reference counting to ensure that objects referenced by multiple other objects are not prematurely destroyed.

We present full examples of this approach below for the ODE/DAE solvers.

## 3. PETSC TIME-STEPPING APPLICATION PROGRAMMING INTERFACE

The PETSc interface for solving time-dependent problems is organized around the following form of a DAE:

$$F(t, u, \dot{u}) = G(t, u), \quad u(t_0) = u_0.$$

If the matrix $F_{\dot{u}}(t) = \partial F/\partial \dot{u}$ is nonsingular, then the equation is an ODE and can be transformed to the standard explicit form ($\dot{u} = Q(t, u)$). This transformation may not lead to efficient algorithms, so often the transformation to explicit form should be avoided. For ODEs with nontrivial mass matrices such as those that arise in the finite element method, the implicit/DAE interface can significantly reduce the overhead to prepare the system for algebraic solvers by having the user assemble the correctly shifted matrix. This interface is also useful for ODE systems, not just DAE systems.

To solve an ODE or DAE one uses the timestep context `TS` created with `TSCreate(comm,&ts)` and then allows the user to set options from the options database with `TSSetFromOptions(ts)`. To define the ODE/DAE, the user needs to provide one or more functions (callbacks). The TS API for providing these functions consists of the following.

— Function $F(t, u, \dot{u})$ is provided, by the user, with

```
1   TSSetIFunction(TS ts,Vec r, (*f)(TS,PetscReal,Vec,Vec,Vec,void*),void *fP);
```

The vector `r` is an optional location to store the residual. The arguments to the function `f()` are the timestep context, current time, input state $u$, input time derivative $\dot{u}$, and the (optional) user-provided context `*fP` that contains data needed by the application-provided call-back routines. . When only $G(t, u)$ is provided, TS automatically assumes that $F(t, u, \dot{u}) = \dot{u}$.
— Function $G(t, u)$, if it is nonzero, is provided, by the user, with

```
TSSetRHSFunction(TS ts,Vec r,(*g)(TS,PetscReal,Vec,Vec,void*),void *gP);
```

Again the vector `r` is an optional location to store the residual. The arguments to the function `g()` are the timestep context, current time, input state $u$, and the (optional) user-provided context `*gP`.

— Jacobian $F_u + (shift)F_{\dot{u}}$

If using a fully implicit or semi-implicit (IMEX) method, one also must provide an appropriate (approximate) Jacobian matrix of $F()$ at the current solution $u^n$ using

```
TSSetIJacobian(TS ts,Mat A,Mat B,
    (*j)(TS,PetscReal,Vec,Vec,PetscReal,Mat,Mat,void*),void *jP);
```

The arguments of `j()` are the timestep context; current time; input state $u$; input derivative $\dot{u}$; $shift$ (described below); matrix $A$ (which defines the Jacobian); matrix $B$, which is optionally different from $A$ (from which the preconditioner is constructed); and the (optional) user-provided context `jP`.

This form for the Jacobian arises because for all implicit and semi-implicit time integrators in PETSc the value of $\dot{u}^n$ is approximated in the ODE/DAE solver algorithms by $(shift)u^n + q(u^{n-1}, ...)$, where the method-specific function $q(u^{n-1}, ...)$ depends only on previous iterations. Hence

$$\frac{dF}{du^n} = \frac{\partial F}{\partial \dot{u}^n}\frac{\partial \dot{u}^n}{\partial u^n} + \frac{\partial F}{\partial u^n}$$
$$= (shift)F_{\dot{u}^n}(t^n, u^n, \dot{u}^n) + F_{u^n}(t^t, u^n, \dot{u}^n).$$

For example, the backward Euler method $\dot{u}^n = (u^n - u^{n-1})/\Delta t$. With $F(u^n) = M\dot{u}^n - f(t, u^n)$ one obtains the expected Jacobian

$$\frac{dF}{du^n} = \frac{\partial(M\dot{u}^n)}{\partial \dot{u}^n}\frac{\partial \dot{u}^n}{\partial u^n} - \frac{\partial f}{\partial u^n}$$
$$= \frac{1}{\Delta t}M - f_{u^n}(t, u^n).$$

In this case the value of *shift* is $1/\Delta t$.

— Jacobian $G_u$

If using a fully implicit method and the function $G()$ is provided, one must also provide an appropriate (approximate) Jacobian matrix of $G()$ using

```
TSSetRHSJacobian(TS ts,Mat A,Mat B,
  (*gj)(TS,PetscReal,Vec,Mat,Mat,void*),void *gjP);
```

The arguments for the function `gj()` are the timestep context, current time, input state $u$, matrix $A$, optional matrix $B$ from which the preconditioning is constructed, and the (optional) user-provided context `gjP`.

Providing appropriate $F()$ and $G()$ and their derivatives for the problem allows for easy runtime switching between explicit, semi-implicit, and fully implicit method.

Providing correctly coded Jacobians is often a major stumbling block for users of ODE/DAE integration packages. PETSc provides three useful tools to help users in this process:

— application of Jacobians via *matrix-free* differencing approaches,
— explicit computation of Jacobians via matrix coloring and differencing, and
— automatic testing of user-provided Jacobian computations.

Finite-difference-based matrix-free application of Jacobians is handled with a special PETSc matrix class that never forms the matrix entries explicitly but merely provides matrix-vector products. For most situations the user simply provides the option `-snes_mf`, which uses the PETSc provided matrix-free matrix class, and either no preconditioner or a user-provided preconditioner or `-snes_mf_operator`, where a standard preconditioner is constructed from some user-provided approximation to the Jacobian. Users who desire more control over the process can utilize

```
1  MatCreateMFFD(MPI_Comm,PetscInt m, PetscInt n,PetscInt M,PetscInt N,Mat *J)
2  MatMFFDSetFunction(Mat J,(*f)(void*,Vec,Vec),void *ctx)
```

The arguments of `MatCreateMFFD()` are the local and global dimensions of the operator, while the arguments of `MatMFFDSetFunction()` include the nonlinear function and optional user-provided context data. A simpler alternative uses the nonlinear function already provided to the nonlinear solver with

```
1  TSGetSNES(TS ts,SNES *snes)
2  MatCreateSNESMF(SNES snes,Mat *J)
```

An explicit matrix representation of the Jacobian via matrix coloring may be constructed by using the option -snes_fd_color. The coloring can be provided in complementary ways, either by providing the nonzero structure of the Jacobian (but not its numerical values) and applying a variety of matrix coloring routines to compute the coloring (this is done by creating a coloring object with `MatColoringCreate()` and from this performing the coloring) or by providing the coloring based on the mesh structure and specific numerical discretization approach used (this is done by calling `DMCreateColoring()`. Once the coloring is provided, the actual computation of the Jacobian entries involves the use of `MatFDColoringCreate()` and `MatFDColoringSetFunction()`, which plays a role similar to `MatMFFDSetFunction()`. Both the matrix-free differencing and the explicit computation of the Jacobian entries support various options for selecting the differencing parameters. The explicit computation of Jacobian entries via differencing can be used to find the locations of Jacobian entries incorrectly provided by the user. In the simplest case this is handled via the option -snes_type test -snes_test_display. Other options include -snes_compare_coloring and -snes_compare_coloring_display.

We now present a complete simple example code demonstrating the use of TS to solve a small set of ODEs: $\dot{u}_0 = -\kappa u_0 u_1$, $\dot{u}_1 = -\kappa u_0 u_1$ $\dot{u}_2 = \kappa u_0 u_1$, $u^0 = [1.0, 0.7, 0.0]^T$.

```
1  /*   Defines the ODE passed to the ODE solver   */
2  IFunction(TS ts,PetscReal t,Vec U,Vec Udot,Vec F,AppCtx *ctx){
3    PetscScalar      *f;
4    const PetscScalar *u,*udot;
5
6    /*  Allow access to the vector entries  */
7    VecGetArrayRead(U,&u); VecGetArrayRead(Udot,&udot); VecGetArray(F,&f);
8    f[0] = udot[0] + ctx->k*u[0]*u[1];
9    f[1] = udot[1] + ctx->k*u[0]*u[1];
10   f[2] = udot[2] - ctx->k*u[0]*u[1];
11   VecRestoreArrayRead(U,&u); VecRestoreArrayRead(Udot,&udot);
12   VecRestoreArray(F,&f);
13 }
14 /*     Defines the Jacobian of the ODE passed to the ODE solver.    */
15 IJacobian(TS ts,PetscReal t,Vec U,Vec Udot,PetscReal a,Mat A,Mat B,AppCtx *
      ctx){
16   PetscInt         rowcol[] = {0,1,2};
17   PetscScalar      J[3][3];
18   const PetscScalar *u,*udot;
19
20   VecGetArrayRead(U,&u); VecGetArrayRead(Udot,&udot);
21   J[0][0] = a + ctx->k*u[1];   J[0][1] = ctx->k*u[0];      J[0][2] = 0.0;
22   J[1][0] = ctx->k*u[1];       J[1][1] = a + ctx->k*u[0];  J[1][2] = 0.0;
23   J[2][0] = -ctx->k*u[1];      J[2][1] = -ctx->k*u[0];     J[2][2] = a;
24   MatSetValues(B,3,rowcol,3,rowcol,&J[0][0],INSERT_VALUES);
25   VecRestoreArrayRead(U,&u); VecRestoreArrayRead(Udot,&udot);
26   MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);
27   MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);
28 }
29 /*   Defines the initial conditions (and the analytic solution)  */
```

```
30  Solution(TS ts,PetscReal t,Vec U,AppCtx *ctx){
31    const PetscScalar *uinit;
32    PetscScalar      *u,d0,q;
33
34    VecGetArrayRead(ctx->initialsolution,&uinit); VecGetArray(U,&u);
35    d0   = uinit[0] - uinit[1];
36    if (d0 == 0.0) q = ctx->k*t;
37    else q = (1.0 - PetscExpScalar(-ctx->k*t*d0))/d0;
38    u[0] = uinit[0]/(1.0 + uinit[1]*q);
39    u[1] = u[0]  - d0;
40    u[2] = uinit[1] + uinit[2] - u[1];
41    VecRestoreArrayRead(ctx->initialsolution,&uinit); VecRestoreArray(U,&u);
42  }
43  /* Creates the TS object, sets functions, options, then solves the ODE */
44  int main(int argc,char **argv){
45    TS              ts;               /* ODE integrator */
46    Vec             U;                /* solution will be stored here */
47    Mat             A;                /* Jacobian matrix */
48    PetscInt        n = 3;
49    AppCtx          ctx;
50
51    PetscInitialize(&argc,&argv,(char*)0,help);
52    /*   Create necessary matrix and vectors  */
53    MatCreate(PETSC_COMM_WORLD,&A);
54    MatSetSizes(A,n,n,PETSC_DETERMINE,PETSC_DETERMINE);
55    MatSetFromOptions(A); MatSetUp(A);
56    MatCreateVecs(A,&U,&ctx.initialsolution);
57    /*   Set runtime option  */
58    ctx.k = .9;
59    PetscOptionsGetScalar(NULL,NULL,"-k",&ctx.k,NULL);
60    /*  Create timestepping solver context  */
61    TSCreate(PETSC_COMM_WORLD,&ts);
62    TSSetProblemType(ts,TS_NONLINEAR);
63    TSSetType(ts,TSROSW);
64    TSSetIFunction(ts,NULL,(TSIFunction) IFunction,&ctx);
65    TSSetIJacobian(ts,A,A,(TSIJacobian)IJacobian,&ctx);
66    /*  Set initial conditions */
67    Solution(ts,0,U,&ctx);
68    /*   Set solver options  */
69    TSSetTimeStep(ts,.001);
70    TSSetMaxSteps(ts,1000);
71    TSSetMaxTime(ts,20.0);
72    TSSetExactFinalTime(ts,TS_EXACTFINALTIME_STEPOVER);
73    TSSetFromOptions(ts);
74    TSMonitorLGSetVariableNames(ts,names);
75
76    TSSolve(ts,U);
77
78    VecDestroy(&ctx.initialsolution); MatDestroy(&A); VecDestroy(&U);
79    TSDestroy(&ts);
80    PetscFinalize();
81  }
```

We now present a complete simple example code demonstrating the use of TS to solve a small set of stiff ODEs written in Python and using the petsc4py binding:

$$\dot{u}_0 = -77.27(u_1 + u_0(1 - 8.375 \times 10^{-6}u_0 - u_1)),$$
$$\dot{u}_1 = \frac{1}{77.27}(u_2 - (1 + u_0)u_1),$$
$$\dot{u}_2 = 0.161(u_0 - u_2),$$
$$u(t = 0) = [1.0, 2.0, 3.0]^T.$$

Oregonator: stiff 3-variable oscillatory ODE system from chemical reactions, problem OREGO in [Hairer and Wanner 2002].

```python
import sys, petsc4py
from matplotlib import pylab
from matplotlib import rc
import numpy as np
petsc4py.init(sys.argv)

from petsc4py import PETSc

class Orego(object):
    n = 3
    comm = PETSc.COMM_SELF
    def evalSolution(self, t, x):
        x.setArray([1, 2, 3])
    def evalFunction(self, ts, t, x, xdot, f):
        f.setArray([xdot[0] - 77.27*(x[1] + x[0]*(1 - 8.375e-6*x[0] - x[1])),
                    xdot[1] - 1/77.27*(x[2] - (1 + x[0])*x[1]),
                    xdot[2] - 0.161*(x[0] - x[2])])
    def evalJacobian(self, ts, t, x, xdot, a, A, B):
        B[:,:] = [[a - 77.27*((1 - 8.375e-6*x[0] - x[1]) -8.375e-6*x[0]),
                   -77.27*(1 - x[0]), 0],
                  [1/77.27*x[1], a + 1/77.27*(1 + x[0]),    -1/77.27],
                  [-0.161, 0, a + 0.161]]
        B.assemble()
        if A != B: A.assemble()
        return True # same nonzero pattern

OptDB = PETSc.Options()
ode = Orego()

J = PETSc.Mat().createDense([ode.n, ode.n], comm=ode.comm)
J.setUp()
x = PETSc.Vec().createSeq(ode.n, comm=ode.comm); f = x.duplicate()

ts = PETSc.TS().create(comm=ode.comm)
ts.setType(ts.Type.ROSW) # use Rosenbrock-W method

ts.setIFunction(ode.evalFunction, f)
ts.setIJacobian(ode.evalJacobian, J)

history = []
def monitor(ts, i, t, x):
    xx = x[:].tolist()
    history.append((i, t, xx))
ts.setMonitor(monitor)

ts.setTime(0.0)
ts.setTimeStep(0.1)
ts.setMaxTime(360)
ts.setMaxSteps(2000)
ts.setExactFinalTime(PETSc.TS.ExactFinalTime.INTERPOLATE)
ts.setMaxSNESFailures(-1) # allow unlimited failures (step will be retried)

# Set a different tolerance on each variable.
vatol = x.duplicate(array=[1e-2, 1e-1, 1e-4])
ts.setTolerances(atol=vatol,rtol=1e-3) # adaptive controller attempts to
    match this tolerance

snes = ts.getSNES()               # Nonlinear solver
snes.setTolerances(max_it=10)     # Stop nonlinear solve after 10 iterations
    (TS will retry with shorter step)
```

```
59   ksp = snes.getKSP()              # Linear solver
     ksp.setType(ksp.Type.PREONLY)    # No Krylov method
     pc = ksp.getPC()                 # Preconditioner
     pc.setType(pc.Type.LU)           # Use a direct solve

64   ts.setFromOptions()              # Apply run-time options, e.g.
         -ts_adapt_monitor -ts_type arkimex -snes_converged_reason
     ode.evalSolution(0.0, x)
     ts.solve(x)

     if OptDB.getBool('plot_history', True):
69       ii = np.asarray([v[0] for v in history])
         tt = np.asarray([v[1] for v in history])
         xx = np.asarray([v[2] for v in history])

         rc('text', usetex=True)
74       pylab.suptitle('Oregonator: TS \\texttt{%s}' % ts.getType())
         pylab.subplot(2,2,1)
         pylab.subplots_adjust(wspace=0.3)
         pylab.semilogy(ii[:-1], np.diff(tt), )
         pylab.xlabel('step number')
79       pylab.ylabel('timestep')

         for i in range(0,3):
             pylab.subplot(2,2,i+2)
             pylab.semilogy(tt, xx[:,i], "rgb"[i])
84           pylab.xlabel('time')
             pylab.ylabel('$x_%d$' % i)
         pylab.show()
```

In figure 1 we show show the output of the OREGO Python code. We have also performed a work-precision diagram illustrating the effect of choosing different tolerances in the TSAdapt (see §5) on the amount of effort and precision.
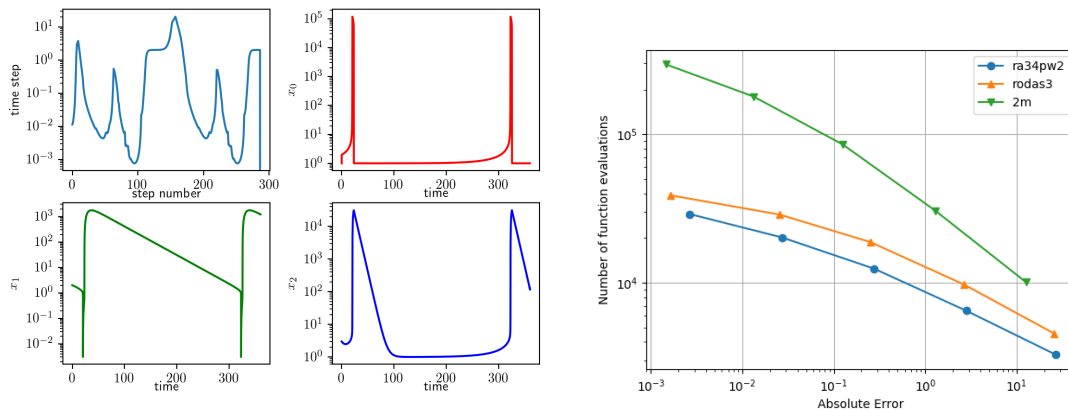


Fig. 1. Oregonator output and work-precision diagram for three Rosenbrock-W methods (§4.2) under different TSAdapt tolerances ($10^{-1}, \dots, 10^{-5}$).

## 4. TIME-STEPPING SCHEMES

This section describes the interfaces for setting the time-stepping schemes and their options. The classes of methods currently implemented in PETSc are described in Ta-

ble I. They cover multistage, multistep, and general linear methods with different stability properties. To address different problem requirements, PETSc provides explicit methods that are fast and accurate, implicit methods that have robust stability properties, and partitioned methods that combine both implicit and explicit integrators. Many of the methods implemented in PETSc allow users to register new schemes by supplying a new set of method coefficients. Most methods offer local error control. Global error estimation is also supported for several integrators. When implicit and semi-implicit methods are used, any of the PETSc linear and nonlinear solvers can be selected either by calling functions within the program or via the PETSc options database. These are fully described in the PETSc users manual [Balay et al. 2017]. The methods are summarized in Table I. The following list details some of the methods and their properties.

— **euler** Explicit Euler method. This is a basic implementation of the simplest time integrator.
— **ssp** Class of strong-stability-preserving multistage explicit methods suitable for hyperbolic partial differential equations (PDEs).
— **beuler, cn, theta** low-order implicit methods. These are suitable for DAEs and when stability is a concern.
— **alpha(2)** Time stepping developed for Navier-Stokes problems [Jansen et al. 2000].
— **glle** Implementation of implicit general linear methods for stiff problems.
— **rk** Implementation of explicit Runge-Kutta methods.
— **eimex** Implementation of extrapolated partitioned Runge-Kutta methods. These methods can have arbitrarily high orders [Constantinescu and Sandu 2010].
— **arkimex** Implementation of additive partitioned Runge-Kutta methods. These are suitable for problems with stiff and nonstiff components.
— **rosw** Implementation of Rosenbrock and W-methods, linearly implicit multistage methods with full or approximated Jacobian matrices. These methods are suitable for stiff, mildly nonlinear problems.
— **glee** Explicit and implicit general linear methods; typically self-starting, with global error estimation [Constantinescu 2018]. With suitable coefficients, these methods include 'euler', 'beuler', 'cn', 'theta', 'ssp', and 'rk'.
— **bdf** Standard backward differentiation methods. These are relatively low-order implicit multistep methods suitable for stiff problems and DAEs.

Explicit methods are conditionally stable. Implicit methods can be conditionally or unconditionally stable. Unconditional stability can be stronger or weaker. In this study we distinguish two types of stability: *A-Stable* methods, which have a stability region that covers the entire real-negative complex half plane, and *L-Stable* or *stiffly accurate* (SA) methods, which are A-Stable methods for which the amplification factor goes to zero as stiffness goes to infinity, thus giving them better stability properties for stiff problems and DAEs.

### 4.1. Partitioned Runge-Kutta
Partitioned methods are aimed at relaxing ODE integrator classification into strictly stiff and nonstiff problems that may have both types of characteristics. Partitioned methods employ two types of integrators: an explicit integrator for the nonstiff problem components and an implicit integrator suitable for the stiff ones [Ascher et al. 1997; Kennedy and Carpenter 2003]. Partitioned Runge-Kutta methods are typically strongly coupled; that is, both integrators participate in each stage calculation. In the literature these methods are known as implicit-explicit or semi-implicit [Giraldo et al. 2013; Zhong 1996].

Table I. List of time integration schemes available in the TS component of PETSc.

| TS Name | Reference | Class | Type | Order |
|---------|-----------|-------|------|-------|
| euler | forward Euler | one-step | explicit | 1 |
| ssp | multistage SSP [Ketcheson 2008] | Runge-Kutta | explicit | $\leq 4$ |
| beuler | backward Euler | one-step | implicit | 1 |
| cn | Crank-Nicolson | one-step | implicit | 2 |
| theta | theta-method | one-step | implicit | $\leq 2$ |
| alpha(2) | alpha-method [Jansen et al. 2000] | one-step | implicit | 2 |
| glle | general linear [Butcher et al. 2007] | general linear | implicit | $\leq 3$ |
| eimex | extrapolated IMEX [Constantinescu and Sandu 2010] | one-step | $\geq 1$, adaptive | |
| arkimex | §4.1 | IMEX Runge-Kutta | IMEX | $1-5$ |
| rosw | §4.2 | Rosenbrock-W | linearly implicit | $1-4$ |
| glee | method with global error estimation [Constantinescu 2018] | general linear | explicit/implicit | $1-4$ |
| bdf | standard BDF methods [Brenan et al. 1996] | multistep | implicit | $1-6$ |

A typical **additive** partitioning of an ODE problem results in the following:

$$\dot{u} = \underbrace{G(t,u)}_{\dot{U}_E} + \underbrace{H(t,u)}_{\dot{U}_I}, \tag{1}$$

where $\dot{U}_E$ denotes the nonstiff RHSFunction and $F(t,u,\dot{u}) = \dot{u} - \dot{U}_I$ the stiff IFunction in PETSc. Integrating this problem explicitly in $G$ and implicitly in $F$ from $u_n$ to $u_{n+1}$ by an additive Runge-Kutta (ARK) method defined by coefficients $(A = \{a_{ij}\}, b, c)$ for the explicit part and $(\tilde{A}, \tilde{b}, \tilde{c})$ for the implicit part is

$$U^{(i)} = u_n + \Delta t \sum_{j=1}^{i-1} a_{ij} \dot{U}_E^{(j)} + \Delta t \sum_{j=1}^{i} \tilde{a}_{ij} \dot{U}_I^{(j)}, \; i = 1, \ldots, s \tag{2a}$$

$$u_{n+1} = u_n + \Delta t \sum_{j=1}^{s} b_j \dot{U}_E^{(j)} + \Delta t \sum_{j=1}^{s} \tilde{b}_j \dot{U}_I^{(j)}, \tag{2b}$$

where $A = \{a_{ij}\}$ is strictly lower triangular, $\tilde{A} = \{\tilde{a}_{ij}\}$ is lower triangular and can have zeros on the diagonal (these correspond to explicit stages), and $\circ^{(i)}$ is stage index $i$.

The implementation of the standard IMEX scheme is as follows. Solve for $U^{(i)}$:

$$F\left(t_n + c_i\Delta t, U^{(i)}, \frac{1}{\Delta t\tilde{a}_{ii}}\left(U^{(i)} - Z\right)\right) = 0, \tag{3a}$$

$$Z = u_n + \Delta t\sum_{j=1}^{i-1}\tilde{a}_{ij}\dot{U}_I^{(j)} + \Delta t\sum_{j=1}^{i-1}a_{ij}\dot{U}_E^{(j)} \tag{3b}$$

$$\dot{U}_I^{(i)} = \frac{1}{\Delta t\tilde{a}_{ii}}\left(U^{(i)} - Z\right), \quad \dot{U}_E^{(i)} = G\left(t_n + c_i\Delta t, U^{(i)}\right), \quad i = 1,\ldots,s \tag{3c}$$

$$u_{n+1} = u_n + \Delta t\sum_{j=1}^{s}b_j\dot{U}_E^{(j)} + \Delta t\sum_{j=1}^{s}\tilde{b}_j\dot{U}_I^{(j)}. \tag{3d}$$

If $\tilde{a}_{ii} = 0$, then (3a) is skipped, and (3c) is modified. This approach allows direct use of these schemes for different types of problems as expressed in Table II.

Lower-order approximations are computed in the same way as for RK and ARK methods by evaluating (2b) with different $b$ and $\tilde{b}$ coefficients.

If one calls `TSARKIMEXSetFullyImplicit()` or use the option `-ts_imex_fully_implicit`, then (3a) solves $F(t,u,\dot{u}) = G(t,u)$ by using only the implicit integrator, and thus turning the time-stepping procedure into a diagonally implicit integrator. This facilitates solving DAEs and implicit ODEs. A summary of casting different problems through the partitioned additive interface is given in Table II. An IMEX formulation for problems such as $M\dot{u} = g(t,u) + h(t,u)$ requires the user to provide $M^{-1}g(t,u)$. General cases such as $F(t,u,\dot{u}) = G(t,u)$ are not amenable to IMEX Runge-Kutta but can be solved by using fully implicit methods, that is, by using the `-ts_imex_fully_implicit` option.

Table II. In PETSc, DAEs and ODEs are formulated as $F(t,u,\dot{u}) = G(t,u)$, where $F()$ is meant to be integrated implicitly and $G()$ explicitly.

| | | |
|---|---|---|
| $\dot{u} = g(t,u)$ | nonstiff ODE | $F(t,u,\dot{u}) := \dot{u}, G(t,u) := g(t,u)$ |
| $\dot{u} = h(t,u)$ | stiff ODE | $F(t,u,\dot{u}) := \dot{u} - h(t,u), G(t,u) := $ `NULL` |
| $M\dot{u} = h(t,u)$ | stiff ODE with mass matrix | $F(t,u,\dot{u}) := M\dot{u} - h(t,u), G(t,u) := $ `NULL` |
| $M\dot{u} = g(t,u)$ | nonstiff ODE with mass matrix | $F(t,u,\dot{u}) := \dot{u}, G(t,u) := M^{-1}g(t,u)$ |
| $\dot{u} = g(t,u) + h(t,u)$ | stiff-nonstiff ODE | $F(t,u,\dot{u}) := \dot{u} - h(t,u), G(t,u) := g(t,u)$ |
| $M\dot{u} = g(t,u) + h(t,u)$ | stiff-non-stiff ODE with mass matrix | $F(t,u,\dot{u}) := M\dot{u} - h(t,u), G(t,u) := M^{-1}g(t,u)$ |
| $h(t,y,\dot{y}) = 0$ | implicit ODE/DAE | $F(t,u,\dot{u}) := h(t,u,\dot{u}), G(t,u) := $ `NULL`; `TSSetEquationType()` set to `TS_EQ_IMPLICIT` or higher |

The *dense output* or *continuous approximation* [] of the solution within one timestep is also supported. This is used to obtain a high-order interpolation on the one hand and a hot-start initial guess for the Newton iterations on the other hand. The $s^* \geq s$ dense output formula for IMEX Runge-Kutta schemes of order $p^*$ is given by

$$u^*(t_n + \theta\Delta t) := u_n + \Delta t\sum_{i=1}^{s^*}B_i^*(\theta)g(t_n + c_i\Delta t, U^{(i)}) + \widehat{B}_i^*(\theta)f(t_n + c_i\Delta t, U^{(i)}), \tag{4}$$

where $\theta \in [0,1]$, $B_i^*(\theta) = \sum_{j=1}^{p^*}b_{ij}^*\theta^j$, and $\widehat{B}_i^*(\theta) = \sum_{j=1}^{p^*}\widehat{b}_{ij}^*\theta^j$. We typically take $b_{ij}^* = \widehat{b}_{ij}^*$. When $\theta > 1$, the dense output is used for extrapolation. This option is set

by `-ts_arkimex_initial_guess_extrapolate` and has the effect of setting the initial guess for all stages based on the dense output extrapolated solution from the previous step. In nonlinear problems this was shown to accelerate the code by up to three times; however, the gains are highly problem dependent.

Table III. List of the IMEX RK schemes currently available in the TS component of PETSc. For each method we list the reference, the total number of stages/implicit stages, the order/stage order, the implicit stability properties, stiff accuracy (SA), the existence of an embedded scheme, and dense output.

| TS Name | Reference | Stages (IM) | Order (stage) | IM Stab. | SA |
|---|---|---|---|---|---|
| 1bee | B Euler + Extrap | 3(3) | 1(1) | L | yes |
| a2 | RK2a + Trap. | 2(1) | 2(2) | A | yes |
| l2 | SSP2(2,2,2)[Pareschi and Russo 2005] | 2(2) | 2(1) | L | yes |
| ars122 | ARS122, [Ascher et al. 1997] | 2(1) | 3(1) | A | yes |
| 2c | [Giraldo et al. 2013] | 3(2) | 2(2) | L | yes |
| 2d | [Giraldo et al. 2013] | 3(2) | 2(2) | L | yes |
| 2e | [Giraldo et al. 2013] | 3(2) | 2(2) | L | yes |
| prssp2 | PRS(3,3,2) [Pareschi and Russo 2005] | 3(3) | 3(1) | L | |
| 3 | [Kennedy and Carpenter 2003] | 4(3) | 3(2) | L | yes |
| bpr3 | [Boscarino et al. 2011] | 5(4) | 3(2) | L | yes |
| ars443 | [Ascher et al. 1997] | 5(4) | 3(1) | L | yes |
| 4 | [Kennedy and Carpenter 2003] | 6(5) | 4(2) | L | yes |
| 5 | [Kennedy and Carpenter 2003] | 8(7) | 5(2) | L | yes |

| TS Name | Embed. | Dense Output | Remarks |
|---|---|---|---|
| 1bee | yes(1) | no | extrapolated BEuler |
| a2 | yes(1) | yes(2) | |
| l2 | es(1) | yes(2) | SSP, SDIRK |
| ars122 | yes(1) | yes(2) | |
| 2c | yes(1) | yes(2) | SDIRK,SSP |
| 2d | yes(1) | yes(2) | SDIRK |
| 2e | yes(1) | yes(2) | SDIRK |
| prssp2 | no | no | SSP, nonSDIRK |
| 3 | yes(2) | yes(2) | SDIRK |
| bpr3 | no | no | SDIRK, DAE-1 |
| ars443 | no | no | SDIRK |
| 4 | yes(3) | yes(2,3) | SDIRK |
| 5 | yes(4) | yes(3) | SDIRK |

## 4.2. Rosenbrock

Rosenbrock methods are linearly implicit versions of implicit Runge-Kutta methods. They use explicit function evaluations and implicit linear solves, and therefore they tend to be faster than the implicit Runge-Kutta methods because at each stage only a linear system needs to be solved, as opposed to the implicit Runge-Kutta methods that require solving a nonlinear system at each stage. An $s$-stage Rosenbrock method is defined by coefficient matrices $A = a_{ij}$, $j < i$ and $\Gamma = \gamma_{i,j}$, $j \le i$ and vector $b_i$, $i = 1, \ldots, s$. The Rosenbrock scheme applied to $\dot{u} = f(t, u)$ computes the solution at step $n + 1$ by

$$k_i = \Delta t f(t_n + c_i \Delta t, u_n + \sum_{j=1}^{i-1} a_{ij} k_j) + \Delta t J \sum_{j=1}^{i} \gamma_{ij} k_j \ \ i = 1, \ldots, s \tag{5a}$$

$$u_{n+1} = u_n + \sum_{i=1}^{s} b_i k_i, \tag{5b}$$

where $J$ is the Jacobian matrix of $f(t,u)$ at $t = t_n$ and $c_i = \sum_{j=1}^{i-1} a_{ij}$. Extensions to DAEs and PDAEs are readily available [Rang and Angermann 2005]. The linear system is defined in terms of the Jacobian matrix, which can be exact or approximated. The latter case leads to W-methods.

We follow the implementation suggested in [Rang and Angermann 2005; Kaps et al. 1985] where the coefficient matrix $\Gamma$ is inverted, and a change of variable is used

$$v_i = \sum_{j=1}^{i} \gamma_{ij} k_j \,, \; i = 1, \ldots, s \,,$$

leading to the following expressions:

$$\left( \frac{1}{\gamma_{ii}\Delta t} M - J \right) v_i = f \left( t_n + c_i \Delta t, v_n + \sum_{j=1}^{i-1} \omega_{ij} v_j \right) + \frac{1}{\Delta t} M \sum_{j=1}^{i-1} d_{ij} v_j \,, \; i = 1, \ldots, s \quad \text{(6a)}$$

$$u_{n+1} = u_n + \sum_{j=1}^{s} m_j v_j \,, \quad \text{(6b)}$$

where $\{d\}_{ij} = \text{diag}(\gamma_{11}^{-1}, \ldots, \gamma_{ss}^{-1}) - \Gamma^{-1}$, $\{\omega\}_{ij} = A\Gamma^{-1}$, $\{m\}_i = b\,\Gamma^{-1}$, $\gamma_i = \sum_{j=1}^{i-1} \gamma_{ij}$, and $M$ is a mass matrix that can be singular, resulting in a DAE. In our implementation we also allow for a noninvertible $\Gamma$ coefficient matrix by applying a correction to (6). This allows us to use methods that have explicit stages. Lower-order approximations are computed in the same way as for RK and ARK methods by evaluating (5b) with different $b$ coefficients. A work-precision diagram with three of these methods is presented in Fig. 1.

For PDEs, much of the source code is responsible for managing the mesh and spatial discretization, while only a small amount handles the time integration. In PETSc the bridge between the mass of code that handles the mesh and discretization and the solver and time integrator is the `DM` object. This object provides the information needed by the solvers and integrators while concealing all the details of the mesh and discretization management. PETSc provides several DM classes including `DMDA` for structured grids with finite difference discretizations and `DMPLEX` for unstructured meshes with finite element or finite volume discretizations. We present an example of a PDE discretized on a structured grid using PETSc's DMDA and time integrators. This example demonstrates an interesting nontrivial pattern formation with a reaction-diffusion equation.

```
#include <petscdm.h>
#include <petscdmda.h>
#include <petscts.h>
4
typedef struct {
  PetscScalar u,v;
} Field;
typedef struct {
9   PetscReal D1,D2,gamma,kappa;
} AppCtx;

int main(int argc,char **argv){
  TS            ts;            /* ODE integrator */
14  Vec           x;            /* solution */
  DM            da;
  AppCtx        appctx;
```

Table IV. List of the Rosenbrock W-schemes currently available in the TS component of PETSc. For each method we listed the reference, the total number of stages and implicit stages, the scheme order and stage-order, the implicit stability properties, stiff accuracy (SA), the existence of an embedded scheme, dense output, the capacity to use inexact Jacobians (-W), and high-order integration of differential algebraic equations (PDAE).

| TS Name | Reference | Stages (IM) | Order (stage) | IM Stab. | SA |
|---|---|---|---|---|---|
| theta1 | classical | 1(1) | 1(1) | L | - |
| theta2 | classical | 1(1) | 2(2) | A | - |
| 2m | Zoltan | 2(2) | 2(1) | L | No |
| 2p | Zoltan | 2(2) | 2(1) | L | No |
| ra3pw | [Rang and Angermann 2005] | 3(3) | 3(1) | A | No |
| ra34pw2 | [Rang and Angermann 2005] | 4(4) | 3(1) | L | Yes |
| rodas3 | [Sandu et al. 1997] | 4(4) | 3(1) | L | Yes |
| sandu3 | [Sandu et al. 1997] | 3(3) | 3(1) | L | Yes |
| assp3p3s1c | unpublished | 3(2) | 3(1) | A | No |
| lassp3p4s2c | unpublished | 4(3) | 3(1) | L | No |
| lassp3p4s2c | unpublished | 4(3) | 3(1) | L | No |
| ark3 | unpublished | 4(3) | 3(1) | L | No |

| TS Name | Embed. | Dense Output | -W | PDAE | Remarks |
|---|---|---|---|---|---|
| theta1 | | - | - | - | - |
| theta2 | - | - | - | - | - |
| 2m | Yes(1) | Yes(2) | Yes | No | SSP |
| 2p | Yes(1) | Yes(2) | Yes | No | SSP |
| ra3pw | Yes | Yes(2) | No | Yes(3) | - |
| ra34pw2 | Yes | Yes(3) | Yes | Yes(3) | - |
| rodas3 | Yes | No | No | Yes | - |
| sandu3 | Yes | Yes(2) | No | No | - |
| assp3p3s1c | Yes | Yes(2) | Yes | No | SSP |
| lassp3p4s2c | Yes | Yes(3) | Yes | No | SSP |
| lassp3p4s2c | Yes | Yes(3) | Yes | No | SSP |
| ark3 | Yes | Yes(3) | Yes | No | IMEX-RK |

```
     PetscInitialize(&argc,&argv,(char*)0,help);
19   appctx.D1    = 8.0e-5;
     appctx.D2    = 4.0e-5;
     appctx.gamma = .024;
     appctx.kappa = .06;
     /*  Create distributed array (DMDA) to manage parallel grid and vectors */
24   DMDACreate2d(PETSC_COMM_WORLD,DM_BOUNDARY_PERIODIC,DM_BOUNDARY_PERIODIC,
         DMDA_STENCIL_STAR,65,65,PETSC_DECIDE,PETSC_DECIDE,2,1,NULL,NULL,&da);
     DMSetFromOptions(da); DMSetUp(da);
     DMDASetFieldName(da,0,"u"); DMDASetFieldName(da,1,"v");
     DMCreateGlobalVector(da,&x);
     /* Create timestepping solver context */
29   TSCreate(PETSC_COMM_WORLD,&ts);
     TSSetType(ts,TSARKIMEX);
     TSARKIMEXSetFullyImplicit(ts,PETSC_TRUE);
     TSSetDM(ts,da);
     TSSetProblemType(ts,TS_NONLINEAR);
34   TSSetRHSFunction(ts,NULL,RHSFunction,&appctx);
     TSSetRHSJacobian(ts,NULL,NULL,RHSJacobian,&appctx);
     /* Set initial conditions */
     InitialConditions(da,x);
     TSSetSolution(ts,x);
39   /*  Set solver options  */
     TSSetMaxTime(ts,2000.0);
     TSSetTimeStep(ts,.0001);
     TSSetExactFinalTime(ts,TS_EXACTFINALTIME_STEPOVER);
     TSSetFromOptions(ts);
44   /*  Solve ODE system  */
```

```
      TSSolve(ts,x);
      VecDestroy(&x); TSDestroy(&ts); DMDestroy(&da);
      PetscFinalize();
   }
49 /*    RHSFunction - Evaluates nonlinear function, F(x). */
   RHSFunction(TS ts,PetscReal ftime,Vec U,Vec F,void *ptr) {
      AppCtx          *appctx = (AppCtx*)ptr;
      DM              da;
      PetscInt        i,j,Mx,My,xs,ys,xm,ym;
54    PetscReal       hx,hy,sx,sy;
      PetscScalar     uc,uxx,uyy,vc,vxx,vyy;
      Field           **u,**f;
      Vec             localU;

59    TSGetDM(ts,&da);
      DMGetLocalVector(da,&localU);
      DMDAGetInfo(da,PETSC_IGNORE,&Mx,&My,PETSC_IGNORE,...)
      hx = 2.50/(PetscReal)(Mx); sx = 1.0/(hx*hx);
      hy = 2.50/(PetscReal)(My); sy = 1.0/(hy*hy);
64    /* Scatter ghost points to local vector  */
      DMGlobalToLocalBegin(da,U,INSERT_VALUES,localU);
      DMGlobalToLocalEnd(da,U,INSERT_VALUES,localU);
      DMDAVecGetArrayRead(da,localU,&u);
      DMDAVecGetArray(da,F,&f);
69    /*  Get local grid boundaries
      DMDAGetCorners(da,&xs,&ys,NULL,&xm,&ym,NULL);
      /*  Compute function over the locally owned part of the grid */
      for (j=ys; j<ys+ym; j++) {
        for (i=xs; i<xs+xm; i++) {
74        uc          = u[j][i].u;
          uxx         = (-2.0*uc + u[j][i-1].u + u[j][i+1].u)*sx;
          uyy         = (-2.0*uc + u[j-1][i].u + u[j+1][i].u)*sy;
          vc          = u[j][i].v;
          vxx         = (-2.0*vc + u[j][i-1].v + u[j][i+1].v)*sx;
79        vyy         = (-2.0*vc + u[j-1][i].v + u[j+1][i].v)*sy;
          f[j][i].u = appctx->D1*(uxx + uyy)-uc*vc*vc+appctx->gamma*(1.0-uc);
          f[j][i].v = appctx->D2*(vxx + vyy)+uc*vc*vc-(appctx->gamma +
              appctx->kappa)*vc;
        }
      }
84    DMDAVecRestoreArrayRead(da,localU,&u);
      DMDAVecRestoreArray(da,F,&f);
      DMRestoreLocalVector(da,&localU);
   }
   RHSJacobian(TS ts,PetscReal t,Vec U,Mat A,Mat BB,void *ctx) {
89    AppCtx          *appctx = (AppCtx*)ctx;  /* application context */
      DM              da;
      PetscInt        i,j,Mx,My,xs,ys,xm,ym;
      PetscReal       hx,hy,sx,sy;
      PetscScalar     uc,vc;
94    Field           **u;
      Vec             localU;
      MatStencil      stencil[6],rowstencil;
      PetscScalar     entries[6];

99    TSGetDM(ts,&da);
      DMGetLocalVector(da,&localU);
      DMDAGetInfo(da,PETSC_IGNORE,&Mx,&My,PETSC_IGNORE,...)
      hx = 2.50/(PetscReal)(Mx); sx = 1.0/(hx*hx);
      hy = 2.50/(PetscReal)(My); sy = 1.0/(hy*hy);
104   DMGlobalToLocalBegin(da,U,INSERT_VALUES,localU);
      DMGlobalToLocalEnd(da,U,INSERT_VALUES,localU);
      DMDAVecGetArrayRead(da,localU,&u);
```

```
      DMDAGetCorners(da,&xs,&ys,NULL,&xm,&ym,NULL);

109   stencil[0].k = 0; stencil[1].k = 0; stencil[2].k = 0;
      stencil[3].k = 0; stencil[4].k = 0; stencil[5].k = 0;
      rowstencil.k = 0;
      for (j=ys; j<ys+ym; j++) {
        stencil[0].j = j-1;
114     stencil[1].j = j+1;
        stencil[2].j = j;
        stencil[3].j = j;
        stencil[4].j = j;
        stencil[5].j = j;
119     rowstencil.k = 0; rowstencil.j = j;
        for (i=xs; i<xs+xm; i++) {
          uc = u[j][i].u;
          vc = u[j][i].v;
          uyy      = (-2.0*uc + u[j-1][i].u + u[j+1][i].u)*sy;
124       vxx      = (-2.0*vc + u[j][i-1].v + u[j][i+1].v)*sx;
          vyy      = (-2.0*vc + u[j-1][i].v + u[j+1][i].v)*sy;
          f[j][i].u = appctx->D1*(uxx + uyy)-uc*vc*vc+appctx->gamma*(1.0-uc);

          stencil[0].i = i; stencil[0].c = 0; entries[0] = appctx->D1*sy;
129       stencil[1].i = i; stencil[1].c = 0; entries[1] = appctx->D1*sy;
          stencil[2].i = i-1; stencil[2].c = 0; entries[2] = appctx->D1*sx;
          stencil[3].i = i+1; stencil[3].c = 0; entries[3] = appctx->D1*sx;
          stencil[4].i = i; stencil[4].c = 0; entries[4] = -2.0*appctx->D1*(sx +
              sy) - vc*vc - appctx->gamma;
          stencil[5].i = i; stencil[5].c = 1; entries[5] = -2.0*uc*vc;
134       rowstencil.i = i; rowstencil.c = 0;
          MatSetValuesStencil(A,1,&rowstencil,6,stencil,entries,INSERT_VALUES);

          stencil[0].c = 1; entries[0] = appctx->D2*sy;
          stencil[1].c = 1; entries[1] = appctx->D2*sy;
139       stencil[2].c = 1; entries[2] = appctx->D2*sx;
          stencil[3].c = 1; entries[3] = appctx->D2*sx;
          stencil[4].c = 1; entries[4] = -2.0*appctx->D2*(sx + sy) + 2.0*uc*vc -
              appctx->gamma - appctx->kappa;
          stencil[5].c = 0; entries[5] = vc*vc;
          rowstencil.c = 1;
144       MatSetValuesStencil(A,1,&rowstencil,6,stencil,entries,INSERT_VALUES);
        }
      }
      DMDAVecRestoreArrayRead(da,localU,&u);
      DMRestoreLocalVector(da,&localU);
149   MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);
      MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);
      MatSetOption(A,MAT_NEW_NONZERO_LOCATION_ERR,PETSC_TRUE);
    }
```

## 5. ADAPTIVE TIME STEPPING AND ERROR CONTROL

PETSc provides several options for automatic timestep control in order to attain a user-specified goal via a `TSAdapt` context. Typically, the goals are related to accuracy. In this case the user provides an absolute (ATOL) and a relative (RTOL) error tolerance. The adaptor controls the timestep in order to meet the specified error tolerance. Most time-stepping methods with adaptivity evaluate a lower-order approximation at each timestep by using a different set of coefficients, denote this solution as $\tilde{u}$. The following weighted error quantity is used for time-step control:

$$werr(t_{[n]}) = \frac{||u(t_{[n]}) - \tilde{u}(t_{[n]})||_{\{1,2,\ldots,\infty\}}}{\text{ATOL} + \text{RTOL}\max(|u(t_{[n]})|, |\tilde{u}(t_{[n]})|)} \ . \tag{7}$$

If $werr(t_{[n]})$ is larger than one, then the estimated local trunctation error at step $n$ exceeds ATOL or RTOL. Otherwise, the estimated error is less than that prescribed by the user, in which case the step is accepted and the next step is adjusted so that it tracks whether $werr(t_{[n]})$ will approach the value one. If the error exceeds the tolerances specified by the user, then the step is rejected and a smaller timestep is taken. This logic is implemented in the "basic" adaptor. A more advanced adaptivity logic based on linear digital control theory and aimed at producing smoother step size sequences is implemented in the "dsp" adapter [Söderlind 2003; Söderlind and Wang 2006].

In many fluid dynamics applications the timestep is restricted by stability considerations as given by the Courant-Friedrichs-Lewy (CFL) condition. TS provides an adapter that controls the timestep so that the CFL stability is not exceeded. Additionally, a special adapter for controlling the global error for the TS glee method [Constantinescu 2018] is available. This adapter can be used wherever the standard (basic) one is used. Similar to the basic adapter, the glee adapter can be used for tracking the absolute and relative errors separately.

A list of timestep adapters is presented in Table V. Custom adapters can be easily registered vis the PETSc API.

Table V. List of time integration adapter schemes currently available in the `TSAdapt` component of PETSc.

| TS Adapt Name | Remarks | Used by |
|---|---|---|
| none | No adaptivity | all |
| basic | Standard timestep adaptivity [Gear 1971] | all with lower-order error approximation |
| dsp | Adapter using control theory [Söderlind 2003] | same as basic |
| cfl | Controls the timestep to match error provided CFL limit | typically TS spp, rk |
| glee | Time step adaptivity with global error estimation[Constantinescu 2018] | typically for TS glee methods, extends TSAdapt basic |

## 6. COMPUTING SENSITIVITIES (DERIVATIVES)

The time-stepping library provides a framework based on discrete forward (tangent linear) and adjoint models for sensitivity analysis for ODEs and DAEs. The ODE/DAE solution process (henceforth called the forward run) can be obtained by using either explicit or implicit solvers in TS, depending on the problem properties. Currently supported method types are TSRK (Runge-Kutta) explicit methods and TSTHETA (Theta) implicit methods.

### 6.1. Discrete adjoint sensitivity

The TSAdjoint routines of PETSc provide the capability to calculate partial derivatives of a given objective function

$$\Psi_i(u_0, p) = \Phi_i(u_F, p) + \int_{t_0}^{t_F} r_i(u(t), p, t)dt \quad i = 1, ..., n_{\text{obj}}, \tag{8}$$

subject to initial conditions $u_0$ and parameters $p$.

Without loss of generality, we assume that the system is integrated with a one-step method,

$$u_{n+1} = \mathcal{N}_n(u_n), \quad n = 0, \dots, N-1, \quad u_0 = \mathcal{I}, \tag{9}$$

where $\mathcal{I}$ are the initial values and the solution at the end of the simulation is given by $u_N$.

To illustrate the approach consider a simple case in which we compute the sensitivities of the terminal function $\psi(u_N)$ to initial values only. We use the Lagrange multipliers $\lambda_0, \ldots, \lambda_N$ to account for the constraint from each timestep, and we define the Lagrangian as

$$\mathcal{L}(\mathcal{I}, u_0, \ldots, u_N, \lambda_0, \ldots, \lambda_N) = \psi(u_N) - \lambda_0^T (u_0 - \mathcal{I}) - \sum_{n=0}^{N-1} \lambda_{n+1}^T (u_{n+1} - \mathcal{N}(u_n)). \quad (10)$$

Differentiating Equation (10) w.r.t. $\mathcal{I}$ and applying the chain rule, we obtain

$$\frac{d\mathcal{L}}{d\mathcal{I}} = \lambda_0^T - \left( \frac{d\psi}{du}(u_N) - \lambda_N^T \right) \frac{\partial u_N}{\partial \mathcal{I}} - \sum_{n=0}^{N-1} \left( \lambda_n^T - \lambda_{n+1}^T \frac{d\mathcal{N}}{dy}(u_n) \right) \frac{\partial u_n}{\partial \mathcal{I}}. \quad (11)$$

By defining $\lambda$ to be the solution of the discrete adjoint model,

$$\lambda_N = \left( \frac{d\psi}{du}(u_n) \right)^T, \quad \lambda_n = \left( \frac{d\mathcal{N}}{du}(u_n) \right)^T \lambda_{n+1}, \quad n = N - 1, \ldots, 0, \quad (12)$$

we obtain the gradient $\nabla_{\mathcal{I}} \mathcal{L} = \nabla_{\mathcal{I}} \psi(u_n) = \lambda_0$.

This model can be expanded to incorporate integral objective functions and calculate parametric sensitivities by augmenting the state variable into a larger system. See [Zhang et al. 2017] for more details.

To efficiently calculate the gradient with adjoint method, one needs to first perform a forward run that solves the original equation and saves the solution trajectory with a checkpointing scheme, initialize the adjoint sensitivity variables, and then perform a backward run that propagates the adjoint sensitivity according to (12). As can be seen from the adjoint equation (12), performing an adjoint step requires trajectory information including the solution vector at the current step and optional stage values if a multistage time-stepping method is used. Applying checkpointing techniques that have partial recomputation of the solution provides a balance between recomputation and storage.

To use the PETSc adjoint solver, one creates two arrays of $n_{\text{cost}}$ vectors $\lambda$ and $\mu$ (if there are no parameters $p$, $\mu$ can be set to NULL). The $\lambda$ vectors have the same dimension and parallel layout as the solution vector for the ODE, and the $\mu$ vectors are of dimension $p$. The vectors $\lambda_i$ and $\mu_i$ should be initialized with the values $d\Phi_i/dy|_{t=t_F}$ and $d\Phi_i/dp|_{t=t_F}$ respectively.

If $F()$ is a function of $p$, one needs to also provide the Jacobian $F_p$ with

```
TSAdjointSetRHSJacobian(TS ts,Mat Amat,
    (*fp)(TS,PetscReal,Vec,Mat,void*),void *ctx)
```

The arguments for the function fp() are the timestep context, current time, $u$, and the (optional) user-provided context. If there is an integral term in the cost function, one must also provide Jacobian terms for the integrand with

```
TSSetCostIntegrand(TS ts,PetscInt numcost,Vec costintegral,
    (*rf)(TS,PetscReal,Vec,Vec,void*),(*drduf)(TS,PetscReal,Vec,Vec*,void*),
    (*drdpf)(TS,PetscReal,Vec,Vec*,void*),void *ctx)
```

where $\text{drduf} = dr/du$, $\text{drdpf} = dr/dp$.

The integral term can be evaluated in either the forward run or the backward run by using the same time-stepping algorithm as for the original equations.

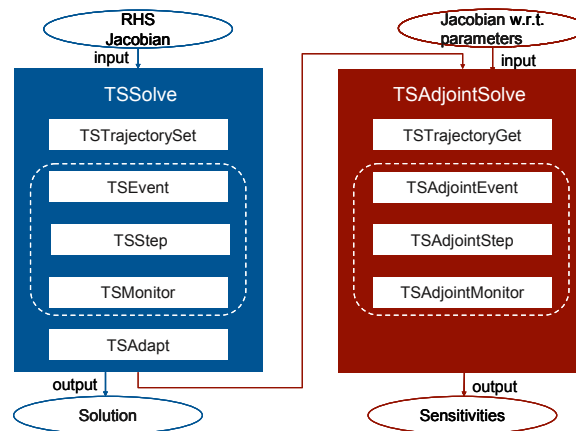The features of PETSc adjoint solver are summarized as follows.

Fig. 2. Structure of PETSc implementation for adjoint sensitivity analysis [Zhang et al. 2017]

— Supports multiobjective sensitivity calculation and integral objective functions.
— Handles hybrid dynamical systems with discontinuities.
— Contains state-of-the-art checkpointing schemes.

The adjoint solver is built on existing components in PETSc's time-stepping library `TS`, as shown in Fig. 2.

(1) The `TSEvent` object, further introduced in Sec. 7, supports detecting events and allows users to add a post-event handler to modify the right-hand side function, reinitialize the DAE system, and apply jump conditions for sensitivity analysis. It is particularly important for the simulation of hybrid dynamical systems.
(2) The `TSTrajectory` object provides a variety of sophisticated online and offline checkpointing schemes that are suitable for single-level storage media (for example, RAM) and multilevel storage media (RAM and external disk/tape). Trajectory information is stored as checkpoints in the forward run by repeatedly calling `TSTrajectorySet` at each timestep. `TSTrajectoryGet` is responsible for obtaining the required trajectory information before an adjoint step starts. It may extract the information from the restored checkpoint directly or recompute from the checkpoint. Recomputation typically happens when checkpoints are stored only at selective timesteps because of limited storage capacity. `TSTrajectorySet` and `TSTrajectoryGet` encapsulate the state-of-the-art checkpointing scheduler `revolve` [Griewank and Walther 2000] that can generate a guaranteed optimal strategy.
(3) `TSAdjointStep` corresponds to the adjoint version of `TSStep`, which fulfills the time-stepping operator $\mathcal{N}(u_n)$. Thus they have similar underlying infrastructure, and their implementations differ from one time-stepping method to another. By design, the inputs for the adjoint solver are either reused or modified from the original `TS` solver.

All the components are compatible with one another and used together within the highly composable solver in order to tackle the difficulties of hybrid systems. The paper [Marin et al. 2017] contains details on using the infrastructure discussed here for solving PDE-constrained optimization problems utilizing the spectral element method.

### 6.2. Discrete forward (tangent linear) sensitivity

The discrete forward (also known as tangent linear) model for a one-step time integration algorithm can be obtained by taking the derivative of (9) with respect to the

parameters. The propagation equation for parameters $p$ can be symbolically described by

$$\mathcal{S}_0 = \frac{d\mathcal{I}}{dp}, \quad \mathcal{S}_{n+1} = \frac{d\mathcal{N}}{du}(u_n)\mathcal{S}_n, \quad n = 0, \ldots, N-1, \tag{13}$$

where $\mathcal{S}_n = dX_n/dp$ is a matrix denoting the solution sensitivities (or so-called trajectory sensitivities in the power system field). Note that each parameter results in one corresponding column of the sensitivity matrix $\mathcal{S}$ and one linear equation to be solved at each timestep. Consequently, the computational cost of the forward approach is linear in the number of parameters for which the sensitivities are calculated. This feature usually limits its application to cases involving few parameters.

Like the discrete adjoint models, the implementation of discrete forward models also depend on the particular time integration algorithm. In principle, these two models are analogous to the well-known forward and reverse modes of algorithmic differentiation (AD) that are applied to high-level abstractions of a computer program. Traditional AD handles a sequence of operations (either a source code line or a binary instruction) while in our case the primitive operation is a time step.

Furthermore, the forward model requires the same ingredients as those needed in the adjoint model. Users may need to provide `TSAdjointSetRHSJacobian()` and TSSet-CostIntegrand() in the same way that they are used for TSAdjoint.

Although forward sensitivities are not used as frequently with gradient-based optimization algorithms as are adjoint sensitivities, they still are convenient for calculating gradients for objective function in the general form (8). Specifically, the total derivative of the scalar function $\Phi(X_N)$ can be computed with

$$\frac{d\Phi}{dp}(X_N) = \frac{\partial\Phi}{\partial X}(X_N)\mathcal{S}_N + \frac{\partial\Phi}{\partial p}(X_N). \tag{14}$$

The total derivative of the integral term in (8) (denote by $q$ for simplicity) to parameters $p$ is given as

$$\frac{dq}{dp} = \int_{t_0}^{t_F} \left( \frac{\partial r}{\partial X}(t, X)\mathcal{S} + \frac{\partial r}{\partial p}(t, X) \right) dt. \tag{15}$$

This integral together with $q$ is calculated automatically by PETSc with the same timestepping algorithm and sequence of timesteps in the discrete approaches for consistency, when users provide the necessary Jacobian callbacks with `TSSetCostIntegrand`. In addition, the forward apporach is useful for obtaining solution sensitivities often required by second-order adjoint sensitivity analysis [Azyurt and Barton 2005].

## 7. HANDLING DISCONTINUITIES AND EVENTS

One characteristic of applications, typically from the control systems world, is the discontinuous nature of equations due to the presence of various time- and state-based nonlinearities such as faults, limiters, and hysterisis. Such discontinuities give rise to the following conditionals[1] introduced in the ODE or DAE equations:

$$\begin{cases} x - x^+ = 0, & \text{if } x \geq x^+ \\ x - x^- = 0, & \text{if } x \leq x^- \\ \dot{x} = f(x), & \text{otherwise.} \end{cases} \tag{16}$$

PETSc supports the handling of such discontinuities through its event-handling mechanism called *TSEvent*. Detecting and locating such discontinuities is done using

---

[1]Equation 16 shows one form for illustrative purposes. In general, the conditionals can include functions of the state variables instead of simple box constraints.

an event handler or root-finding method. A switching function $h(t, x) = 0$ is propagated along with the equations. The mechanism of event detection and location is illustrated in Fig. 3.
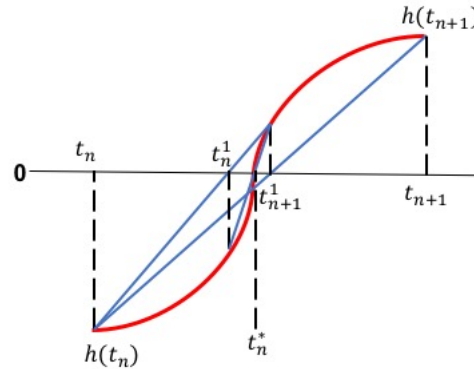


Fig. 3.   Detection and location of nonlinearities

The time-stepper checks for the *zero-crossing* of the event function at every timestep. Specific directions of zero-crossing—positive only, negative only, or both—can be provided. The zero-crossing of an event is detected by the sign change of the event function, namely, $sign(h(t_n)) \neq sign(h(t_{n+1}))$. If this condition is true, the event is said to be detected and the solution rolled back to $t_n$. By using interpolation and successively shrinking the time boundaries, the zero-crossing of the event function is detected when its value is within a specified tolerance. At this time instant, $t_n^*$ in Fig. 3, the discontinuity is applied, and an additional step is taken to synchronize with $t_{n+1}$. TSEvent also incorporates further improvements to avoid duplicate steps (by utilizing the Illinois algorithm [Dowell and Jarratt 1971]), and it speeds the detection of event zero-crossing by using the Anderson–Björck method. [Galdino 2011]. In the case of multiple events detected during the same timestep, the event detection mechanism uses the smallest interpolated timestep from the list of events.

Figure 4 presents a simple example illustrating the usage of TSEvent for a bouncing ball.

Events can be set to TS through the application interface function *TSSetEventHandler()*, which has the following form:

```
TSSetEventHandler(TS ts,PetscInt nevents,PetscInt direction[],PetscBool
    terminate[],(*eventfun)(TS,PetscReal t,Vec X,PetscScalar h[],void
    *ctx),(*posteventfun)(TS ts,PetscInt nevents_det,PetscInt
    event_id[],PetscReal t,Vec X,PetscBool forwardsolve,void *ctx),void
    *ctx);
```

Here, *nevents* is the number of local events to be located, *direction[]* is an array of zero-crossing direction for each event, and *terminate[]* array controls terminating TS time-stepping after an event has been located. The event function $h(t, x)$ is set through the callback function *\*eventfun*; and, optionally, a post-event function *(\*posteventfun)* can be set that is called after an event (or simultaneous multiple events) are located. Specific actions following an event can be performed through the post-event function.

For event functions having widely differing scales or range of values, finer control on locating the events can be provided through the *TSSetEventTolerances()* function:

```
TSSetEventTolerances(TS,PetscReal tol, PetscReal tols[]);
```
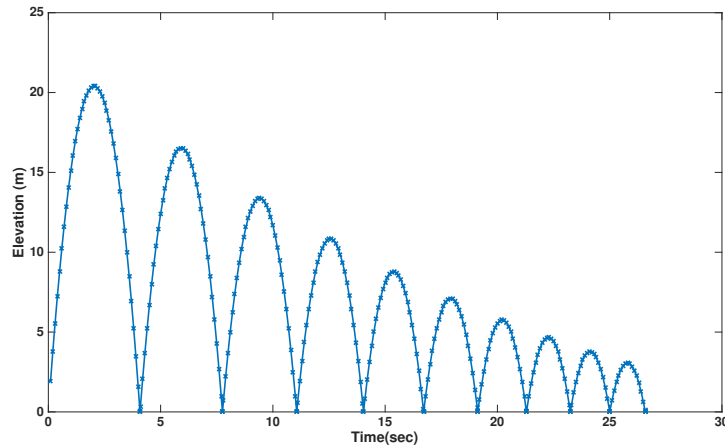
Fig. 4.   Bouncing ball example: The dynamics of the bouncing ball are described by the equations $\dot{u} = v$ and $\dot{v} = -9.8$. The ball velocity $v$ is attenuated by a factor of 0.9 every time it hits the ground $u = 0$.

A single tolerance *tol* can be used for all the events, or tolerances for each event can be set via the *tols* array.

## 8. MONITORING AND VISUALIZATION

Often users of ODE solver packages do not know much about even the qualitative properties of the ODE they are solving; for example, they may not even know whether it is stiff or which parts of the ODE are stiff. To help users understand the qualitative properties of the solution, PETSc/TS provides an extensible approach that allows monitoring and visualizing the solution as well as solution properties, such as maximum values of the solution or eigenvalues of the Jacobian.

Monitoring and visualization in PETSc are organized around the `PetscViewer` object, which is an abstraction of ASCII and binary files, as well as graphics APIs. Objects can be "viewed" with varying levels of refinement based on the viewer used and options set for the viewer. For example, `MatView(A,viewer);` can display minimalistically, with ASCII text, the size of the matrix and the number of nonzeros or the entire matrix in binary format in a file or as an image of the sparsity pattern depending on the viewer used. The reduction of the object from its parallel representation is handled automatically by the PETSc libraries. In addition to viewing the PETSc data objects `Vec` and `Mat`, one can (in fact, doing so is desirable) view the solver objects, for example, `TS`. With an ASCII viewer it prints information about the type of solver being used and all its options; for binary viewers it saves the state of the object that can be reloaded into memory with `TSLoad()`; and for graphics viewers it displays the relationship of the solver with the other solvers in process, for example, that a SNES nonlinear solver object is embedded in a TS object and that a KSP linear solver object is embedded in the nonlinear solver object if Newton's method is being used. Here we display the output of a `TSView()` on a particular ODE solver in ASCII.

```
TS Object: 1 MPI processes
  type: rosw
  maximum steps=1000
  maximum time=20
  total number of nonlinear solver iterations=108
  total number of nonlinear solve failures=0
```

```
total number of linear solver iterations=108
total number of rejected steps=0
  Rosenbrock-W ra34pw2
  Abscissa of A       =  0.000000  0.871733  0.731580  1.000000
  Abscissa of A+Gamma =  0.435867  0.871733  0.731580  1.000000
TSAdapt Object:   1 MPI processes
  type: basic
  number of candidates 1
    Basic: clip fastest decrease 0.1, fastest increase 10
    Basic: safety factor 0.9, extra factor after step rejection 0.5
SNES Object:   1 MPI processes
  type: ksponly
  maximum iterations=50, maximum function evaluations=10000
  tolerances: relative=1e-08, absolute=1e-50, solution=1e-08
  total number of linear solver iterations=1
  total number of function evaluations=1
  Jacobian is never rebuilt
  SNESLineSearch Object:     1 MPI processes
    type: basic
    maxstep=1.000000e+08, minlambda=1.000000e-12
    tolerances: relative=1.000000e-08, absolute=1.000000e-15, lambda=1.000000e-08
    maximum iterations=1
  KSP Object:     1 MPI processes
    type: preonly
    maximum iterations=10000, initial guess is zero
    tolerances:  relative=1e-05, absolute=1e-50, divergence=10000
    left preconditioning
    using NONE norm type for convergence test
  PC Object:     1 MPI processes
    type: lu
      LU: out-of-place factorization
      tolerance for zero pivot 2.22045e-14
      matrix ordering: nd
      factor fill ratio given 5, needed 1
        Factored matrix follows:
          Mat Object:               1 MPI processes
            type: seqaij
            rows=3, cols=3
            package used to perform factorization: petsc
            total: nonzeros=9, allocated nonzeros=9
            total number of mallocs used during MatSetValues calls =0
              using I-node routines: found 1 nodes, limit used is 5
    linear system matrix = precond matrix:
    Mat Object:       1 MPI processes
      type: seqaij
      rows=3, cols=3
      total: nonzeros=9, allocated nonzeros=15
      total number of mallocs used during MatSetValues calls =0
        using I-node routines: found 1 nodes, limit used is 5
```

Viewing of solver objects can usually be controlled at runtime via the options database. For example, `-ts_view` produces ASCII output about the solver, whereas `-ts_view draw` produces a graphical display of the solver.

In addition to static views of PETSc data and solver objects, we provide numerous ways of dynamically viewing the solution and properties of the solution, from within the program or via the options database. This process is handled via "monitor" callback functions that can be attached to solver objects. For TS this is done with

```
TSMonitorSet(TS ts,(*monitor)(TS ts,PetscInt timestep,PetscReal time,Vec u
    ,void*mctx),void *mctx,(*mdestroy)(void**mctx));
```

The `monitor()` function provided is called at the beginning and at the end of each timestep, and it can present the solution information in any way the user likes. Various monitors may be set for the same solver. PETSc provides a variety of default monitors that

— print the current timestep and time,
— save the current solution to a binary or vtk file,
— display the current solution using a variety of graphical approaches using X windows or OpenGL, and
— display the eigenvalues of the current operator, which is useful for understanding the stability of the scheme being used.

In addition, the `monitor()` routines can compute and track information over the lifetime of the simulation, for example, maximum and minimum values of the solution or conserved quantities. The idea is that rather than requiring users to modify the actual ODE integrator code to track any property of the solution or solution process, simple monitor routines are provided. Many of these monitoring routines can be controlled from the command line, for example `-ts_monitor_lg_timestep` allows one to graphically monitor the changes in the adapted timestep as the computation proceeds.
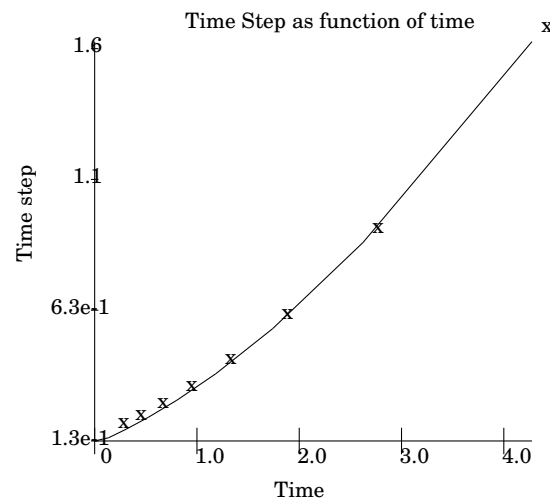


The nonlinear and linear solvers also provide the same type of flexible monitoring of the convergence process, with many available default monitors allowing one to track how well the selected solvers are working.

PETSc provides a simple but powerful API and options for gathering performance information about the solution time, for example, time in the linear solvers and time in the computation of the Jacobian. These allow users to quickly focus in on the portions of the computation that are the most time consuming and either select alternative algorithms or further optimize the implementation. These are discussed in the PETSc users manual [Balay et al. 2017].

## 9. SUPPORT FOR SPECIFIC APPLICATION DOMAINS

Many application areas have their own vocabulary and methodology for describing their problem that are often distinct from the language of ODEs. Although underlying

their simulation is a set of ODEs or DAEs, they never work directly with this form; rather, they express their problems at a higher level of abstraction. Users can easily take advantage of these higher levels with the PETSc ODE and DAE solvers, which allow the users to work with their own natural language for defining the problem and working with it. We already provide this higher-level interface for two application areas: power systems analysis and chemical reactions. We expect to do more in the future in collaboration with application partners.

### 9.1. Electrical Power Grid via DMNetwork

Applications in an electrical power grid span a large range of temporal and spatial scales that entail problems involving secure, stable, and efficient planning and operation of the grid. A list of potential applications suitable for PETSc usage is given in [Abhyankar et al. 2011]. PETSc's time-stepping library TS has been used mainly for applications assessing the impacts of large disturbances, such as short circuits and equipment outages, on the stability of the grid. In such applications, called *transient stability analysis* in electrical power grid parlance, the stability of the grid is determined through a time-domain simulation of the power grid differential-algebraic equations. The differential equations, $f(t, y, x)$, describe the dynamics of electromechanical generators and motors, while the algebraic equations, $g(t, x, y)$, are used for the electrical network comprising transmission lines, transformers, and other connecting equipment.

$$\dot{x} = f(t, x, y) \tag{17}$$
$$0 = g(t, x, y) \tag{18}$$

Different time-stepping schemes, including adaptive stepping and event handling, are compared in [Abhyankar et al. 2017a] for the solution of transient stability problems. Rosenbrock schemes were found to be optimal in terms of speedup and accuracy. In [Abhyankar et al. 2017b] the authors present experiments to achieve real-time simulation speed using PETSc's time-stepping and linear solvers. Results show that real-time simulation speed was achieved on fairly large electrical grid. Similar real-time simulation speed results have also been reported in [Abhyankar and Flueck 2012]. The authors in [Jin et al. 2017] compare parallel transient stability algorithms using MPI and OpenMP. GridPACK [Palmer et al. 2018], a software library for developing parallel power grid applications that uses PETSc's core solvers, is used in this work. Efficient calculation of sensitivities of power grid dynamics trajectories to initial conditions using a discrete adjoint scheme is described in [Zhang et al. 2017].

DMNetwork [Abhyankar et al. 2014; Maldonado et al. 2017] is a relatively new subclass of PETSc's data management class DM that provides functionality for efficiently managing and migrating data and topology for networks and collections of networks. It handles the complex node-edge relationships typically found in unstructured network problems such as electrical grids, water networks, and gas networks; and it provides simple abstractions to query the network topology and associate physics with nodes/edges, acting as a middle layer between PETSc solvers and the application physics. DMNetwork has been used for several network applications, including electrical grids [Abhyankar et al. 2013], water networks with over 1 billion unknowns [Maldonado et al. 2017], and gas networks [Jalving et al. 2017].

### 9.2. Chemical Reactions via TCHEM

TCHEM [Safta et al. 2011] is an open source implementation of many of the reaction network chemistry capabilities of the commercial ChemKin package [chemkin 2017]. TCHEM provides the code for the ODE function evaluation and its Jacobian compu-

tation. It can read ChemKin data files and construct the appropriate needed function evaluations. PETSc provides an easy-to-use interface to TCHEM. In the code listing below we demonstrate how the TCHEM function and Jacobian routines can easily be wrapped and called from PETSc, thus merging TCHEM's chemistry capabilities with PETSc's ODE integrators.

```
FormRHSFunction(TS ts,PetscReal t,Vec X,Vec F,void *ptr){
  User             user = (User)ptr;
  PetscScalar      *f;
  const PetscScalar *x;

  VecGetArrayRead(X,&x); VecGetArray(F,&f);
  PetscMemcpy(user->tchemwork,x,(user->Nspec+1)*sizeof(x[0]));
  user->tchemwork[0] *= user->Tini; /* Dimensionalize */
  TC_getSrc(user->tchemwork,user->Nspec+1,f);TC
  f[0] /= user->Tini;              /* Non-dimensionalize */
  VecRestoreArrayRead(X,&x); VecRestoreArray(F,&f);
}
FormRHSJacobian(TS ts,PetscReal t,Vec X,Mat Amat,Mat Pmat,void *ptr){
  User             user = (User)ptr;
  const PetscScalar *x;
  PetscInt         M = user->Nspec+1,i;

  VecGetArrayRead(X,&x);
  PetscMemcpy(user->tchemwork,x,(user->Nspec+1)*sizeof(x[0]));
  VecRestoreArrayRead(X,&x);
  user->tchemwork[0] *= user->Tini;  /* Dimensionalize temperature */
  TC_getJacTYN(user->tchemwork,user->Nspec,user->Jdense,1);
  for (i=0; i<M; i++) user->Jdense[i + 0*M] /= user->Tini;
  for (i=0; i<M; i++) user->Jdense[0 + i*M] /= user->Tini;
  for (i=0; i<M; i++) user->rows[i] = i;
  MatSetOption(Pmat,MAT_ROW_ORIENTED,PETSC_FALSE);
  MatSetOption(Pmat,MAT_IGNORE_ZERO_ENTRIES,PETSC_TRUE);
  MatZeroEntries(Pmat);
  MatSetValues(Pmat,M,user->rows,M,user->rows,user->Jdense,INSERT_VALUES);
  MatAssemblyBegin(Pmat,MAT_FINAL_ASSEMBLY);
  MatAssemblyEnd(Pmat,MAT_FINAL_ASSEMBLY);
}
```

### REFERENCES

S. Abhyankar, J. Brown, M. Knepley, F. Meier, and B. Smith. 2014. Abstractions for expressing network problems in PETSc. In *SIAMNS'14*.

S. Abhyankar, E. Constantinescu, and A. Flueck. 2017a. Variable-step multi-stage integration methods for fast and accurate power system dynamics simulation. In *IREP'17*.

S. Abhyankar, E. Constantinescu, B. Smith, A. Flueck, and D. Maldonado. 2017b. Parallel dynamics simulation using a Krylov-Schwarz linear solution scheme. *IEEE Transactions on Smart Grid* 8 (2017).

S. Abhyankar and A. Flueck. 2012. Real-time power system dynamics simulation using a parallel block-Jacobi preconditioned Newton-GMRES scheme. In *HiPCNA-PG12*.

Shrirang Abhyankar, Barry Smith, Hong Zhang, and A. Flueck. 2011. Using PETSc to Develop Scalable Applications for Next-Generation Power Grid. In *Proceedings of the 1st International Workshop on High Performance Computing, Networking and Analytics for the Power Grid*. ACM. http://www.mcs.anl.gov/uploads/cels/papers/P1957-0911.pdf

S. Abhyankar, B. F. Smith, and E. Constantinescu. 2013. Evaluation of overlapping restricted additive Schwarz preconditioning for parallel solution of very large power flow problems. In *HiPCNA-PG13*.

U.M. Ascher and L.R. Petzold. 1998. *Computer methods for ordinary differential equations and differential-algebraic equations*. Society for Industrial Mathematics.

U.M. Ascher, S.J. Ruuth, and R.J. Spiteri. 1997. Implicit-explicit Runge-Kutta methods for time-dependent partial differential equations. *Applied Numerical Mathematics* 25 (1997), 151–167.

Derya B. Azyurt and Paul I. Barton. 2005. Cheap Second Order Directional Derivatives of Stiff ODE Embedded Functionals. *SIAM Journal on Scientific Computing* 26, 5 (2005), 1725–1743. DOI:http://dx.doi.org/10.1137/030601582

Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Todd Munson, Karl Rupp, Barry F. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang. 2017. *PETSc Users Manual*. Technical Report ANL-95/11 - Revision 3.8. Argonne National Laboratory.

Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. 1997. Efficient Management of Parallelism in Object Oriented Numerical Software Libraries. In *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen (Eds.). Birkhauser Press, 163–202.

S. Boscarino, L. Pareschi, and G. Russo. 2011. Implicit-Explicit Runge-Kutta schemes for hyperbolic systems and kinetic equations in the diffusion limit. (2011). Arxiv preprint arXiv:1110.4375.

K.E. Brenan, S.L. Campbell, S.L.V. Campbell, and L.R. Petzold. 1996. *Numerical solution of initial-value problems in differential-algebraic equations*. Society for Industrial Mathematics.

J.C. Butcher. 2008. *Numerical Methods for Ordinary Differential Equations* (second ed.). Wiley.

J.C. Butcher, Z. Jackiewicz, and W.M. Wright. 2007. Error propagation of general linear methods for ordinary differential equations. *Journal of Complexity* 23, 4-6 (2007), 560–580. DOI:http://dx.doi.org/10.1016/j.jco.2007.01.009

chemkin 2017. ANSYS Chemkin-Pro Website. http://www.ansys.com/products/fluids/ansys-chemkin-pro. (2017).

E.M. Constantinescu. 2018. Generalizing global error estimation for ordinary differential equations by using coupled time-stepping methods. *J. Comput. Appl. Math.* 332, Supplement C (2018), 140 – 158. DOI:http://dx.doi.org/10.1016/j.cam.2017.05.012

E.M. Constantinescu and A. Sandu. 2010. Extrapolated implicit-explicit time stepping. *SIAM Journal on Scientific Computing* 31, 6 (2010), 4452–4477. DOI:http://dx.doi.org/10.1137/080732833

M. Dowell and P. Jarratt. 1971. A modified regula falsi method for computing the root of an equation. *BIT* 11 (1971), 168–174. Issue 2.

Sergio Galdino. 2011. A family of regula falsi root-finding methods. In *Proceedings of the 2011 World Congress on Engineering and Technology*.

C.W. Gear. 1971. *Numerical initial value problems in ordinary differential equations*. Prentice Hall PTR.

F.X. Giraldo, J.F. Kelly, and E.M. Constantinescu. 2013. Implicit-explicit formulations of a three-dimensional nonhydrostatic unified model of the atmosphere (NUMA). *SIAM Journal on Scientific Computing* 35, 5 (2013), B1162–B1194. DOI:http://dx.doi.org/10.1137/120876034

Andreas Griewank and Andrea Walther. 2000. Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Trans. Math. Software* 26, 1 (2000), 19–45.

E. Hairer, S.P. Nørsett, and G. Wanner. 2008. *Solving Ordinary Differential Equations I: Nonstiff Problems*. Springer. DOI:http://dx.doi.org/10.1007/978-3-540-78862-1

E. Hairer and G. Wanner. 2002. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*. Springer.

M. Heroux, R. Bartlett, V.H.R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, and A. Williams. 2003. *An Overview of Trilinos*. Technical Report SAND2003-2927. Sandia National Laboratories.

A.C. Hindmarsh, P.N. Brown, K.E. Grant, S.L. Lee, R. Serban, D.E. Shumaker, and C.S. Woodward. 2005. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software (TOMS)* 31, 3 (2005), 363–396.

J. Jalving, S. Abhyankar, K. Kim, M. Herald, and V. Zavala. 2017. A graph-based computational framework for simulation and optimization of coupled infrastructure networks. *IET Generation, Transmission, and Distribution* 11 (2017). Issue 12.

K.E. Jansen, C.H. Whiting, and G.M. Hulbert. 2000. A generalized-$\alpha$ method for integrating the filtered Navier–Stokes equations with a stabilized finite element method. *Computer Methods in Applied Mechanics and Engineering* 190, 3 (2000), 305–319.

S. Jin, Z. Huang, R. Diao, D. Wu, and Y. Chen. 2017. Comparative implementation of high performance computing for power system dynamic simulations. *IEEE Transactions on Smart Grid* 8 (2017).

P. Kaps, S.W.H. Poon, and T.D. Bui. 1985. Rosenbrock methods for stiff ODEs: A comparison of Richardson extrapolation and embedding technique. *Computing* 34, 1 (1985), 17–40.

C.A. Kennedy and M.H. Carpenter. 2003. Additive Runge-Kutta schemes for convection-diffusion-reaction equations. *Appl. Numer. Math.* 44, 1-2 (2003), 139–181. DOI:http://dx.doi.org/10.1016/S0168-9274(02)00138-1

D.I. Ketcheson. 2008. Highly Efficient Strong Stability-Preserving Runge–Kutta Methods with Low-Storage Implementations. *SIAM Journal on Scientific Computing* 30, 4 (2008), 2113–2136. DOI:http://dx.doi.org/10.1137/07070485X

D. Maldonado, S. Abhyankar, B. Smith, and H. Zhang. 2017. Scalable multiphysics network simulation using PETSc DMNetwork. http://www.mcs.anl.gov/papers/P7065-0617.pdf. (2017).

Oana Marin, Emil Constantinescu, and Barry Smith. 2017. *PDE constrained optimization, error estimation and control with spectral elements using PETSc and TAO*. Preprint ANL/MCS-P9031-1117. ANL.

MATLAB. 2014. *version 8.1.0 (R2013a)*. The MathWorks Inc., Natick, Massachusetts.

NAG. 2018. The NAG Library, The Numerical Algorithms Group (NAG). Oxford, United Kingdom. (2018). www.nag.com

B. Palmer et al. 2018. GridPACK Web page. https://www.gridpack.org/wiki/index.php/Main_Page. (2018).

L. Pareschi and G. Russo. 2005. Implicit-Explicit Runge-Kutta Schemes and Applications to Hyperbolic Systems with Relaxation. *Journal of Scientific Computing* 25, 1 (2005), 129–155.

L.R. Petzold. 1992. DASSL, Solution of Differential Algebraic Equation. (1992).

J. Rang and L. Angermann. 2005. New Rosenbrock W-methods of order 3 for partial differential algebraic equations of index 1. *BIT Numerical Mathematics* 45, 4 (2005), 761–787.

Cosmin Safta, Habib N Najm, and Omar Knio. 2011. TChem-a software toolkit for the analysis of complex kinetic models. *Sandia Report, SAND2011-3282* (2011).

A. Sandu, J.G. Verwer, J.G. Blom, E.J. Spee, G.R. Carmichael, and F.A. Potra. 1997. Benchmarking stiff ode solvers for atmospheric chemistry problems II: Rosenbrock solvers. *Atmospheric Environment* 31, 20 (1997), 3459–3472.

G. Söderlind. 2003. Digital filters in adaptive time-stepping. *ACM Transactions on Mathematical Software (TOMS)* 29, 1 (2003), 1–26.

G. Söderlind and L. Wang. 2006. Adaptive time-stepping and computational stability. *J. Comput. Appl. Math.* 185, 2 (2006), 225–243.

H. Zhang, S.S. Abhyankar, E.M. Constantinescu, and Mihai Anitescu. 2017. Discrete adjoint sensitivity analysis of hybrid dynamical systems with switching. *IEEE Transactions on Circuits and Systems I: Regular Papers* 64, 5 (2017), 1247–1259. DOI:http://dx.doi.org/10.1109/TCSI.2017.2651683

X. Zhong. 1996. Additive semi-implicit Runge-Kutta methods for computing high speed nonequilibrium reactive flows. *J. Comput. Phys.* 128 (1996), 19–31.