Time Integration
(with hands-on examples using SUNDIALS)

Presented to
ATPESC 2020 Participants

Daniel R. Reynolds
Professor, SMU
SUNDIALS Team Member, ARKODE Lead Developer

Date 08/04/2020

U.S. DEPARTMENT OF ENERGY | Office of Science

ATPESC Numerical Software Track

EXASCALE COMPUTING PROJECT

Argonne NATIONAL LABORATORY | BERKELEY LAB | Sandia National Laboratories | Rensselaer | SMU | Lawrence Livermore National Laboratory

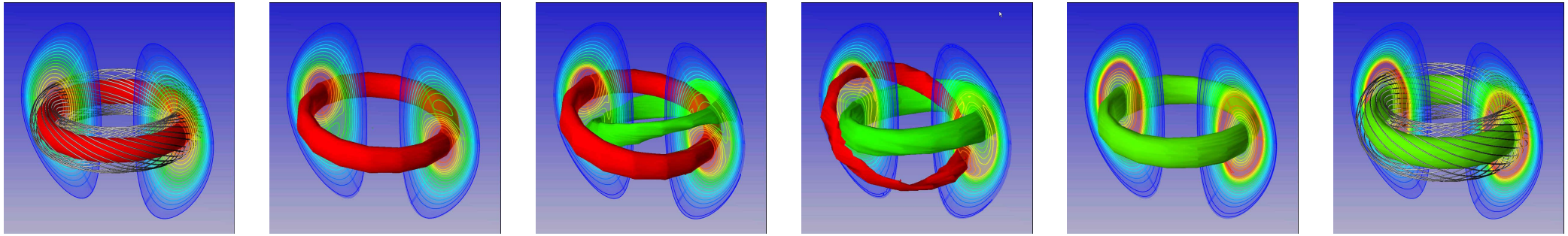
Time integrators in the HPC “landscape”

Most models of physical systems are formulated in terms of the *rate of change* of some variable, e.g. $\frac{\mathrm{d}u}{\mathrm{d}t}$

– Newton’s 2nd law: $\mathbf{f} = m\mathbf{a} \quad \Rightarrow \quad \frac{\mathrm{d}\mathbf{v}}{\mathrm{d}t} = \frac{\mathbf{f}}{m}$

– Chemical rate equations: $A + B \rightarrow P \quad \Rightarrow \quad \frac{\mathrm{d}[P]}{\mathrm{d}t} = k(T)[A][B]$

- Time integrators are used to track changes in solutions as time proceeds, allowing studies of the ‘evolution’ of a model.



”Sawtooth” reconnection in a tokamak (NIMROD)

Time integrators in the HPC "landscape"

Unlike spatial discretization or visualization that live at the bottom/top of the software stack, respectively, time integrators typically live in the "middle." Consider some PDE systems,

$$\partial_t \rho + \nabla \cdot (\rho \mathbf{u}) = 0$$
$$\partial_t \mathbf{u} + \mathbf{u} \cdot \nabla \mathbf{u} + \frac{\nabla p}{\rho} = \mathbf{g}$$
$$\partial_t e + \mathbf{u} \cdot \nabla e + \frac{p}{\rho} \nabla \cdot \mathbf{u} = 0$$

$$\partial_t \mathbf{u} + \mathbf{u} \cdot \nabla \mathbf{u} - \nu \nabla^2 \mathbf{u} = -\nabla \left(\frac{p}{\rho_0} \right) + \mathbf{g}$$
$$\nabla \cdot \mathbf{u} = 0$$

- Using a "method of lines" approach, after spatial discretization, one considers the resulting ODE/DAE system:

$$\dot{y} = f(t, y), \quad y(t_0) = y_0 \qquad F(t, y, \dot{y}) = 0, \quad y(t_0) = y_0, \quad \dot{y}(t_0) = \dot{y}_0$$

  - y contains *all* discretized solution components; f or F encodes the physics & spatial discretization

Time integrator overview

- Let $y_n \approx y(t_n), \quad t_{n+1} = t_n + \Delta t_n$. Then instead of requiring the solution at all time values, we only compute the solution at the finite set of times $\{t_n\}_{n=0}^{N}$.
- A "time marching" scheme computes these time-evolved solutions using a prescribed update formula:

$$y_{n+1} = \Phi(\Delta t_n, y_{n+1}, y_n, \ldots)$$

  e.g., explicit Euler, $y_{n+1} = y_n + \Delta t_n \, f(t_n, y_n)$, and implicit Euler, $y_{n+1} = y_n + \Delta t_n \, f(t_{n+1}, y_{n+1})$

- Time integrator types (explicit, implicit, IMEX):
  - If Φ depends on y_{n+1} then the method is *implicit*, and requires a nonlinear solve of the form

$$\mathbf{F}(y) \equiv y - \Phi(\Delta t_n, y, y_n, \ldots) = \mathbf{0}$$

  - If Φ does not depend on y_{n+1} then the method is *explicit,* in that the updated solution may be explicitly constructed using known data.
  - Implicit-explicit (IMEX) methods arise when only some parts of Φ depend on y_{n+1}.
  - *Multirate* methods use different time step sizes $\Delta t_n \gg \delta t_n$ to evolve separate problem components.

Time integrator overview (continued)

- Time integration methods have multiple mechanisms for achieving increased accuracy:
  - "One-step" methods use multiple internal stages per step [Runge-Kutta, Rosenbrock].
    - More work per-step; amenable to problems with spatial adaptivity & hyperbolic effects.
  - "Multistep" methods retain a longer history of previous solutions [Adams-Bashforth, BDF].
    - Less work per-step; amenable to problems with strong reaction and diffusion effects.
- Linear stability: a method is numerically stable if for a desired Δt_n, floating-point roundoff error stays "controlled" throughout the simulation (vs growing out of control). For a brief refresher, see here.
  - "A-stable": linearly stable no matter the Δt_n -- this is only possible with implicit methods[1].
  - Otherwise, the method has a maximum stable step size Δt_n for any given problem (in PDEs, this is frequently given by the *CFL condition*, wherein $\Delta t_n \propto \Delta x$ or $\Delta t_n \propto \Delta x^2$).
  - *Stability ≠ accuracy* – just because a solution does not blow up, it is not necessarily accurate.

[1]So-called "exponential" methods are explicit and may be A-stable, but require *significantly* more work per-step than traditional explicit methods. I know of no open-source HPC library that provides these.

Choosing between explicit and implicit methods

Explicit Methods	Implicit Methods
+ easy to conceptualize	+ less/nonexistent stability limits
+ easy to code	+ steps over fastest dynamics
+ no algebraic solvers required	− requires algebraic solvers
− stability limits on step sizes	− solvers generally couple all solution unknowns
− tracks fastest dynamics	− increased code complexity

- IMEX: a bit of both – one chooses the splitting to balance 'cheaper' algebraic solvers and stability.
- "Stiffness" helps us choose: "*The stepsize needed to maintain stability of the forward Euler method is much smaller than that required to represent the solution accurately.*" (Ascher and Petzold, 1998)
  - Depends on Jacobian eigenvalues, system dimension, accuracy requirements, length of simulation.
  - For stability, stiff problems generally require implicit or IMEX methods, with robust implicit solvers.
- DAEs nearly always require implicit methods to maintain stability due to the algebraic constraint.
- Multirate methods may be preferable if the 'slow' operator is much more costly than the 'fast'.

Adaptive time-step selection

- *Stability alone should never dictate the time steps used in an application.*
- Given a maximum stable step size, adaptive methods select Δt_n to obtain a desired solution accuracy:
  - At each internal step, computes both the solution and an estimate of the error introduced in that step.
  - If that *local truncation error* is small enough the step is accepted; otherwise a new step size is chosen that should provide sufficient accuracy, and the step is recomputed.
  - Advanced "error controllers" adapt these step sizes to meet a variety of objectives:
    - minimize failed steps
    - maximize step sizes
    - maintain smooth transitions in the step sizes as integration proceeds
- Temporal adaptivity can lead to *much* more efficient (and accurate) results.

"Solving" Initial-Value Problems with SUNDIALS

- SUNDIALS' integrators consider initial-value problems of a variety of types:
  - Standard IVP [CVODE]: $\dot{y}(t) = f(t, y(t)), \quad y(t_0) = y_0$
  - Linearly-implicit, split [ARKODE]: $M\,\dot{y}(t) = f_1(t, y(t)) + f_2(t, y(t)), \quad y(t_0) = y_0$
  - Multirate [ARKODE/MRIStep]: $\dot{y} = f^F(t, y) + f^S(t, y), \quad y(t_0) = y_0$
  - Differential-algebraic form [IDA]: $F(t, y(t), \dot{y}(t)) = 0, \quad y(t_0) = y_0, \quad \dot{y}(t_0) = \dot{y}_0$
- By "integrate" we adapt time steps (and/or method order) to meet user-specified tolerances:

$$\left[\frac{1}{N}\sum_{k=1}^{N}\left(\frac{\text{error}_k}{\text{rtol}\,|y_k| + \text{atol}_k}\right)^2\right]^{1/2} < 1$$

  - $\text{error} \in \mathbb{R}^N$ is the estimated temporal error in the time step
  - $y \in \mathbb{R}^N$ is the previous time-step solution
  - $\text{rtol} \in \mathbb{R}$ encodes the desired relative solution accuracy (number of significant digits)
  - $\text{atol} \in \mathbb{R}^N$ is the 'noise' level for any solution component (protects against $y_k = 0$)

Other DOE Time Integration Packages

- PETSc's *TS* module provides a unified interface for implicit, explicit, & IMEX ODEs and DAEs:

$$F(t, y, \dot{y}) = G(t, y), \quad y(t_0) = y_0$$

  - $F(t,y,y')$ – stiff portion; $G(t,y)$ – nonstiff portion

- Trilinos includes both an older *Rythmos* module for ODEs and DAEs:

$$F(t, y(t), \dot{y}(t)) = 0, \quad y(t_0) = y_0, \quad \dot{y}(t_0) = \dot{y}_0$$

  As well as a newer *Tempus* module for ODEs:

$$\begin{aligned} M(t,y)\,\dot{y}(t) &= G(t,y) + F(t,y), & y(t_0) &= y_0 \\ M\,\ddot{y}(t) + C\,\dot{y}(t) + K\,y(t) + F(t) &= 0, & y(t_0) &= y_0,\ \dot{y}(t_0) = \dot{y}_0 \end{aligned}$$

  - Top: $G(t,y)$ stiff, $F(t,y)$ nonstiff; Bottom: Newmark integrators for second-order ODEs

- All perform temporal adaptivity, and provide a rich set of algebraic solvers for implicit time integration methods.

Implicit methods require a nonlinear solver for $\mathbf{F}(\mathbf{x}) = \mathbf{0}$

The PETSc team is presenting on nonlinear solvers in sessions parallel to this one, so I'll only give a high-level idea, leaving details for them.

Nonlinear solvers must be iterative, since few nonlinear equations admit analytical solutions.

The two largest classes of nonlinear solvers are *fixed-point* vs *Newton-based*.

- FP typically use only $\mathbf{F}$ and converge linearly, but may have a large domain of convergence.
- Newton uses both $\mathbf{F}$ and the Jacobian $J(\mathbf{x}) \equiv \frac{\partial \mathbf{F}(\mathbf{x})}{\partial \mathbf{x}}$ (or an approx.):
  - Each iteration requires a linear solve with the matrix J (see linear solvers talks this AM).
  - Typically converge quadratically (or superlinearly, depending on how well J is solved).
  - For most problems, Newton is algorithmically scalable – as the mesh is refined, the number of iterations remains fixed, so scalability hinges on the linear system solver.

Take Away Messages

- SUNDIALS, PETSc, and Trilinos provide a wide variety of high quality, scalable ODE/DAE integrators and nonlinear solvers.
- PDEs can be converted to ODEs/DAEs via spatial semi-discretization, and then integrated using ODE/DAE libraries.
- Stiffness is an important characteristic of ODEs, and helps dictate which methods are appropriate for any given problem.
- Adaptive time-stepping provides an inexpensive means to combine algorithmic efficiency and solution quality.
- Scalability of implicit and IMEX methods hinges on selection of robust and scalable algebraic solvers; while Newton methods can handle nonlinearities, robustness and scalability of the inner linear solver is critical (and often problem-dependent).

Hands-on lessons

Switch over to web-based hands-on lesson instructions – webpage

Agenda:

1. Explicit time integration (`HandsOn1.exe`)
2. Implicit / IMEX time integration (`HandsOn2.exe`)
3. Preconditioning (`HandsOn3.exe`)

CASC

Center for Applied
Scientific Computing

SMU®

Lawrence Livermore
National Laboratory

Linear Stability – A brief refresher

A fundamental question for any time integration method is how well it handles errors due to floating-point roundoff. To this end, we consider the simple "Dahlquist" test problem:

$$y'(t) = \lambda y(t), \quad y(0) = 1$$

- Here, y corresponds to the normalized floating-point error, and λ to the largest eigenvalue of the Jacobian of a prototypical ODE right-hand side function (assumed to satisfy $\Re(\lambda) < 0$).
- The true solution to this problem is just $y(t) = e^{\lambda t}$, which decays to zero as $t \to \infty$, indicating that roundoff errors should decay as the simulation proceeds.
- The numerical method, on the other hand, computes approximate solutions

$$y_{n+1} = \Phi(\Delta t_n, y_{n+1}, y_n, \ldots)$$

  that may (or may not) similarly satisfy the similar requirement that $y_n \to 0 \text{ as } n \to \infty$.
- Generally, this decay in numerical roundoff error will only occur for specific values of $\Delta t\lambda = z \in \mathbb{C}$. We therefore define the *stability region* for a method as $S = \{z \in \mathbb{C} : \Phi_z(y_n) \to 0 \text{ as } n \to \infty\}$

Linear Stability Example – Explicit Euler

Consider the explicit Euler method: $y_{n+1} = y_n + \Delta t f(t_n, y_n)$

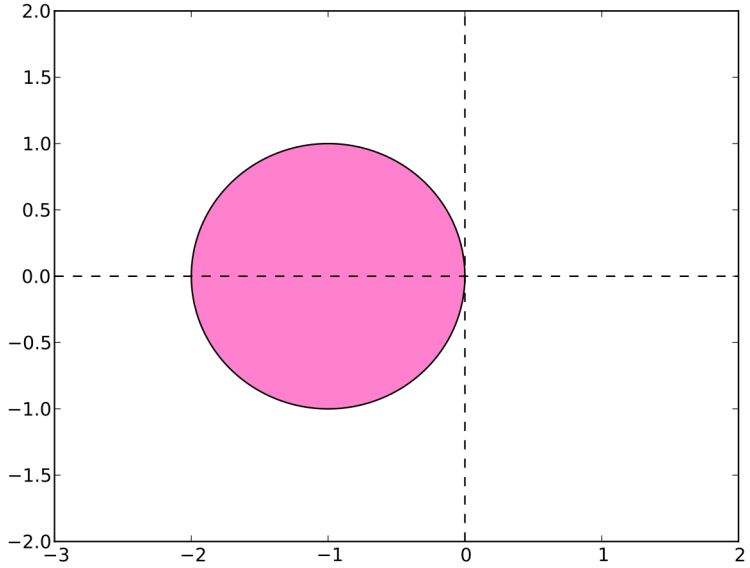
For the Dahlquist test problem, this becomes

$$y_{n+1} = y_n + \Delta t \lambda y_n = (1 + \Delta t \lambda) y_n = (1 + \Delta t \lambda)^2 y_{n-1} = \ldots = (1 + \Delta t \lambda)^{n+1} y_0 = (1 + \Delta t \lambda)^{n+1}$$

which only decays to zero if $|1 + \Delta t \lambda| < 1$.

Hence the explicit Euler linear stability region is

$$S = \{z \in \mathbb{C} \ : \ |1 + z| < 1\}$$



From https://en.wikipedia.org/wiki/Euler_method

Linear Stability Example – Implicit Euler

Consider the implicit Euler method: $y_{n+1} = y_n + \Delta t f(t_{n+1}, y_{n+1})$

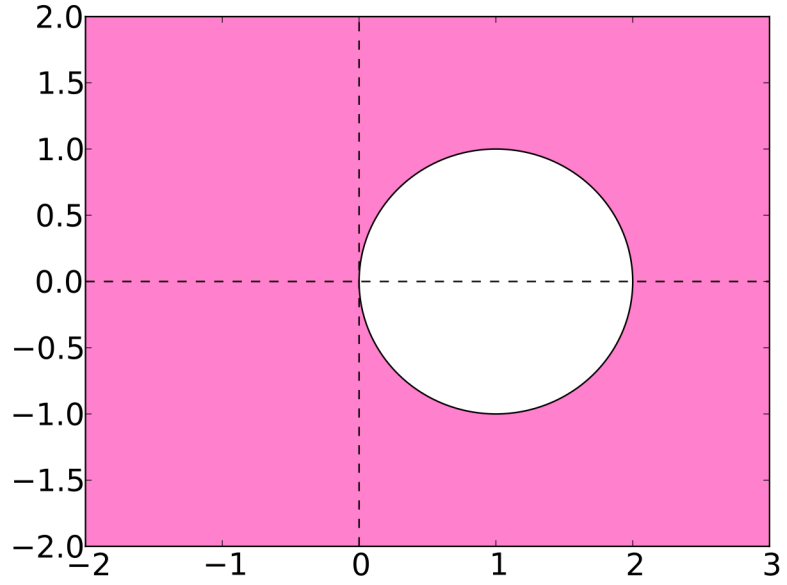
For the Dahlquist test problem, this becomes

$$y_{n+1} = y_n + \Delta t \lambda y_{n+1} \quad \Leftrightarrow \quad y_{n+1} = (1 - \Delta t \lambda)^{-1} y_n = \ldots = (1 - \Delta t \lambda)^{-(n+1)}$$

which only decays to zero if $|1 - \Delta t \lambda| > 1$.

Hence the explicit Euler linear stability region is

$$S = \{z \in \mathbb{C} \ : \ |1 - z| > 1\}$$



From https://en.wikipedia.org/wiki/Backward_Euler_method

Why use an integrator library (instead of "rolling your own")

- Many applications (particularly early in the development process) prefer complete control over their software stack and build system, and therefore choose to implement all numerical methods manually.
- While this can work, the resulting methods may be overly simplistic (e.g., straight out of "Numerical Recipes") or even buggy, and do not benefit from advanced "expert" features.
- Integrator libraries, on the other hand, are typically bug-free, heavily tested, and admit numerous benefits:
  - Time adaptivity (Δt_n) – providing approximate solutions of requested accuracy with minimal work. Libraries request your desired *accuracy,* not step size.
  - Seamless integration with scalable algebraic solver libraries for implicit and IMEX problems.
  - Include many advanced options for later use: temporal root-finding, forward/adjoint sensitivity analysis, globalization options (nonlinear solvers), …
- For more information:
  - SUNDIALS: https://computing.llnl.gov/projects/sundials
  - PETSc: https://www.mcs.anl.gov/petsc/
  - Trilinos: https://trilinos.github.io/