Runge–Kutta Implementation, Adaptive and Multirate Time Integration

Daniel R. Reynolds

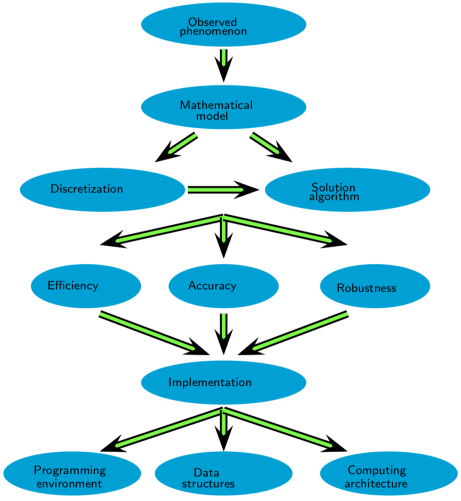
Department of Mathematics, Southern Methodist University,
Department of Mathematics & Statistics, University of Maryland Baltimore County

VIASM Summer School on Advanced Numerical Methods for
Deterministic and Stochastic Differential Equations
9-12 June 2025

Outline

Outline

Context



(courtesy of U. Ascher)

Getting set up with Numpy and Matplotlib

This lecture (and multiple that follow) will have demos and hands-on coding exercises in Python, that will use a variety of well-established Python modules. These are housed on GitHub in the public repository https://github.com/drreynolds/viasm.

To be able to run these codes, we ask you to follow the "Setup" instructions in that repository "README.md" file.

For those unfamiliar with scientific computing in Python, we provide two demonstration scripts to familiarize you with the Numpy and Matplotlib modules, which are used extensively in scientific computing:

- `numpy_demo.py` – basic array creation, indexing, and mathematical operations.
- `matplotlib_demo.py` – basic MATLAB-like plotting of functions and data.

Implementation of time-stepping methods

Time-stepping methods are used to solve ordinary differential equation (ODE) initial value problems (IVPs) of the form

$$y'(t) = f(t,y), \quad a < t < b, \quad y(a) = y_a. \tag{1}$$

We approximate solutions to this at only a discrete set of times,

$$a = t_0 < t_1 < \cdots < t_N = b.$$

Note: we can equivalently numerically evolve backwards in time if given $y(b) = y_b$, in which case $b = t_0 > t_1 > \cdots > t_N = a$, although we will generally consider the forward evolution here.

We denote the spacing between these times as $h_n = t_{n+1} - t_n > 0$, and construct approximate solutions

$$y_{n+1} \approx y(t_{n+1}), \quad n = 0, 1, \ldots, N.$$

Implementation of time-stepping methods

Time marching schemes compute these approximations using a prescribed update formula:

$$y_{n+1} = \Phi\left(t_n, h_n, y_{n+1}, y_n, y_{n-1}, \ldots\right).$$

- A method is defined by specifying Φ.
- If Φ does not depend on y_{n+1} then the method is *explicit*, in that y_{n+1} may be explicitly constructed using known data.
- If Φ depends on y_{n+1} then the method is *implicit*, and requires a nonlinear solve of the form

$$F(y) = y - \Phi\left(t_n, h_n, y, y_n, y_{n-1}, \ldots\right) = 0.$$

The simplest time-stepping method is *forward Euler*, which is an explicit and uses the update formula

$$y_{n+1} = y_n + h_n f(t_n, y_n).$$

A similarly simple method is *backward Euler*, which is an implicit method with update formula

$$y_{n+1} = y_n + h_n f(t_{n+1}, y_{n+1}).$$

Standard IVP solvers

IVP solver software (e.g., in MATLAB, Python, etc.) generally follows a standard interface, which includes the following:

- A function that defines the ODE, e.g., `f(t,y)`.
- An input for the time interval under consideration, e.g., `[t0,tf]`. This is often split into a set of discrete times where the solution is desired, e.g., `[t0, t1, ..., tf]`.
- An input for initial condition, `y0`.
- An input for the requested fixed step size to use, e.g., `h`, or input specifying the desired solution accuracy, e.g., `rtol` and `atol` for relative and absolute tolerances.
- The software then takes time steps, overwriting the internal solution after each step, and only storing the solution at the requested times, which it returns to the user.

This strategy is shown in the `ForwardEuler.py` code, that we'll build on to construct more advanced methods.

Outline

Explicit Runge–Kutta (ERK) methods

As discussed earlier today, an s-stage ERK method for the time step $y_n \to y_{n+1}$ with size h_n has the general form

$$z_i = y_n + h_n \sum_{j=1}^{i-1} a_{ij} f(t_n + c_j h_n, z_j), \quad i = 1, \ldots, s,$$

$$y_{n+1} = y_n + h_n \sum_{j=1}^{s} b_j f(t_n + c_j h_n, z_j),$$

To eliminate redundant calculations (at the expense of some additional storage), we save the RHS function evaluations following each internal stage, leading to the equivalent algorithm

$$\left.\begin{aligned} z_i &= y_n + h_n \sum_{j=1}^{i-1} a_{ij} k_j, \\ k_i &= f(t_n + c_i h_n, z_i) \end{aligned}\right\} \quad i = 1, \ldots, s,$$

$$y_{n+1} = y_n + h_n \sum_{j=1}^{s} b_j k_j.$$

Explicit Runge–Kutta (ERK) methods

Due to it's explicit structure, the ERK method is easy to implement by looping over the stages $i = 1, \ldots, s$, to compute the intermediate values z_i and k_i, then using these to compute the final solution y_{n+1}.

We can look over my implementation in Python: `ERK.py`.

We can use this to run a few experiments in `driver_explicit_fixed.py`.

Explicit method stability limitations

We also earlier discussed the stability limitations of explicit Runge–Kutta methods, which can perform poorly for stiff problems.

We can see this experimentally by running the code `driver_explicit_stability.py`.

It is somewhat beyond the scope of this summer course, but for those of you interested in generating plots of the linear stability regions of Runge–Kutta methods (both explicit and implicit), I have provided a Python function for this in `RK_stability.py`.

Implicit methods require nonlinear solvers

Earlier, we discussed the need for implicit methods when tackling stiff problems (e.g., those resulting from parabolic PDEs), but how do these work?

As I mentioned earlier, implicit methods are characterized by an update formula of the form

$$y_{n+1} = \Phi\left(t_n, h_n, y_{n+1}, y_n, y_{n-1}, \ldots\right),$$

which is equivalent to the root-finding problem

$$0 = F(y) := y - \Phi\left(t_n, h_n, y, y_n, y_{n-1}, \ldots\right).$$

Thus, to compute the next time step y_{n+1}, we must solve the nonlinear equation $F(y_{n+1}) = 0$ for the unknown y_{n+1}. This is typically done using a Newton–Raphson method, which requires an initial guess and an iterative solution procedure.

Nonlinear solvers – the Newton-Raphson method

Definition (Newton-Raphson iteration)

Given a function $F(x) : \mathbb{R}^m \to \mathbb{R}^m$ and an initial guess $x_0 \in \mathbb{R}^m$, the Newton-Raphson iteration for finding a solution $x_* \in \mathbb{R}^m$ such that $F(x_*) = 0$ proceeds as

$$x_{j+1} = x_j - (F_x(x_j))^{-1} F(x_j), \quad j = 0, 1, \ldots$$

Note: although this is *written* using the inverse matrix $(F_x(x_j))^{-1}$, that is **rarely** used in practice. Instead, once breaks the iteration j into two phases:

Step 1: solve $F_x(x_j)\Delta x_j = F(x_j)$ for $\Delta x_j \in \mathbb{R}^m$,

Step 2: update $x_{j+1} = x_j - \Delta x_j$.

Nonlinear solvers – the Newton-Raphson method

Theorem (Newton-Raphson convergence)

Assume:

(a) $F : \mathcal{D} \to \mathbb{R}^m$ *where* $\mathcal{D} \subset \mathbb{R}^m$ *is open.*

(b) $F \in C^1(\mathcal{D})$ *and* F_x *is Lipschitz continuous on* $\mathcal{D}$ *with constant* κ.

(c) $x_* \in \mathcal{D}$, *and* $F_x(x_*)$ *is invertible.*

Then for any $\sigma \in (0,1)$, $\exists \varepsilon > 0$ *and* $c > 0$ *such that if* $\|x_0 - x_*\| < \varepsilon$, *Newton's method will generate* $\lim_{j\to\infty} x_j = x_*$ *(i.e., the method converges),*

$$\|x_{j+1} - x_*\| \le c\|x_j - x_*\|^2, \quad \text{and} \quad \|x_{j+1} - x_*\| < \sigma\|x_j - x_*\|,$$

i.e., with a sufficiently good initial guess, convergence is quadratic, and guaranteed.

Nonlinear solvers – the Newton-Raphson method

Due to the high cost of computing the Jacobian $F_x(x_j)$ and solving the linear systems $F_x(x_j)\Delta x_j = F(x_j)$, practitioners typically employ a *modified Newton-Raphson iteration*:

$$x_{j+1} = x_j - (F_x(x_\ell))^{-1} F(x_j), \quad j = 0, 1, \ldots$$

where $\ell \leq j$ is held fixed for some number of iterations.

- Multiple linear systems are solved using *the same* Jacobian matrix.
- Each time $F_x(x_\ell)$ is recomputed, we factorize the matrix $F_x(x_\ell) = LU$ where L and U are lower and upper triangular, respectively (typically $\mathcal{O}(m^3)$).
- At each iteration j we solve using these factors (typically $\mathcal{O}(m^2)$ each).
- While this saves computational cost, it can slow convergence to just better than linear.

Nonlinear solvers – stopping criteria

Since time steppers only approximate the true IVP solution, we do not iterate Newton's method until $F(x_j) = 0$; we stop when we believe the nonlinear solution is "within tolerance" of the true solution.

If we believe that the IVP method would have solution y_{n+1} that satisfies

$$\frac{|y_{n+1,i} - y_i(t_{n+1})|}{atol_i + rtol|y_{n+1,i}|} < 1, \quad i = 1, \ldots, m,$$

then we stop the Newton-Raphson iteration when

$$\|\Delta x\|_{WRMS} := \left[\frac{1}{m}\sum_{i=1}^{m}\left(\frac{|\Delta x_i|}{atol_i + rtol|x_i|}\right)^2\right]^{1/2} < 1.$$

This algorithm is implemented in the Python module `ImplicitSolver.py`.
We can see its use for the backward Euler method, where

$$F(y_{n+1}) := y_{n+1} - y_n - h_n f(t_{n+1}, y_{n+1}) \quad \Rightarrow \quad F_y(y) := I - h_n f_y(t_{n+1}, y).$$

in `BackwardEuler.py`.

Diagonally Implicit Runge–Kutta (DIRK) methods

Diagonally-implicit Runge–Kutta (DIRK) methods were introduced earlier as well – if we denote the internal stage times as $t_{n,i} = t_n + c_i h_n$, the stages $z_i, i = 1, \ldots, s$ have the form

$$z_i = y_n + h_n \sum_{j=1}^{i} a_{ij} f(t_{n,j}, z_j),$$

$$\Leftrightarrow$$

$$0 = F^{(i)}(z_i) = z_i - h_n a_{ii} f(t_{n,i}, z_i) - y_n - h_n \sum_{j=1}^{i-1} a_{ij} f(t_{n,j}, z_j),$$

$$\Rightarrow$$

$$F_z^{(i)}(z) = I - h_n a_{ii} f_y(t_{n,i}, z).$$

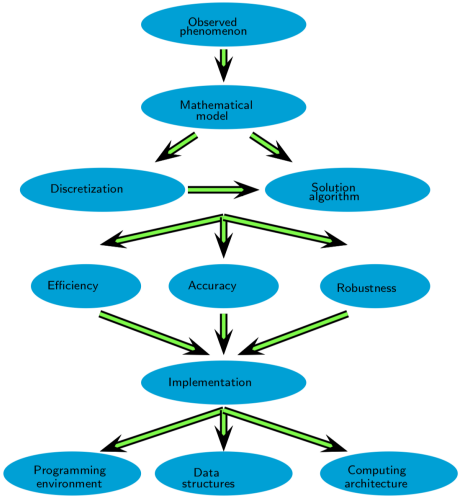
Thus, if we can perform the backward Euler method, then we can perform a DIRK method by solving a sequence of backward Euler-like equations for the stages z_i.

This is implemented in `DIRK.py`, and is tested in `driver_implicit_fixed.py` – we will go through these together if we ahve time.

Also, if we have time I'll explain the approach for solving fully implicit Runge–Kutta methods.

Outline

Context



(courtesy of U. Ascher)

State-of-the-practice

While the methods that we have discussed so far are used widely in practice, a fundamental question arises: what step size h_n should be used?

- In practice, since explicit methods are stability limited, scientists will often select a stepsize h_n that is *just small enough* to ensure linear stability. However, without extensive testing, it is entirely unclear how accurate the resulting solution will be.
- For implicit methods that are frequently not stability limited, the choice of stepsize h_n is even less obvious.
- As a result, practitioners will typically use a fixed stepsize $h_n = h$ for all time steps. This may either be chosen arbitrarily, or through repeated experimentation.
- This may not be ideal, as the solution may evolve at different rates at different times, and thus a fixed stepsize may be too large (leading to poor accuracy) or too small (leading to excessive cost).

Asymptotic convergence

Recall the earlier introduction regarding the difference between *local* error,

$$\ell_n = \|y_{n+1} - y(t_{n+1})\| \leq Ch_n^{p+1}$$

where it is assumed that y_n exactly equals $y(t_n)$, and *global* error,

$$e_n = \|y_{n+1} - y(t_{n+1})\| \leq Ch^p,$$

where $h = \max_j h_j$, that accounts for accumulation of temporal error from one step to the next.

Suppose we have a way to estimate ℓ_n. We could then *adapt* h_n to achieve a variety of objectives:

(a) compute approximate solutions y_n that meet a desired *accuracy*,

(b) perform as few time steps as possible to meet objective (a), and

(c) if a nonlinear solver is required at each step, to increase the *robustness* of the overall scheme.

Temporal adaptivity (local error control)

Temporal adaptivity can be easily incorporated into standard time marching approaches.

Instead of pre-defining the full set of discrete times, $a = t_0 < t_1 < \cdots < t_N = b$, we begin with $t_0 = a$ and an initial (typically small) h_0, and initialize a counter $n = 0$.

Then within the time-marching loop, we:

(i) Compute a "candidate" $y^* \approx y(t_n + h_n)$ and estimate $\ell^* \approx y(t_n + h_n) - y^*$.

(ii) If $\|\ell^*\|$ is "small enough":

  (a) Set $t_{n+1} = t_n + h_n$, $y_{n+1} = y^*$, and $n = n + 1$.

  (b) Estimate a potentially-larger h_n for the next step.

(iii) Else: reduce h_n and recompute the step.

We'll revisit ℓ^* in a moment, but we'll first discuss both "small enough" and how to estimate h_n.

Assessing "small enough"

Recall our discussion of how to stop the Newton-Raphson iteration:

If we believe that the IVP method would have solution y_{n+1} that satisfies

$$\frac{|y_{n+1,i} - y_i(t_{n+1})|}{atol_i + rtol|y_{n+1,i}|} < 1$$

Here:

- $y_{n+1,i} - y_i(t_{n+1})$ corresponds with ℓ_i.
- $rtol \in \mathbb{R}_+$ is a relative solution tolerance (num. sig. digits). If $y_{n,i} \neq 0$ for $i = 1, \ldots, m$,

$$\left|\frac{\ell_i}{y_{n+1,i}}\right| \leq rtol \quad \Leftrightarrow \quad \frac{|\ell_i|}{rtol|y_{n+1,i}|} \leq 1.$$

- $atol \in \mathbb{R}_+^m$ is an absolute solution tolerance (the "noise" level), e.g. for $i = 1, \ldots, m$

$$|\ell_i| \leq atol_i \quad \Leftrightarrow \quad \frac{|\ell_i|}{atol_i} \leq 1.$$

Assessing "small enough"

- If both $|\ell_i| \le rtol|y_{n+1,i}|$ and $|\ell_i| \le atol_i$, then $\ell_i < atol_i + rtol|y_{n+1,i}|$.
- Since we do not know y_{n+1} at the start of the step, it is common to use $rtol|y_{n,i}|$ instead.
- Including an addtional error "bias" $\beta \ge 1$, the *error test* is then

$$\beta|\ell_i| \le (atol_i + rtol|y_{n,i}|), \quad \text{for } i = 1, \ldots, m,$$

$$\Leftrightarrow$$

$$\frac{\beta|\ell_i|}{atol_i + rtol|y_{n,i}|} \le 1, \quad \text{for } i = 1, \ldots, m,$$

$$\Rightarrow$$

$$\|\ell^*\|_{WRMS} := \left[\frac{1}{m}\sum_{i=1}^{m}\left(\ell_i w_{n,i}\right)^2\right]^{1/2} \le 1, \quad \text{where} \quad w_{n,i} = \frac{\beta}{atol_i + rtol|y_{n,i}|}.$$

Note: for brevity we'll omit the $_{WRMS}$ from here onward (though it should be assumed).

Basic stepsize control (the "I-controller")

Whatever the outcome of the error test, we need to select a new step size h for the next step attempt. Assume the numerical method has global order p, i.e.

$$\|\ell_n\| \le C h_n^{p+1}$$

If we have estimated ℓ_n based on a step attempt with h_n, we may "solve" for $C \approx \dfrac{\|\ell_n\|}{h_n^{p+1}}$.

Assuming that C will remain relatively constant from one step attempt to the next, we estimate a desired next step $\tilde{h}$ to exactly attain the target error tolerance:

$$1 = C\tilde{h}^{p+1} = \|\ell_n\| \frac{\tilde{h}^{p+1}}{h_n^{p+1}} \quad \Leftrightarrow \quad \tilde{h} = \frac{h_n}{\|\ell_n\|^{1/(p+1)}}.$$

- This will automatically grow/shrink the step size based on whether $\|\ell_n\|^{1/(p+1)} \le 1$.
- Implementations must guard against an estimated error of $\|\ell_n\| = 0$.
- Some implementations incorporate a safety factor $safe < 1$ here: $\tilde{h} = \dfrac{safe\ h_n}{\|\ell_n\|^{1/(p+1)}}$, instead of including the error bias β.

Embedded Runge–Kutta methods

The most critical component of temporal adaptivity is an efficient and accurate means of estimating ℓ_n.

In the 1960's, Erwin Fehlberg came up with a clever strategy: enhance an RK method with a second set of b coefficients that reuse the stored RHS vectors k_i to compute an *embedded* solution $\tilde{y}_{n+1}$:

$$\left.\begin{aligned} z_i &= y_n + h_n \sum_{j=1}^{i-1} a_{ij} k_j, \\ k_i &= f(t_{n,i}, z_i) \end{aligned}\right\} \quad i = 1, \ldots, s,$$

$$y_{n+1} = y_n + h_n \sum_{j=1}^{s} b_j k_j, \quad \tilde{y}_{n+1} = y_n + h_n \sum_{j=1}^{s} \tilde{b}_j k_j.$$

Armed with two different solutions to the same time step, their difference $y_{n+1} - \tilde{y}_{n+1}$ can approximate

$$\ell_n \;\approx\; y_{n+1} - \tilde{y}_{n+1} \;=\; h_n \sum_{j=1}^{s} \left(b_j - \tilde{b}_j\right) k_j.$$

I-controller implementation

- Typically, $\tilde{y}_{n+1}$ has global order $q = p - 1$, and thus the local error estimate satisfies

$$\begin{aligned}\|\ell_n\| = \|y_{n+1} - \tilde{y}_{n+1}\| &= \|y_{n+1} - y(t_{n+1}) + y(t_{n+1}) - \tilde{y}_{n+1}\| \\ &\le \|y_{n+1} - y(t_{n+1})\| + \|y(t_{n+1}) - \tilde{y}_{n+1}\| \\ &\le C_1 h_n^{p+1} + C_2 h_n^{q+1} \le C h_n^{\min\{p,q\}+1}.\end{aligned}$$

- The files `AdaptiveERK.py` and `AdaptiveDIRK.py` include embedded ERK and DIRK methods, respectively, and employ this I-controller for time step adaptivity. We'll discuss `AdaptiveERK.py` if time permits.

- Tests for both of the adaptive ERK and DIRK solvers are in `driver_explicit_adaptive.py` and `driver_implicit_adaptive.py`.

- `driver_adaptive_timescale.py` shows how adaptive RK methods can *automatically* determine the dynamical time scale of an IVP, eliminating the need for exhaustive testing to determine a fixed time step size.

Additional use of temporal adaptivity – assessing stiffness

- We earlier discussed the idea of *stiffness* as the situation where $y(t)$ has characteristic time scale τ, but the RHS is large, e.g., $\mathrm{Lip}(f) \gg \tau^{-1}$.

- Ascher and Petzold (1998) call a problem stiff if: "The stepsize needed to maintain stability of the forward Euler method is much smaller than that required to represent the solution accurately."

- Stiffness can depend on the eigenvalues of f_y, the dimension of the IVP (m), accuracy requirements $(rtol, atol)$, and even the length of the simulation, $[a, b]$.

- I find that an excellent utility to assess the stiffness of a problem is to run it using both adaptive explicit and adaptive implicit methods. An example is included in the script `driver_adaptive_stability.py`.

Homework – 1

Write a Python file named `ReactionDiffusion.py` to implement the RHS function, RHS Jacobian, and initial condition for the following IVP for $u = u(x,t)$:

$$\begin{aligned} \partial_t u &= \partial_{xx}^2 u + \frac{1}{1+u^2} + \Phi(x,t), \quad (x,t) \in [0,1] \times [0,1], \\ u(t,0) &= u(t,1) = 0, \quad t \in [0,1], \\ u(0,x) &= x(1-x), \end{aligned}$$

where Φ is chosen such that the exact solution of the problem is $u(x,t) = x(1-x)e^t$, the operator ∂_{xx}^2 is discretized using second-order centered finite-differences on a uniform grid with $N_x = 201$ nodes, and where u is stored as a 1D numpy array with N_x entries.

Homework – 2

If you struggle to implement the Jacobian correctly, then if your RHS is named `f`, the following code can be used to provide a decent approximation:

```
def J(t,y):
   nx = len(y)
   f0 = f(t,y)
   Jac = np.zeros((nx,nx))
   for j in range(nx):
      ytemp = y.copy()
      delta = 1e-8 * (abs(y[j]) if abs(y[j]) > 1e-15 else 1.0)
      ytemp[j] += delta
      ftemp = f(t,ytemp)
      Jac[:,j] = (ftemp - f0)/delta
   return Jac
```

Homework – 3

Write Python "driver" files that either

```
import ReactionDiffusion as RD
```

or

```
from ReactionDiffusion import *
```

and that evolve this problem and compute the approximation error using

1. a fixed-step explicit Runge–Kutta method of your choosing – you'll need to experiment to determine a stable time step size to use;
2. a fixed-step diagonally-implicit Runge–Kutta method of your choosing – you may want to experiment to determine an accurate time step size to use;
3. an adaptive explicit Runge–Kutta method of your choosing;
4. an adaptive diagonally-implicit Runge–Kutta method of your choosing.

Outline

Multirate applications

A "multirate" application is characterized as having a subset of components that evolve on a significantly faster/slower time scale than the others.

- If these are sufficiently separated, practitioners will traditionally reformulate the problem analytically, e.g., by approximating the slower process as steady state while evolving the faster one.
- Otherwise, one should include both in the calculation, but how should h_n be selected?
- If h_n is set based on the faster processes, then the problem can be evolved accurately, but some computations will be extraneous (e.g., frequent nearly-zero updates to the slow components).
- If h_n is set based on the slower processes, then there can be considerable error in the faster components.
- Multirate methods are those that use time steps of varying size to evolve distinct components.

Multirate applications

We consider a prototypical multirate IVP as having two coupled evolutionary processes:

$$y'(t) = f^F(t,y) + f^S(t,y), \quad a < t < b, \quad y(a) = y_a.$$

- $f^S(t,y)$ contains the "slow" dynamics, that in isolation would accurately and efficiently be evolved with time step H. This is generally assumed to be *much more costly to evaluate* than $f^F(t,y)$.
- $f^F(t,y)$ contains the "fast" dynamics, that in isolation would accurately and efficiently be evolved with time steps $h \ll H$.
- Either f^F or f^S could be stiff or nonstiff, thereby desiring implicit or explicit treatment. Here, we'll assume that f^S is nonstiff, but in my talk on Thursday I'll discuss implicit and implicit-explicit multirate methods.
- While time scales can evolve dynamically, it is assumed that $h_n < H_n$ in general. Here we'll focus on fixed-step methods, but in my talk on Thursday I'll discuss multirate temporal adaptivity.

Historical "subcycling" techniques

- Practitioners have traditionally evolved multirate IVPs by treating each term separately. Mathematically, this corresponds with a *Lie–Trotter* operator splitting approach: to evolve $y_n \to y_{n+1}$ one takes the steps:

$$\frac{\mathrm{d}}{\mathrm{d}t}y^{\{1\}}(t) = f^{\{1\}}\left(t, y^{\{1\}}\right), \quad t_n < t < t_n + H, \quad y^{\{1\}}(t_n) = y_n,$$
$$\frac{\mathrm{d}}{\mathrm{d}t}y^{\{2\}}(t) = f^{\{2\}}\left(t, y^{\{2\}}\right), \quad t_n < t < t_n + H, \quad y^{\{2\}}(t_n) = y^{\{1\}}(t_{n+1}),$$

  and sets $y_{n+1} = y^{\{2\}}(t_n + H)$.

- Mathematically, the ordering here is unimportant (i.e., $f^{\{1\}}$ could be either f^F or f^S), although in practice some disciplines vigorously debate the merits of "slow-first" versus "fast-first" schemes.

- Each IVP is tackled independently using different "standard" approaches (e.g., a single backward Euler step for f^S, and an adaptive ERK45 for f^F).

Historical "subcycling" techniques

- A competing method that "symmetrizes" Lie–Trotter is the *Strang–Marchuk* splitting approach: to evolve $y_n \to y_{n+1}$ one takes the steps:

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}t}y^{\{1\}}(t) &= f^{\{1\}}\left(t, y^{\{1\}}\right), \quad & t_n < t < t_n + \tfrac{H}{2}, \quad & y^{\{1\}}(t_n) = y_n, \\
\frac{\mathrm{d}}{\mathrm{d}t}y^{\{2\}}(t) &= f^{\{2\}}\left(t, y^{\{2\}}\right), \quad & t_n < t < t_n + H, \quad & y^{\{2\}}(t_n) = y^{\{1\}}(t_n + \tfrac{H}{2}), \\
\frac{\mathrm{d}}{\mathrm{d}t}y^{\{1\}}(t) &= f^{\{1\}}\left(t, y^{\{1\}}\right), \quad & t_n + \tfrac{H}{2} < t < t_n + H, \quad & y^{\{1\}}(t_n + \tfrac{H}{2}) = y^{\{2\}}(t_n + H),
\end{aligned}$$

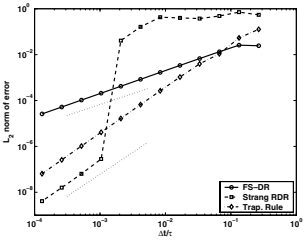
  and sets $y_{n+1} = y^{\{1\}}(t_{n+1})$.

- Again the ordering is unimportant mathematically, but if f^S is stiff then it is typically assigned to $f^{\{2\}}$ to minimize the number of implicit solves.

Shorcomings of historical subcycling approaches

Generally poor accuracy:

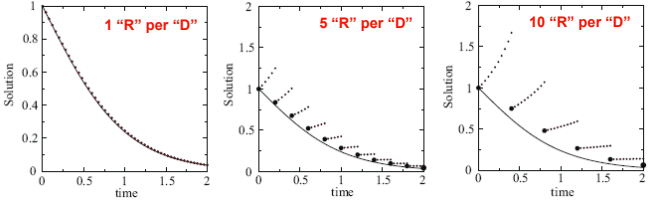
- *No matter the accuracy of each sub-solver*, Lie–Trotter is at best $\mathcal{O}(H)$ and Strang–Marchuk is $\mathcal{O}(H^2)$.
- Extrapolation or deferred correction can improve this but at significant cost.



Convergence of splitting approaches (brusselator) [Ropp & Shadid, 2005].

Poor stability:

- Even "stable" step sizes for each part can result in unstable modes.



Subcycling stability (reaction-diffusion) [Estep et al., 2008].

Multirate infinitesimal (MRI) methods

Modern multirate methods enhance both accuracy and stability by strengthening coupling between the time scales, using a Runge–Kutta-like approach. A single explicit MRI time step proceeds as:

1. Let: $z_1 = y_n$.
2. For each slow stage $z_i,\ i = 2, \ldots, s$:
   a) Define: $r_i(t) = \sum_{j=1}^{i} \gamma_{i,j} \left(\frac{t - t_n}{(c_i - c_{i-1})H} \right) f^S\left(t_{n,j}, z_j\right)$.
   b) Evolve: $v_i'(t) = f^F\left(t, v_i\right) + r_i(t)$, for $t \in [t_{n,i-1}, t_{n,i}],\ v_i(t_{n,i-1}) = z_i$.
   c) Let: $z_i = v_i(t_{n,i})$.
3. Let: $y_{n+1} = z_s$.

- The coefficients $0 = c_1 < c_2 < \ldots < c_s = 1$ give rise to the stage times, $t_{n,i} = t_n + c_i H$.
- The original MIS method [Schlegel et al., 2009] had $\gamma_{i,j}(\theta)$ independent of θ, with coefficients computed from an "outer" ERK method.
- The newer MRI method [Sandu, 2019] had $\gamma_{i,j}(\theta)$ a polynomial in θ.

Multirate method comparison

Multirate method implementations that treat the slow time scale explicitly are provided in the files:

- Lie–Trotter subcycling (slow-fast) is in `LTSubcycling.py`.
- Strang–Marchuk subcycling (fast-slow-fast) is in `SMSubcycling.py`.
- MRI is in `MRI.py`

Each of these can use any of our other codes to evolve the fast time scale (e.g., ERK, AdaptiveDIRK).

In `driver_multirate.py` we compare their convergence on a simple multirate IVP

$$\begin{pmatrix} u \\ v \end{pmatrix}' = \begin{bmatrix} G & e \\ e & -1 \end{bmatrix} \begin{pmatrix} \frac{u^2-r-1}{2u} \\ \frac{v^2-s-2}{2v} \end{pmatrix} + \begin{pmatrix} \frac{r'(t)}{2u} \\ \frac{s'(t)}{2v} \end{pmatrix}$$

where $r(t) = \frac{1}{2}\cos(t)$ and $s(t) = \cos(\omega t)$, and the first row corresponds with f^S while the second row corresponds with f^F.

MRI theory – analyzing order of accuracy

Lemma

M steps of size h for an s-stage Runge–Kutta method may be written as a single step of size mH for a Runge–Kutta method having at most $(s+1)M$ stages. Furthermore, this larger RK method satisfies the same order conditions as the underlying method.

Proof.

Proof idea: consider the 2-stage Heun ERK method for an autonomous IVP: $z_2 = y_n + hf(y_n)$, $y_{n+1} = y_n + \frac{h}{2}f(y_n) + \frac{h}{2}f(z_2)$. Three Heun steps corresponds with one step of size $H = 3h$ for:

$$
\begin{aligned}
z_2 &= y_n + \tfrac{H}{3}f(y_n) \\
z_3 &= y_n + \tfrac{H}{6}f(y_n) + \tfrac{H}{6}f(z_2) \\
z_4 &= y_n + \tfrac{H}{6}f(y_n) + \tfrac{H}{6}f(z_2) + \tfrac{H}{3}f(z_3) \\
z_5 &= y_n + \tfrac{H}{6}f(y_n) + \tfrac{H}{6}f(z_2) + \tfrac{H}{6}f(z_3) + \tfrac{H}{6}f(z_4) \\
z_6 &= y_n + \tfrac{H}{6}f(y_n) + \tfrac{H}{6}f(z_2) + \tfrac{H}{6}f(z_3) + \tfrac{H}{6}f(z_4) + \tfrac{H}{3}f(z_5) \\
y_{n+1} &= y_n + \tfrac{H}{6}f(y_n) + \tfrac{H}{6}f(z_2) + \tfrac{H}{6}f(z_3) + \tfrac{H}{6}f(z_4) + \tfrac{H}{6}f(z_5) + \tfrac{H}{6}f(z_6)
\end{aligned}
$$

□

MRI theory – analyzing order of accuracy

Thus if we assume that the inner IVP

$$v_i'(t) = f^F\left(t, v_i\right) + r_i(t), \quad t \in [t_{n,i-1}, t_{n,i}], \quad v(t_{n,i-1}) = z_i,$$

is solved using M_i steps of an inner Runge–Kutta method, its calculations may be encoded as a single step of size $t_{n,i} - t_{n,i-1} = \Delta c_i H$ via an inner Butcher table $\{A^i, b^i, c^i\}$, with s^i stages, i.e.,

$$\nu_{i,j} = z_i + \Delta c_i H \sum_{k=1}^{j-1} a^i_{j,k} \left(f^F\left(t_{n,i,k}, \nu_{i,k}\right) + r_i(t_{n,i,k}) \right), \quad j = 1, \ldots, s^i$$

$$z_{i+1} = z_i + \Delta c_i H \sum_{j=1}^{s^i} b^i_j \left(f^F\left(t_{n,i,j}, \nu_{i,j}\right) + r_i(t_{n,i,j}) \right),$$

where $\nu_{i,j}$ is the stage j of the inner method, and $t_{n,i,k} = t_{n,i-1} + c^i_k \Delta c_i H$.

MRI theory – analyzing order of accuracy

If we insert the definition of $r_i(t)$, the explicit MRI method may be written as:

1. Let $z_1 = y_n$.
2. For each slow stage $z_i, i = 2, \ldots, s$, compute:

$$\nu_{i,j} = z_i + \Delta c_i H \sum_{k=1}^{j-1} a^i_{j,k} \left(f^F\left(t_{n,i,k}, \nu_{i,k}\right) + \sum_{\ell=1}^{i} \gamma_{i,\ell} \left(\frac{t_{n,i,k} - t_n}{\Delta c_i H} \right) f^S(t_{n,\ell}, z_\ell) \right), \quad j = 1, \ldots, s^i$$

$$z_{i+1} = z_i + \Delta c_i H \sum_{j=1}^{s^i} b^i_j \left(f^F\left(t_{n,i,j}, \nu_{i,j}\right) + \sum_{\ell=1}^{i} \gamma_{i,\ell} \left(\frac{t_{n,i,j} - t_n}{\Delta c_i H} \right) f^S(t_{n,\ell}, z_\ell) \right),$$

3. Let $y_{n+1} = z_s$

Note that although this looks somewhat complicated, each fast stage $\nu_{i,j}$ and each slow stage z_i are comprised of *linear combinations* of f^F evaluations over $\nu_{i,k}$ and f^S evaluations over z_ℓ.

MRI theory – analyzing order of accuracy

Thus, MRI methods are equivalent to *Generalized-structure Additive Runge–Kutta* (GARK) methods [Sandu & Günther, 2015] with a 2-way partitioning:

$$z_i^{\{S\}} = y_n + H\sum_{j=1}^{s^S} a_{i,j}^{\{S,S\}} f^S(z_j^{\{S\}}) + H\sum_{j=1}^{s^F} a_{i,j}^{\{S,F\}} f^F(z_j^{\{F\}}), \quad i = 1,\ldots,s^S$$

$$z_i^{\{F\}} = y_n + H\sum_{j=1}^{s^S} a_{i,j}^{\{F,S\}} f^S(z_j^{\{S\}}) + H\sum_{j=1}^{s^F} a_{i,j}^{\{F,F\}} f^F(z_j^{\{F\}}), \quad i = 1,\ldots,s^F$$

$$y_{n+1} = y_n + H\sum_{i=1}^{s^S} b_i^{\{S\}} f^S(z_i^{\{S\}}) + H\sum_{i=1}^{s^F} b_i^{\{F\}} f^F(z_i^{\{F\}}),$$

that corresponds with a generalized Butcher tableau $\begin{array}{cc} A^{\{S,S\}} & A^{\{S,F\}} \\ A^{\{F,S\}} & A^{\{F,F\}} \\ \hline b^{\{S\}} & b^{\{F\}} \end{array}$.

MRI theory – analyzing order of accuracy

Order conditions for GARK methods through $\mathcal{O}\left(H^4\right)$ are included in [Sandu & Günther, 2015], thus to analyze a given family of MRI methods, one may:

1. Assume that the Runge–Kutta method used for the "fast" evolution has sufficiently high order of accuracy (and thus its table satisfies standard RK order conditions).

2. Determine the GARK coefficients $A^{\{S,S\}}$, $A^{\{S,F\}}$, $A^{\{F,S\}}$, $A^{\{F,F\}}$, $b^{\{S\}}$, and $b^{\{F\}}$ that correspond with those in the MRI method under consideration.

3. Test the GARK order conditions to determine which are satisfied/violated, e.g. for $\mathcal{O}\left(H^2\right)$:

$$
\begin{aligned}
b^{\{S\}} \cdot \mathbf{1}^{\{s^S\}} &= 1, \\
b^{\{S\}} \cdot c^{\{S,S\}} &= \tfrac{1}{2}, \\
b^{\{S\}} \cdot c^{\{S,F\}} &= \tfrac{1}{2},
\end{aligned}
\qquad\qquad
\begin{aligned}
b^{\{F\}} \cdot \mathbf{1}^{\{s^F\}} &= 1, \\
b^{\{F\}} \cdot c^{\{F,S\}} &= \tfrac{1}{2}, \\
b^{\{F\}} \cdot c^{\{F,F\}} &= \tfrac{1}{2}.
\end{aligned}
$$

Advertisement – workshop talk

My talk on Friday is titled: "Flexible Time Integration Methods for Multiphysics Models"

Multiphysics models couple two or more physical processes together in a single simulation, e.g., stiff, nonstiff, and multirate. Understandably, such simulations prove challenging for "monolithic" time integration methods that treat all processes using a uniform approach.

I'll discuss recent work on time integration methods that extend what I've presented here:

- mixed implicit-explicit Runge–Kutta and multirate methods
- adaptive control over both H_n and h_n in MRI methods.

Homework

Update your `ReactionDiffusion.py` to add RHS functions that split the problem as:

$$f^F = \partial^2_{xx} u,$$
$$f^S = \frac{1}{1+u^2} + \Phi(x,t).$$

Write Python "driver" files that evolve your original problem and compute the solution error using:

1. a Lie–Trotter subcycling method with the "ERK1" Butcher table (i.e., forward Euler) for f^S, and an adaptive explicit solver for f^F,
2. a Strang–Marchuk subcycling method with a second-order ERK Butcher table for f^S, and an adaptive explicit solver for f^F,
3. an $\mathcal{O}\left(H^3\right)$ or $\mathcal{O}\left(H^4\right)$ MRI method for f^S, and an adaptive explicit solver for f^F.

For each of these you will need to experiment with selecting the appropriate slow step size, H.