Numerical Solvers for Astrochemistry Networks (Chapter 3 from Bovino & Grassi, 2023)

Daniel R. Reynolds

Department of Mathematics & Statistics, University of Maryland Baltimore County

A decade of KROME: towards the next generation of astrochemical modelling
13-17 July 2026

Outline

Outline

Example codes

All codes related to this lecture are located in https://github.com/drreynolds/acbook-time-integration:

```
git clone https://github.com/drreynolds/acbook-time-integration.git
```

This depends on a few Python packages, which can be installed via `pip`:

```
pip install -r requirements.txt
```

or `conda`:

```
conda install --file requirements.txt
```

Initial-value problems and differential-algebraic equations

Models for time-evolving chemical networks typically take one of two basic forms:

- First-order systems of initial-value problems (IVPs):

$$\dot{\mathbf{y}}(t) = \mathbf{f}(t, \mathbf{y}), \quad a < t < b, \quad \mathbf{y}(a) = \mathbf{y}_a,$$

- First-order systems of differential-algebraic equations (DAEs):

$$\mathbf{0} = \mathbf{F}(t, \mathbf{y}, \dot{\mathbf{y}}), \quad a < t < b, \quad \mathbf{y}(a) = \mathbf{y}_a, \quad \dot{\mathbf{y}}(a) = \dot{\mathbf{y}}_a.$$

Example

Consider the simple chemical network

$$\mathrm{A} + \mathrm{D} \underset{k_2}{\overset{k_1}{\rightleftharpoons}} \mathrm{B}$$

$$\mathrm{B} \underset{k_4}{\overset{k_3}{\rightleftharpoons}} \mathrm{C}.$$

This corresponds with the IVP system

$$\begin{aligned}
\frac{\mathrm{d}n(\mathrm{A})(t)}{\mathrm{d}t} &= k_2\, n(\mathrm{B}) - k_1\, n(\mathrm{A})\, n(\mathrm{D}),\\
\frac{\mathrm{d}n(\mathrm{B})(t)}{\mathrm{d}t} &= k_1\, n(\mathrm{A})\, n(\mathrm{D}) - (k_2 + k_3)\, n(\mathrm{B}) + k_4\, n(\mathrm{C}),\\
\frac{\mathrm{d}n(\mathrm{C})(t)}{\mathrm{d}t} &= k_3\, n(\mathrm{B}) - k_4\, n(\mathrm{C}),\\
\frac{\mathrm{d}n(\mathrm{D})(t)}{\mathrm{d}t} &= k_2\, n(\mathrm{B}) - k_1\, n(\mathrm{A})\, n(\mathrm{D}).
\end{aligned}$$

Example (continued)

The steady-state of this system corresponds with solutions of a system of nonlinear algebraic equations,

$$\begin{aligned}
0 &= k_2\, n(\mathrm{B}) - k_1\, n(\mathrm{A})\, n(\mathrm{D}),\\
0 &= k_1\, n(\mathrm{A})\, n(\mathrm{D}) - (k_2 + k_3)\, n(\mathrm{B}) + k_4\, n(\mathrm{C}),\\
0 &= k_3\, n(\mathrm{B}) - k_4\, n(\mathrm{C}),\\
0 &= k_2\, n(\mathrm{B}) - k_1\, n(\mathrm{A})\, n(\mathrm{D}).
\end{aligned}$$

However, if only k_3 and k_4 from are much larger than k_1 and k_2, then on the time scale of evolution for $n(\mathrm{A})$ and $n(\mathrm{B})$, the third equation can be well approximated as being in steady state, i.e., one could instead consider the DAE model

$$\begin{aligned}
\frac{\mathrm{d}n(\mathrm{A})(t)}{\mathrm{d}t} &= k_2\, n(\mathrm{B}) - k_1\, n(\mathrm{A})\, n(\mathrm{D})\\
\frac{\mathrm{d}n(\mathrm{B})(t)}{\mathrm{d}t} &= k_1\, n(\mathrm{A})\, n(\mathrm{D}) - (k_2 + k_3)\, n(\mathrm{B}) + k_4\, n(\mathrm{C}),\\
0 &= k_3\, n(\mathrm{B}) - k_4\, n(\mathrm{C}),\\
\frac{\mathrm{d}n(\mathrm{D})(t)}{\mathrm{d}t} &= k_2\, n(\mathrm{B}) - k_1\, n(\mathrm{A})\, n(\mathrm{D}).
\end{aligned}$$

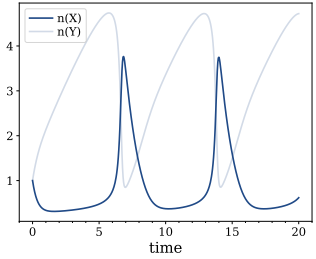
Recurring Example – Brusselator [Prigogine, 1980]

We'll revisit two examples in this lecture and codes – the first is the "Brusselator" system modeling an autocatalytic reaction,

$$\begin{aligned} \mathrm{A} &\xrightarrow{k_1} \mathrm{X} \\ \mathrm{B} + \mathrm{X} &\xrightarrow{k_2} \mathrm{Y} + \mathrm{D} \\ 2\mathrm{X} + \mathrm{Y} &\xrightarrow{k_3} 3\mathrm{X} \\ \mathrm{X} &\xrightarrow{k_4} \mathrm{E}. \end{aligned}$$

If A, B are in excess and can be modeled as constant, X, Y satisfy

$$\begin{aligned} \frac{\mathrm{d}n(\mathrm{X})(t)}{\mathrm{d}t} &= k_1 n(\mathrm{A}) - k_2 n(\mathrm{B}) n(\mathrm{X}) + k_3 n(\mathrm{X})^2 n(\mathrm{Y}) - k_4 n(\mathrm{X}), \\ \frac{\mathrm{d}n(\mathrm{Y})(t)}{\mathrm{d}t} &= k_2 n(\mathrm{B}) n(\mathrm{X}) - k_3 n(\mathrm{X})^2 n(\mathrm{Y}). \end{aligned}$$



Brusselator solution for $k_1 = k_2 = k_3 = k_4 = 1$, $n(\mathrm{A}) = 1$ and $n(\mathrm{B}) = 3$, on $t \in [0, 20]$, with initial conditions $n(\mathrm{X})(0) = n(\mathrm{Y})(0) = 1$.

PYTHON codes for this IVP, including $\mathbf{f}(t, \mathbf{y})$ and its Jacobian are in example1.py.

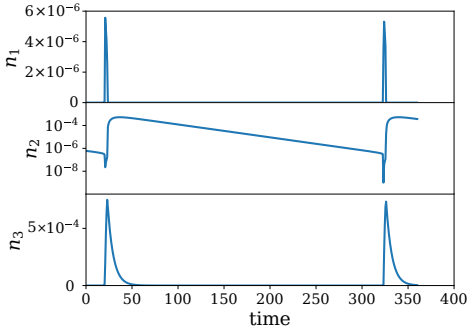
Recurring Example – Oregonator [Field et al., 1972]

The second is the "Oregonator" system, modeling the Belousov–Zhabotinskii chemical reaction:

$$\frac{\mathrm{d}n_1(t)}{\mathrm{d}t} = -k_1 n_1 n_2 + k_2 n_1 - k_3 n_1^2 + k_4 n_2,$$
$$\frac{\mathrm{d}n_2(t)}{\mathrm{d}t} = -k_1 n_1 n_2 - k_4 n_2 + k_5 n_3,$$
$$\frac{\mathrm{d}n_3(t)}{\mathrm{d}t} = k_2 n_1 - k_5 n_3,$$

with rate coefficients

$$k_1 = 2.57555802 \times 10^{8}, \qquad k_2 = 7.72667406 \times 10^{1},$$
$$k_3 = 1.28777901 \times 10^{7}, \qquad k_4 = 1.29421790 \times 10^{-2},$$
$$k_5 = 1.60972376 \times 10^{-1},$$



and initial conditions $n_1(0) = 5.025 \times 10^{-11}$, $n_2(0) = 6 \times 10^{-7}$, and $n_3(0) = 7.236 \times 10^{-8}$.

PYTHON codes for this IVP, including $\mathbf{f}(t, \mathbf{y})$ and its Jacobian are in example2.py.

Outline

Time-stepping methods

Instead of computing the solution at all $t \in [a, b]$, IVP solvers compute the solution at a discrete set of times $a = t_0 < t_1 < \cdots < t_f = b$. We denote $\mathbf{y}_n \approx \mathbf{y}(t_n)$ and $h_n := t_{n+1} - t_n$.

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

$$\mathbf{y}_{n+1} = \Phi\left(t_n, h_n, \mathbf{y}_{n+1}, \mathbf{y}_n, \mathbf{y}_{n-1}, \ldots\right),$$

  e.g., *fwd* and *bwd Euler*, $\mathbf{y}_{n+1} = \mathbf{y}_n + h_n \mathbf{f}(t_n, \mathbf{y}_n)$, and $\mathbf{y}_{n+1} = \mathbf{y}_n + h_n \mathbf{f}(t_{n+1}, \mathbf{y}_{n+1})$.

- Time integrator types:
  - If Φ depends on $\mathbf{y}_{n+1}$ then the method is *implicit*, and requires a nonlinear solve of the form

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

  - Else the method is *explicit*, in that $\mathbf{y}_{n+1}$ may be explicitly constructed using known data.
  - *Implicit-explicit (ImEx)* methods arise when only some parts of Φ depend on $\mathbf{y}_{n+1}$.
  - *Multirate* methods use different step sizes $H_n \gg h_n$ for separate problem components.

Linear stability

The numerical stability of an IVP solver is typically analyzed using the *Dahlquist test problem*

$$\dot{y}(t) = \lambda y(t), \quad y(0) = 1, \quad t \geq 0. \tag{1}$$

- One generally considers $\lambda \in \mathbb{C}$ to be an eigenvalue of the Jacobian matrix, $J_{i,j}(t, \mathbf{y}) := \frac{\partial f_i(t,\mathbf{y})}{\partial y_j}$.
- *Stable* IVPs require $\Re(\lambda) < 0$, i.e., the solution to (1) satisfies $y(t) \to 0$ as $t \to \infty$.

- Plugging (1) into forward Euler we obtain

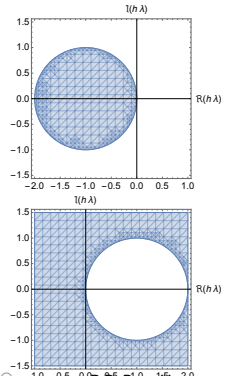
$$y_{n+1} = y_n + h\lambda y_n = (1 + h\lambda)y_n = (1 + h\lambda)^{n+1} y_0,$$

and thus $y_n \to 0$ as $n \to \infty$ only when $|1 + h\lambda| < 1$.

- Plugging (1) into backward Euler similarly gives

$$y_{n+1} = y_n + h\lambda y_{n+1} \quad \Leftrightarrow \quad y_{n+1} = (1 - h\lambda)^{-1} y_n,$$

and thus $y_n \to 0$ as $n \to \infty$ only when $|1 - h\lambda| > 1$.



Linear stability example

In the linear_stability_test.py PYTHON code, we test both forward and backward Euler on the Dahlquist problem with $\lambda = -100$ over the time interval $0 < t < 0.4$, using step sizes of $h = \{0.005, 0.01, 0.02, 0.04\}$.

$$y(t) = e^{-100t} \quad \Rightarrow \quad y(0.4) \approx 4.2 \times 10^{-18}.$$

	Approximation at $t = 0.4$			
Method	$h = 0.005$	$h = 0.01$	$h = 0.02$	$h = 0.04$
Forward Euler	4.3e-18	4.3e-18	1.0	5.9e4
Backward Euler	8.2e-15	9.1e-13	2.9e-10	1.0e-7

Backward Euler keeps floating-point errors in check, while forward Euler only does so for the two smallest step sizes, and $h = 0.02$ lies on the border of its linear stability region for this problem.

Stiffness and time scale estimation

- There are many problems where a maximally stable step size is *completely inaccurate*, and others where a stable step provides more accurate solutions than desired.
- The question of stability thus differs from the question of accuracy, although *both* are critically important in the decision over appropriate step sizes h to use in any given calculation.
- The "battle" between these considerations is related to a problem property called *stiffness*. Quoting from Ascher & Petzold [1998]:

  > *An IVP is stiff in some interval $[a, b]$ if the step size needed to maintain stability of the forward Euler method is much smaller than the step size required to represent the solution accurately.*

Although its definition is somewhat subjective, "stiffness" arises from a variety of factors:

(a) the eigenvalues λ_i of the Jacobian matrix J,

(b) the desired solution accuracy,

(c) the length of the time integration interval,

Item (a) is particularly relevant in astrochemistry, since some chemical reactions occur much more rapidly than others, thereby giving rise to widely varying eigenvalues $|\lambda_i|$.

Stiffness and time scale estimation (continued)

Revisiting our Brusselator and Oregonator examples, at their initial conditions they have Jacobians

$$J_{\rm B} = \begin{bmatrix} -2 & 1 \\ 1 & -1 \end{bmatrix} \quad \text{and} \quad J_{\rm O} \approx \begin{bmatrix} -77.3 & 0 & 0 \\ -155 & -0.0259 & 0.161 \\ 77.3 & 0 & -0.161 \end{bmatrix},$$

with corresponding eigenvalues $\lambda(J_{\rm B}) \approx \{-2.62, -0.382\}$ and $\lambda(J_{\rm O}) \approx \{-77.3, -0.161, -0.0259\}$.

- At their initial conditions, the Oregonator's eigenvalues vary much more than the Brusselator, indicating a higher degree of stiffness.
- The `maximum_stiffness` utility functions compute $|\lambda_{max}|/|\lambda_{min}|$ throughout each problem's time interval, with Brusselator maximally ~ 65, and Oregonator $\sim 5.3 \times 10^6$.

This gives rise to a fundamental question: when some reactions occur on faster time scales than others (i.e., $|\lambda_1| \gg |\lambda_2|$), which species need to be resolved accurately versus stably?

- If the "fast" reactions must be accurate, then an explicit method like forward Euler can be used, where h is selected such that the accuracy of λ_1 is prioritized.
- If only the "slow" reactions need to be accurate, with the "fast" reactions stable, then h can be selected to track λ_2; this would require an implicit method like backward Euler.

Outline

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).

Standard IVP solvers

IVP solver software (e.g., PYTHON, MATLAB, JULIA, 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., $[t_0, t_f]$. This is often split into a set of discrete times where the solution is desired, e.g., $[t_0, t_1, \ldots, t_f]$.
- An input for initial condition, `y0`.
- An input for 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.

Local Error Control

Many time integration methods produce not only $\mathbf{y}_{n+1}$, but also an estimate of the local error

$$\ell_n \approx \|\mathbf{y}_{n+1} - \mathbf{y}(t_{n+1})\| \le C_1 h_n^{p+1},$$

where p is the global order of the method, and C_1 is a constant that depends on the problem and the method.

With this estimate, adaptive methods pick h_n to ensure that each solution component $y_{n+1,i}$ satisfies

$$\frac{|\ell_{n,i}|}{atol_i + rtol|y_{n+1,i}|} < 1 \quad \Leftrightarrow \quad |\ell_{n,i}| < atol_i + rtol|y_{n+1,i}|$$

which attempts to encode both relative and absolute solution tolerances, $rtol$ and $atol$,

$$\left|\frac{\ell_{n,i}}{y_{n+1,i}}\right| \le rtol \quad \text{and} \quad |\ell_{n,i}| \le atol_i.$$

This is encoded in the *weighted root-mean-square (WRMS) norm*

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

Local error control (continued)

- There exist numerous "controller" algorithms for actually performing this adaptivity (e.g., I, PI, PID, ...)
- The controller itself intimately depends on the method order p, with higher-order methods generally resulting in larger h_n that satisfy the error tolerance.
- These algorithms only control the estimated local error within each time step. Some applications are more/less sensitive in how local errors accumulate into the global error, so no automated method can guarantee that the global error is within the desired tolerance. Thus, practitioners frequently tighten the tolerances to be slightly stricter than the desired global accuracy.
- I find that adaptive methods can provide a useful utility to assess the stiffness of a problem. By solving the same problem twice, using explicit and implicit methods with the same p, one can compare the number of steps required.

Outline

Implicit methods require nonlinear solvers

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

$$\mathbf{y}_{n+1} = \Phi\left(t_n, h_n, \mathbf{y}_{n+1}, \mathbf{y}_n, \mathbf{y}_{n-1}, \ldots\right),$$

which is equivalent to the root-finding problem

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

Thus, to compute the next time step $\mathbf{y}_{n+1}$, we must solve the nonlinear equation $\mathbf{F}(\mathbf{y}_{n+1}) = \mathbf{0}$ for the unknown $\mathbf{y}_{n+1}$. 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 $\mathbf{F}(\mathbf{y}) : \mathbb{R}^m \to \mathbb{R}^m$ and an initial guess $\mathbf{y}^{(0)} \in \mathbb{R}^m$, the *Newton-Raphson iteration* for finding a vector $\mathbf{y}^{(*)} \in \mathbb{R}^m$ such that $\mathbf{F}(\mathbf{y}^{(*)}) = \mathbf{0}$ proceeds as

$$\mathbf{y}^{(k+1)} = \mathbf{y}^{(k)} - \left(J_F(\mathbf{y}^{(k)})\right)^{-1} \mathbf{F}(\mathbf{y}^{(k)}), \quad k = 0, 1, \ldots,$$

where $J_F := \partial \mathbf{F}(\mathbf{y}) / \partial \mathbf{y}$ is the Jacobian of $\mathbf{F}(\mathbf{y})$ with respect to $\mathbf{y}$. Note: although the iteration is *written* using the inverse matrix J_F^{-1}, this is not used in practice. Instead, once breaks the iteration into two phases:

Step 1: solve $\left(J_F(\mathbf{y}^{(k)})\right) \Delta\mathbf{y}^{(k)} = \mathbf{F}(\mathbf{y}^{(k)})$ for $\Delta\mathbf{y}^{(k)} \in \mathbb{R}^m$,

Step 2: update $\mathbf{y}^{(k+1)} = \mathbf{y}^{(k)} - \Delta\mathbf{y}^{(k)}$.

Nonlinear solvers – the Newton-Raphson method

- When provided with a good initial guess, $\mathbf{y}^{(0)}$, Newton-Raphson exhibits *rapid* convergence.
- However, at each iteration it requires (a) construction of J_F, and (b) solution of a potentially large linear system of equations.
- If J_F is dense and $\mathbf{y} \in \mathbb{R}^m$, then its construction costs $\mathcal{O}\left(m^2\right)$ work and its solution $\mathcal{O}\left(m^3\right)$ work.
- Thus, practitioners frequently employ the *modified* Newton iteration,

$$\mathbf{y}^{(k+1)} = \mathbf{y}^{(k)} - \left(J_F(\mathbf{y}^{(j)})\right)^{-1} \mathbf{F}(\mathbf{y}^{(k)}), \quad k = 0, 1, \ldots,$$

  where $j \leq k$ is held fixed for some number of iterations While reducing construction/factorization costs, this can slow convergence, with this slowdown more pronounced the longer J_F is lagged.
- To construct J_F, practitioners increasingly employ computer algebra systems such as Sympy or Mathematica to analytically determine formulas for each entry of the $m \times m$ Jacobian matrix.
- We'll discuss techniques for linear solvers in a moment.

Nonlinear solvers – stopping criteria

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

When coupled with time adaptive methods, where we only require each time step to satisfy

$$\|\mathbf{y}_{n+1} - \mathbf{y}(t_{n+1})\|_{WRMS} \approx \|\ell_n\|_{WRMS} < 1,$$

then it is typical to we stop the Newton-Raphson iteration when

$$\|\Delta\mathbf{y}^{(k)}\|_{WRMS} < 1.$$

The file implicit_solver.py provides a CPU-based Newton-Raphson and modified Newton solver.

We can see its use for the backward Euler method, where

$$\mathbf{F}(\mathbf{y}_{n+1}) := \mathbf{y}_{n+1} - \mathbf{y}_n - h_n\mathbf{f}(t_{n+1}, \mathbf{y}_{n+1}) \quad \Rightarrow \quad J_F(\mathbf{y}) := I - h_n\partial\mathbf{f}(t_{n+1}, \mathbf{y})/\partial\mathbf{y}.$$

in backward_euler.py.

Linear solvers

- The dominant cost of an implicit time integration method is typically the solution of the linear systems of equations arising within Newton's method.
- For ODE systems arising from networks with m chemical species, each linear system

$$\left(J_F(\mathbf{y}^{(k)})\right)\,\Delta\mathbf{y}^{(k)} = \mathbf{F}(\mathbf{y}^{(k)})$$

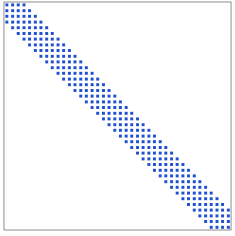
  corresponds with an $m \times m$ matrix.
- Gaussian Elimination with Partial Pivoting is "standard" for Newton's method, or LU Factorization with Partial Pivoting is standard for modified Newton's method, but both assume the Jacobian is dense, and have cost $\mathcal{O}(m^3)$.
- If the Jacobian has a special structure (e.g., banded, block diagonal, sparse, etc.), then specialized linear solvers can be used to reduce the cost of these solves.
- Or for very large problems, one may instead consider an iterative linear solver.

Structured matrices – Banded

A **banded** matrix J_F is one wherein the row i and column j entry, $(J_F)_{i,j}$ satisfies the property

$$(J_F)_{i,j} = 0, \quad \text{for all } |i-j| > b,$$

and where b is the *half-bandwidth* of the matrix.



- *Diagonal* matrices have $b = 0$
- *Tridiagonal* matrices have $b = 1$,
- *Pentadiagonal* matrices have $b = 2$, etc

Gaussian elimination may leverage the banded structure to require only $\mathcal{O}(b^2 m)$ operations. The *standard* banded matrix solver is LAPACK, wrapped within nearly every high-level software package:

- PYTHON wraps these in `scipy.linalg.solve_banded`.
- MATLAB notices the banded matrices automatically and wraps LAPACK from it's `A\b` operator.
- JULIA wraps these in the `BandedMatrices.jl` package.

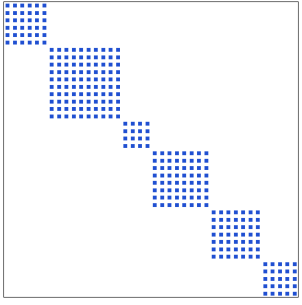
Structured matrices – Block diagonal

Block diagonal matrices arise frequently in chemistry applications that implicitly couple chemical species within each spatial cell, but that treat couplings between cells explicitly (*semi-implicit* or *implicit-explicit*).

These are like banded matrices, but with additional zeros that decouple the blocks, such that $A\mathbf{x} = \mathbf{b}$ becomes

$$A_i \mathbf{x}_i = \mathbf{b}_i, \quad i = 1, \ldots, \text{nblocks}$$

- This may be easily handled by solving each block independently, and then concatenating the results.
- As before, for each block LAPACK provides dense linear solvers that are wrapped elsewhere.

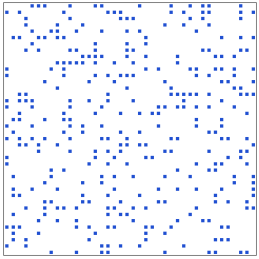


Structured matrices – Sparse

Many large-scale chemical reaction networks are "sparse", although they may not have a clearly discernible pattern. Sparse solvers can leverage the already-zero entries, gaining efficiency in terms of both computational effort and memory storage.

Two fundamental questions arise.

- Multiple storage formats exist: dictionary of keys (DOK), list of lists (LIL), coordinate list (COO), compressed sparse row (CSR, CRS, or Yale), and compressed sparse column (CSC or CCS).
- "Fill in": initially-zero entries become nonzero within the elimination process. *Reordering* algorithms convert $A\mathbf{x} = \mathbf{b}$ to $(P_1 A P_2)(P_2^{-1}\mathbf{x}) = P_1\mathbf{b}$, where P_1, P_2 are easily invertible, such that elimination on $P_1 A P_2$ minimizes fill in.



Popular open-source sparse solvers include SuiteSparse/KLU and SuperLU for serial and shared-memory computers, or SuperLU-DIST and MUMPS for distributed-memory computers.

Interfaces to some of these solvers are included in Matlab, Python and Julia.

Iterative linear solvers

- While direct solvers are highly robust, they may require more memory storage and computation than some applications can afford.
- For *large* systems of equations, iterative linear solvers may be used in an attempt to limit these costs.
- Iterative linear solvers begin with a *guess* $\mathbf{x}^{(0)}$ for the solution $\mathbf{x}^*$ to $A\mathbf{x} = \mathbf{b}$.
- The method then constructs a sequence $\left\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots\right\}$, in the hopes that at some point $\|\mathbf{x}^* - \mathbf{x}^{(l)}\|$ is small, and where $\|\cdot\|$ is some reasonable choice of vector norm.
- These are typically stopped using similar criteria as the nonlinear solvers, e.g., $\|\mathbf{r}^{(l)}\|_{WRMS} < 1$, where $\mathbf{r}^{(l)} = \mathbf{b} - A\mathbf{x}^{(l)}$ is the *residual vector*.

Krylov iterative linear solvers

While many iterative linear solvers exist, Krylov methods are particularly useful for chemistry. Reaction networks typically have non-symmetric Jacobian J, so the most widely-used Krylov method for reaction networks is the Generalized Minimum Residual (GMRES) method.

- Each GMRES iteration requires one matrix-vector product, $J_F\mathbf{v}$. This can be approximated without ever constructing J_F via a Gâteaux derivative,

$$J_F(\mathbf{y})\mathbf{v} \approx \frac{1}{\sigma}\left[\mathbf{F}\left(\mathbf{y}+\sigma\mathbf{v}\right)-\mathbf{F}\left(\mathbf{y}\right)\right],$$

  for appropriate $\sigma \ll 1$. In the case of Newton's method applied to the backward Euler scheme, this approximation becomes

$$J_F(\mathbf{y})\mathbf{v} \approx \mathbf{v} - \frac{h}{\sigma}\left[\mathbf{f}\left(t_{n+1},\mathbf{y}+\sigma\mathbf{v}\right)-\mathbf{f}\left(t_{n+1},\mathbf{y}\right)\right].$$

- GMRES requires at most m iterations to converge for a problem of size $\mathbf{x} \in \mathbb{R}^m$.
- However, GMRES stores mK data, where K is the maximum number of allowed iterations. Thus the faster GMRES converges, the better savings of both computational time and memory.

Preconditioned Krylov methods

Krylov methods may be accelerated with an effective *preconditioner*, P.

- P should approximate J_F^{-1} but be inexpensive to apply.
- This is used to convert $J_F\mathbf{x} = \mathbf{b}$ to the equivalent system $(J_F P)(P^{-1}\mathbf{x}) = \mathbf{b}$.
- GMRES is applied to $(J_F P)\mathbf{w} = \mathbf{b}$, and then $\mathbf{x}$ is then computed via $\mathbf{x} = P\mathbf{w}$.
- If $P \approx J_F^{-1}$ then $J_F P \approx I$, in which case GMRES should converge rapidly.
- "Optimal" preconditioners are unfortunately highly problem-dependent. Attractive options for stiff chemical networks include using a reduced-order network, machine learning techniques, or reduced-precision arithmetic.

Preconditioning Example – Oregonator (in example2_preconditioning.py)

Let's precondition the Oregonator with a matrix that includes only its fastest chemical reactions.

- Consider backward Euler where $J_F(t,y;h) = I - hJ(t,y)$, with J the true Oregonator Jacobian.
- $P(t,y;\gamma) = \left(I - h\tilde{J}(t,y)\right)^{-1}$, where $\tilde{J}$ zeros "slow" reaction rates, $k_2 = k_4 = k_5 = 0$.
- We compute the maximum of $|\lambda_{max}|/|\lambda_{min}|$ for both J_F and $J_F P$, over 501 time points, and consider a variety of h from 0.18 to 0.03:

h	0.18	0.09	0.06	0.045	0.036	0.03
J_F	25000	12000	8200	6100	4900	4100
$J_F P$	28	101	11	6.6	4.9	4.0

- As h decreases, so does the eigenvalue spread, thereby making it easier for GMRES to converge.
- The effect of P is even more pronounced, collapsing the eigenvalues to a small interval for even the largest step sizes, & dramatically improving the potential performance of iterative linear solvers.

Outline

Higher-order accuracy

There is much more to solving IVPs that arise in astrochemistry than the basics covered here. While we've focused on explicit and implicit Euler, those are the *lowest-order* methods available.

Techniques for higher-order accurate IVP methods for the time step $\mathbf{y}_n \to \mathbf{y}_{n+1}$ include:

(a) *Multi-derivative* methods (e.g., Taylor Series) use additional problem information, e.g., $\ddot{\mathbf{y}}(t) = \frac{\mathrm{d}}{\mathrm{d}t}\mathbf{f}(t, \mathbf{y}(t))$, ..., from within the time step $[t_n, t_{n+1}]$.

(b) *Multi-stage* methods (e.g., Runge–Kutta) evaluate $\mathbf{f}(t, y)$ at multiple locations within the time step $[t_n, t_{n+1}]$.

(c) *Multi-step* methods (e.g., BDF) use solution and/or right-hand side values $\mathbf{y}_k$ and $\mathbf{f}(t_k, \mathbf{y}_k)$ for $k < n$, i.e., from *outside* the time step $[t_n, t_{n+1}]$.

Advanced solvers also exist for multi-physics problems that *partition* the IVP system, e.g., ImEx, and multirate methods – multiple of these are included in the SUNDIALS package, that now provides native PYTHON interfaces (`pip install sundials4py`).

Homework

Consider the following moderately stiff chemical reaction network for $t \in [0, 320]$:

$$\begin{aligned}
\dot{y}_1 &= -1.71\, y_1 + 0.43\, y_2 + 8.32\, y_3 + 0.0007, & y_1(0) &= 1,\\
\dot{y}_2 &= 1.71\, y_1 - 8.75\, y_2, & y_2(0) &= 0,\\
\dot{y}_3 &= -10.03\, y_3 + 0.43\, y_4 + 0.035\, y_5, & y_3(0) &= 0,\\
\dot{y}_4 &= 8.32\, y_2 + 1.71\, y_3 - 1.12\, y_4, & y_4(0) &= 0,\\
\dot{y}_5 &= -1.745\, y_5 + 0.43\, y_6 + 0.43\, y_7, & y_5(0) &= 0,\\
\dot{y}_6 &= -280\, y_6 y_8 + 0.69\, y_4 + 1.71\, y_5 - 0.43\, y_6 + 0.69\, y_7, & y_6(0) &= 0,\\
\dot{y}_7 &= 280\, y_6 y_8 - 1.81\, y_7, & y_7(0) &= 0,\\
\dot{y}_8 &= -280\, y_6 y_8 + 1.81\, y_7, & y_8(0) &= 0.0057.
\end{aligned}$$

Study example2.py and example2_comparison.py. Write PYTHON codes to evolve network this using:

(a) backward Euler, (b) DIRK3, and (c) the BDF3, each using a variety of fixed step sizes h,

(d) `solve_ivp` with `'method=BDF'`, and (e) `solve_ivp` with `'method=RK45'`, both using rtol$= 10^{-5}$ and atol$= 10^{-10}$. For BDF, use the `jac` option to provide the Jacobian matrix.

You'll need to experiment to find appropriate h values for (a)-(c). Which methods do best/worst?

To access these slides