Modeling & Simulating Chemically Reacting Systems

Flemming Holtorf

Computer Science and Artificial Intelligence Laboratory

Department of Chemical Engineering

Massachusetts Institute of Technology

July 2022

Goals for this session

1. Build your own integrator module in Python
   - **Learn** how (generic) state-of-the-art Runge-Kutta time steppers work
     - adaptive time-stepping (performance)
     - implicit time-stepping (stability)
   - . . . and implement some of it

2. Build your own toy reactor model
   - **Learn** a template for how to model reacting systems
   - implement a simple reactor model

3. Build a simplified ionospheric reaction model (see Schunk [1988])

Time-stepping: Runge-Kutta methods

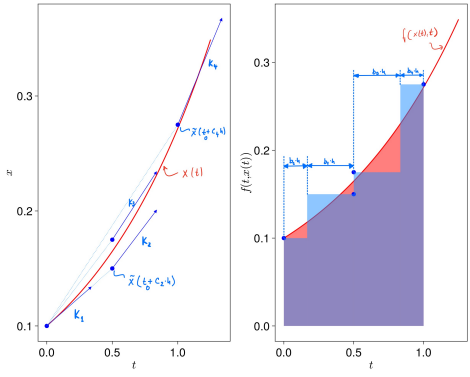
- Time-stepping schemes:

$$x(t+h) = x(t) + \underbrace{\int_t^{t+h} f(x(\tau),\tau)\,d\tau}_{\text{approximate}}$$

- Runge-Kutta method

$$\int_t^{t+h} f(x(\tau),\tau)\,d\tau \approx h\sum_{i=1}^{s} b_i k_i$$

$$\text{where } k_i = \underbrace{f(x(t)+h\sum_{j=1}^{s} a_{i,j}k_j,\, t+c_j h)}_{\approx x(t+c_j h)}$$



- Usually $\epsilon_{LTE} = O(h^{p+1})$ with $p \sim s$

$\Longrightarrow$ **tune accuracy/cost trade-off by choosing the right Runge-Kutta method**

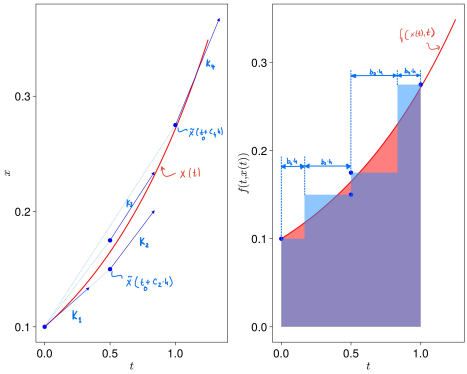
Time-stepping: Butcher tableaus
Explicit Runge-Kutta method

$$
\begin{array}{c|cccc}
c_1 & & & & \\
c_2 & a_{2,1} & & & \\
\vdots & \vdots & \ddots & & \\
c_s & a_{s,1} & \cdots & a_{s,s-1} & \\
\hline
 & b_1 & b_2 & \cdots & b_s
\end{array}
$$

Successively evaluate:

$$k_1 = f(x(t), t)$$
$$k_2 = f(x(t) + h a_{2,1} k_1, t + c_2 h)$$
$$\vdots$$
$$k_s = f(x(t) + h \sum_{j=1}^{s-1} a_{s,j} k_j, t + c_s h)$$

Update: $\boxed{\hat{x}(t+h) = x(t) + h \sum_{i=1}^{s} b_i k_i}$



Time-stepping: Butcher tableaus

Explicit Runge-Kutta method

$$
\begin{array}{c|ccccc}
c_1 & & & & & \\
c_2 & a_{2,1} & & & & \\
\vdots & \vdots & \ddots & & & \\
c_s & a_{s,1} & \cdots & a_{s,s-1} & & \\
\hline
 & b_1 & b_2 & \cdots & b_s
\end{array}
$$

Successively evaluate

$$
\begin{aligned}
k_1 &= f(x(t), t) \\
k_2 &= f(x(t) + h a_{2,1} k_1, t + c_2 h) \\
&\vdots \\
k_s &= f\left(x(t) + h \sum_{j=1}^{s-1} a_{s,j} k_j, t + c_s h\right)
\end{aligned}
$$

Update: $\hat{x}(t+h) = x(t) + h \sum_{i=1}^{s} b_i k_i$

Example: Explicit Euler

$$
\begin{array}{c|c}
c_1 = 0 & \\
\hline
 & b_1 = 1
\end{array}
$$

Evaluate:

$$k_1 = f(x(t), t)$$

Update:

$$
\begin{aligned}
\hat{x}(t+h) &= x(t) + h \sum_{i=1}^{s} b_i k_i \\
&= x(t) + h f(x(t), t)
\end{aligned}
$$

Time-stepping: Butcher tableaus

Explicit Runge-Kutta method

$$
\begin{array}{c|cccc}
c_1 & & & & \\
c_2 & a_{2,1} & & & \\
\vdots & \vdots & \ddots & & \\
c_s & a_{s,1} & \cdots & a_{s,s-1} & \\
\hline
 & b_1 & b_2 & \cdots & b_s
\end{array}
$$

Successively evaluate:

$$
\begin{aligned}
k_1 &= f(x(t), t) \\
k_2 &= f(x(t) + h a_{2,1} k_1, t + c_2 h) \\
&\vdots \\
k_s &= f\left(x(t) + h \sum_{j=1}^{s-1} a_{s,j} k_j, t + c_s h\right)
\end{aligned}
$$

Update: $\boxed{\hat{x}(t+h) = x(t) + h \sum_{i=1}^{s} b_i k_i}$

General/Implicit Runge-Kutta method

$$
\begin{array}{c|cccc}
c_1 & a_{1,1} & a_{1,2} & \cdots & a_{1,s} \\
c_2 & a_{2,1} & a_{2,2} & \cdots & a_{2,s} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
c_s & a_{s,1} & a_{s,2} & \cdots & a_{s,s} \\
\hline
 & b_1 & b_2 & \cdots & b_s
\end{array}
$$

Solve nonlinear equation system:

$$
\begin{aligned}
k_1 &= f\left(x(t) + h \sum_{j=1}^{s} a_{1,j} k_j, t + c_1 h, \right) \\
k_2 &= f\left(x(t) + h \sum_{j=1}^{s} a_{2,j} k_j, t + c_2 h\right) \\
&\vdots \\
k_s &= f\left(x(t) + h \sum_{j=1}^{s} a_{s,j} k_j, t + c_s h\right)
\end{aligned}
$$

Exercise 1: Implementing a generic explicit Runge-Kutta stepper

Question 1

Complete the function `explicit_RK_stepper(f,x,t,h,a,b,c)` in `runge_kutta.py`. It takes inputs

- `f` – right-hand-side $f(x,t)$ of the ODE to be integrated.
- `x` – current state $x(t)$
- `t` – current time t
- `h` – time step
- `a,b,c` – coefficients of the Runge-Kutta-Method. `b` and `c` are lists and `a` is organized as follows: `a = [[a_21], [a_31, a_32], ..., [a_s1, a_s2, ..., a_s(s-1)]]`

and returns the RK prediction $\hat{x}(t+h)$.

Explicit RK update

$$\hat{x}(t+h) = x(t) + \sum_{i=1}^{s} b_i k_i \text{ where}$$

$$k_1 = f(x(t), t)$$

$$k_2 = f(x(t) + h a_{2,1} k_1, t + c_2 h)$$

$$\vdots$$

$$k_s = f(x(t) + h \sum_{j=1}^{s-1} a_{s,j} k_j, t + c_s h)$$

Exercise 1: Implementing an explicit Runge-Kutta stepper

Question 2

Test your integrator by applying it to the test problem

$$\frac{dx}{dt} = -2x(t), \quad x(0) = 3, \ t \in [0, 2]$$

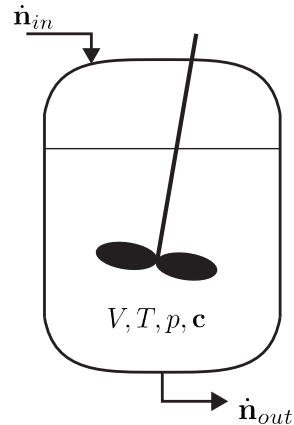
To that end, you only need to complete the function `dormand_prince_stepper(f,x,t,h)` in `main.py`. Please use `explicit_RK_stepper(f,x,t,h,a,b,c)` as implemented in the previous problem to do so.
The coefficients for the Dormand-Prince method Dormand and Prince [1980] are provided in the template.

Continuously Stirred Tank Reactors (CSTR)

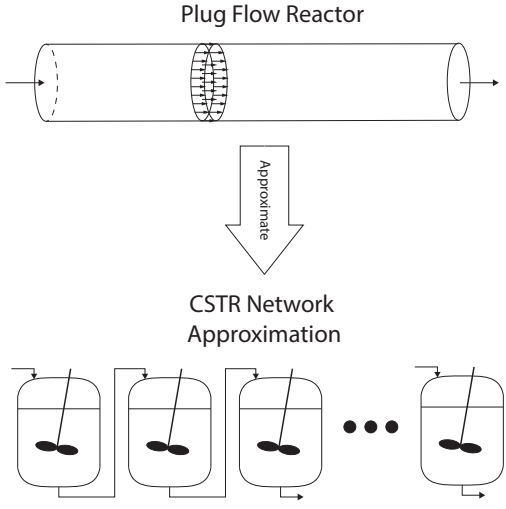
- No spatial variations of intrinsic state variables: $(T, p, \mathbf{c})$

- For externally imposed V, T, p the reactor is thermodynamically fully described by mole balances for all species:

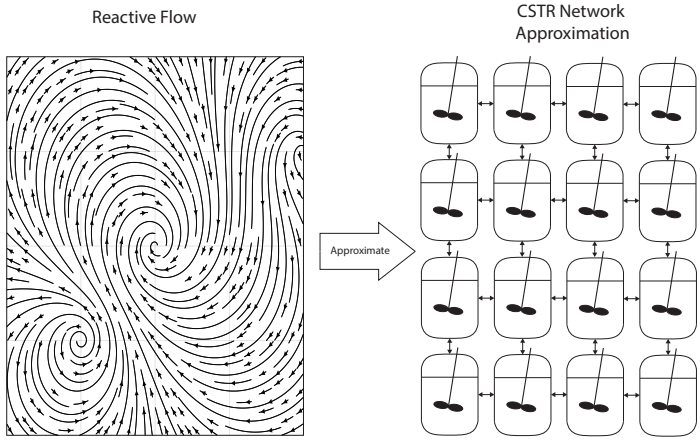
$$\frac{dn_i}{dt} = \dot{n}_{i,in} - \dot{n}_{i,out} + V \sum_{k=1}^{N_R} \underbrace{r_{i,j}(\mathbf{c}, p, T)}_{\text{reactive flux}}$$



CSTRs – The key building blocks for modeling reacting systems



CSTRs – The key building blocks for modeling reacting systems

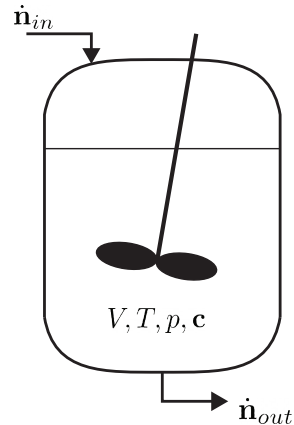


Common CSTR simplifications

- isochoric ($V = \text{const.}$) operation (usually implying $\dot{V}_{in} = \dot{V}_{out}$)

- Letting $D := \frac{\dot{V}}{V}$ and using that $n_i = Vc_i$ yields

$$\frac{dc_i}{dt} = D\left(c_{i,in} - c_i\right) + \sum_{k=1}^{N_R} r_{i,j}(\mathbf{c}, p, T)$$



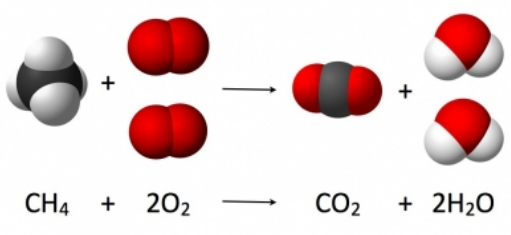
Modeling Reactions – Stoichiometry

Stoichiometry of Chemical Reactions

$$\nu_1^- A_1 + \nu_2^- A_2 + \cdots + \nu_N^- A_N \rightarrow \nu_1^+ A_1 + \nu_2^+ A_2 + \cdots + \nu_N^+ A_N$$

Example: Methane Combustion

$$\begin{aligned}
&\mathrm{CH_4}: && \nu^-_{\mathrm{CH_4}} = 1, \nu^+_{\mathrm{CH_4}} = 0 \\
&\mathrm{O_2}: && \nu^-_{\mathrm{O_2}} = 2, \nu^+_{\mathrm{O_2}} = 0 \\
&\mathrm{CO_2}: && \nu^-_{\mathrm{CO_2}} = 0, \nu^+_{\mathrm{CO_2}} = 1 \\
&\mathrm{H_2O}: && \nu^-_{\mathrm{H_2O}} = 0, \nu^+_{\mathrm{H_2O}} = 2
\end{aligned}$$



Modeling Reactions – Stoichiometry

- Reactive flux of species i due to reaction j:

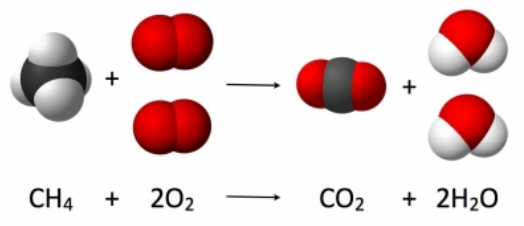
$$\underbrace{r_{i,j}(\mathbf{c}, T, p)}_{\text{reactive flux}} = (\nu_{ij}^{+} - \nu_{ij}^{-}) \underbrace{r_j(\mathbf{c}, T, p)}_{\text{reaction rate}}$$

- Summarize stoichiometry information in stoichiometry matrix

$$\mathbf{S} \in \mathbb{Z}^{\#\text{species} \times \#\text{reactions}} \text{ such that } \mathbf{S}_{i,j} = \nu_{i,j}^{+} - \nu_{i,j}^{-}$$

Example: Methane Combustion

$$\mathbf{S} = \begin{array}{c} \\ \mathrm{CH_4} \\ \mathrm{O_2} \\ \mathrm{CO_2} \\ \mathrm{H_2O} \end{array} \begin{array}{c} \text{reaction 1} \\ \begin{bmatrix} -1 \\ -2 \\ 1 \\ 2 \end{bmatrix} \end{array}$$



Implications for CSTR model

- In vector-valued notation, the mole balances describing a CSTR take a compact and computationally form:

$$\underbrace{\frac{d\mathbf{n}}{dt}}_{\begin{bmatrix} \frac{dn_1}{dt} \\ \frac{dn_2}{dt} \\ \vdots \\ \frac{dn_N}{dt} \end{bmatrix}} = \underbrace{\dot{\mathbf{n}}_{in}}_{\begin{bmatrix} \dot{n}_{1,in} \\ \dot{n}_{2,in} \\ \vdots \\ \dot{n}_{N,in} \end{bmatrix}} - \underbrace{\dot{\mathbf{n}}_{out}}_{\begin{bmatrix} \dot{n}_{1,out} \\ \dot{n}_{2,out} \\ \vdots \\ \dot{n}_{N,out} \end{bmatrix}} + V \underbrace{\mathbf{S}\mathbf{r}(\mathbf{c}, T, p)}_{\begin{bmatrix} \sum_{j=1}^{N_R} r_{1,j}(\mathbf{c}, T, p) \\ \sum_{j=1}^{N_R} r_{2,j}(\mathbf{c}, T, p) \\ \vdots \\ \sum_{j=1}^{N_R} r_{N,j}(\mathbf{c}, T, p) \end{bmatrix}} = \mathbf{S} \begin{bmatrix} r_1(\mathbf{c}, T, p) \\ r_2(\mathbf{c}, T, p) \\ \vdots \\ r_{N_R}(\mathbf{c}, T, p) \end{bmatrix}$$

- . . . or for isochoric operation:

$$\frac{d\mathbf{c}}{dt} = D(\mathbf{c}_{in} - \mathbf{c}) + \mathbf{S}\mathbf{r}(\mathbf{c}, T, p)$$

Modeling Reactions – Reaction Rates

- The theory of chemical kinetics is deep, rich and beyond the scope of this session.
- Most commonly mass-action kinetics are used (if appropriate or not ...):

$$r_j(\mathbf{c}, T, p) = k(T, p) \prod_{i=1}^{N} c_i^{\nu_{i,j}^-}$$

- The temperature variation of the rate coefficient k(T,p) is frequently modeled via the *extended Arrhenius* equation:

$$k(T, p) = A(p)\, T^{n(p)} e^{-\frac{E_a(p)}{k_B T}}$$

- Pressure dependence is hard, so people frequently assume $A(p), n(p), E_a(p) \approx$ const. in the pressure range of interest.

Exercise 2: Modeling a toy reactor

Toy Reaction Network

$$\mathrm{A} \xrightarrow{k_1} \mathrm{B} \underset{k_3}{\overset{k_2}{\rightleftharpoons}} 2\,\mathrm{C}$$

Assumptions

- The reactor is closed: $D = \dot{n}_{i,in} = \dot{n}_{i,out} = 0$.
- The fluid volume, temperature and pressure are constant.
- The reactions follow mass action kinetics with constant rate coefficients:

$$k_1 = 100\,\mathrm{s}^{-1},\ k_2 = 0.25\,\mathrm{s}^{-1},\ k_3 = 1\,\mathrm{cm}^3/\mathrm{mol/s}$$

Question 1

How does the general CSTR model look like for this setting?

Question 2

State the stoichiometry matrix associated with this network. Implement a `numpy` array carrying this matrix.

Question 3

Write a function `reaction_rates(c,k)` that takes in a vector `k` containing the rate coefficients, and a vector `c` of the concentrations of species A, B, & C and returns the reaction rates of the respective reactions as `numpy` array.

Question 4

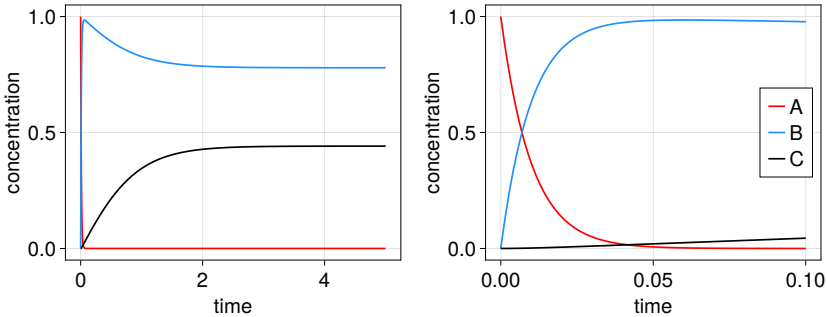
Write a function `reactor(c,k,S)` that computes the right-hand-side of the CSTR model.

Question 5

Simulate the reactor for different step sizes h using the previously implemented interface.

Stiffness – the curse of modeling reactions

Dynamics at different time scales



Small steps are only needed to accurately simulate fast dynamics.

Can we adjust our step sizes according to the dynamics on the fly?

Order of Accuracy

Definition (Local Truncation Error)

Given an explicit ODE

$$\frac{dx}{dt} = f(x, t)$$

The local truncation error is defined as the error between the exact solution at $x(t + h)$ and the numerical approximation $\hat{x}(t + h)$ after *one* time step:

$$\epsilon_{LTE} = \|x(t + h) - \hat{x}(t + h)\|$$

Definition (Consistency & Order)

An integration method is called consistent if $\epsilon_{LTE} = o(h)$. It is called order p if $\epsilon = O(h^{p+1})$.

Order of Accuracy

Example: Explicit Euler

Given an explicit ODE:

$$\frac{dx}{dt} = f(x, t).$$

The explicit Euler integration update is defined

$$\hat{x}(t + h) = x(t) + hf(x(t), t)$$

The exact solution can be Taylor expanded around t (under weak smoothness assumptions)

$$x(t + h) = x(t) + \underbrace{\left.\frac{dx}{dt}\right|_t}_{=f(x(t),t)} h + \frac{1}{2} \left.\frac{d^2x}{dt^2}\right|_t h^2 + O(h^3)$$

Thus, (again under very weak assumptions)

$$\epsilon_{LTE} = \|x(t + h) - \hat{x}(t + h)\| = \left\| \frac{1}{2} \left.\frac{d^2x}{dt^2}\right|_t h^2 + O(h^3) \right\| = O(h^2)$$

Adaptive Runge-Kutta methods

Idea: Interweave two Runge-Kutta methods with different orders of accuracy to estimate ϵ_{LTE} and choose step size to keep error below threshold.

Recall that

$$\epsilon_{LTE} = \|\hat{x}(t+h) - x(t+h)\|$$

Thus, given $\hat{x}(t+h)$ and $\hat{x}'(t+h)$, we can conclude that

$$\epsilon'_{LTE} = \|\hat{x}'(t+h) - x(t+h)\| = \|\hat{x}'(t+h) - \hat{x}(t+h)\| + O(\epsilon_{LTE})$$

... but how do we do this in practice?

If both methods are order $s' < s$, respectively, we have that $\epsilon_{LTE} = O(\epsilon'_{LTE})$. Thus, the local truncation error of both methods can be estimated as

$$\epsilon_{LTE}, \epsilon'_{LTE} = O(\|\hat{x}(t+h) - \hat{x}'(t+h)\|)$$

Adaptive Runge-Kutta methods

Solution: Use two Runge-Kutta methods of order p and $p-1$ that share the same intermediate steps for minimal computation.

Adaptive Runge-Kutta method

$$
\begin{array}{c|ccc}
c_1 & a_{1,1} & a_{1,2} & \cdots & a_{1,s} \\
c_2 & a_{2,1} & a_{2,2} & \cdots & a_{2,s} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
c_s & a_{s,1} & a_{s,2} & \cdots & a_{s,s} \\
\hline
 & b_1 & b_2 & \cdots & b_s \\
 & b'_1 & b'_2 & \cdots & b'_s
\end{array}
$$

Procedure: Simple step size control

1: compute RK update $x(t+h)$ and error estimate ϵ
2: **while** $\epsilon > \texttt{rtol} \cdot \|x(t+h)\| + \texttt{atol}$ **do**
3: half step size: $h \leftarrow h/2$
4: recompute $x(t+h)$ and ϵ
5: **end while**
6: double step size: $h \leftarrow 2h$
7: **return** x(t+h), h

Error estimate: $\epsilon_{LTE} \approx \|\hat{x}(t+h) - \hat{x}'(t+h)\| = \left\| h \sum_{i=1}^{s} (b_i - b'_i) k_i \right\|$

Exercise 3: Implementing an adaptive, explicit Runge-Kutta method

Question 1

Complete the function `adaptive_explicit_RK_stepper(f,x,t,h,a,b,c,b_control)` in . The inputs shall be understood as

- `f` – right-hand-side $f(x, t)$ of the ODE to be integrated.
- `x` – current state of the ODE
- `t` – current time
- `h` – time step
- `a,b,c,b_control` – coefficients of the Runge-Kutta-Method. `b`, `b_control` and `c` are lists and `a` is organized as follows: `a` = [[a_21], [a_31, a_32], ..., [a_s1, a_s2, ..., a_s(s-1)]]

The function returns the RK update $\hat{x}(t + h)$ and an estimate of ϵ_{LTE}.

Exercise 3: Implementing an adaptive, explicit Runge-Kutta method

Question 2

Complete the function `adaptive_integrate` in `runge_kutta.py`. It shall implement the integration via an adaptive integrator such that the one implemented in `adaptive_explicit_RK_stepper`. Follow the pseudo-code below.

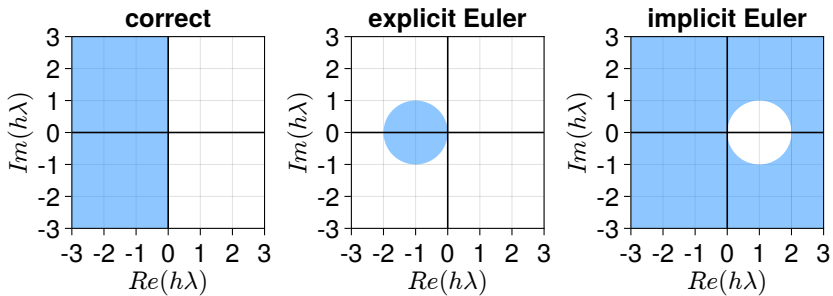
Pseudo-code for adaptive integration:

1: Given $x(t)$, compute RK update $\hat{x}(t+h)$ and error estimate ϵ_{LTE}
2: **if** $\epsilon_{LTE} > \texttt{rtol} \cdot \|x(t+h)\| + \texttt{atol}$ **then**
3: Reject the step and half step size: $h \leftarrow h/2$
4: Go to Line 1
5: **else**
6: Accept the step: $t \leftarrow t+h, x(t) \leftarrow \hat{x}(t+h)$
7: Double step size: $h \leftarrow 2h$
8: Go to Line 1
9: **end if**
10: Repeat until end of time horizon is reached.

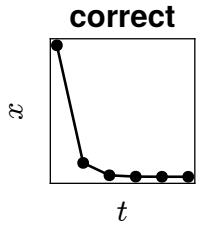
Stability

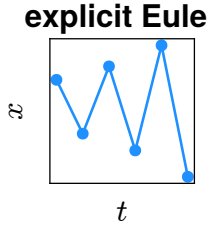
Dahlquist Stability Analysis

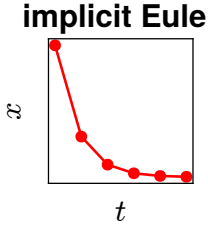
Apply numerical integrator with stepsize h to Dahlquist's test equation $\frac{dx}{dt} = \lambda x$

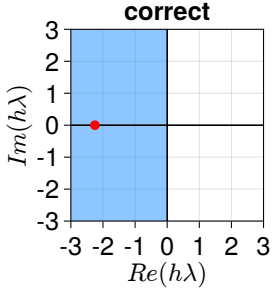


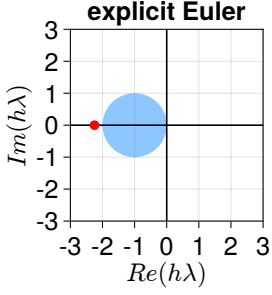
Stability

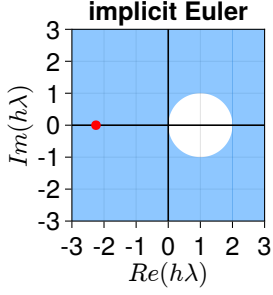

correct
x
t


explicit Euler
x
t

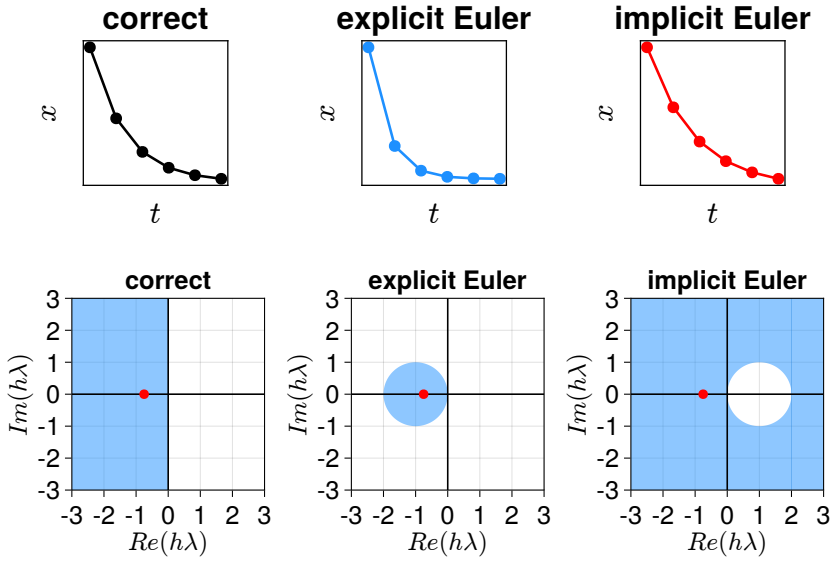

implicit Euler
x
t


correct
Im(hλ)
Re(hλ)
3
2
1
0
-1
-2
-3

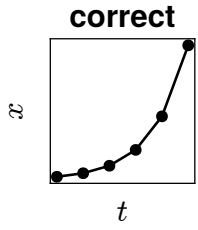

explicit Euler
Im(hλ)
Re(hλ)
3
2
1
0
-1
-2
-3

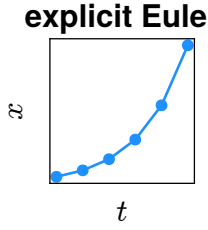

implicit Euler
Im(hλ)
Re(hλ)
3
2
1
0
-1
-2
-3

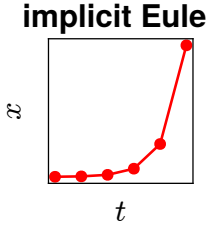
Stability

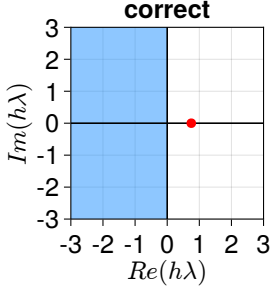

correct
explicit Euler
implicit Euler
x
t
Im(hλ)
Re(hλ)
3
2
1
0
-1
-2
-3

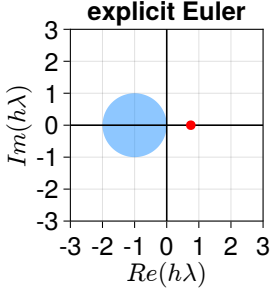
Stability

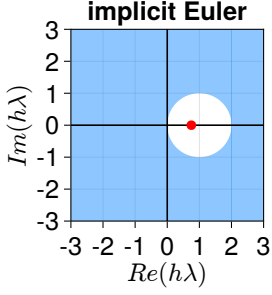

correct
x
t


explicit Euler
x
t

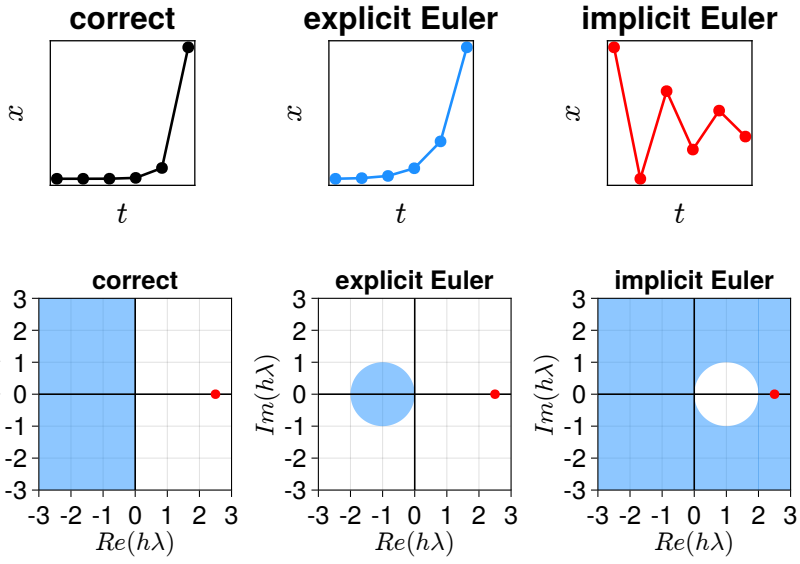

implicit Euler
x
t


correct
Im(hλ)
Re(hλ)
3
2
1
0
-1
-2
-3


explicit Euler
Im(hλ)
Re(hλ)
3
2
1
0
-1
-2
-3


implicit Euler
Im(hλ)
Re(hλ)
3
2
1
0
-1
-2
-3

Stability


correct
explicit Euler
implicit Euler
x
t
Im(hλ)
Re(hλ)
3
2
1
0
-1
-2
-3

Implicit time-stepping

- Superior stability properties
- ... but need to solve an implicit equation system (find x such that $g(x) = 0$)
  Consider for example implicit Euler:

$$\text{find } \hat{x}(t+h) \text{ such that } 0 = x(t) + hf(t+h, \hat{x}(t+h)) - \hat{x}(t+h)$$

- Usually some numerical equation solving technique is required (Newton, fixed-point iteration, ...)

How to solve implicit equation systems in python

- We will use `scipy.optimize.fsolve` to do this
- To that end, our equation solving problem is to be formulated as

$$\text{find } x \text{ such that } 0 = f(x)$$

- We then supply a python function implementing $f(x)$ & an initial guess for the solution x_0.

```
from scipy.optimize import fsolve
import numpy as np
# right-hand-side
def f(x):
    return np.exp(x) - 4*x
# initial guess
x0 = 2.0
# solve statment assings solution to x
x = fsolve(f, x0)
```

- Analogous syntax for vector-valued functions

Exercise 4: Comparing integrators

Question 1

Complete the code template `integrators.py` and compare the stability and accuracy properties of the integrators below. To that end, plot

- the traces of $c_A(t), c_B(t), c_C(t)$ as predicted by each method
- the time needed for integration
- the accuracy as measured by $|c_A(0.01) - c_{A,0}/e|$

for step sizes $h \in \{1 \times 10^{-4}, 1 \times 10^{-3}, 1 \times 10^{-2}, 1 \times 10^{-1}, 1, 1 \times 10^{1}\}$

Method	**Time-stepping rule**
Explicit Euler	$x_{k+1} = x_k + hf(x_k, t_k)$
Implicit Euler[1]	$x_{k+1} = x_k + hf(x_{k+1}, t_k + h)$
Heun's Method	$y = x_k + hf(x_k, t_k)$ $x_{k+1} = x_k + \frac{h}{2}\left(f(x_k, t_k) + f(y, t_k + h)\right)$
Crank-Nicolson Method[1]	$x_{k+1} = x_k + \frac{h}{2}\left(f(x_k, t_k) + f(x_{k+1}, t_k + h)\right)$

[1] Hint: use `scipy.optimize.fsolve` to solve the equation system for the implicit methods.

Disclaimer

- Runge-Kutta methods are not the end of the story
  - linear multistep methods (Adams-Bashforth, Adams-Moulton, **BDF**)
  - symplectic/geometric integrators
  - low rank integrators
  - ...

- If you can, do **NOT** implement these yourself. Others have done the work for you (and most likely better)
  - `DifferentialEquations.jl` in Julia
  - `scipy.integrate` in Python
  - ...

Exercise 5: Simplified ionosphere chemistry (derived from [Schunk, 1988])

Chemistry

$$\mathrm{O^+ + N_2} \xrightarrow{k_1} \mathrm{NO^+ + N}$$

$$\mathrm{O^+ + O_2} \xrightarrow{k_2} \mathrm{O + {O_2}^+}$$

$$\mathrm{{O_2}^+ + e} \xrightarrow{k_3} \mathrm{2\,O}$$

$$\mathrm{{N_2}^+ + O} \xrightarrow{k_4} \mathrm{O^+ + N_2}$$

$$\mathrm{{N_2}^+ + O_2} \xrightarrow{k_5} \mathrm{{O_2}^+ + N_2}$$

$$\mathrm{{O_2}^+ + N} \xrightarrow{k_6} \mathrm{NO^+ + O}$$

$$\mathrm{NO^+ + e} \xrightarrow{k_7} \mathrm{N + O}$$

Ionization

$$\mathrm{O} \underset{k_8}{\overset{k_9}{\rightleftharpoons}} \mathrm{O^+ + e}$$

$$\mathrm{O_2} \underset{k_{10}}{\overset{k_{11}}{\rightleftharpoons}} \mathrm{{O_2}^+ + e}$$

$$\mathrm{N_2} \underset{k_{12}}{\overset{k_{13}}{\rightleftharpoons}} \mathrm{{N_2}^+ + e}$$

Question

Complete the provided template. Simulate the system at the provided conditions. For simulation, use `scipy.integrate.solve_ivp` and specify `'LSODA'` as method high-performance method.

Rate coefficients

$$k_1(T) = 1.533 \times 10^{-12} - 5.92 \times 10^{-13} \frac{T}{300\,\mathrm{K}} + 8.6 \times 10^{-14} \left(\frac{T}{300\,\mathrm{K}}\right)^2$$

$$k_2(T) = 2.82 \times 10^{-11}$$

$$k_3(T) = 1.6 \times 10^{-7} \left(\frac{300\,\mathrm{K}}{T}\right)^{0.55}$$

$$k_4(T) = 1 \times 10^{-11} \left(\frac{300\,\mathrm{K}}{T}\right)^{0.23}$$

$$k_5(T) = 5 \times 10^{-11} \frac{300\,\mathrm{K}}{T}$$

$$k_6(T) = 1.2 \times 10^{-10}$$

$$k_7(T) = 1 \times 10^{-11} \left(\frac{300\,\mathrm{K}}{T}\right)^{0.85}$$

$$k_8(T) = k_{10} = k_{12} = 1 \times 10^{-8}$$

$$k_9(T) = k_{11} = k_{13} = 1 \times 10^{-5}$$

Thank you!

github.com/FHoltorf

holtorf@mit.edu

References

JR Dormand and PJ Prince. A family of embedded Runge-Kutta formulae. *Journal of computational and applied mathematics*, 6(1):19–26, 1980.

RW Schunk. A mathematical model of the middle and high latitude ionosphere. *Pure and applied geophysics*, 127(2):255–303, 1988.