Danmarks Tekniske Universitet

DTU

02686
Scientific Computing for Differential Equations

Authors

Pierre Segonne - s182172

May 26, 2020

Contents

Introduction

Ordinary Differential Equations (ODEs) appear naturally when describing dynamical systems such as in physics ([1] page 4, [2]), chemistry ([3]), or economics ([4], chapters 15 and 16).

In their most general form, differential equations can be spelled out as

$$\boldsymbol{x}'(t) = f(t, \boldsymbol{x}(t)) \tag{1}$$

where $\dot{\boldsymbol{x}} \in \mathbb{R}^n$ is the temporal derivative of $\boldsymbol{x} \in \mathbb{R}^n$. While in some of their earliest applications - such as the study of the trajectory of an object thrown into the air, or the simple pendulum under the assumption of small angles - they admit analytical solutions, many actually don't have closed form solutions. It is therefore necessary to elaborate numerical methods that can approximate with great accuracy solutions of differential equations.

Such is the goal of this report. Throughout this document will be detailed numerical methods that can approximate solutions to differential equations. Their mathematical derivation will be given and considerations about their accuracy and stability will be detailed. Such discussions will then make it clear what advantages each method offer in terms of computational efficiency and exactitude and therefore when they should be used.

Initial value problems (IVP) will be specifically used here. In their general form they consist of a differential equation as described in 1, given an initial value for the solution $\boldsymbol{x}$.

$$\boldsymbol{x}'(t) = f(t, \boldsymbol{x}(t)), \qquad \boldsymbol{x}(t_0) = \boldsymbol{x}_0 \tag{2}$$

Their approximation will be computed over a certain interval of time, $I = [t_0, t_N]$ with a step size that will be noted $h = \frac{t_N - t_0}{N}$.

Numerical methods will be introduced together with a *Python* implementation, whose interface and parameters will be described at length to allow an easy use by any interested reader.

Default ODE solver

A default ODE solver will serve as a baseline comparison to all methods introduced in this report. It uses the *Scipy ode* interface[1]. This interface allows to compute an approximated solution to any provided differential equation with different methods. For the baseline, the method specified with the *dopri5* denominator, which actually implements the Dormand Prince 4(5) method following [5].

[1] https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.ode.html

Test Problems

The Van der Pol Oscillator

The Van der Pol oscillator arises as the solution of the second order differential equation:

$$y''(t) = \mu(1 - y(t)^2)y'(t) - y(t) \tag{3}$$

which can be re-arranged into the general ODE form exposed above in 1. Noting $\boldsymbol{x}(t) = \begin{bmatrix} y(t) \\ y'(t) \end{bmatrix}$ results in the following:

$$\boldsymbol{x}'(t) = \begin{bmatrix} y'(t) \\ y''(t) \end{bmatrix} = \begin{bmatrix} y'(t) \\ \mu(1 - y(t)^2)y'(t) - y(t) \end{bmatrix} = f(t, \boldsymbol{x}(t)) \tag{4}$$

The function

$$f(t, \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}) = \begin{bmatrix} x_2 \\ \mu(1 - x_1^2)x_2 - x_1 \end{bmatrix} \tag{5}$$

has the following Jacobian matrix:

$$J(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}) = \begin{bmatrix} 0 & 1 \\ -2\mu x_1 x_2 - 1 & \mu(1 - x_1^2) \end{bmatrix} \tag{6}$$

The parameter μ controls the sharpness of the limit cycle, and as a result, the stiffness of the equation generating the oscillator. Indeed, with an increased sharpness, some regions of the solution curve display greater variation which can then induce error in the approximate solution.

The following figures, 1 and 2, which represent numerical solutions for the Van der Pol oscillator, for $\mu = 2$ and $\mu = 12$ respectively using the default ODE solver presented above, reveal the influence of μ on the stiffness of the problem. For $\mu = 12$, the spikes of $x_2(t)$ induce a greater variation of the solution than for $\mu = 2$

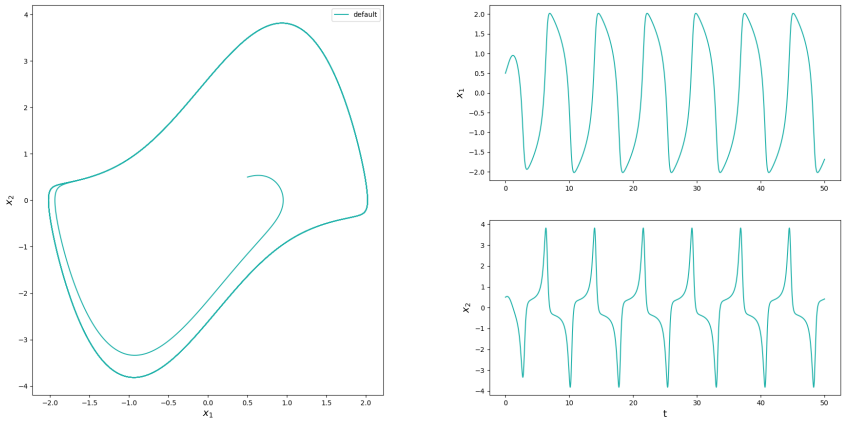


Figure 1: Van der Pol oscillator simulation with default ode solver for parameter $\mu = 2$

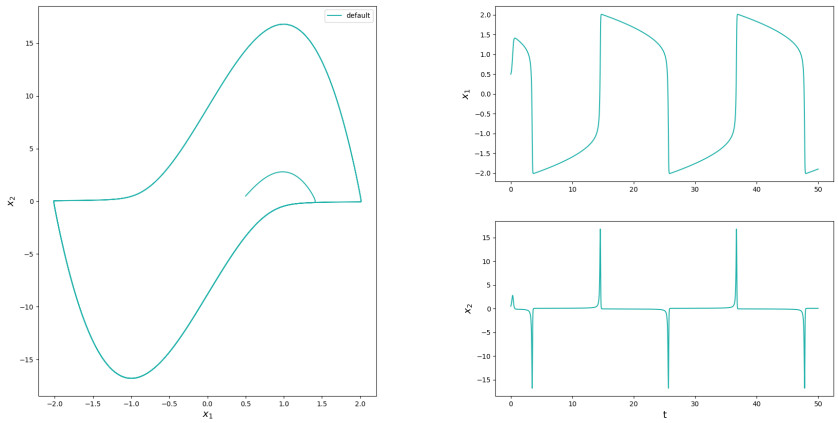


Figure 2: Van der Pol oscillator simulation with default ode solver for parameter $\mu = 12$

The A-CSTR

The Adiabatic Continuous Stirred Reactor (A-CSTR) describes a chemical reaction conducted in an adiabatic CSTR.

The reaction is the following

$$Na_2S_2O_3 + 2H_2O_2 \rightarrow \frac{1}{2}Na_2SO_4 + \frac{1}{2}Na_2S_3O_6 + 2H_2O$$

which can be simplified to

$$A + 2B \rightarrow \frac{1}{2}C + \frac{1}{2}D + 2E$$

By noting C_A, C_B the concentration of reactants A and B and T the output temperature, the following system of differential equations hold:

$$\boldsymbol{x}'(t) = \begin{bmatrix} C'_A(t) \\ C'_B(t) \\ T'(t) \end{bmatrix} = \begin{bmatrix} \frac{F}{V}(C_{A,in} - C_A(t)) + R_A(C_A(t), C_B(t), T(t)) \\ \frac{F}{V}(C_{B,in} - C_B(t)) + R_B(C_A(t), C_B(t), T(t)) \\ \frac{F}{V}(T_{in} - T(t)) + R_T(C_A(t), C_B(t), T(t)) \end{bmatrix} = f(t, \boldsymbol{x}(t)) \tag{7}$$

where

$$\begin{aligned} R_A(C_A, C_B, T) &= -r(C_A, C_B, T) \\ R_B(C_A, C_B, T) &= -2r(C_A, C_B, T) \\ R_T(C_A, C_B, T) &= \beta r(C_A, C_B, T) \end{aligned} \tag{8}$$

and

$$r(C_A, C_B, T) = k(T)C_A C_B \tag{9}$$

and finally

$$k(T) = k_0 \exp\left(\frac{-E_a}{RT}\right) \tag{10}$$

which is the Arrhenius equation. It is applied on temperatures evaluated in Kelvins.

The function f that defines the right hand side (RHS) of the differential equation,

$$f(t, \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}) = \begin{bmatrix} \frac{F}{V}(C_{A,in} - x_1) - k_0 \exp\left(\frac{-E_a}{Rx_3}\right) x_1 x_2 \\ \frac{F}{V}(C_{B,in} - x_2) - 2k_0 \exp\left(\frac{-E_a}{Rx_3}\right) x_1 x_2 \\ \frac{F}{V}(T_{in} - x_3) + \beta k_0 \exp\left(\frac{-E_a}{Rx_3}\right) x_1 x_2 \end{bmatrix} \tag{11}$$

has the following Jacobian:

$$J(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}) = \begin{bmatrix} -\frac{F}{V} - k_0 \exp\left(\frac{-E_a}{Rx_3}\right) x_2 & -k_0 \exp\left(\frac{-E_a}{Rx_3}\right) x_1 & -\frac{k_0 E_a}{Rx_3^2} \exp\left(\frac{-E_a}{Rx_3}\right) x_1 x_2 \\ -2k_0 \exp\left(\frac{-E_a}{Rx_3}\right) x_2 & -\frac{F}{V} - 2k_0 \exp\left(\frac{-E_a}{Rx_3}\right) x_1 & -2\frac{k_0 E_a}{Rx_3^2} \exp\left(\frac{-E_a}{Rx_3}\right) x_1 x_2 \\ \beta k_0 \exp\left(\frac{-E_a}{Rx_3}\right) x_2 & \beta k_0 \exp\left(\frac{-E_a}{Rx_3}\right) x_1 & -\frac{F}{V} + \beta \frac{k_0 E_a}{Rx_3^2} \exp\left(\frac{-E_a}{Rx_3}\right) x_1 x_2 \end{bmatrix} \tag{12}$$

This 3D model can further be reduced to a single dimension, as detailed in [6]. With $C_A(T) = C_{A,in} + \frac{1}{\beta}(T_{in} - T)$ and $C_B(T) = C_{B,in} + \frac{2}{\beta}(T_{in} - T)$, the 1D model can be written as:

$$T'(t) = \frac{F}{V}(T_{in} - T(t)) + R_T(C_A(T(t)), C_B(T(t)), T(t)) \tag{13}$$

and here the function f that defines the RHS of the equation,

$$f(t, x) = \frac{F}{V}(T_{in} - x) + \beta k_0 \exp\left(\frac{-E_a}{Rx}\right)(C_{A,in} + \frac{1}{\beta}(T_{in} - x))(C_{B,in} + \frac{2}{\beta}(T_{in} - x)) \tag{14}$$

has the following Jacobian:

$$\begin{aligned} J(x) = & -\frac{F}{V} + \frac{k_0 E_a}{Rx^2} \exp\left(\frac{-E_a}{Rx}\right)(C_{A,in} + \frac{1}{\beta}(T_{in} - x))(C_{B,in} + \frac{2}{\beta}(T_{in} - x)) \\ & - k_0 \exp\left(\frac{-E_a}{Rx}\right)(C_{B,in} + \frac{2}{\beta}(T_{in} - x)) \\ & - 2k_0 \exp\left(\frac{-E_a}{Rx}\right)(C_{A,in} + \frac{1}{\beta}(T_{in} - x)) \end{aligned} \tag{15}$$

In both models, the flow parameter F plays a key role in the evolution of the states. When the flow is high, the reactor gets filled with reactants A and B, at a low temperature (0.5 °C). The concentration of reactants therefore increases and the temperature decreases. On the other hand, when the flow is reduced, the exothermic reaction will heat up the reactor and the reactants will be consumed in the process, thus reducing their concentrations. Controlling the flow during the reaction is thus paramount to the control of the evolution of the reaction states. For this reason, the value of the flow, similarly as what is done in [6]

(Figure 1.), evolves during the time interval studied. Figure 3 demonstrates such evolution.

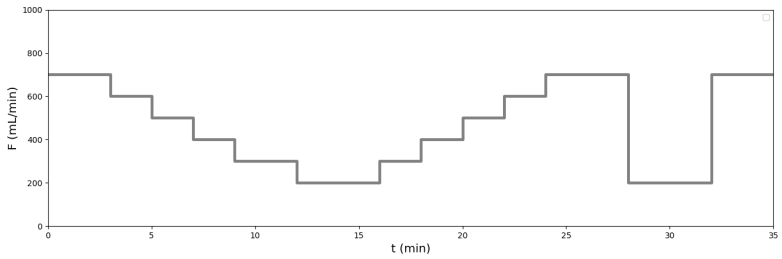


Figure 3: Evolution of the flow F over the integration period

Please note that all time dynamics were considered on a minute basis and the parameters of the reaction, available in TODO Annex, were adapted accordingly.

Figure 4 demonstrates the approximation of the 3D state of the reaction as determined by the default ode solver.

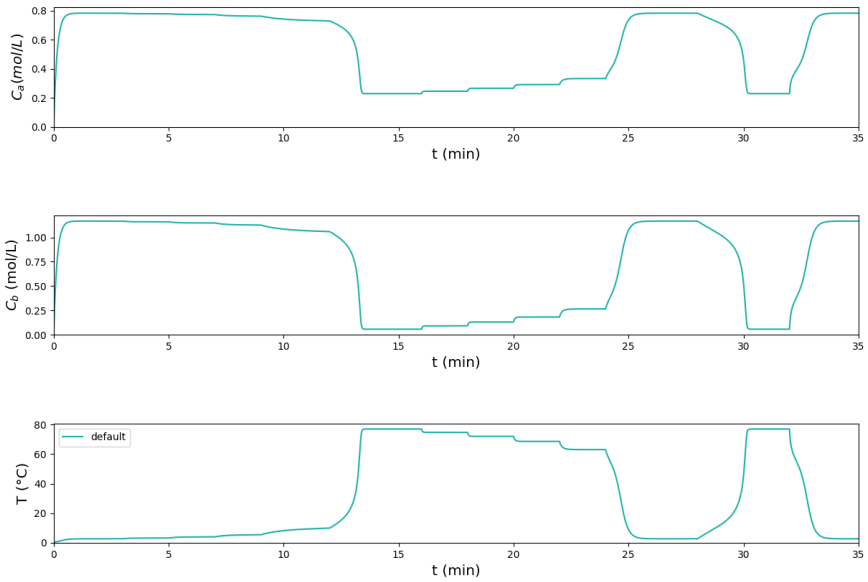


Figure 4: Evolution of the state in the 3D model

And Figure 5 its 1D counterpart.

Python Interface

All implementations of the methods that will be detailed in this report expose the same interface and follow the same structure for easier compatibility and reliability. All the solvers are encapsulated in a function that receives the following parameters:

```
1 def ode_solver(f, J, t0, tf, N, x0, adaptive_step_size=False, **kwargs):
```

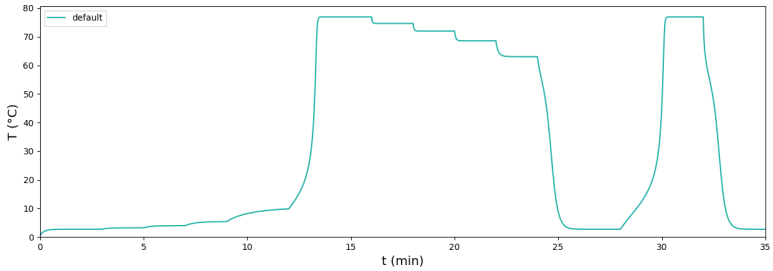


Figure 5: Evolution of the temperature in the 1D model

Where,

- *t0* is the initial time
- *tf* is the final time
- N is the number of steps to use in the simulation
- *adaptive_step_size* is an optional Boolean option to specify whether the step size should be adapted during the simulation
- *kwargs* are any key word arguments passed down for various purposes, such as tolerances specifications or parameters of the problem function.

f represents the RHS of the differential equation and is structured as:

```
def f(t, X, **kwargs):
    f = ...
    return f
```

J represents the Jacobian of the RHS and is structured as:

```
def J(t, X, **kwargs):
    J = ...
    return J
```

MISC

TODO, mathematical analysis

- Global Error IE
- Classical RK
- Classical RK, adaptive step size
- Dormand-Prince, adaptive step size
- Build own method

- Derive ESDIRK23

  TODO, code

- esdrik23, refacto?

- sde

- own rk

Notes:

F is a flow rate in CSTR, which we control.

What is meant by show with different step sizes is that it should be shown with rough steps what happens

Yes. A- and L-stability are very important properties among other properties of the ESDIRK23 method. A-stability implies that ESDIRK23 can be applied to stiff systems of differential equations. L-stability means that it can be applied to index-1 DAE systems (can be considered as extremly stiff systems of differential equations). Explicit RK methods cannot be A-stable and hence not L-stable. Therefore, they are not well suited for stiff systems of differential equations, as the time-step size would be limited by stability and very small.

ref for test eq

Definition of stiffness

Derivation of RK4

1 Explicit ODE solver

1.1 Description of the Explicit Euler Method

Let's consider again an IVP in its general form:

$$\boldsymbol{x}'(t) = f(t, \boldsymbol{x}(t)), \qquad \boldsymbol{x}(t_0) = \boldsymbol{x}_0 \tag{16}$$

Assuming that x is infinitely differentiable, it is possible to take its Taylor series, evaluated in t_{k+1} with respect to t_k, separated by $\Delta t = t_{k+1} - t_k = h$:

$$\boldsymbol{x}(t_{k+1}) = \sum_{n=0}^{\infty} \frac{\boldsymbol{x}^{(n)}(t_k)}{n!}(t_{k+1} - t_k)^n = \sum_{n=0}^{\infty} \frac{\boldsymbol{x}^{(n)}(t_k)}{n!}h^n \tag{17}$$

And by restricting the series to its first two terms, n=0 and n=1, the following approximation arises:

$$\boldsymbol{x}(t_{k+1}) \simeq \boldsymbol{x}(t_k) + h\boldsymbol{x}'(t_k) = \boldsymbol{x}(t_k) + hf(t_k, \boldsymbol{x}(t_k)) \tag{18}$$

Therefore, if the evaluation of the RHS of the ODE for all t_k is possible, and if $\boldsymbol{x}_0 = \boldsymbol{x}(t_0)$ is known, this approximation then constitutes a valid way to iteratively and numerically solve the ODE. Instead of determining the exact solution $\boldsymbol{x}(t_k)$ for every t_k, a numerical approximation $\boldsymbol{x}_k \simeq \boldsymbol{x}(t_k)$ can be computed.

It is called the explicit Euler method.

1.2 Python Implementation

The implementation of the explicit Euler method is straightforward as any new iteration $\boldsymbol{x}_{k+1}$ only depends on the previous iteration $\boldsymbol{x}_k$ and the evaluation of the RHS function at the previous iteration $f(t_k, \boldsymbol{x}_k)$.

```
def ode_solver(f, J, t0, tf, N, x0, **kwargs):

    dt = (tf - t0)/N

    T = [t0]
    X = [x0]

    for k in range(N):
        f_eval = f(T[-1], X[-1], **kwargs)
        X.append(X[-1] + dt*f_eval)
        T.append(T[-1] + dt)

    T = np.array(T)
    X = np.array(X)

    return X, T
```

Listing 1: Implementation of the explicit Euler method

Figure 6 helps demonstrate how the method works. In black is plotted the true solution of a simple exponential decay and in green the approximation computed by the explicit Euler method.

On the left, the step size h was deliberately chosen too large (0.5). The initial estimation of the slope of the decay is indeed tangent to the slope around 0 but as the step size is too large, the approximation overshoots the true solution. At the following point, the slope is re-estimated, and again the large step size leads to an even greater approximation error. At the end, these accumulated errors result in a poor approximation of the true solution.

On the right, the step size was chosen much smaller (0.003). Due to this small size, the overshooting effect witnessed on the left plot is greatly reduced and the approximation of the true solution seems visually acceptable.

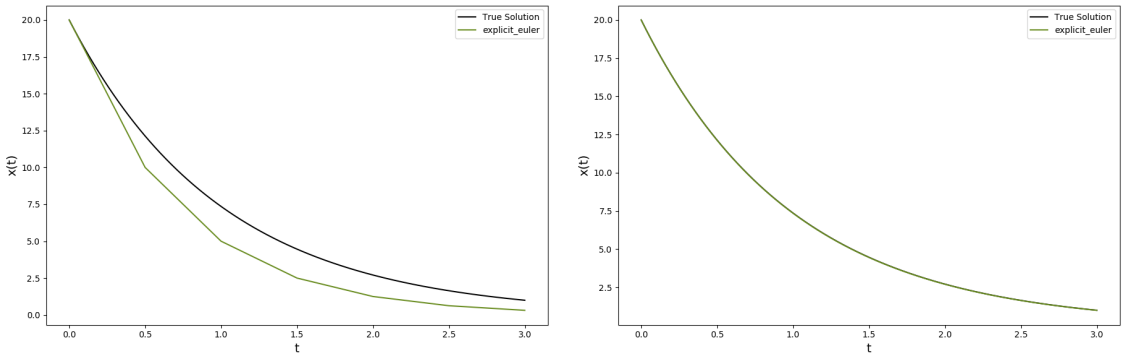


Figure 6: Explicit Euler approximation for large and small step sizes

This demonstrates that the explicit Euler method is greatly dependent on the speed at which varies the solution, and on the adopted step size. Particularly, it shows that even if it is very simple and computationally efficient, it is not suited for stiff real problems, which would require step sizes much too small to be practical to reach adequate accuracy.

This dependence is further illustrated in Figure 7, which clearly shows that the method might even diverge for too large step sizes.

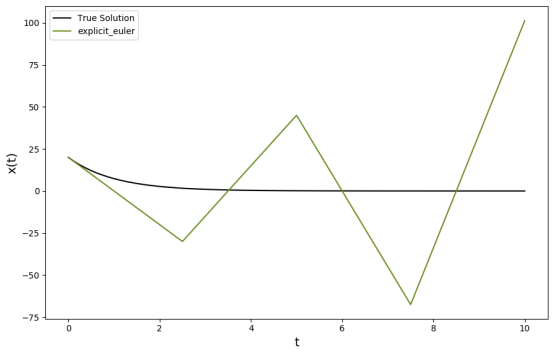


Figure 7: Divergence of the explicit Euler method.

1.3 Python Implementation with Adaptive Step Size

Including a step size controller in the method is a good way to alleviate the aforementioned shortcomings. Controlling the step size during the simulation can be notably achieved using an asymptotic step size controller. Such controller requires the error to be estimated at every iteration to efficiently modify the value of the step size.

The intuitive idea behind the step size controller, is that for a given absolute tolerance *atol* and a relative tolerance *reltol*, a step $k+1$ will be accepted only if the local error l_k at that step is below the threshold

$$l_k < \max_{i=1\ldots D}(atol,\ |x_k|_i reltol) \tag{19}$$

To avoid impacting the performance of the method too drastically, considerations regarding the control of the step size must be local. Thus, the local truncation error will be used by the step size controller to determine whether a step should be accepted or not. An efficient way to estimate the local error is to perform step doubling. From previous iteration $\boldsymbol{x}_{k-1}$, not only the new iteration $\boldsymbol{x}_k$, but also a doubled step with half step size $\frac{h}{2}$ - $\hat{\boldsymbol{x}}_{k-\frac{1}{2}}$, $\hat{\boldsymbol{x}}_k$ - can be computed. Their difference then provides an asymptotic estimate of the local truncation error. To illustrate why, let's consider again test equation,

$$x'(t) = \lambda x(t), \qquad x(0) = x_0 \tag{20}$$

For a step size, h, it follows that:

$$\boldsymbol{x}_k = \boldsymbol{x}_{k-1} + hf(t_{k-1}, \boldsymbol{x}_{k-1}) = \boldsymbol{x}_{k-1}(1+h\lambda)$$

$$\begin{aligned} \hat{\boldsymbol{x}}_{k-\frac{1}{2}} &= \boldsymbol{x}_{k-1} + \frac{h}{2} f(t_{k-1}, \boldsymbol{x}_{k-1}) = \boldsymbol{x}_{k-1}(1+\frac{h}{2}\lambda) \\ \hat{\boldsymbol{x}}_k &= \hat{\boldsymbol{x}}_{k-\frac{1}{2}} + \frac{h}{2} f(t_{k-\frac{1}{2}}, \hat{\boldsymbol{x}}_{k-\frac{1}{2}}) = \boldsymbol{x}_{k-1}(1+\frac{h}{2}\lambda)^2 \end{aligned} \tag{21}$$

And thus, the estimated error e_k will be at each iteration (For insights into the lower equation of 22, see section 3.2)

$$\begin{aligned} |\boldsymbol{e}_k| &= |\hat{\boldsymbol{x}}_k - \boldsymbol{x}_k| = |\boldsymbol{x}_{k-1}|(\frac{h}{2}\lambda)^2 \\ e_k &= \max_{i=1\ldots D} |\boldsymbol{e}_k|_i = \mathcal{O}(h^2) \approx |l_k| \end{aligned} \tag{22}$$

and can then be used to compute

$$r_k = \max_{i=1\ldots D}([\frac{|\boldsymbol{e}_k|}{\max(\text{abstol}, \hat{\boldsymbol{x}}_k \text{reltol})}]_i) \tag{23}$$

which is provides a direct way to to know whether the step should be accepted ($r < 1$) or not.

For a given numerical method which approximates the true solution of an ODE with order p (See again Section 3.2 for details) the local truncation error adopts the asymptotic

behaviour, for small h,

$$l_k = \mathcal{O}(h^{p+1}) = F(\boldsymbol{x}_k)h^{p+1} + \mathcal{O}(h^{p+2}) \approx F(\boldsymbol{x}_k)h^{p+1} \tag{24}$$

Where F is a given function of the step $\boldsymbol{x}_k$. One can then wonder what step size h' would lead to the equality, for a given error tolerance ϵ.

$$\epsilon = F(\boldsymbol{x}_k)h'^{p+1} \tag{25}$$

These two equations result in an effective method to update the step size, with respect to the asymptotic behaviour of the error of the method

$$h' = h(\frac{\epsilon}{l_k})^{\frac{1}{p+1}} \simeq h(\frac{\epsilon}{e_k})^{\frac{1}{p+1}} \tag{26}$$

In practice, for the explicit Euler method, the order is $p = 1$ and the next step size can be determined from the previous step size with the following asymptotic heuristic, whether the step was accepted or not. The notation h' refers to the new step size computed after each trial from the previous step size h, de-correlated from the step k, as several step sizes might be experimented with for a single step.

$$h' = \max(\text{min\_factor}, \min(\sqrt{\frac{\epsilon}{r_{k+1}}}, \text{max\_factor}))h \tag{27}$$

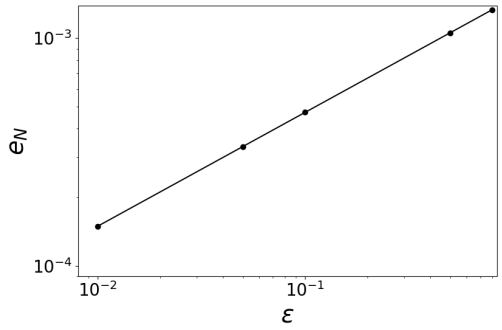
Equation 26 can be understood pretty intuitively. It was indeed seen that $e_{k+1} = \mathcal{O}(h^2)$. Thus if e_{k+1} is above its asymptotic behaviour, then $\frac{h^2}{e_{k+1}} < 1$, resulting in a new step size smaller, and thus a smaller error, closer to what is expected. Conversely, if $\frac{h^2}{e_{k+1}} > 1$, the step size will increase.

This demonstrates a key property of methods with adaptive step sizes. When the solution to be approximated displays very little variation, a large step size will be adopted to efficiently approximate it, and when the solution displays large variations, a small step size will be adopted to generate an accurate approximation.

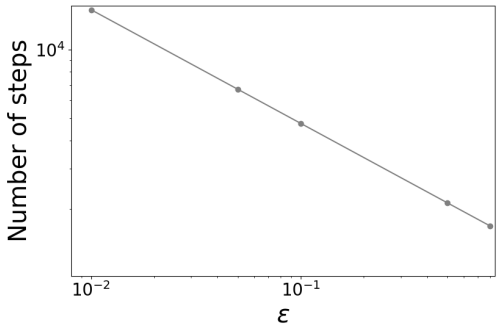
Figure 8 demonstrates the influence of the error tolerance ϵ on the approximation. As it represents the error tolerance for the error of the method, decreasing its value will result in a lower global error e_N, at the cost of more steps, and vice versa. In practice the value $\epsilon = 0.8$ is commonly used as a good trade-off between performance and accuracy.

In Figure 9 is presented the computed approximation for the explicit Euler method with adaptive step size. The same initial configuration was provided than on the left of Figure 6, with an initial step size of 0.5. The right part shows clearly the effect of step size controller. For the first trials, the acceptance criterion, that depends on the estimated error, is too large and the step size decreases incrementally after each non accepted trials. At the third trial (iteration 4), the step size reaches around 0.001 and the steps are finally accepted. The resulting simulation, that can be seen on the left part of Figure 9, is indistinguishable from the true solution.

With this adaptive step size, it becomes impossible to obtain a poor approximation due to an initially large step size. The controller also allows to take advantage of the structure



(a) Evolution of the global error e_N depending on the error tolerance ϵ, caeteris paribus, for the explicit Euler method on the test equation.



(b) Evolution of the number of simulation steps depending on the error tolerance ϵ, caeteris paribus, for the explicit Euler method on the test equation.

Figure 8: Influence of the error tolerance ϵ on the simulation.

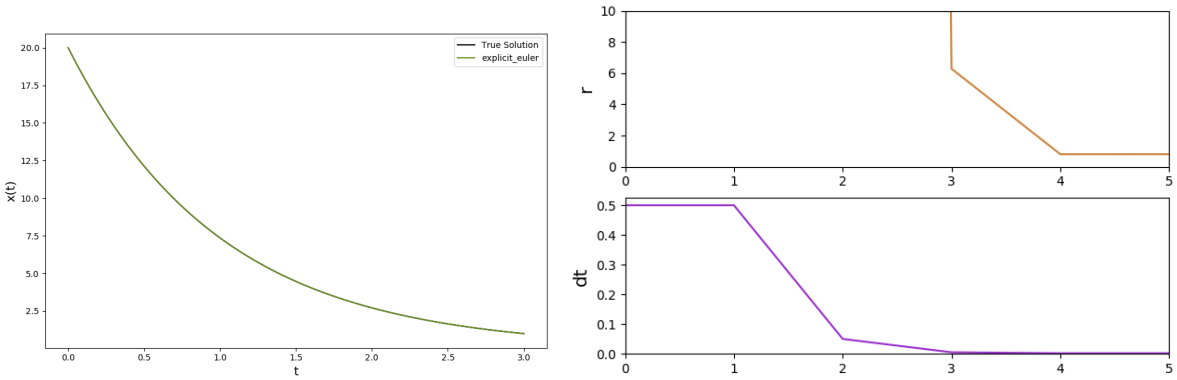


Figure 9: Explicit Euler approximation with adaptive step size

of the solution; if some sections of the solution present low variations, the step size will gradually re-increase and thus speed up the simulation process. Nevertheless, the adaptive step size will not allow the explicit Euler method to be effective to approximate solutions with great variations along their whole trajectory, as it will still require an unpractical step size value to stay close to the true solution.

```
def ode_solver(f, J, t0, tf, N, x0, adaptive_step_size=False, **kwargs):
    dt = (tf - t0)/N

    T = [t0]
    X = [x0]

    kwargs, abstol, reltol, epstol, facmax, facmin =
   parse_adaptive_step_params(kwargs)

    t = t0
    x = x0
```

```
11
12          while t < tf:
13              if (t + dt > tf):
14                  dt = tf - t
15
16              f_eval = f(t, x, **kwargs)
17
18              accept_step = False
19              while not accept_step:
20                  # Take step of size dt
21                  x_1 = x + dt*f_eval
22
23                  # Take two steps of size dt/2
24                  x_hat_12 = x + (dt/2)*f_eval
25                  t_hat_12 = t + (dt/2)
26                  f_eval_12 = f(t_hat_12, x_hat_12, **kwargs)
27                  x_hat = x_hat_12 + (dt/2)*f_eval_12
28
29                  # Error estimation
30                  e = np.abs(x_1 - x_hat)
31                  r = np.max(np.abs(e) / np.maximum(abstol, np.abs(x_hat)*
    reltol))
32
33                  accept_step = (r <= 1)
34                  if accept_step:
35                      t = t + dt
36                      x = x_hat
37
38                      T.append(t)
39                      X.append(x)
40
41                  dt = np.maximum(facmin, np.minimum(np.sqrt(epstol/r),
    facmax)) * dt
42
43      T = np.array(T)
44      X = np.array(X)
45
46      return X, T
```

Listing 2: Implementation of the explicit Euler method with adaptive step size

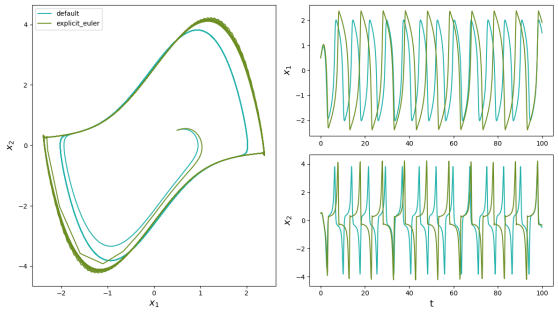
The error tolerance ϵ can be seen as the ideal step size that the method should aim for, if possible. Its value should therefore be set while keeping in mind the properties of the function. For example, it was seen on Figure 7 that for a step size too large, the method is divergent on the test equation. Setting for the controller such a large value to aim for can potentially create a divergence as well.

1.4 Test on the Van der Pol Problem

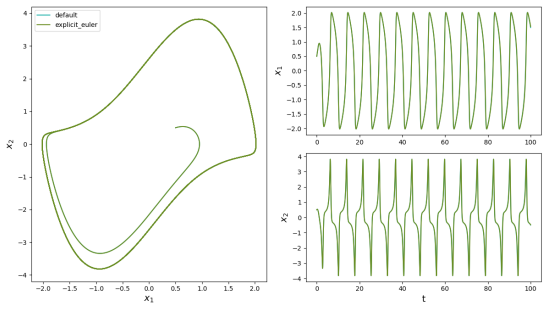
1.4.1 For $\mu = 2$

The explicit Euler method was first tested on the Van der Pol problem with parameter $\mu = 2$. As mentioned when the problem was described, for small values of μ, the problem is not so stiff and we can reasonably expect the method to perform not too poorly in its approximation.

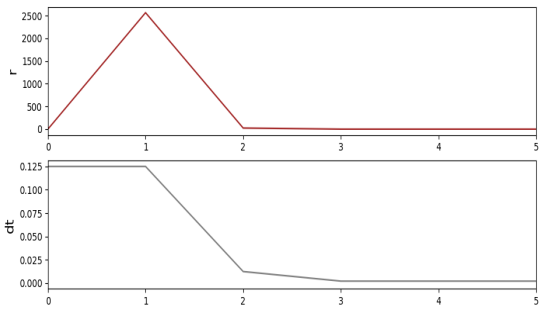
First, the method was applied with a large step size, $h = 0.125$ for a time interval of $[0, 100]$ with initial coordinates $\boldsymbol{x}_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}$. The following figures display the results of the different simulations



(a) Explicit Euler method on the Van der Pol problem. $\mu = 2$ and $h = 0.125$



(b) Explicit Euler method with adaptive step size on the Van der Pol problem. $\mu = 2$ and $h_0 = 0.125$



(c) Evolution of error and step size for first few iterations of the step size controller. Van der Pol problem, $\mu = 2$ and $h_0 = 0.125$

Figure 10: Simulation results for large step size on the Van der Pol problem. $\mu = 2$ and $h = 0.125$

The accuracy of the standard explicit Euler method is pretty poor, as can be seen on Figure 10, the too large step size results quickly in a divergence with the default ODE solver, which can be expected to provide an efficient numerical approximation. Nevertheless, adopting a step size controller allows a clear improvement in performance. The step size rapidly decreases to a more sensible value and the resulting simulation is indistinguishable from the output of the default solver.

The method was also ran with a much smaller step size, $h = 0.001$, resulting in the simulation displayed in 11

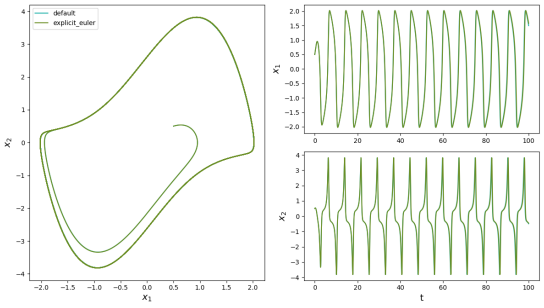


Figure 11: Explicit Euler method on the Van der Pol problem. $\mu = 2$ and $h = 0.001$

It thus seems that with a smaller step size, the method is able to correctly simulate the solution for the given parameter.

1.4.2 For $\mu = 12$

The case $\mu = 12$ was then investigated to study how the increased variation in the problem influenced the resulting numerical simulation. Here again, the simulation was first ran with an initially large step size, $h = 0.025$. It is interesting to note that for some larger step size values, the solution would diverge completely and result in an overflow.

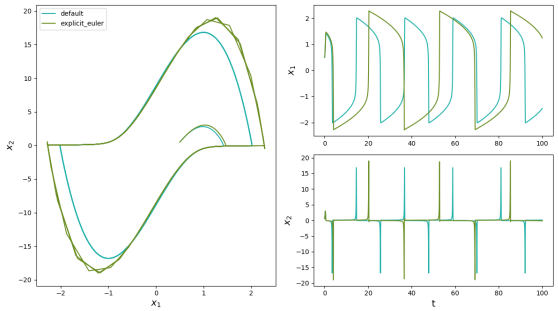
The increased stiffness of the problem is apparent (Figure 12) given the inability of the resulting approximation to handle the peaks of the oscillator, which correspond to areas with more variations in the solution. Here again, introducing the step size controller allows for a much more precise solution. It is interesting to note the variations observed in the step size; cyclically, the step size is allowed to rise to speed up the simulation, probably corresponding to ares of low variation (like the slopes leading to the peaks of the oscillator).

The considerably smaller step size $h = 0.001$ was again used and resulted in simulation displayed in 13.

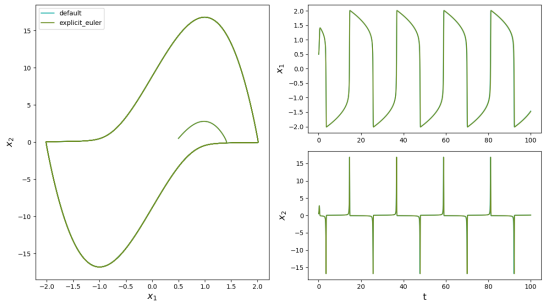
Contrary to what was seen for $\mu = 2$, here decreasing the step size to the same level is not enough to reach a satisfactory approximation. After a few cycles, a clear divergence appears between the default ODE solver's solution and the explicit Euler method's solution. This is a good demonstration of why this method should not be used for stiff problems.

1.5 Test on the A-CSTR Problem

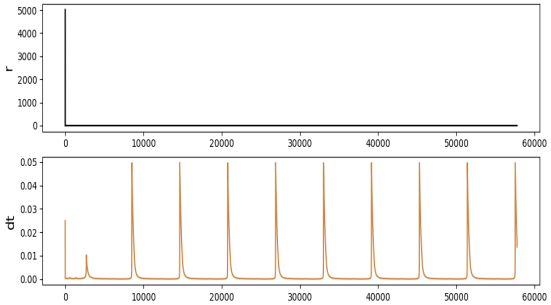
Similarly to what was done for the Van der Pol problem, the explicit Euler method as executed for both the CSTR 1D and 3D models with first a large step size, $h = 0.175$, and later with a considerably smaller step size $h = 0.00035$.



(a) Explicit Euler method on the Van der Pol problem. $\mu = 12$ and $h = 0.025$



(b) Explicit Euler method with adaptive step size on the Van der Pol problem. $\mu = 12$ and $h_0 = 0.025$



(c) Evolution of error and step size for the step size controller. Van der Pol problem, $\mu = 12$ and $h_0 = 0.025$

Figure 12: Simulation results for large step size on the Van der Pol problem. $\mu = 12$ and $h = 0.025$

1.5.1 CSTR 1D

Figure 14 displays the result of the simulation with a large step size. It is obvious that the standard explicit Euler method cannot handle the quick variations of the temperature T when it increases sharply, at around 15 minutes. The version with the adaptive step size displays again a greatly more accurate solution, but contrary as what was seen on the Van der Pol oscillator, cannot approximate perfectly the evolution of the temperature. Indeed, around minute 23, it displays some lag in its adaptation to the new temperature threshold. This is probably because the solution right before the step did not display much variation, and consequently, the step size controller increased the step size before the step.

Nevertheless, decreasing the step size to $h = 0.00035$ allows the explicit Euler method to approximate accurately the solution as can be seen in Figure 15

1.5.2 CSTR 3D

The CSTR 3D model displays the same behaviour than its 1D counterpart, as can be seen in Figure 16 and 17.

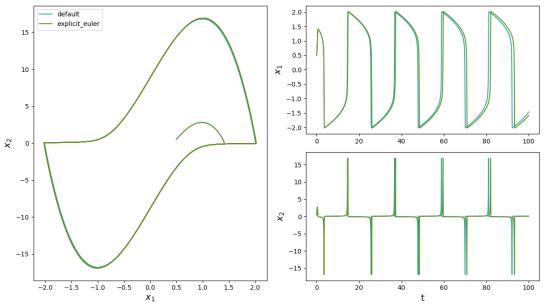
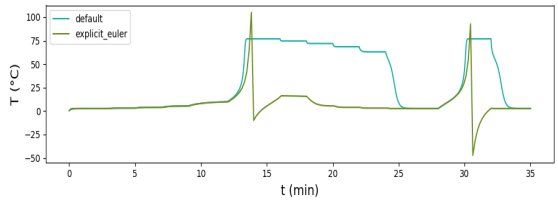
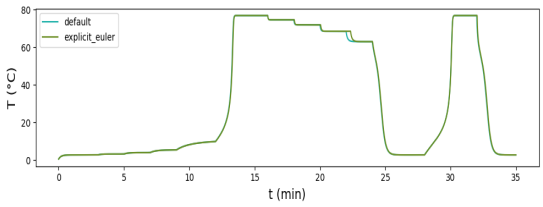


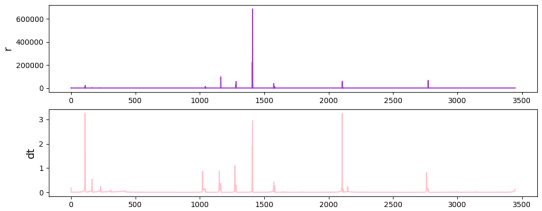
Figure 13: Explicit Euler method on the Van der Pol problem. $\mu = 12$ and $h = 0.001$



(a) Explicit Euler method on the CSTR 1D problem. $h = 0.175$



(b) Explicit Euler method with adaptive step size on the CSTR 1D problem. $h_0 = 0.175$



(c) Evolution of error and step size for the step size controller. CSTR 1D problem, $h_0 = 0.175$

Figure 14: Simulation results for large step size on the CSTR 1D problem. $h = 0.175$

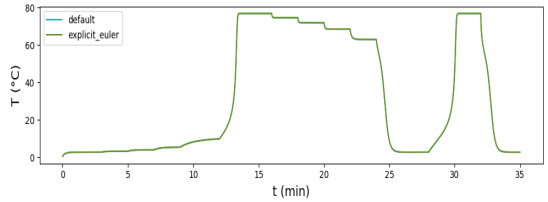
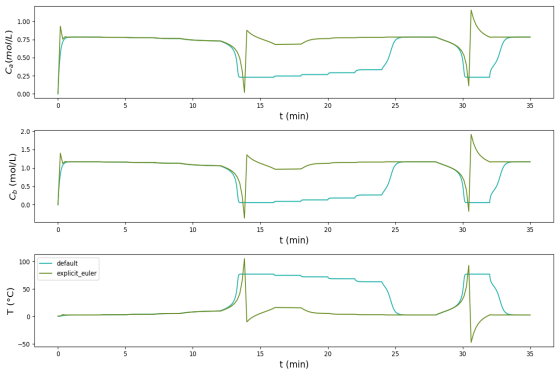
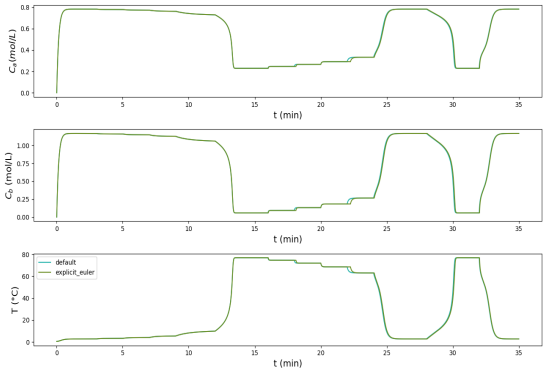


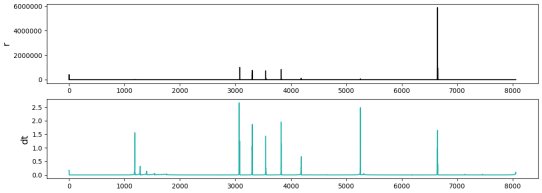
Figure 15: Explicit Euler method on the CSTR 1D problem. $h = 0.00035$



(a) Explicit Euler method on the CSTR 3D problem. $h = 0.175$



(b) Explicit Euler method with adaptive step size on the CSTR 3D problem. $h_0 = 0.175$



(c) Evolution of error and step size for the step size controller. CSTR 3D problem, $h_0 = 0.175$

Figure 16: Simulation results for large step size on the CSTR 3D problem. $h = 0.175$

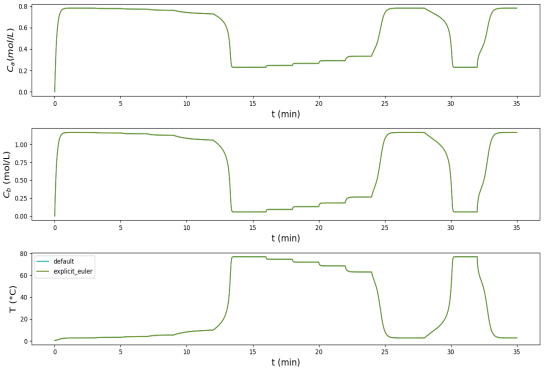


Figure 17: Explicit Euler method on the CSTR 3D problem. $h = 0.00035$

2 Implicit ODE solver

2.1 Description of the Implicit Euler algorithm

Again, coming back to the general form of an IVP:

$$\boldsymbol{x}'(t) = f(t, \boldsymbol{x}(t)), \qquad \boldsymbol{x}(t_0) = \boldsymbol{x}_0 \tag{28}$$

Assuming again that $\boldsymbol{x}$ is infinitely differentiable, it is possible to take its Taylor series, evaluated this time in t_k with respect to t_{k+1} :

$$\boldsymbol{x}(t_k) = \sum_{n=0}^{\infty} \frac{\boldsymbol{x}^{(n)}(t_{k+1})}{n!}(t_k - t_{k+1})^n = \sum_{n=0}^{\infty} \frac{\boldsymbol{x}^{(n)}(t_k)}{n!}(-h)^n \tag{29}$$

And by restricting the series to its first two terms, n=0 and n=1, the following approximation arises:

$$\boldsymbol{x}(t_k) \simeq \boldsymbol{x}(t_{k+1}) - h\boldsymbol{x}'(t_{k+1}) = \boldsymbol{x}(t_k) - hf(t_{k+1}, \boldsymbol{x}(t_{k+1})) \tag{30}$$

And therefore

$$\boldsymbol{x}(t_{k+1}) \simeq \boldsymbol{x}(t_k) + hf(t_{k+1}, \boldsymbol{x}(t_{k+1})) \tag{31}$$

Obtaining iteratively $\boldsymbol{x}(t_{k+1})$ here from $\boldsymbol{x}(t_k)$ is not as straightforward as for the explicit Euler method. Indeed, $\boldsymbol{x}(t_{k+1})$ depends on the value of $f(t_{k+1}, \boldsymbol{x}(t_{k+1}))$, i.e, on itself. Let R: $\mathbb{R}^n \to \mathbb{R}^n$ be defined as:

$$R(\boldsymbol{x}(t_{k+1})) = \boldsymbol{x}(t_{k+1}) - \boldsymbol{x}(t_k) - hf(t_{k+1}, \boldsymbol{x}(t_{k+1})) = 0 \tag{32}$$

Determining $\boldsymbol{x}(t_{k+1})$ is therefore equivalent to determining the multivariate root of the residual R. In most cases, no analytical root can be found, so here again a numerical and iterative approximation must be used. Considering that $\boldsymbol{x}_{k+1}^{[i]}$ is the ith iteration of the approximation of $\boldsymbol{x}(t_{k+1})$, the first order Taylor expansion of R in $\boldsymbol{x}_{k+1}^{[i+1]}$ with respect to $\boldsymbol{x}_{k+1}^{[i]}$ is:

$$R(\boldsymbol{x}_{k+1}^{[i+1]}) \simeq R(\boldsymbol{x}_{k+1}^{[i]}) + \frac{\partial R}{\partial \boldsymbol{x}}(\boldsymbol{x}_{k+1}^{[i]})(\boldsymbol{x}_{k+1}^{[i+1]} - \boldsymbol{x}_{k+1}^{[i]}) = 0 \tag{33}$$

Noting $\boldsymbol{x}_{k+1}^{[i+1]} = \boldsymbol{x}_{k+1}^{[i]} + \Delta\boldsymbol{x}$, and solving, for $\Delta\boldsymbol{x}$

$$-\frac{\partial R}{\partial \boldsymbol{x}}(\boldsymbol{x}_{k+1}^{[i]})\Delta\boldsymbol{x} = R(\boldsymbol{x}_{k+1}^{[i]}) \tag{34}$$

thus constitutes an update rule for the approximation of $\boldsymbol{x}(t_{k+1})$. It is called Newton's method.

Therefore, here again if the evaluation of f and its Jacobian is possible for every t_k and if $\boldsymbol{x}(t_0) = \boldsymbol{x}_0$ is known, a numerical method to solve the ODE appears. Iteratively,

it is possible to determine an approximation $\boldsymbol{x}_{k+1} \simeq \boldsymbol{x}(t_{k+1})$ from the previous iteration $\boldsymbol{x}_k \simeq \boldsymbol{x}(t_k)$ using Newton's method. It is called the implicit Euler method.

2.2 Python Implementation

```
def newtons_method(f, J, t, x, dt, x_init, tol, max_iters, **kwargs):
    k = 0
    x_iter = x_init
    t_iter = t + dt
    f_eval = f(t_iter, x_iter, **kwargs)
    J_eval = J(t_iter, x_iter, **kwargs)

    R = x_iter - dt*f_eval - x
    I = np.eye(x.shape[0])
    while ((k < max_iters) & (norm(R, np.inf) > tol)):
        k += 1
        M = I - dt*J_eval
        dx_iter = np.linalg.solve(M,R)
        x_iter -= dx_iter
        f_eval = f(t_iter, x_iter, **kwargs)
        J_eval = J(t_iter, x_iter, **kwargs)
        R = x_iter - dt*f_eval - x

    return x_iter
```

Listing 3: Implementation of Newton's Method for the implicit Euler method

```
def ode_solver(f, J, t0, tf, N, x0, adaptive_step_size=False, **kwargs):

    dt = (tf - t0)/N

    T = [t0]
    X = [x0]

    kwargs, newtons_tol, newtons_max_iters = parse_newtons_params(kwargs)


    for k in range(N):

        # Use explicit form to start off newton
        f_eval = f(T[-1], X[-1], **kwargs)
        x_init = X[-1] + dt*f_eval
        X.append(newtons_method(f, J, T[-1], X[-1], dt, x_init,
   newtons_tol, newtons_max_iters, **kwargs))
        T.append(T[-1] + dt)

    T = np.array(T)
    X = np.array(X)

    return X, T
```

Listing 4: Implementation of the implicit Euler method

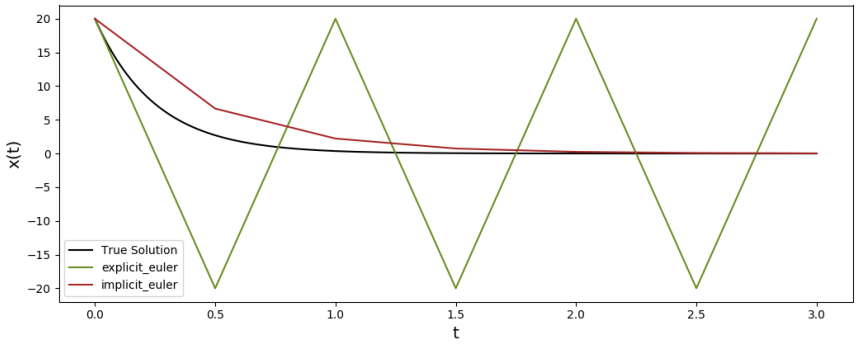


Figure 18: TODO

2.3 Python Implementation with Adaptive Step Size

```
def ode_solver(f, J, t0, tf, N, x0, adaptive_step_size=False, **kwargs):

    dt = (tf - t0)/N

    T = [t0]
    X = [x0]
    controllers = {
        'r': [0],
        'dt': [dt]
    }

    kwargs, newtons_tol, newtons_max_iters = parse_newtons_params(kwargs)

    kwargs, abstol, reltol, epstol, facmax, facmin =
   parse_adaptive_step_params(kwargs)

    t = t0
    x = x0

    while t < tf:
        if (t + dt > tf):
            dt = tf - t

        f_eval = f(t, x, **kwargs)
        accept_step = False
        while not accept_step:
            # Take initial guess step of size dt
            x_1_init = x + dt*f_eval
            x_1 = newtons_method(f, J, t+dt, x, dt, x_1_init, newtons_tol,
    newtons_max_iters, **kwargs)

            # Take two steps of size dt/2
            x_hat_12_init = x + (dt/2)*f_eval
            t_hat_12 = t + (dt/2)
            x_hat_12 = newtons_method(f, J, t_hat_12, x, dt/2,
```

```
    x_hat_12_init, newtons_tol, newtons_max_iters, **kwargs)
34
35              f_eval_12 = f(t_hat_12, x_hat_12, **kwargs)
36              x_hat_init = x_hat_12 + (dt/2)*f_eval_12
37              x_hat = newtons_method(f, J, t+dt, x_hat_12, dt/2, x_hat_init,
     newtons_tol, newtons_max_iters, **kwargs)
38
39              # Error estimation
40              e = np.abs(x_1 - x_hat)
41              r = np.max(np.abs(e) / np.maximum(abstol, np.abs(x_hat)*reltol
    ))
42
43              accept_step = (r <= 1)
44              if accept_step:
45                  t = t + dt
46                  x = x_hat
47
48                  T.append(t)
49                  X.append(x)
50              controllers['dt'].append(dt)
51              controllers['r'].append(r)
52
53              dt = np.maximum(facmin, np.minimum(np.sqrt(epstol/r), facmax))
     * dt
54
55     T = np.array(T)
56     X = np.array(X)
57     controllers['dt'] = np.array(controllers['dt'])
58     controllers['r'] = np.array(controllers['r'])
59
60     return X, T, controllers
```

Listing 5: Implementation of the implicit Euler method with adaptive step size

2.4 Test on the Van der Pol Problem

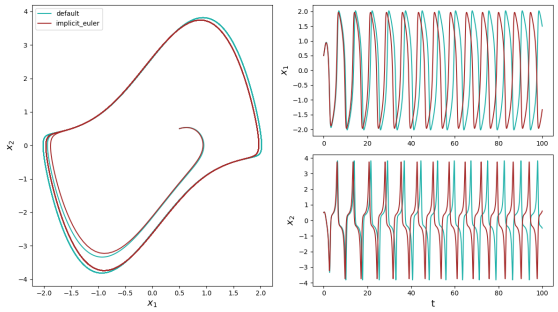
2.4.1 For $\mu = 2$

2.4.2 For $\mu = 12$

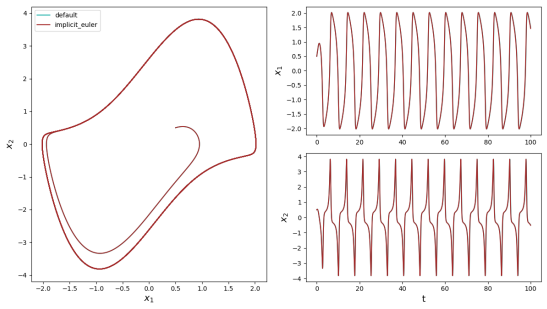
2.5 Test on the A-CSTR Problem

2.5.1 CSTR 1D

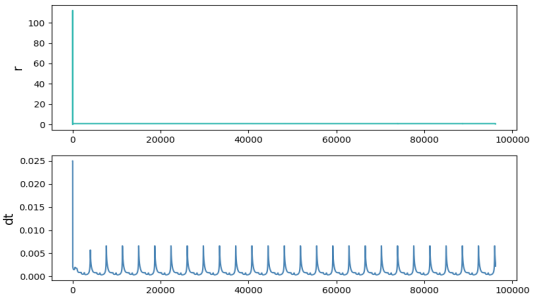
2.5.2 CSTR 3D



(a) Implicit Euler method on the Van der Pol problem. $\mu = 2$ and $h = 0.025$



(b) Implicit Euler method with adaptive step size on the Van der Pol problem. $\mu = 2$ and $h_0 = 0.025$



(c) Evolution of error and step size for the step size controller. Van der Pol problem, $\mu = 2$ and $h_0 = 0.025$

Figure 19: Simulation results for large step size on the Van der Pol problem. $\mu = 2$ and $h = 0.025$

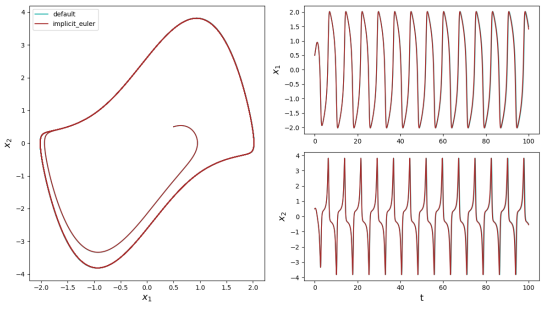
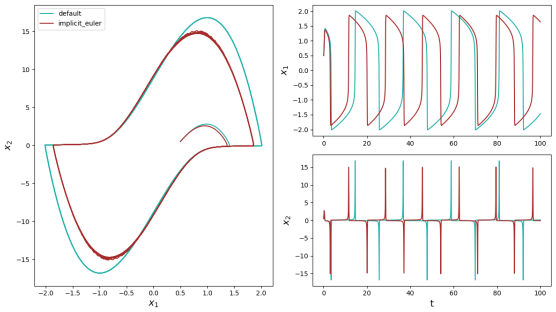
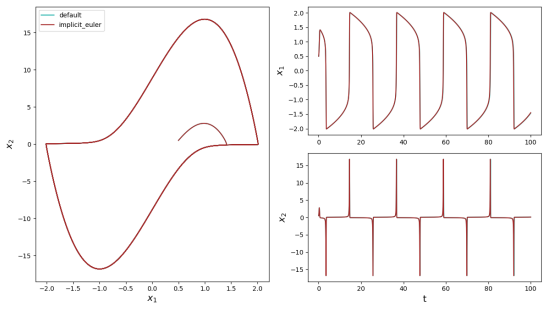


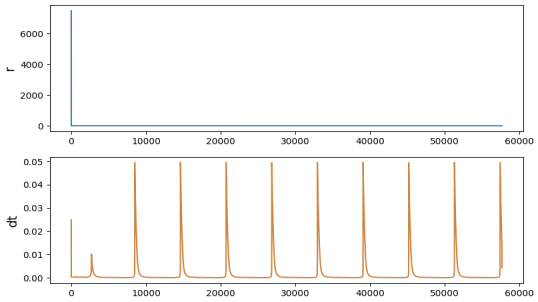
Figure 20: Implicit Euler method on the Van der Pol problem. $\mu = 2$ and $h = 0.001$



(a) Implicit Euler method on the Van der Pol problem. $\mu = 12$ and $h = 0.025$



(b) Implicit Euler method with adaptive step size on the Van der Pol problem. $\mu = 12$ and $h_0 = 0.025$



(c) Evolution of error and step size for the step size controller. Van der Pol problem, $\mu = 12$ and $h_0 = 0.025$

Figure 21: Simulation results for large step size on the Van der Pol problem. $\mu = 12$ and $h = 0.025$

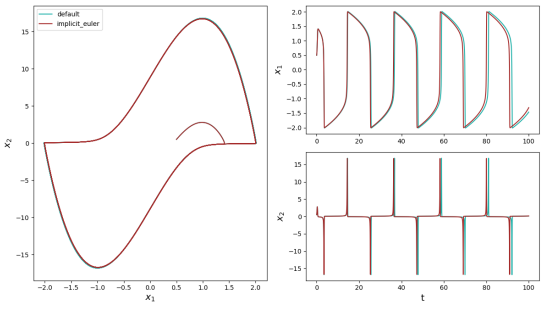
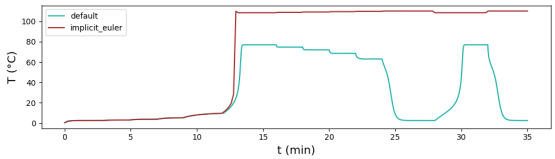
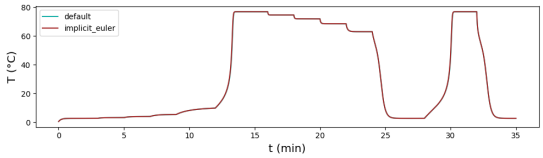


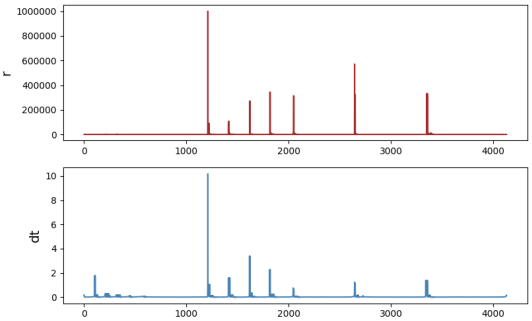
Figure 22: Implicit Euler method on the Van der Pol problem. $\mu = 12$ and $h = 0.001$



(a) Implicit Euler method on the CSTR 1D problem. $h = 0.175$



(b) Implicit Euler method with adaptive step size on the CSTR 1D problem. $h_0 = 0.175$



(c) Evolution of error and step size for the step size controller. CSTR 1D problem, $h_0 = 0.175$

Figure 23: Simulation results for large step size on the CSTR 1D problem. $h = 0.175$

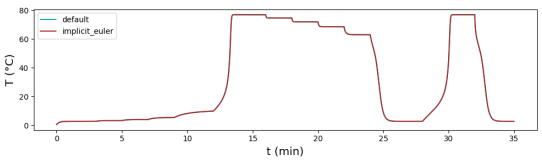
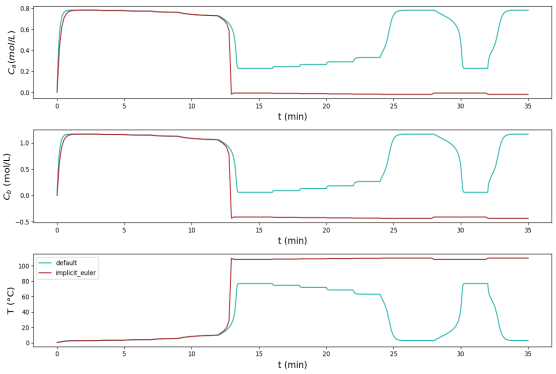
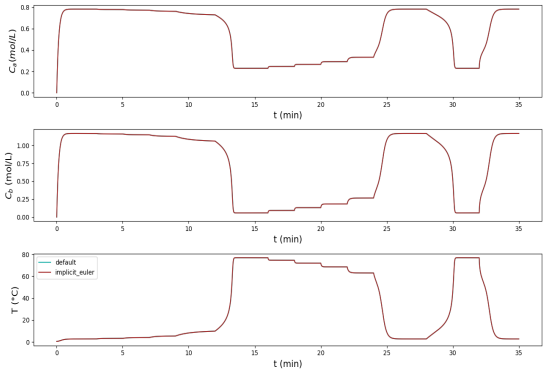


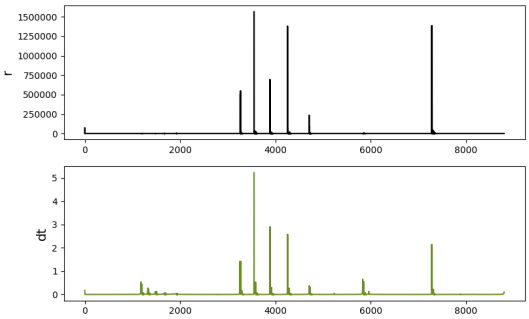
Figure 24: Implicit Euler method on the CSTR 1D problem. $h = 0.00035$



(a) Implicit Euler method on the CSTR 3D problem. $h = 0.175$



(b) Implicit Euler method with adaptive step size on the CSTR 3D problem. $h_0 = 0.175$



(c) Evolution of error and step size for the step size controller. CSTR 3D problem, $h_0 = 0.175$

Figure 25: Simulation results for large step size on the CSTR 3D problem. $h = 0.175$

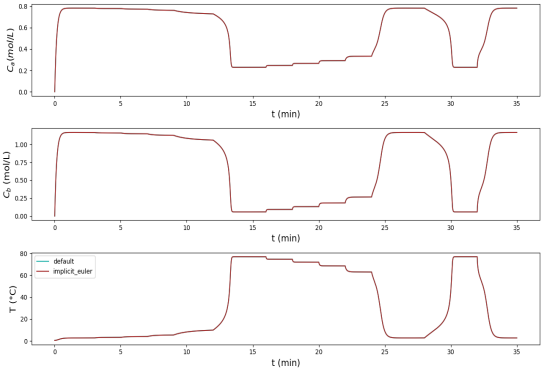


Figure 26: Implicit Euler method on the CSTR 3D problem. $h = 0.00035$

3 Test equation for ODEs

3.1 Analytical Solution

Let's now consider the scalar test equation

$$x'(t) = \lambda x(t), \qquad x(0) = x_0 \tag{35}$$

In the general case, it is immediately verifiable that

$$x(t) = x_0 e^{\lambda t} \tag{36}$$

is an analytical solution to this test equation. Indeed $x'(t) = \lambda x_0 e^{\lambda t} = \lambda x(t)$.

3.2 Local and Global Truncation Errors

For any numerical method, it is inevitable that at each step (for non trivial problems), some error will separate the computed approximation and the true solution. The error at each step is called the local truncation error. Along the simulation the local errors will accumulate into a global truncation error. The study of truncation errors for a given method reveals fundamental properties and allow the distinction between better and poorer methods. They are defined as follows:

For $t_k \in I$, x_k the numerical approximation of $x(t_k)$ the exact analytical solution, and $x_{k-1}(t_k)$ the analytical solution that has the numerical approximation $x_{k-1} \simeq x(t_{k-1})$ as initial value, the **local error** l_k is defined as:

$$l_k = x_k - x_{k-1}(t_k) \tag{37}$$

For $t_k \in I$, and x_k the numerical approximation of $x(t_k)$ the exact analytical solution, the **global error** e_k is defined as:

$$e_k = x_k - x(t_k) \tag{38}$$

Figure 27 illustrates the definition of the truncation errors. In black is displayed $x(t)$, the true exact analytical solution of the IVP. In green is shown the approximation computed through the explicit Euler method with a rough step size ($h = 0.5$), x_0, x_1 and x_2. In blue is depicted the analytical solution, $x_1(t)$ that would have t_1 and x_1 as starting points.

The red errors l_1 and l_2 correspond to the local truncation error, i.e the difference between the approximation and the true solution that uses the previous point of the approximation as starting point. And the purple error e_2 corresponds to the global error at t_2, which is the final difference between the approximation and the true solution.

The local and global truncation error are generally expressed asymptotically as a function of the step size. Indeed they therefore allow to determine what accuracy to expect from a numerical method for a given step size.

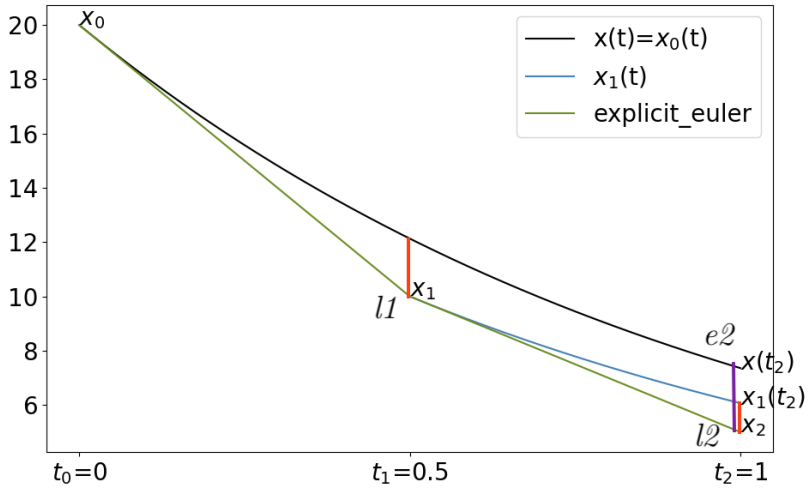


Figure 27: Illustration of the truncation errors for the explicit Euler method

3.3 Error Expression for Euler Methods

3.3.1 Explicit Euler Method

Local Truncation Error

To compute the local truncation error for the explicit Euler method at step t_k, it is first required to expressed the two terms used x_k and $x_{k-1}(t_k)$:

$$\begin{cases} x_k = (1+\lambda h)x_{k-1} \\ x_{k-1}(t_k) = x_{k-1}e^{\lambda(t_k - t_{k-1})} = x_{k-1}e^{\lambda h} \simeq x_{k-1}(1+\lambda h + \frac{(\lambda h)^2}{2} + \frac{(\lambda h)^3}{6} + ...) \end{cases} \tag{39}$$

Which thus allow to determine an asymptotic value for the local truncation error at step k, l_k:

$$l_k = x_k - x_{k-1}(t_k) \simeq -x_{k-1}(\frac{(\lambda h)^2}{2} + \frac{(\lambda h)^3}{6} + ...) = \mathcal{O}(h^2) \tag{40}$$

Global Truncation Error

The top part of Equation 39 indicates that the terms x_0, x_1, ..., x_k of the approximation computed through the explicit Euler method follow a geometric progression of reason $1+h\lambda$. Its kth term can thus be expressed as

$$x_k = x_0(1+h\lambda)^k \tag{41}$$

The expression $(1+x)^k$ can be expressed in 0 with its Taylor expansion:

$$(1+x)^k = \sum_{n=0}^{\infty} \binom{k}{n} x^n \tag{42}$$

For $t_0 = 0$, the final time step of the simulation t_f is directly proportional to the step size $t_f = Nh$ where N is the number of steps. The global error at the end of the simulation will therefore be:

$$\begin{aligned} e_N &= x_N - x(t_f) \\ &= x_0((1+h\lambda)^N - e^{Nh\lambda}) \\ &= x_0((1 + Nh\lambda + \frac{N(N-1)}{2}(h\lambda)^2) - (1 + Nh\lambda + \frac{(Nh\lambda)^2}{2})) + \mathcal{O}(h^3) \\ &= -x_0\frac{N}{2}(h\lambda)^2 + \mathcal{O}(h^3) \\ &= -x_0\frac{t_f\lambda^2}{2}h + \mathcal{O}(h^2) \end{aligned} \tag{43}$$

The leading term of the global truncation error evolves with h. As expected, this is characteristic of an order 1 method.

Furthermore, for the explicit Euler method in a general setting the global error can be bounded under certain assumptions by an expression of order 1 in h. Let's consider the Taylor expansions of the true solution $x(t)$ and of the RHS function f, truncated after order 2 for $x(t)$ and 1 in the second variable of f.

$$\begin{cases} x(t) \simeq x(T) + x'(T)(T-t) + x''(T)\frac{(T-t)^2}{2} \\ f(T,x) \simeq f(T,X) + \frac{\partial f}{\partial x}(T,X)(x-X) \end{cases} \tag{44}$$

The above approximation, applied for $t = t_k$ and $T = t_{k-1}$, $x = x(t_{k-1})$ and $X = x_{k-1}$ results in

$$\begin{aligned} x(t_k) &= x(t_{k-1}) + x'(t_{k-1})(t_k - t_{k-1}) + x''(t_{k-1})\frac{(t_k - t_{k-1})^2}{2} \\ &= x(t_{k-1}) + f(t_{k-1}, x(t_{k-1}))h + x''(t_{k-1})\frac{h^2}{2} \\ &= x(t_{k-1}) + (f(t_{k-1}, x_{k-1}) + \frac{\partial f}{\partial x}(t_{k-1}, x_{k-1})(x(t_{k-1}) - x_{k-1}))h + x''(t_{k-1})\frac{h^2}{2} \\ &= x(t_{k-1}) + (f(t_{k-1}, x_{k-1}) - \frac{\partial f}{\partial x}(t_{k-1}, x_{k-1})e_{k-1})h + x''(t_{k-1})\frac{h^2}{2} \end{aligned} \tag{45}$$

And thus,

$$\begin{aligned} e_k &= x_k - x(t_k) \\ &= x_{k-1} - x(t_{k-1}) + \frac{\partial f}{\partial x}(t_{k-1}, x_{k-1})e_{k-1}h - x''(t_{k-1})\frac{h^2}{2} \\ &= e_{k-1}(1 + \frac{\partial f}{\partial x}(t_{k-1}, x_{k-1}))h - x''(t_{k-1})\frac{h^2}{2} \end{aligned} \tag{46}$$

This expression is quite interesting in itself, as it provides an intuition on the accumulation of errors along the simulation. Using the definition of f given by the IVP, it can be expanded further.

$$x''(t) = \frac{dx'(t)}{dt} = \frac{df}{dt}(t, x(t)) \tag{47}$$

f is a multivariate function, for which both arguments display a dependency in time. To get the derivative of f with regard to time, the multivariate chain rule must be applied. It goes as follows

$$\frac{df}{dx}(g_1(x), g_2(x)) = \frac{dg_1(x)}{dx}\frac{\partial f}{\partial g_1}(g_1(x), g_2(x)) + \frac{dg_2(x)}{dx}\frac{\partial f}{\partial g_2}(g_1(x), g_2(x)) \tag{48}$$

And therefore implies that

$$\frac{df}{dt}(t, x(t)) = \frac{\partial f}{\partial t}(t, x(t)) + \frac{dx(t)}{dt}\frac{\partial f}{\partial x}(t, x(t)) = \frac{\partial f}{\partial t}(t, x(t)) + f(t, x(t))\frac{\partial f}{\partial x}(t, x(t)) \tag{49}$$

Let's assume that the second derivative of x is bounded (by continuity on a closed support for example). It then admits a maximum M. Let's also assume that f is Lipchitz continous on its second argument for a constant K. These two assumptions can be expressed as

$$\begin{cases} K = \max |\frac{\partial f}{\partial x}(t, x(t))| \\ M = \max |\frac{\partial f}{\partial t}(t, x(t)) + f(t, x(t))\frac{\partial f}{\partial x}(t, x(t)) \end{cases} \tag{50}$$

Resulting, by triangular inequality, in the following:

$$|e_k| \leq (1 + Kh)|e_{k-1}| + M\frac{h^2}{2} \tag{51}$$

Which finally yields the bound in O(h) for the global error of the explicit Euler method in the general case:

$$|e_k| \leq \frac{M}{2K}(e^{Kt_k} - 1)h \tag{52}$$

3.3.2 Implicit Euler Method

Local Truncation Error

To compute the local truncation error for the implicit Euler method at step t_k, it is first required to expressed the two terms used x_k and $x_{k-1}(t_k)$:

$$\begin{cases} x_k = x_{k-1} + \lambda h x_k \Rightarrow x_k = \frac{1}{1-\lambda h} x_{k-1} \\ x_{k-1}(t_k) = x_{k-1}(1 + \lambda h + \frac{(\lambda h)^2}{2} + \frac{(\lambda h)^3}{6} + ...) \end{cases} \tag{53}$$

The development of the inverse function into its Taylor expansion, $\frac{1}{1-\lambda h} = 1 + \lambda h + (\lambda h)^2 + (\lambda h)^3 + ...$, yields an asymptotic value for the local truncation error at step k, l_k:

$$l_k = x_k - x_{k-1}(t_k) = x_{k-1}(\frac{(\lambda h)^2}{2} + \frac{5(\lambda h)^3}{6} + ...) = \mathcal{O}(h^2) \tag{54}$$

Global Truncation Error

The top part of Equation 53 indicates that the terms x_0, x_1, ..., x_k of the approximation computed through the implicit Euler method follow a geometric progression of reason $\frac{1}{1-h\lambda}$. Its kth term can thus be expressed as

$$x_k = x_0(\frac{1}{1-h\lambda})^k \tag{55}$$

The expression $(\frac{1}{1-x})^k$ can be expressed in 0 with its Taylor expansion

$$(\frac{1}{1-x})^k = \sum_{n=0}^{\infty} \binom{-k}{n} (-x)^n \tag{56}$$

For $t_0 = 0$, the final time step of the simulation t_f is directly proportional to the step size $t_f = Nh$, where N is the number of steps. The global error at the end of the simulation will therefore be:

$$\begin{aligned} e_N &= x_N - x(t_f) \\ &= x_0((\frac{1}{1-h\lambda})^N - e^{Nh\lambda}) \\ &= x_0((1 + Nh\lambda + \frac{N(N+1)}{2}(h\lambda)^2) - (1 + Nh\lambda + \frac{(Nh\lambda)^2}{2})) + \mathcal{O}(h^3) \\ &= x_0\frac{N}{2}(h\lambda)^2 + \mathcal{O}(h^3) \\ &= x_0\frac{t_f\lambda^2}{2}h + \mathcal{O}(h^2) \end{aligned} \tag{57}$$

As expected, the accumulation of errors along the simulation result in a global order with leading term in $\mathcal{O}(h)$. The implicit Euler method has thus an accuracy order of 1.

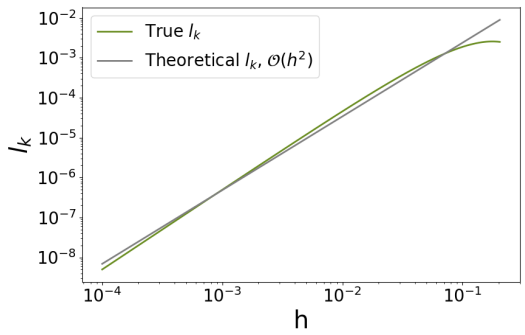
3.4 Plot of Local Error

Figure 28 displays the local truncation error l_k[2] for both Euler methods as a function of the step size h. They both behave as expected, as they match closely the expected asymptotic behaviour of $l_k = \mathcal{O}(h^2)$ displayed in grey.

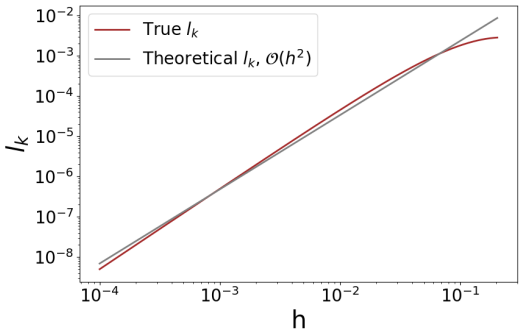
The asymptotic behaviour of the local error of the method informs about its order. Indeed, the accuracy of a numerical method is said to be of order p if $l_k = \mathcal{O}(h^{p+1})$.

These plots, supplemented by equations 40 and 54 therefore prove that both Euler methods are of order 1.

[2]taken at $k = 10$



(a) Local truncation error for the explicit Euler method

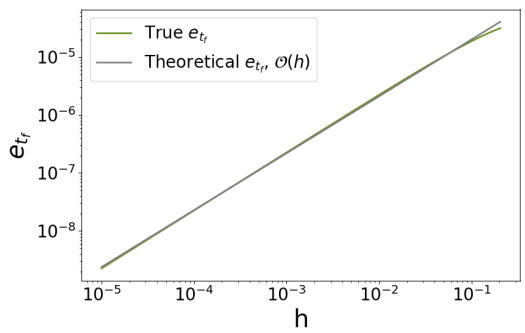


(b) Local truncation error for the implicit Euler method

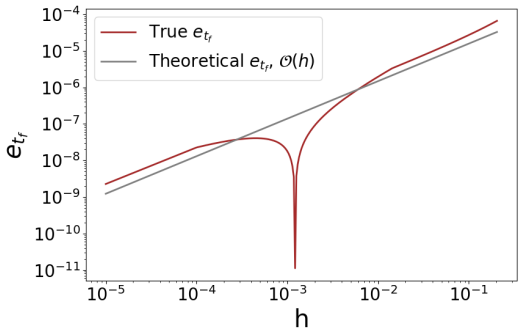
Figure 28: Local truncation errors for Euler methods

3.5 Plot of Global Error

Figure 29 displays the global truncation error e_{t_f} for both Euler methods as a function of the step size h. The explicit Euler method behaves perfectly as expected, as it lies very close to the theoretical asymptotic bound, $\mathcal{O}(h)$. On the other hand, the implicit Euler method displays a clear drop in its final global error for around $h = 0.0013$, indicating that there is a sweet spot for its performance on such problem. It also asymptotically follows the expected behaviour of $\mathcal{O}(h)$.



(a) Global truncation error for the explicit Euler method



(b) Global truncation error for the implicit Euler method

Figure 29: Global truncation errors for Euler methods

3.6 On Stability

The test equation provides a way to study the stability of numerical ODE solvers. For the test equation

$$x'(t) = \lambda x(t), \qquad x(0) = 1, \qquad \lambda \in \mathbb{C}, \qquad Re(\lambda) < 0 \tag{58}$$

the analytical solution $x(t)$ admits $\lim_{t\to\infty} x(t) = 0$.

Naturally, the numerical method's approximation x_k is expected to converge as well $\lim_{k\to\infty} x_k = 0$. If it does, then it is stable.

The region of stability for the method can thus be defined as:

$$\mathcal{D} = \{(h, \lambda) \in \mathbb{R} \times \mathbb{C} \mid \lim_{k\to\infty} x_k = 0\} \tag{59}$$

Explicit Euler Method

In the case of the explicit Euler method, for a given k, the following holds:

$$\begin{aligned} x_{k+1} &= x_k + hf(t_k, x_k) \\ &= x_k(1 + \lambda h) \end{aligned} \tag{60}$$

The series of terms of the approximation x_0, x_1 ... x_{k+1} therefore follows a geometric progression of reason $(1 + \lambda h)$ and its kth term is therefore

$$x_k = x_0(1 + \lambda h)^k = (1 + \lambda h)^k \tag{61}$$

This allows to infer that the region of stability for the explicit Euler method is:

$$\mathcal{D}_{ee} = \{(h, \lambda) \in \mathbb{R} \times \mathbb{C} \mid |1 + \lambda h| < 1\} \tag{62}$$

Implicit Euler Method

Similarly, for the implicit Euler method, for a given k, the following holds:

$$\begin{aligned} x_{k+1} &= x_k + hf(t_k, x_{k+1}) \\ &= x_k + \lambda h x_{k+1} \\ &= x_k(\frac{1}{1 - \lambda h}) \end{aligned} \tag{63}$$

Here again the terms of the approximation follow a geometric progression of reason $(\frac{1}{1-\lambda h})$ and therefore:

$$x_k = x_0(\frac{1}{1 - \lambda h})^k = (\frac{1}{1 - \lambda h})^k \tag{64}$$

which implies that the region of stability for the implicit Euler method is:

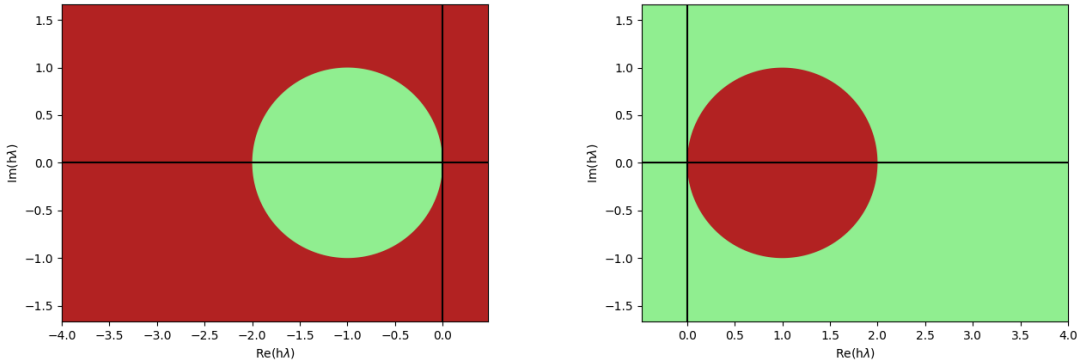
$$\mathcal{D}_{ie} = \{(h, \lambda) \in \mathbb{R} \times \mathbb{C} \mid |1 - \lambda h| > 1\} \tag{65}$$

Figure 30 displays the region of stability of both the explicit Euler method (left) and implicit Euler method (right) in the complex plane $\lambda h \in \mathbb{C}$. The green region is the stable region and the red region is the unstable region.

Such plots can be used to guess whether a method is A-stable or not. A method is

said to be **A-stable** is its region of absolute stability $\mathcal{D}$ contains the entire left complex half plane, i.e $\mathcal{D} \in \{(h, \lambda) \in \mathbb{R} \times \mathbb{C} \mid Re(\lambda h) < 0\}$

The explicit Euler method is therefore not A-stable, while its implicit counterpart is. It is furthermore important to note that A-stability gives the fundamental property that for any A-stable method, the step size should be chosen on accuracy considerations and not on stability considerations. This is illustrated in Figure 18, where even though the step size is very large, the implicit Euler method does not diverge from the true solution while the explicit Euler method does.



(a) Stability region for the explicit Euler method

(b) Stability region for the implicit Euler method

Figure 30: Stability regions for Euler methods

4 Solvers for SDEs

The term of Stochastic Differential Equations (SDEs) describes any form of differential equation where at least one of the terms involved is a stochastic process. A common form of SDE is obtained when adding a standard gaussian perturbation on the right hand side of any ODE. In this case, and in a general form, the SDE can be written as:

$$d\boldsymbol{x}(t) = f(t, \boldsymbol{x}(t))dt + g(\boldsymbol{x}(t))d\boldsymbol{w}(t) \tag{66}$$

Where g is a diffusion function and $\boldsymbol{w}$ is a Wiener process.

In the general case, when the interval [a,b] is divided in N sub-intervals with boundaries t_0, t_1, ..., t_{N-1} of constant length $h = t_{k+1} - t_k$ the following holds:

$$\begin{aligned} \int_a^b f(t)dt &= \lim_{N\to\infty} \sum_{i=0}^{N-1} hf(t_i) \\ \int_a^b f(t)d\boldsymbol{w}(t) &= \lim_{N\to\infty} \sum_{i=0}^{N-1} f(t_i)\Delta\boldsymbol{w}_i \ , \ \Delta\boldsymbol{w}_i \sim \mathcal{N}(0, h) \quad \text{(Ito's Integral)} \end{aligned} \tag{67}$$

In particular, one can hope that for a and b sufficiently close to each other, as would be t_k and t_{k+1} in a numerical approximation, the following approximation would not deviate too much from truth:

$$\begin{aligned} \int_{t_{k+1}}^{t_k} f(t)dt &\simeq hf(t_c) \ , \ t_c \in [t_k, t_{k+1}] \\ \int_{t_{k+1}}^{t_k} f(t)d\boldsymbol{w}(t) &\simeq f(t_k)\Delta\boldsymbol{w}_k \ , \ \Delta\boldsymbol{w}_k \sim \mathcal{N}(0, h) \end{aligned} \tag{68}$$

Therefore providing a direct way to iteratively approximate solution of such SDEs, as the integration of equation 66 between t_k and t_{k+1} yields:

$$\boldsymbol{x}_{k+1} - \boldsymbol{x}_k = hf(t_c, \boldsymbol{x}(t_c)) + g(\boldsymbol{x}_k)\Delta\boldsymbol{w}_k \ , \ t_c \in [t_k, t_{k+1}] \ , \ \Delta\boldsymbol{w}_k \sim \mathcal{N}(0, h) \tag{69}$$

Please note that for this section, and the next, which detail numerical methods for SDEs, the interface of the solvers is slightly modified to account for the stochastic processes and is detailed in Listing 6

```
def sde_solver(f, J, g, dW, t0, tf, N, x0, **kwargs):
    ...
```

Listing 6: Interface for SDE solver

4.1 Multivariate Standard Wiener Process

A standard Wiener Process defined over [0, T] is a continuous random variable $\boldsymbol{w}(t)$ that satisfies:

- $\boldsymbol{w}(0) = \mathbf{0}$
- $\forall (s,t) \in [0,T]^2$, $s < t$, $\boldsymbol{w}(t) - \boldsymbol{w}(s) \sim \mathcal{N}(0, t-s)$
- $\forall (s,t,u) \in [0,T]^3$, $s < t < u$, the increment $\boldsymbol{w}(u) - \boldsymbol{w}(t)$ is independent of past values $\boldsymbol{w}(s)$

It can be discretized into $\boldsymbol{W}$ which is defined as follows, for a step h

- $\boldsymbol{W}_0 = 0$
- $\boldsymbol{W}$ has independent gaussian increments: $\boldsymbol{W}_{k+1} - \boldsymbol{W}_k \sim \mathcal{N}(0, h)$

It is therefore pretty straightforward to implement such process. A random generator can directly provide $N \times k$ gaussian samples[3], and a cumulative sum can then directly generate $\boldsymbol{W}_0$, $\boldsymbol{W}_1$, ..., $\boldsymbol{W}_N$. Such implementation can be found in Listing 7

```
def wiener_process(T, N, dims=2):
    W = np.zeros((N, dims))
    dt = T / N
    dW = np.random.normal(loc=0, scale=dt, size=(N-1, dims))
    W[1:, :] = np.cumsum(dW, axis=0)

    # adapt size of dW array
    dW = np.vstack((np.zeros((1, dims)), dW))

    return dW, W
```

Listing 7: Implementation of a multivariate Wiener process

The previous implementation allows the generation of Figure 31.

4.2 Explicit-Explicit Method

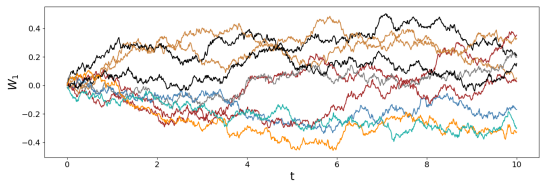
For the Explicit-Explicit method, both the non-stochastic and the stochastic parts of the RHS of the iteration equation are explicit. This means that equation 69 is applied for $t_c = t_k$. For $\Delta w_k \sim \mathcal{N}(0, h)$, the iteration equation can therefore be defined as:

$$\boldsymbol{x}_{k+1} - \boldsymbol{x}_k = h f(t_k, \boldsymbol{x}_k) + g(\boldsymbol{x}_k) \Delta w_k \tag{70}$$

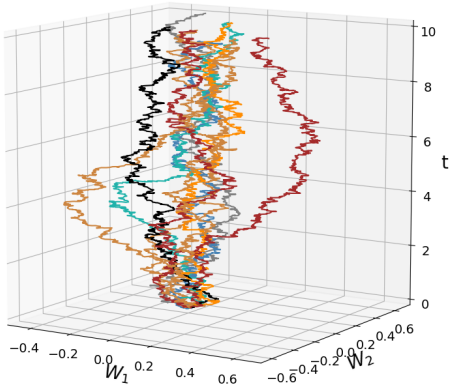
The implementation of such method can be directly drawn from the implementation of the explicit Euler method. Only the shochastic terms $g(\boldsymbol{x}_k)\Delta w_k$ to each iteration, as can be seen in Listing 8

```
def sde_solver(f, J, g, dW, t0, tf, N, x0, **kwargs):
    dt = (tf - t0) / N

    T = [t0]
    X = [x0]
```

[3] k is here the dimension of the process



(a) 10 Realisations of a 1D Wiener process



(b) 10 Realisations of a 2D Wiener process

Figure 31: Realisations of Wiener processes

```
    for k in range(N):
        f_eval = f(T[-1], X[-1], **kwargs)
        g_eval = g(T[-1], X[-1], **kwargs)
        X.append(X[-1] + dt * f_eval + dW[k] * g_eval)
        T.append(T[-1] + dt)

    T = np.array(T)
    X = np.array(X)

    return X, T
```

Listing 8: Implementation of the explicit-explicit SDE solver

4.3 Implicit-Explicit Method

On the other hand, when equation 69 is applied for $t_c = t_{k+1}$, the non-stochastic term of its RHS becomes implicit as for the implicit Euler method. For $\Delta w_k \sim \mathcal{N}(0, h)$, its iteration equation can thus be written:

$$\boldsymbol{x}_{k+1} - \boldsymbol{x}_k = hf(t_{k+1}, \boldsymbol{x}_{k+1}) + g(\boldsymbol{x}_k)\Delta w_k \tag{71}$$

Similarly as for the implicit Euler equation, the dependency between $\boldsymbol{x}_{k+1}$ and $f(\boldsymbol{x}_{k+1})$ needs to be handled with an approximation method, such as Newton's method. Nevertheless, the stochastic term must be accounted for in the residual equation

$$R_k(\boldsymbol{x}_{k+1}) = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k - hf(t_{k+1}, \boldsymbol{x}_{k+1}) - g(\boldsymbol{x}_k)\Delta w_k = 0 \tag{72}$$

For this reason, the newton's method implementation was slightly modified to accept g and Δw_k as arguments. Listing 9 details such implementation.

```
def sde_newtons_method(f, J, psi, t, x, dt, x_init, tol, max_iters, **
    kwargs):
    k = 0
    x_iter = x_init
    t_iter = t + dt
    f_eval = f(t_iter, x_iter, **kwargs)
    J_eval = J(t_iter, x_iter, **kwargs)

    R = x_iter - dt*f_eval - psi
    I = np.eye(x.shape[0])
    while ((k < max_iters) & (norm(R, np.inf) > tol)):
        k += 1
        M = I - dt*J_eval
        dx_iter = np.linalg.solve(M,R)
        x_iter -= dx_iter
        f_eval = f(t_iter, x_iter, **kwargs)
        J_eval = J(t_iter, x_iter, **kwargs)
        R = x_iter - dt*f_eval - psi

    return x_iter
```

Listing 9: Implementation of Newton's method for the implicit-explicit SDE solver

The rest of the implementation is identical to the implementation detailed earlier of the implicit Euler method as proves Listing 10

```
def sde_solver(f, J, g, dW, t0, tf, N, x0, **kwargs):
    dt = (tf - t0) / N

    T = [t0]
    X = [x0]

    kwargs, newtons_tol, newtons_max_iters = parse_newtons_params(kwargs)

    for k in range(N):
        # Use explicit form to start off newton
        f_eval = f(T[-1], X[-1], **kwargs)
        g_eval = g(T[-1], X[-1], **kwargs)
        psi = X[-1] + g_eval*dW[k]
        x_init =  dt * f_eval + psi
        X.append(sde_newtons_method(f, J, psi, T[-1], X[-1], dt, x_init,
    newtons_tol, newtons_max_iters, **kwargs))
        T.append(T[-1] + dt)

    T = np.array(T)
    X = np.array(X)

    return X, T
```

Listing 10: Implementation of the implicit-explicit SDE solver

4.4 Test on the Van der Pol Problem

4.4.1 For $\mu = 2$

4.4.2 For $\mu = 12$

4.5 Test on the A-CSTR Problem

4.5.1 CSTR 1D

4.5.2 CSTR 3D

5 Test equation for SDEs

5.1 Analytical Solution

Let's consider the geometric Brownian motion described by the following equation

$$dx(t) = \lambda x(t)dt + \sigma x(t)d\omega(t) = x(t)(d\omega(t) + \lambda dt) \tag{73}$$

Ito's formula can be used to derive its analytical solution. For any twice continously differentiable f: $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$, its multivariate Taylor expansion is:

$$df = \frac{\partial f(t,x)}{\partial t}dt + \frac{\partial f(t,x)}{\partial x}dx + \frac{1}{2}\frac{\partial^2 f(t,x)}{\partial x^2}dx^2 \tag{74}$$

To avoid any later confusion in its terms, let's write $\partial_1 f$ the partial derivative of f with regard to its first argument, $\partial_2 f$ its partial derivative with regard to its second argument and $\partial_{2,2} f$ its second partial derivative with regard to its second argument. The expansion can be thus re-written:

$$df = \partial_1 f(z_1, z_2)dz_1 + \partial_2 f(z_1, z_2)dz_2 + \frac{1}{2}\partial_{2,2} f(z_1, z_2)dz_2^2 \tag{75}$$

If this expansion is then applied for $f(t, x(t)) = log(x(t))$, with $x(t)$ solution of the geometric Brownian motion equation, the following holds

$$df(t, x(t)) = \partial_1 f(t, x(t))dt + \partial_2 f(t, x(t))dx(t) + \frac{1}{2}\partial_{2,2} f(t, x(t))dx(t)dx(t) \tag{76}$$

The function f does not depend on its first argument, allowing the simplification

$$df(t, x(t)) = \partial_2 f(t, x(t))dx(t) + \frac{1}{2}\partial_{2,2} f(t, x(t))dx(t)dx(t) \tag{77}$$

Finally, as $\partial_2 f(t, x(t)) = \frac{1}{x(t)}$ and $\partial_{2,2} f(t, x(t)) = -\frac{1}{x(t)}^2$, Equation 73 can be inserted into 77 to yield:

$$\begin{aligned} d\log(x(t)) &= \frac{dx(t)}{x(t)} - \frac{1}{2}\frac{dx(t)dx(t)}{x(t)^2} \\ &= (\sigma d\omega(t) + \lambda dt) - \frac{1}{2}(\sigma d\omega(t) + \lambda dt)^2 \end{aligned} \tag{78}$$

$(\sigma d\omega(t) + \lambda dt)^2$ can be developed into $(\sigma^2 d\omega(t)^2 + 2\sigma\lambda d\omega(t)dt + \lambda^2 dt^2)$. In the limit $dt \to 0$, the terms dt^2 and $d\omega(t)dt$ become negligible in comparison to $d\omega(t)^2 = \mathcal{O}(dt)$ (due to the quadratic variance of the Wiener process). Leading to the following simplification:

$$d\log(x(t)) = \sigma d\omega(t) + (\lambda - \frac{\sigma^2}{2})dt \tag{79}$$

Finally, the integration of such expression, for a time interval [0, t],

$$\int_{x_0}^{x(t)} d\log(x(t)) = \int_{\omega_0}^{\omega(t)} \sigma d\omega(t) + \int_0^t (\lambda - \frac{\sigma^2}{2})dt \tag{80}$$

yields, because, by definition $\omega_0 = 0$,

$$\log(x(t)) = \log(x_0) + \sigma\omega(t) + (\lambda - \frac{\sigma^2}{2})t \tag{81}$$

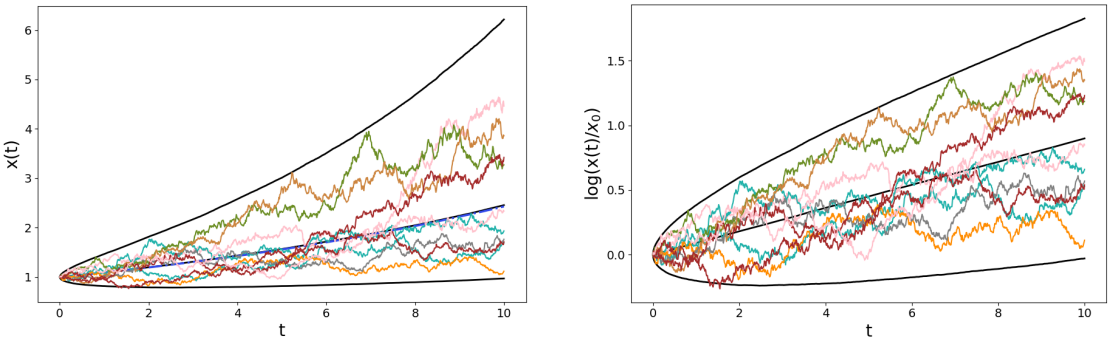
Finally providing an analytical solution to the geometric Brownian motion equation.

$$x(t) = x_0 e^{(\lambda - \frac{\sigma^2}{2})t + \sigma\omega(t)} \tag{82}$$

5.2 Comparison of the Numerical Solution to the Analytical Solution

For comparisons of the analytical solution to numerical methods, the following parameters were used: $\lambda = 0.1$, $\sigma = 0.15$, $x_0 = 1$ and $N = 1000$ (Number of simulation points)

Figure 32 displays in colour 10 realisations of the analytical solution of the geometric Brownian motion. It allows to get an understanding of what realisations of such process look like. Additionally, were added on black the mean and mean ± 1.96 standard deviation computed over 10 000 realisations.



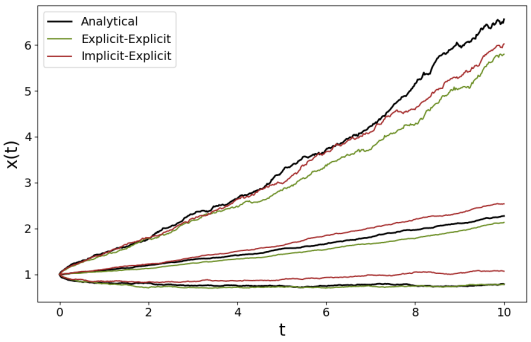
(a) 10 Realisations of the analytical solution of the test equation for SDEs. The blue central dotted line is the solution of the equation without any stochastic term

(b) 10 Realisations of the analytical solution of the test equation for SDEs. Logarithmic values

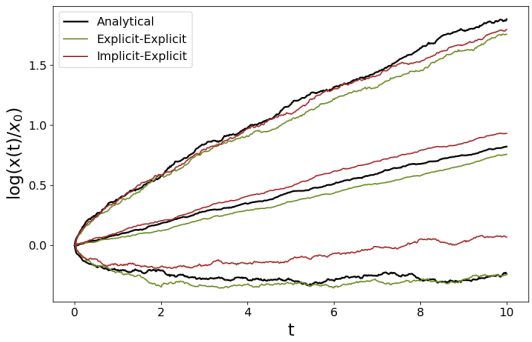
Figure 32: 10 Realisations of the analytical solution of the test equation for SDEs. The black lines correspond to the mean averaged over 10000 realisations, and the 95% confidence interval around it

Figure 33, compares the mean and the mean ± 1.96 of the analytical solution with their counterparts from the two implemented numerical solutions, the explicit-explicit method (green) and implicit-explicit method (red). The top row only uses 100 realisations to compute the resulting trajectories. They clearly display more variation than the bottom row which

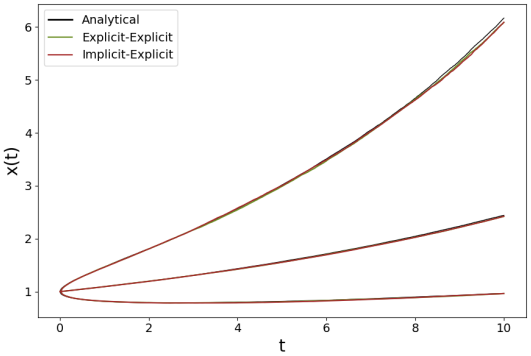
uses 10 000 realisations. For the bottom row, the estimates are so close from each other that the resulting trajectories are almost indistinguishable from each other, thus indicating good performance from the numerical methods.



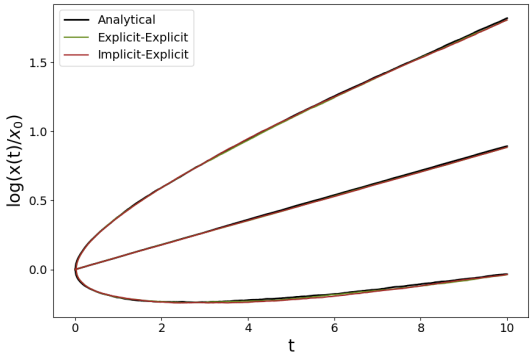
(a) Comparison of the analytical solution against the numerical methods for 100 realisations



(b) Comparison of the analytical solution against the numerical methods for 100 realisations. Logarithmic values



(c) Comparison of the analytical solution against the numerical methods for 10 000 realisations

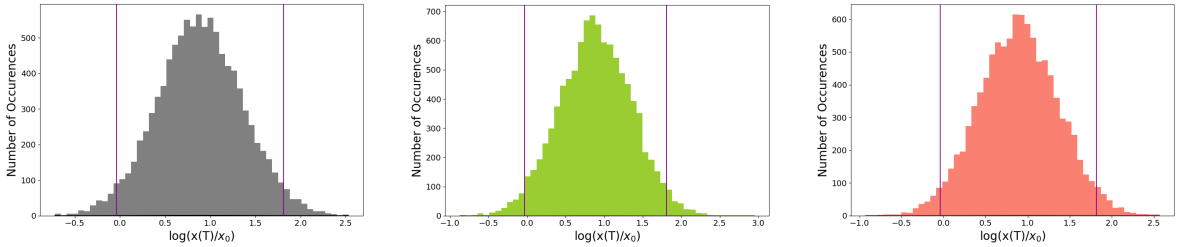


(d) Comparison of the analytical solution against the numerical methods for 10 000 realisations. Logarithmic values

Figure 33

5.3 Distribution of the Final State of the Numerical Solution

For the 10 000 realisations mentioned in the previous part, it is interesting to look at the distribution of the final states $x(T)$ of the analytical solution and of the numerical methods. Figure 34 display such distributions in the logarithmic space as they are expected to follow a normal distribution in their logarithmic form. This expectation is met by the experimental results as all three resulting distributions follow closely the shape of a bell curve.



(a) Distribution of the final states x(T) for the analytical solution. Logarithmic values

(b) Distribution of the final states x(T) for the explicit-explicit method. Logarithmic values

(c) Distribution of the final states x(T) for the implicit-explicit method. Logarithmic values

Figure 34: Comparison of the distribution of the final state x(T) between the analytical solution and numerical methods. The logarithmic values are used. The magenta vertical lines are the boundaries determined by the mean ±1.96 standard deviation, which correspond to the 95% confidence interval

5.4 Comparison of Moments

Unsurprisingly, the moments of the three resulting distributions of $x(T)$ have very similar moments, as shown by Tables 1 and 2. This explains why the distributions of $x(T)$ are so similar for the analytical solution and the numerical simulations and why the juxtaposed mean and mean ± 1.96 standard deviation curved are almost indistinguishable at their right limit on the bottom row of Figure 33.

Method	Mean	Std
Analytical	2.426	1.603
Explicit-Explicit	2.429	1.599
Implicit-Explicit	2.433	1.606

Table 1: Moments of the distribution of x(T) for both the analytical solution and numerical methods

Method	Mean	Std
Analytical	0.886	0.472
Explicit-Explicit	0.888	0.469
Implicit-Explicit	0.889	0.474

Table 2: Moments of the distribution of x(T) for both the analytical solution and numerical methods. Logarithmic values

5.5 Order of the Explicit-Explicit Method

Figure 35 displays the recorded errors $e_N = |x(T) - x_N|$ averaged over 1000 realisations for the explicit-explicit method. The error behaves entirely as expected, as displaying the same behaviour as the theoretical asymptotical limit of $\mathcal{O}(h)$, displayed in grey. Note that the number of realisations was reduced to 1000 limit the execution time required to compute the set of errors $\{e_N\}$

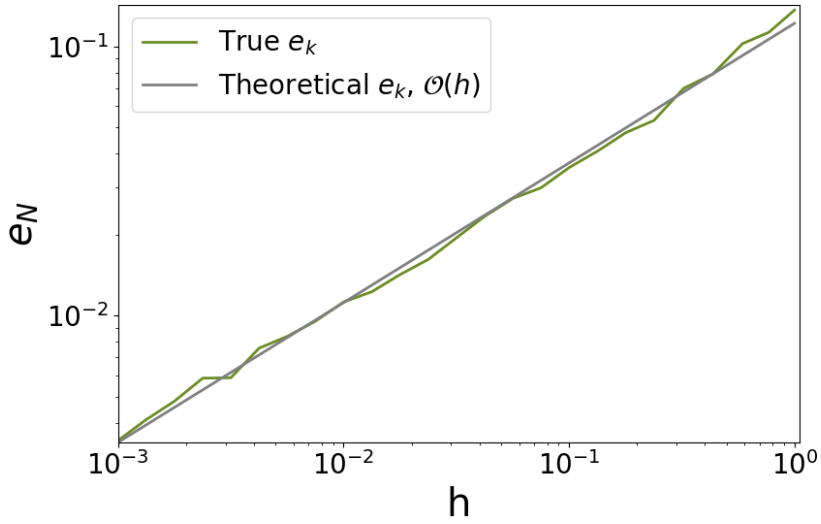


Figure 35: Error of the explicit-explicit method as a function of the step size h. Logarithmic scale

5.6 Order of the Implicit-Explicit Method

Figure 35 displays the recorded errors $e_N = |x(T) - x_N|$ averaged over 1000 realisations for the implicit-explicit method. The error behaves entirely as expected, as displaying the same behaviour as the theoretical asymptotical limit of $\mathcal{O}(h)$, displayed in grey.

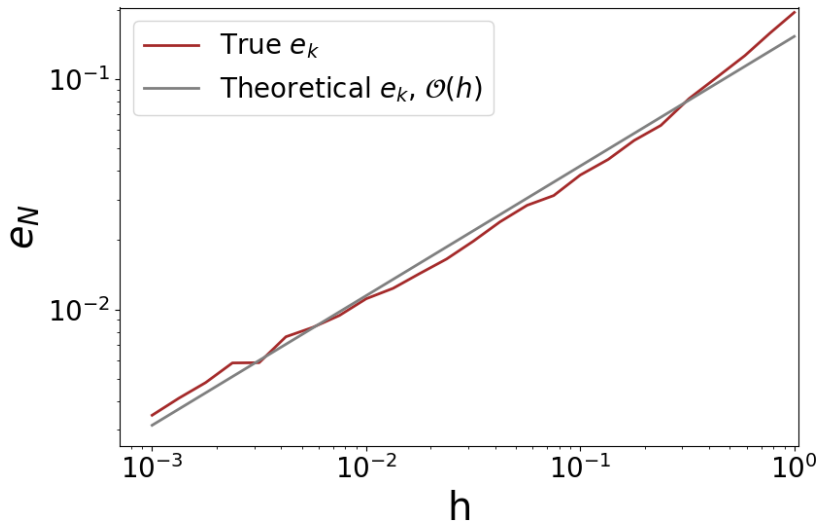


Figure 36: Error of the implicit-explicit method as a function of the step size h. Logarithmic scale

6 Classical Runge-Kutta method with fixed time step size

6.1 Method Description

Runge Kutta methods are stage methods that generalize on the principles previously described with the Euler methods. Coming back to a general IVP:

$$\boldsymbol{x}'(t) = f(t, \boldsymbol{x}(t)), \qquad \boldsymbol{x}(t_0) = \boldsymbol{x}_0 \tag{83}$$

Integrating it between two simulation points, t_{k+1} and t_k, results in:

$$\boldsymbol{x}(t_{k+1}) - \boldsymbol{x}(t_k) = \int_{t_k}^{t_{k+1}} f(t, \boldsymbol{x}(t))dt \tag{84}$$

Approximating $f(t, \boldsymbol{x}(t))$ by $f(t_k, \boldsymbol{x}_k)$ results in the explicit Euler method and with $f(t_{k+1}, \boldsymbol{x}_{k+1})$ in the implicit Euler method. Restricting the approximation of f to a boundary point of the interval $[t_k, t_{k+1}]$ limits the attainable accuracy order of the method to 1. The midpoint approximation $f(t_{k+\frac{1}{2}}, \frac{x_{k+1}+x_k}{2})$ makes use of information about both x_k and x_{k+1} to provide a better approximation for f. In doing so, it reaches an order of accuracy of 2 (see [7] page 328).

It is nevertheless an implicit method. To use intermediary points while keeping the method explicit, it is possible to first apply the explicit update on a intermediary step size and then apply recursively the explicit update on the estimated point. That is what the explicit midpoint method does for example.

$$\begin{aligned} \hat{\boldsymbol{x}}_{k+\frac{1}{2}} &= \boldsymbol{x}_k + \frac{h}{2} f(t_k, \boldsymbol{x}_k) \\ \boldsymbol{x}_{k+1} &= \boldsymbol{x}_k + h f(t_k + \frac{h}{2}, \hat{\boldsymbol{x}}_{k+\frac{1}{2}}) \end{aligned} \tag{85}$$

Figure 37 demonstrates visually how the explicit midpoint method works.

Using the test equation $x'(t) = f(t, x(t)) = \lambda t$, it appears that the explicit midpoint method has an accuracy order of 2. Indeed,

$$\begin{aligned} \hat{x}_{k+\frac{1}{2}} &= x_k + \frac{h}{2}\lambda x_k = x_k(1 + \frac{h}{2}\lambda) \\ x_{k+1} &= x_k + h\lambda \hat{x}_{k+\frac{1}{2}} = x_k(1 + h\lambda + \frac{(h\lambda)^2}{2}) \end{aligned} \tag{86}$$

And therefore,

$$l_{k+1} = x_{k+1} - x_k(t_{k+1}) = x_k(1 + h\lambda + \frac{(h\lambda)^2}{2} - e^{h\lambda}) = \mathcal{O}(h^3) \tag{87}$$

This shows what is so interesting about using intermediary points in the interval. Their use allow to increase the maximum order of accuracy attainable. Runge-Kutta methods (of

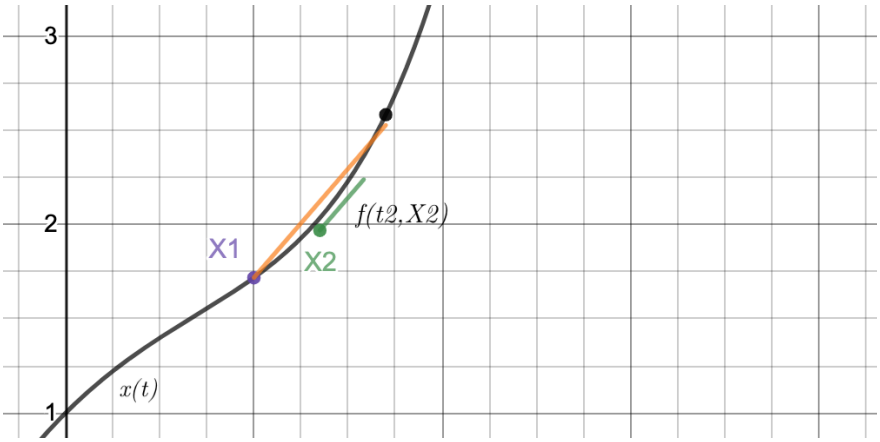


Figure 37: Illustration of basic explicit Runge Kutta method.

order greater than 1) use this mechanism to increase their accuracy.

In their most general form, the Runge-Kutta methods can thus be written as:

$$\begin{aligned} \boldsymbol{x}_{k+1} &= \boldsymbol{x}_k + h \sum_{i=1}^{s} b_i f(t_k + c_i h, \boldsymbol{X}_i) \\ \boldsymbol{X}_i &= \boldsymbol{x}_k + h \sum_{j=1}^{s} a_{ij} f(t_k + c_j h, \boldsymbol{X}_j) \end{aligned} \tag{88}$$

Where the different X_i designate the intermediary points used. The coefficients $\{a_{ij}\}_{(i,j)\in[1,s]^2}$, $\{b_i\}_{i\in[1,s]}$ and $\{c_i\}_{i\in[1,s]}$ can be arranged in a Butcher's tableau

$$\begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \dots & a_{1s} \\ c_2 & a_{21} & a_{22} & \dots & a_{2s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s1} & a_{s2} & \dots & a_{ss} \\ \hline & b_1 & b_2 & \dots & b_s \end{array}$$

The classical Runge-Kutta method (RK4) is a method of order $s = 4$ with the following Butcher tableau:

$$\begin{array}{c|cccc} 0 & & & & \\ \frac{1}{2} & \frac{1}{2} & & & \\ \frac{1}{2} & 0 & \frac{1}{2} & & \\ 1 & 0 & 0 & 1 & \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array}$$

The intermediary steps can therefore be described with the following updates

$$\begin{cases} T_1 = t_k \quad X_1 = x_k \\ T_2 = t_k + \frac{1}{2}h \quad X_2 = x_k + h\frac{1}{2}f(T_1, X_1) \\ T_3 = t_k + \frac{1}{2}h \quad X_3 = x_k + h\frac{1}{2}f(T_2, X_2) \\ T_4 = t_k + h \quad X_4 = x_k + hf(T_3, X_3) \end{cases} \tag{89}$$

Resulting in the following update rule for x_k

$$\begin{cases} t_{k+1} = t_k + h \\ x_{k+1} = x_k + h(\frac{1}{6}f(T_1, X_1) + \frac{1}{3}f(T_2, X_2) + \frac{1}{3}f(T_3, X_3) + \frac{1}{6}f(T_4, X_4)) \end{cases} \tag{90}$$

It is interesting to visualise the different intermediary steps for the method. Figure 38 illustrates the dynamics involved on a simple example. The function $x(t) = e^t - t^2$, which admits along its trajectory the function $f(t, x(t)) = e^t - 2t$ as the RHS of any ODE describing it is shown in black. The RK4 method is applied on a rough step, between $t = 1$ and $t = 1.7$, and in orange is displayed the numerical estimation of the method, which lies very close to the true solution value, despite the large step size. Figure 38 thus provides an intuitive representation how accurate the RK4 method is.

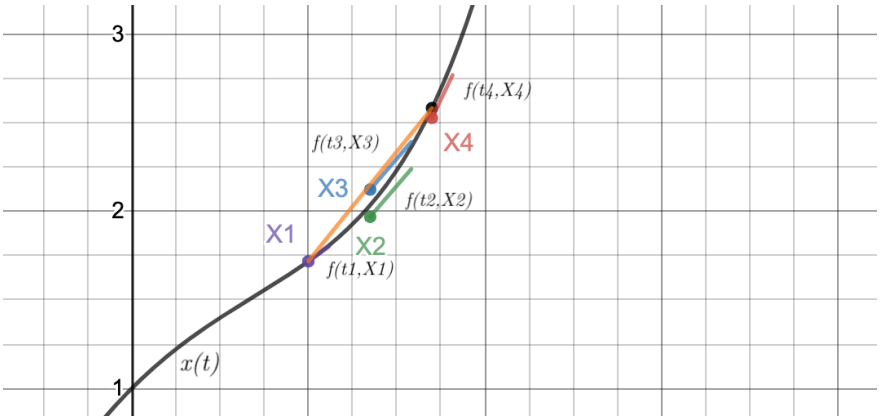


Figure 38: Illustration of the stages of the classical Runge Kutta method. In purple, green, blue and red are the intermediary stages to produce the orange estimation

6.2 Implementation

The general form of a Runge-Kutta step, as presented in Equation 88 can be simplified using matrix notations. Noting $\boldsymbol{K} = \begin{bmatrix} f(t_k + c_1 h, \boldsymbol{X_1}) \\ \vdots \\ f(t_k + c_s h, \boldsymbol{X_s}) \end{bmatrix}$, and $\boldsymbol{B} = \begin{bmatrix} b_1 \\ \vdots \\ b_s \end{bmatrix}$ the upper equation becomes

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + h\boldsymbol{B}^T\boldsymbol{K} \tag{91}$$

Furthermore, assuming that the matrix $\boldsymbol{A}$ of the Butcher tableau is strictly lower triangular (no diagonal elements), which corresponds to the set of all explicit Runge-Kutta methods, such as RK4, each row of $\boldsymbol{K}$ can be computed using the previous rows using,

$$\begin{aligned} \boldsymbol{X}_i &= \boldsymbol{x}_k + h\boldsymbol{A}_{i,:i} \begin{bmatrix} f(t_1, \boldsymbol{X}_1) \\ \vdots \\ f(t_{i-1}, \boldsymbol{X}_{i-1}) \end{bmatrix} = \boldsymbol{x}_k + h\boldsymbol{A}_{i,:i}\boldsymbol{K}_{:i)} \\ \boldsymbol{K}_i &= f(t_k + c_i h, \boldsymbol{X}_i) = f(t_k + c_i h, \boldsymbol{x}_k + h\boldsymbol{A}_{i,:i}\boldsymbol{K}_{:i}) \end{aligned} \tag{92}$$

Where $\boldsymbol{K}_i$ is the ith row of $\boldsymbol{K}$, $\boldsymbol{K}_{:i}$ the (i-1)th first rows of $\boldsymbol{K}$ and $\boldsymbol{A}_{i,:i}$ the ith row of $\boldsymbol{A}$ truncated to its first (i-1) elements. This provides an elegant and concise method to implement an iterative step of any explicit RK method. Listing 11 details its *Python* implementation.

```python
def rk_step(f, t, x, dt, butcher_tableau, **kwargs):
    A = butcher_tableau['A']
    B = butcher_tableau['B']
    C = butcher_tableau['C']

    P = len(C)
    K = np.zeros((P, x.shape[0]))

    K[0, :] = f(t, x, **kwargs)

    for p, (a, c) in enumerate(zip(A[1:], C[1:]), start=1):
        t_p = t + dt * c
        x_p = x + dt * (K.T @ a)
        K[p, :] = f(t_p, x_p, **kwargs)

    return x + dt * (K.T @ B)
```

Listing 11: Implementation of the method agnostic rk step

The generic explicit RK step needs to be given the proper Butcher tableau elements in order to compute the approximation corresponding to the RK4 method. Listing 12 shows how the *rk4_step* function provides a wrapper for the generic *rk_step* function.

```python
# Butcher Tableau
C = np.array([0, 1 / 2, 1 / 2, 1])
A = np.array([
    [0, 0, 0, 0],
    [1 / 2, 0, 0, 0],
    [0, 1 / 2, 0, 0],
    [0, 0, 1, 0],
])
B = np.array([1 / 6, 1 / 3, 1 / 3, 1 / 6])


def rk4_step(f, t, x, dt, **kwargs):
    butcher_tableau = {
        'A': A,
        'B': B,
```

```
        'C': C,
    }

    return rk_step(f, t, x, dt, butcher_tableau, **kwargs)[0]  # no error
    estimation.
```

Listing 12: Implementation of the RK4 step as a wrapper of the general RK step

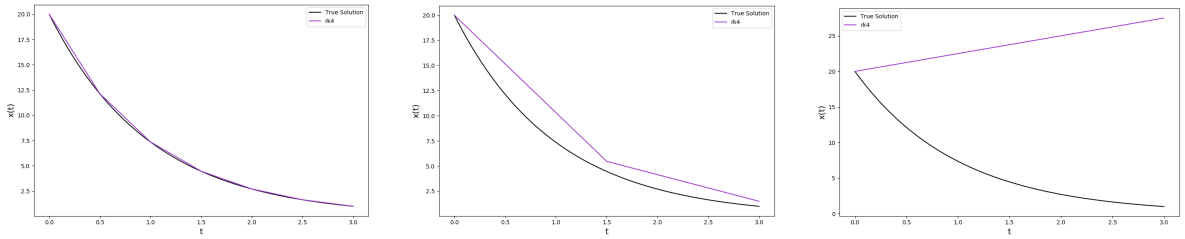
Finally, the interface and structure of the RK4 solver are identical to the interface and structure of the explicit Euler method as they are both purely explicit methods. Listing 13 displays them.

```
def ode_solver(f, J, t0, tf, N, x0, adaptive_step_size=False, **kwargs):
    dt = (tf - t0) / N

    T = [t0]
    X = [x0]
    for k in range(N):
        X.append(rk4_step(f, T[-1], X[-1], dt, **kwargs))
        T.append(T[-1] + dt)

    T = np.array(T)
    X = np.array(X)

    return X, T
```

Listing 13: Interface for the RK4 solver

6.3 Test Equation

Figure 39 displays solutions computed with the RK4 method for different step sizes. The method stays convergent for large values of h, such as $h = 1.5$ as shown by the middle figure, and is quite accurate for a step size as big as $h = 0.5$ as shown on the left. On the right, the step size $h = 3$ appears to be too large to lead to a convergent simulation.



(a) $h = 0.5$ (b) $h = 1.5$ (c) $h = 3$

Figure 39: Comparison of the RK4 method against the true solution to the test equation for various step sizes.

6.4 Order of the RK4 Method

Deriving the order of accuracy of a Runge-Kutta method is not as immediate as what was seen in Section 3 for the Euler methods. For the general IVP,

$$\boldsymbol{x}'(t) = f(t, \boldsymbol{x}(t)), \qquad \boldsymbol{x}(t_0) = \boldsymbol{x}_0 \tag{93}$$

If the solution is assumed smooth over the integration interval (condition that can be restricted to p-times differentiable for order p consideration), it is possible to use the Taylor expansion of the true solution at the time step t_{k+1} with regard to the time step t_k up to order p

$$\boldsymbol{x}(t_{k+1}) = \boldsymbol{x}(t_k) + \sum_{n=1}^{p} \frac{h^n}{n!} \frac{d^n \boldsymbol{x}}{dt^n}(t_k) + \mathcal{O}(h^{p+1}) \tag{94}$$

The evaluation of the respective nth order derivatives $\frac{d^n \boldsymbol{x}}{dt^n}(t_k)$ can leverage the considered differential equation,

$$\begin{cases} \frac{d\boldsymbol{x}}{dt}(t_k) = f(t_k, \boldsymbol{x}(t_k) \\ \frac{d^2\boldsymbol{x}}{dt^2}(t_k) = \frac{df}{dt}(t_k, \boldsymbol{x}(t_k)) \\ \frac{d^3\boldsymbol{x}}{dt^3}(t_k) = \frac{d^2 f}{dt^2}(t_k, \boldsymbol{x}(t_k)) \\ \dots \end{cases} \tag{95}$$

The RHS function f is a multivariate composite function. For the test equation, problem studied in this study to determine the theoretical order of a numerical method, f can nevertheless be restricted to $f(t, \boldsymbol{x}(t)) = f(\boldsymbol{x}(t))$ leading to the simplification of its derivatives, as the univariate and not multivariate chain rule can be applied.

$$\begin{cases} \frac{df}{dt}(t, \boldsymbol{x}(t)) = \frac{df}{dt}(\boldsymbol{x}(t)) = \boldsymbol{x}'(t) f'(\boldsymbol{x}(t)) & = f(t, \boldsymbol{x}(t)) f'(t, \boldsymbol{x}(t)) \\ \frac{d^2 f}{dt^2}(t, \boldsymbol{x}(t)) = \frac{d}{dt}[f(t, \boldsymbol{x}(t)) f'(t, \boldsymbol{x}(t))] & = f''(t, \boldsymbol{x}(t)) f(t, \boldsymbol{x}(t)) f(t, \boldsymbol{x}(t)) \\ & \quad + f'(t, \boldsymbol{x}(t)) f'(t, \boldsymbol{x}(t)) f(t, \boldsymbol{x}(t)) \\ \dots \end{cases} \tag{96}$$

Which results, for the nth order derivatives of the true solution

$$\begin{cases} \frac{d\boldsymbol{x}}{dt}(t_k) & = f(t_k, \boldsymbol{x}(t_k) \\ \frac{d^2\boldsymbol{x}}{dt^2}(t_k) & = f(t_k, \boldsymbol{x}(t)) f'(t_k, \boldsymbol{x}(t_k)) \\ \frac{d^3\boldsymbol{x}}{dt^3}(t_k) & = f''(t_k, \boldsymbol{x}(t_k)) f(t_k, \boldsymbol{x}(t_k)) f(t_k, \boldsymbol{x}(t_k)) \\ & \quad + f'(t_k, \boldsymbol{x}(t_k)) f'(t_k, \boldsymbol{x}(t_k)) f(t_k, \boldsymbol{x}(t_k)) \\ \dots \end{cases} \tag{97}$$

The nth order derivatives of the true solution thus become increasingly complex as n grows. The terms have nevertheless a very interesting property. They follow the structure of rooted trees. More specifically, the nth order derivative will appear as the sum of rooted trees of order n, each weighted by the number of ways it can be labeled with an ordered set

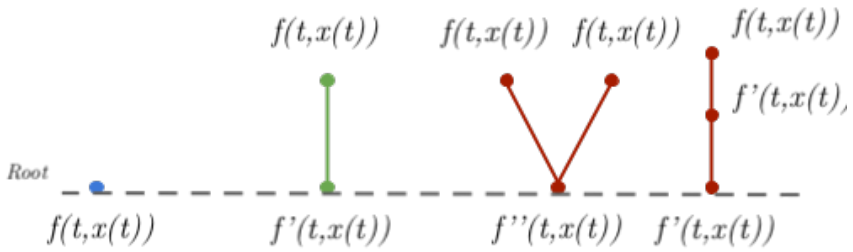


Figure 40: Derivatives seen as a rooted trees, T_1, T_2, $T_{3,1}$ and $T_{3,2}$ respectively

of nodes[4]. The corresponding rooted trees are displayed in Figure 40.

In the rooted tree representation, each node of the tree is an occurrence of f and its differentiation degree corresponds to the number of children of the node. Equation 98 formalises how to use the rooted tree representation to generate the nth derivative of x.

The function $F(\tau)(t)$ returns the product of the derivatives of f as nodes of the tree τ, evaluated at time t, and $\alpha(\tau)$ is the number of possible labeling of the tree τ with an ordered set of nodes.

$$\begin{cases} \frac{d\boldsymbol{x}}{dt}(t_k) = \alpha(\tau_1)F(\tau_1)(t_k) \\ \frac{d^2\boldsymbol{x}}{dt^2}(t_k)) = \alpha(\tau_2)F(\tau_2)(t_k) \\ \frac{d^3\boldsymbol{x}}{dt^3}(t_k)) = \alpha(\tau_{3,1})F(\tau_{3,1})(t_k) + \alpha(\tau_{3,2})F(\tau_{3,2})(t_k) \\ \dots \end{cases} \tag{98}$$

The properties of rooted trees of order n are well documented and are summed up in Table 3 with $r(\tau)$ being the order of the tree τ, $\sigma(\tau)$ its number of symmetries, and $\gamma(\tau)$ its density.

Now that all the necessary function have been introduced, and considering T_n as the set of all rooted trees of order n, equation 94 can be expressed again as:

$$\boldsymbol{x}(t_{k+1}) = \boldsymbol{x}(t_k) + \sum_{n=1}^{p} \sum_{\tau \in T_n} \frac{h^n}{n!} \alpha(\tau) F(\tau)(t_k) + \mathcal{O}(h^{p+1}) \tag{99}$$

And because, by definition $\alpha(\tau) = \frac{r(\tau)!}{\sigma(\tau)\gamma(\tau)}$, equation 94 finally becomes:

$$\boldsymbol{x}(t_{k+1}) = \boldsymbol{x}(t_k) + \sum_{\tau \in \{T_1 \cup \dots \cup T_p\}} \frac{h^{r(\tau)}}{\sigma(\tau)\gamma(\tau)} F(\tau)(t_k) + \mathcal{O}(h^{p+1}) \tag{100}$$

[4]This actually accounts for the plurality of some expression of derivatives of f. The fourth derivative of x is for example $\frac{d^4\boldsymbol{x}}{dt^4} = f'''ff + 3f''ff'f + f'f'f'f + f'f''ff$

τ	τ_1 	τ_2 	$\tau_{3,1}$ 	$\tau_{3,2}$ 
r(τ)	1	2	3	3
$\sigma(\tau)$	1	1	2	1
$\gamma(\tau)$	1	2	3	6
$\alpha(\tau)$	1	1	1	1
F(τ)	f	f'f	f"ff	f'f'f

Table 3: Rooted tree properties

Let's unroll now a similar expansion for the approximation. For simplicity's sake and to make the expansion more understandable, it will be restricted to an explicit RK method of order 3. See Section 3 in J. Butcher's book [8] for full details.

The expansion will still be conducted on the test equation for which the dependency of f on its first argument is eliminated. This enable to simplify the expression with the simplification $f(t_k, \boldsymbol{x}_k) = f(\boldsymbol{x}_k)$ and thus the Taylor expansion of $f(t_k, \boldsymbol{x}_k + \alpha)$ is simplified to:

$$f(t, \boldsymbol{x}_k + \alpha) = f(\boldsymbol{x}_k + \alpha) = f(\boldsymbol{x}_k) + (\boldsymbol{x}_k - \alpha) f'(\boldsymbol{x}_k) + ... \tag{101}$$

It is possible to take the Taylor expansion of f in the intermediate steps to obtain a complete expansion of the approximation. As the RK method used here will be of order 3, the expression of the stage values only need terms up to h^2. Concretely the expansion is:

$$
\begin{aligned}
T_1 &= t_k \\
\boldsymbol{X}_1 &= \boldsymbol{x}_k \\
h\boldsymbol{F}_1 &= hf(T_1, \boldsymbol{X}_1) = hf(\boldsymbol{x}_k)
\end{aligned}
$$

$$
\begin{aligned}
T_2 &= t_k + c_2 h \\
\boldsymbol{X}_2 &= \boldsymbol{x}_k + ha_{21}\boldsymbol{F}_1 = \boldsymbol{x}_k + ha_{21}f(t_k, \boldsymbol{x}_k) \\
h\boldsymbol{F}_2 &= hf(\boldsymbol{x}_k) + h^2 a_{21} f(\boldsymbol{x}_k) f'(\boldsymbol{x}_k) + h^3 \frac{(a_{21} f(\boldsymbol{x}_k))^2}{2} f''(\boldsymbol{x}_k) + \mathcal{O}(h^3)
\end{aligned}
$$

$$
\begin{aligned}
T_3 &= t_k + c_3 h \\
\boldsymbol{X}_3 &= \boldsymbol{x}_k + h(a_{31}\boldsymbol{F}_1 + a_{32}\boldsymbol{F}_2) \\
&= \boldsymbol{x}_k + a_{31} hf(\boldsymbol{x}_k) + a_{32} hf(\boldsymbol{x}_k) + h^2 a_{32} a_{21} f(\boldsymbol{x}_k) f'(\boldsymbol{x}_k) + \mathcal{O}(h^3) \\
&= \boldsymbol{x}_k + (a_{31} + a_{32}) hf(\boldsymbol{x}_k) + h^2 a_{32} a_{21} f(\boldsymbol{x}_k) f'(\boldsymbol{x}_k) + \mathcal{O}(h^3) \\
h\boldsymbol{F}_3 &= hf(\boldsymbol{x}_k) + (a_{31} + a_{32}) h^2 f(\boldsymbol{x}_k) f(\boldsymbol{x}_k) + a_{32} a_{21} h^3 f'(\boldsymbol{x}_k) f'(\boldsymbol{x}_k) f(\boldsymbol{x}_k) \\
&\quad + \frac{1}{2}(a_{31} + a_{32})^2 h^3 f''(\boldsymbol{x}_k) f(\boldsymbol{x}_k) f(\boldsymbol{x}_k) + \mathcal{O}(h^4)
\end{aligned}
\tag{102}
$$

$$
\begin{aligned}
\boldsymbol{x}_{k+1} &= \boldsymbol{x}_k + h(b_1 + b_2 + b_3) f(\boldsymbol{x}_k) + h^2 (b_2 a_{21} + b_3 (a_{31} + a_{32})) f'(\boldsymbol{x}_k) f(\boldsymbol{x}_k) \\
&\quad + \frac{1}{2} h^3 (b_2 a_{21}^2 + b_3 (a_{31} + a_{32})^2) f''(\boldsymbol{x}_k) f(\boldsymbol{x}_k) f(\boldsymbol{x}_k) \\
&\quad + h^3 b_3 a_{32} a_{21} f'(\boldsymbol{x}_k) f'(\boldsymbol{x}_k) f(\boldsymbol{x}_k) + \mathcal{O}(h^4)
\end{aligned}
$$

Here again the expressions of the derivatives of f related to the rooted trees appear, f, $f'f$, $f''ff$, $f'f'f$, preceded by a polynomial expression of the elements of the Butcher tableau. This polynomial expression, will be defined, for each rooted tree appearing, by $\Phi(\tau)$. It is also worth noting that any term in the expansion of the approximation is divided by the number of symmetries of the associated rooted tree.

Furthermore, J. Butcher introduces the following equalities:

$$
\begin{aligned}
\forall k \in [1, p] \ , \quad B(p) &:= \sum_{j=1}^{s} b_j c_j^{k-1} = \frac{1}{k} \\
\forall (i, k) \in [1, s] \times [1, \eta] \ , \quad C(\eta) &:= \sum_{j=1}^{s} a_{ij} c_j^{k-1} = \frac{c_i^k}{k}
\end{aligned}
\tag{103}
$$

They define the stage order of a Runge-Kutta method; it is the largest n such as both $B(n)$ and $C(n)$ hold. In particular, it is desired here to have a stage order greater than 1, so $C(1)$ must hold, resulting in the following additional equation

$$\sum_{j=1}^{s} a_{ij} = c_i \tag{104}$$

Which is also cited by in A. Iserles in [9] page 39 as a necessary condition for order 1. This enables the reformulation

$$\begin{aligned} c_2 &= a_{21} \\ c_3 &= a_{31} + a_{32} \end{aligned} \tag{105}$$

These considerations can be generalised to result in the expansion of the approximation given in Equation 106, where the expression of the polynomials $\Phi(\tau)$ can be found for the first 3 orders in Table 4.

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \sum_{\tau \in \{T_1 \cup ... \cup T_p\}} \frac{h^{r(\tau)}}{\sigma(\tau)} \Phi(\tau) F(\tau)(t_k) + \mathcal{O}(h^{p+1}) \tag{106}$$

τ	τ_1	τ_2	$\tau_{3,1}$	$\tau_{3,2}$
$\sigma(\tau)$	1	1	2	1
r(τ)	1	2	3	3
$\Phi(\tau)$	$\sum b_i$	$\sum b_i c_i$	$\sum b_i c_i^2$	$\sum b_i a_{ij} c_j$

Table 4: Polynomials in butcher's tableau coefficients for rooted trees

It is now obvious, by comparing the expression of the true solution in Equation 100 to the expression of the numerical approximation in Equation 106, that in order for the local error at any step (0 for example) to be of order p + 1 (asymptotic behaviour following $\mathcal{O}(h^{p+1})$) all the terms in the sum must cancel. This results in the order conditions presented in Equation 107

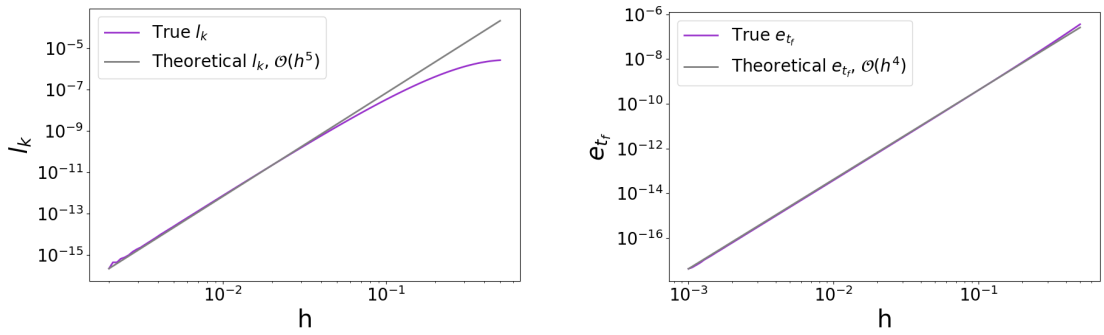
$$\forall \tau \in \{T_1 \cup ... \cup T_p\} \ , \quad \Phi(\tau) = \frac{1}{\gamma(\tau)} \tag{107}$$

Which are equivalent to Theorem 315A from J. Butcher's book [8].

With all these considerations in mind, one can now verify that the RK4 method satisfies all conditions up to p = 4, and therefore prove that its order of accuracy is at least 4. Furthermore, it is shown in [1] page 88, that no method with 4 stages can reach an accuracy of 5, thus proving that the order of the RK4 method is indeed 4.

Figure 41 displays in logarithmic values the variation of the truncation errors when the step size changes. In both cases, the comparison to the theoretical asymptotic value ($\mathcal{O}(h^5)$

for the local error and $\mathcal{O}(h^4)$ for the global error) match closely, proving that the accuracy of the implementations corresponds indeed to the theoretical order of accuracy, 4.



(a) Local truncation error for the RK4 method. Logarithmic values

(b) Global truncation error for the RK4 method. Logarithmic values

Figure 41: Truncation errors for the RK4 method

6.5 Stability of the RK4 Method

As described in Subsection 3.6, the stability of a numerical method can be derived from the test equation

$$x'(t) = \lambda x(t), \qquad x(0) = 1, \qquad \lambda \in \mathbb{C}, \qquad Re(\lambda) < 0 \tag{108}$$

Using $f(t, x(t)) = \lambda x(t)$ in Equation 88, yields that for any $i \in [1, s]$,

$$X_i = x_k + h \sum_{j=1}^{s} a_{ij} \lambda X_j \tag{109}$$

That expression can be rearranged into matrix form

$$\boldsymbol{X} = x_k \mathbf{1} + h\lambda \boldsymbol{A}\boldsymbol{X} \tag{110}$$

Where $\boldsymbol{A} \in \mathbb{R}^{s \times s}$ is the A matrix from the Butcher tableau, $\boldsymbol{X} = [X_1, \ldots, X_s]^T$ and $\mathbf{1} \in \mathbb{R}^s$ is a vector of ones. Assuming that $(\boldsymbol{I} - h\lambda \boldsymbol{A})$ is invertible, then

$$\boldsymbol{X} = x_k (\boldsymbol{I} - h\lambda \boldsymbol{A})^{-1} \mathbf{1} \tag{111}$$

The first equation of 88 can also be expressed in a matrix form,

$$x_{k+1} = x_k + h\lambda \boldsymbol{B}^T \boldsymbol{X} \tag{112}$$

Which, becomes, when the RHS of Equation 111 is inserted into $\boldsymbol{X}$,

$$x_{k+1} = x_k (1 + h\lambda \boldsymbol{B}^T (\boldsymbol{I} - h\lambda \boldsymbol{A})^{-1} \mathbf{1}) = x_k R(h\lambda) \tag{113}$$

Where R is defined as the following polynomial on $\mathbb{C}$,

$$R(z) = 1 + z\boldsymbol{B}^T(\boldsymbol{I} - z\boldsymbol{A})^{-1}\mathbf{1} \tag{114}$$

This provides a direct way to determine the stability region of any Runge-Kutta method. Indeed, the terms $\{x_k\}$ follow a geometric progression of reason $R(h\lambda)$, and therefore,

$$x_k = R(h\lambda)^k x_0 \tag{115}$$

Implying that $\lim_{k\to\infty} x_k = 0$ only when $|R(h\lambda)| < 1$. The stability region is therefore:

$$\mathcal{D}_{RK} = \{(h, \lambda) \in \mathbb{R} \times \mathbb{C} \mid |R(h\lambda)| < 1\} \tag{116}$$

Specifically, for the RK4 method, Figure 42 displays the stability region of its stability polynomial.

$$R(h\lambda) = 1 + h\lambda + \frac{(h\lambda)^2}{2} + \frac{(h\lambda)^3}{6} + \frac{(h\lambda)^4}{24} \tag{117}$$

The red region corresponds to $|R(h\lambda)| > 1$, where the method is unstable, and green, yellow and orange regions correspond to $|R(h\lambda)| < 1$ where the method is stable. RK4 is clearly not A-stable as the region of absolute stability is only a portion of the complex negative half plane. This suggests that RK4 is not well suited to handle stiff problems. It is worth noting as well that the stability region of the RK4 method far exceeds the stability region of the explicit Euler method, indicating that the set of step sizes h for which the method is convergent is not as limited for a given λ.

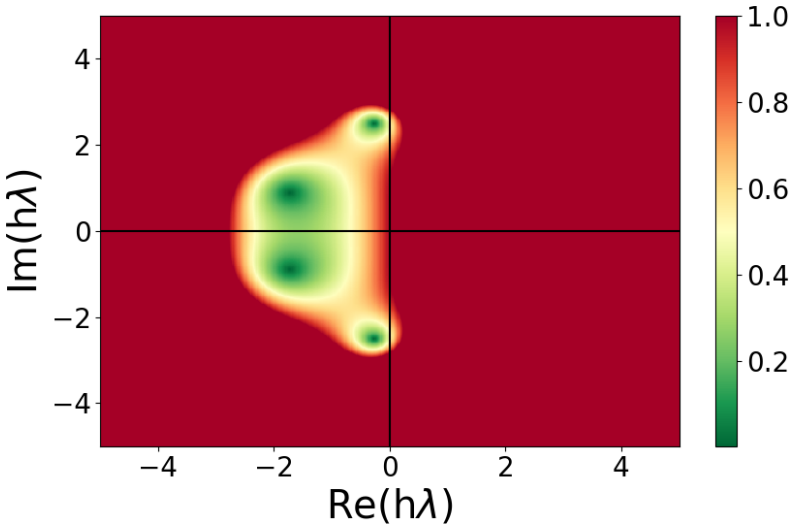
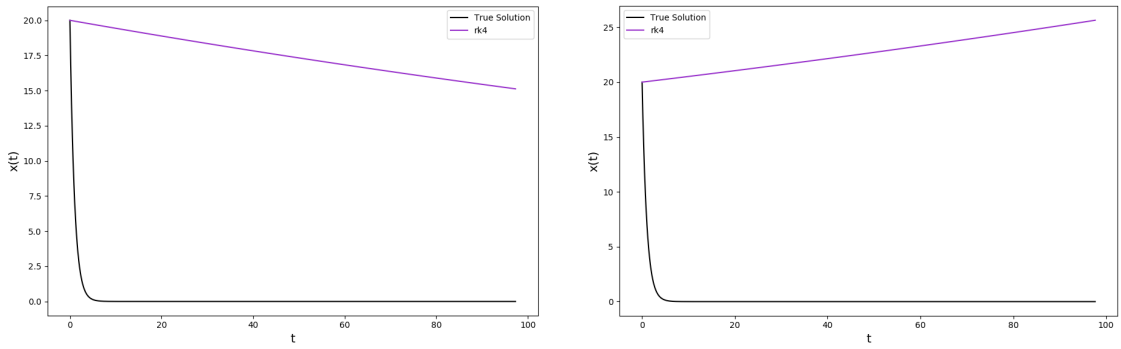


Figure 42: Region of absolute stability (green) for the RK4 method

With this knowledge at hand, the divergence shown on Figure 39 for the step $h = 3$ could have been predicted. Indeed, for the given $\lambda = -1$, and $h = 3$, the stability polynomial

can be evaluated at $R(h\lambda) = \frac{11}{8} > 1$, showing directly that the couple (h, λ) does not lie in the stability region. The strictly positive solution to $R(h\lambda) = 1$ for $\lambda = -1$ is actually located around $h = 2.785$. Figure 43 demonstrates this boundary. On the left side is displayed the approximation computed with the RK4 method for h = 2.78, and even though the convergence rate is very slow, the solution is gradually decreasing towards 0, as it should. On the right side is displayed the approximation computed with the RK4 method for h = 2.79, for which the divergence is very clear.



(a) $h = 0.278$ (b) $h = 0.279$

Figure 43: RK4 method on the test equation for $\lambda = -1$ at the stability boundary.

6.6 Test on the Van der Pol Problem

6.6.1 For $\mu = 2$

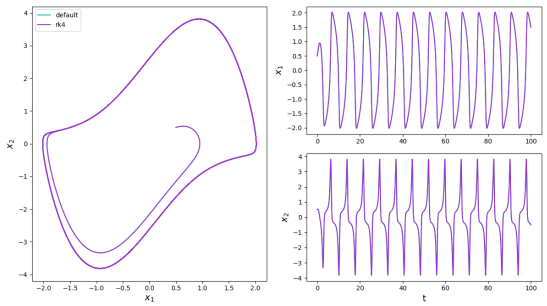
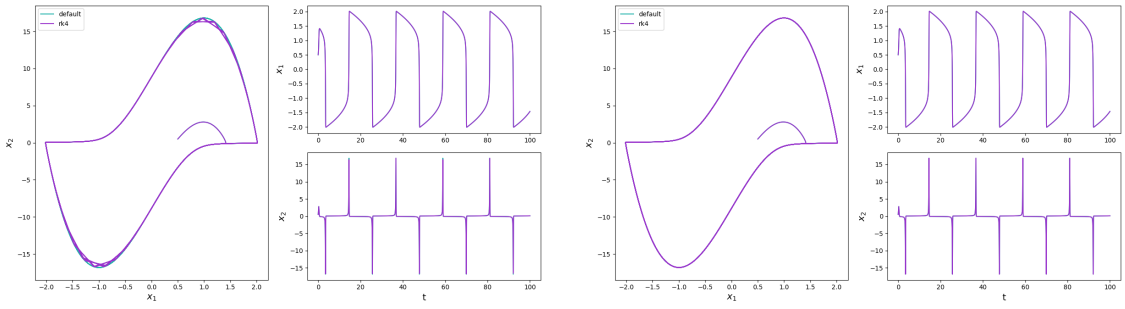
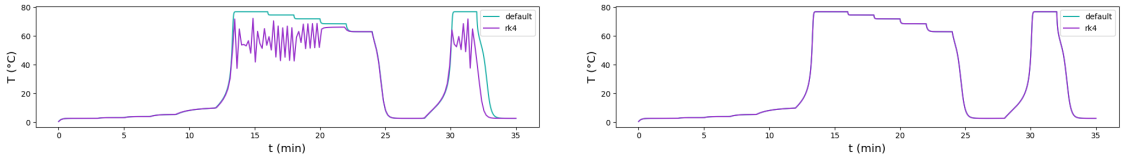


Figure 44: Classical Runge-Kutta method on the Van der Pol problem. $\mu = 2$ and $h = 0.025$



(a) $h = 0.025$ (b) $h = 0.001$

Figure 45: Classical Runge-Kutta method on the Van der Pol problem. $\mu = 12$



(a) $h = 0.175$ (b) $h = 0.00035$

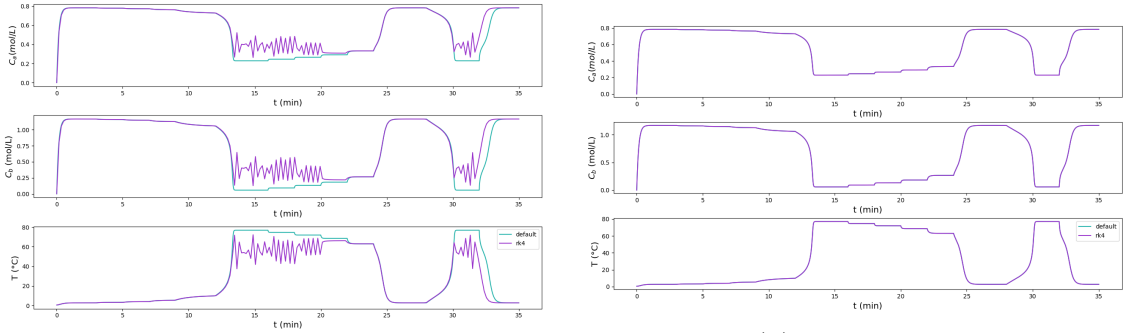
Figure 46: Classical Runge-Kutta method on the CSTR 1D problem.

6.6.2 For $\mu = 12$

6.7 Test on the A-CSTR Problem

6.7.1 CSTR 1D

6.7.2 CSTR 3D



(a) $h = 0.175$ (b) $h = 0.00035$

Figure 47: Classical Runge-Kutta method on the CSTR 3D problem

7 Classical Runge-Kutta method with adaptive time step

7.1 Method Description

One of the shortcomings of the classical Runge-Kutta method is that it is not A-Stable. The choice of the step size h must therefore not only depend on accuracy considerations but also take convergence into account. This might force the use of very small step sizes for problems that display quick variations of the true solution, and the additional computations performed at each step by the method to determine the intermediary values X_1, ..., X_s can result in impractical overall computing times.

Adding a step size controller to the method, which can increase the step size when the solution displays low variations and decrease it when the variations of the solution increase, is a good way to use the classical Runge-Kutta method more efficiently.

Similarly to what has precedently been described, notably in section 1.3, an asymptotic step size controller using step doubling to estimate the local error will improve the standard implementation of the method. It is worth noting that the expression of the asymptotic step size controller is slightly different to what has been presented earlier, due to a larger accuracy order of the method

$$h' = \max(\text{min\_factor}, \min((\frac{\epsilon}{r_{k+1}})^{\frac{1}{p+1}}, \text{max\_factor}))h \tag{118}$$

7.2 Implementation

```
def ode_solver(f, J, t0, tf, N, x0, **kwargs):
    dt = (tf - t0) / N

    T = [t0]
    X = [x0]
    controllers = {
        'r': [0],
        'dt': [dt]
    }

    kwargs, abstol, reltol, epstol, facmax, facmin =
   parse_adaptive_step_params(kwargs)

    p = P_RK4
    t = t0
    x = x0

    while t < tf:
        if (t + dt > tf):
            dt = tf - t

        accept_step = False
        while not accept_step:
            # Take step of size dt
```

```
            x_1 = rk4_step(f, t, x, dt, **kwargs)

            # Take two steps of size dt/2
            x_hat_12 = rk4_step(f, t, x, dt / 2, **kwargs)
            t_hat_12 = t + (dt / 2)
            x_hat = rk4_step(f, t_hat_12, x_hat_12, dt / 2, **kwargs)

            # Error estimation
            e = np.abs(x_1 - x_hat)
            r = np.max(np.abs(e) / np.maximum(abstol, np.abs(x_hat) *
    reltol))

            accept_step = (r <= 1)
            if accept_step:
                t = t + dt
                x = x_hat

                T.append(t)
                X.append(x)
            controllers['dt'].append(dt)
            controllers['r'].append(r)

            dt = np.maximum(facmin, np.minimum((epstol / r)**(1 / (p + 1))
    , facmax)) * dt

     T = np.array(T)
     X = np.array(X)
     controllers['dt'] = np.array(controllers['dt'])
     controllers['r'] = np.array(controllers['r'])

     return X, T, controllers
```

Listing 14: Implementation of the RK4 method with asymptotic step size controller

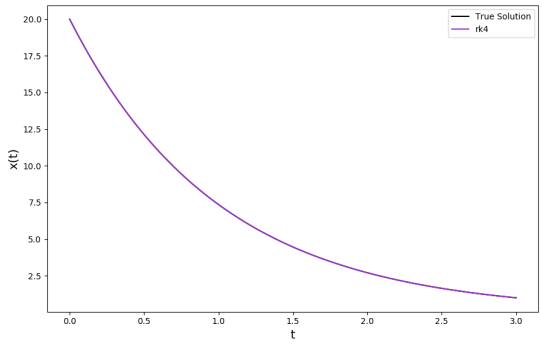
7.3 Test Equation

Figure 48 displays the output of the RK4 method with adaptive step size on the test equation for the parameters $x_0 = 20$, $t_f = 3$, $\lambda = -1$. As expected, the step size is automatically reduced after a few trials to a level that produces a very accurate approximation.

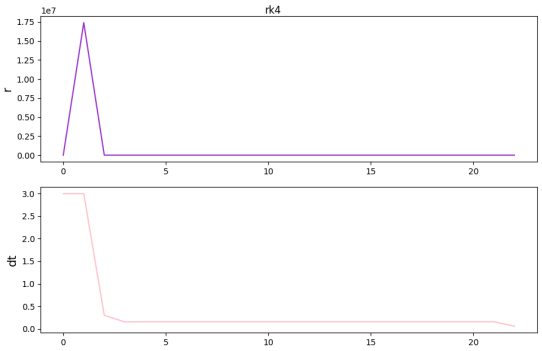
It is interesting to study the influence of some of the parameters of the step size controller on the generated approximation.

First, the minimum factor *min_factor* with which is updated the step size must be strictly lower than 1. Otherwise, the step size will never decrease, cancelling all benefits of the method. The closer to 1 that factor will be, the slower the decrease of the step size will be, as shown in Figure 49.

On another hand, the maximum factor *max_factor* with which is updated the step size should be strictly greater than 1. If it is lower or equal than one, no increase of the step size and thus no increase in the performance of the method can be expected. Figure 50 reveals that dependency for a simulation running all the way to $t_f = 100$, and shows that a larger maximum factor enables to reach high step values for zones of low variation more quickly.

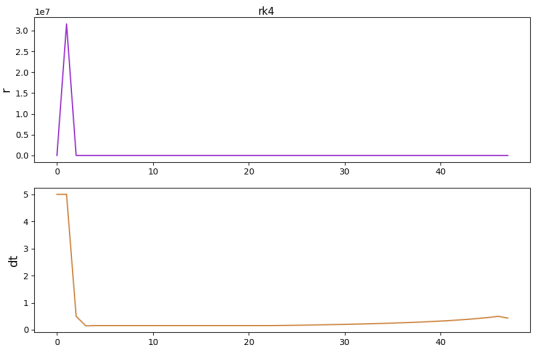


(a) Compariosn of the approximation generated by the RK4 method with the true solution.

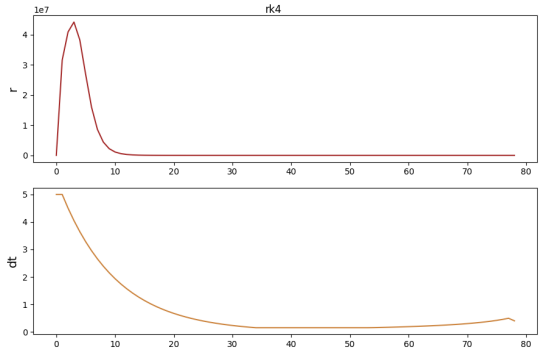


(b) Step acceptance criteria and evolution of the step size

Figure 48: RK4 method with step size controller. $h_0 = 3$



(a) min_factor = 0.1



(b) min_factor = 0.9

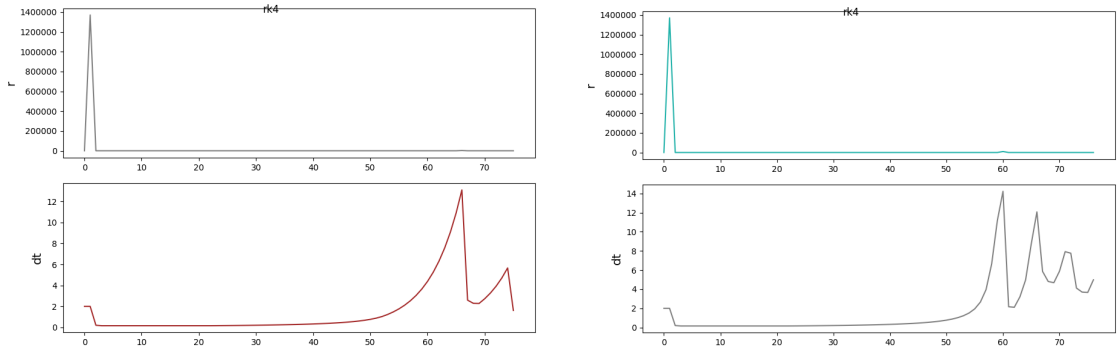
Figure 49: Influence of the parameter min_factor. The closer to one min_factor is, the slower the decrease of the step size (bottom line) can be.

Nevertheless, this comes at the cost of a potential increased unstability of the step size, with the step size controller overshooting the range of correct step sizes and having to re-adjust later on.

Finally, the greater ϵ is, the stronger the variations induced by the difference between the estimated error and its asymptotic behaviour will be.

7.4 Order and Stability

The underlying method, the classical Runge-Kutta method does not change with the introduction of the step size controller. Its accuracy order is therefore unchanged. For the same reason, the stability region of the method is the same as the stability region of the classical Runge Kutta. Nevertheless, the step size controller eliminates the need to consider stability of the method when defining the step size of the simulation. This is clearly shown in Figure 48; the step size is originally $h = 3$, which leads to a divergence of the standard



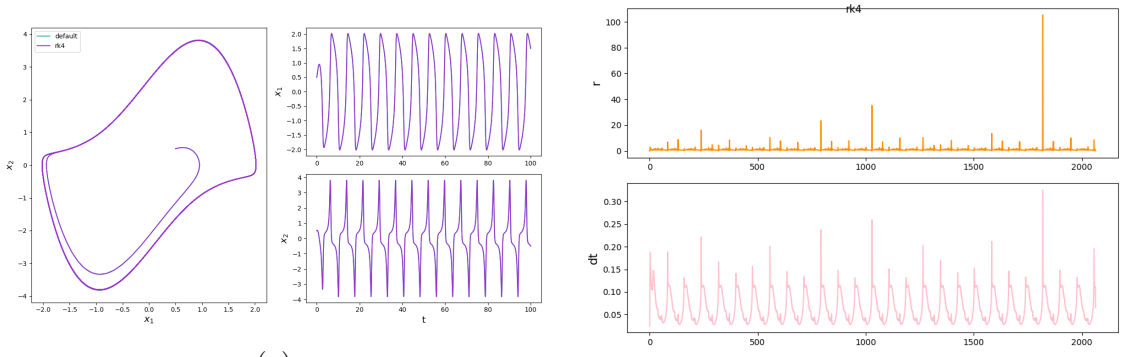
(a) max_factor = 1.2 (b) max_factor = 10

Figure 50: Influence of the parameter max_factor. The closer to one min_factor is, the slower the increase of the step size (bottom line) can be

implementation, while here it is quite clear that step size controller prevents that divergence alltogether.

7.5 Test on the Van der Pol Problem

7.5.1 For $\mu = 2$



(a) (b)

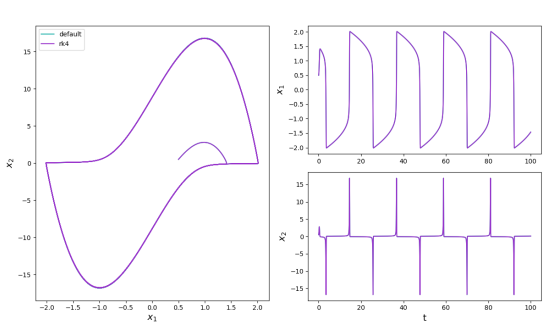
Figure 51

7.5.2 For $\mu = 12$

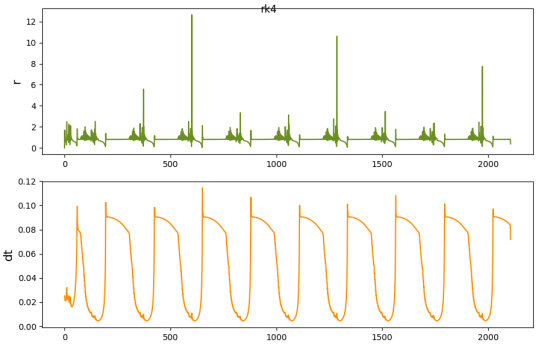
7.6 Test on the A-CSTR Problem

7.6.1 CSTR 1D

7.6.2 CSTR 3D

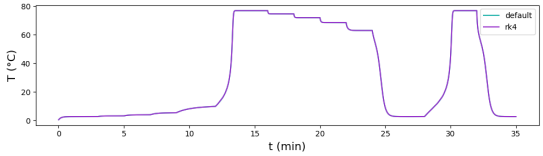


(a)

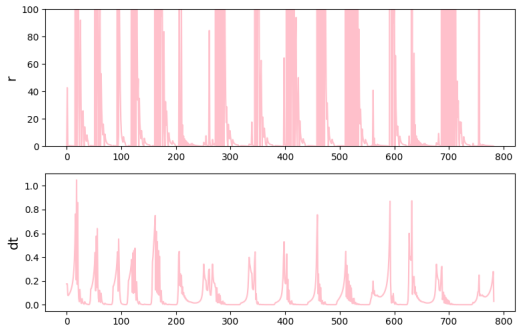


(b)

Figure 52

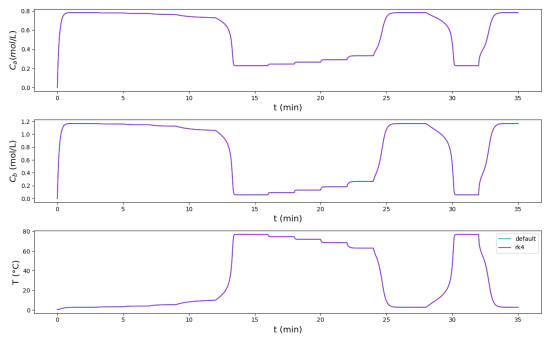


(a)

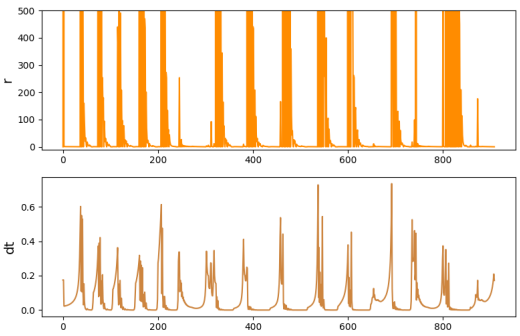


(b)

Figure 53



(a)



(b)

Figure 54

8 Dormand-Prince 5(4)

Motivation behind embedded error estimator is that the cost to estimate the error by step doubling is 3n vs n for the embedded estimator even though it is less accurate (n being the number of total steps N)

For demonstration of the error controller, use test equation with low number of points and plot true local error vs estimated error along the trajectory

To test order of embedded error estimator, plot as a function of the step size any error ek vs true local error lk.

8.1 Method Description

The previous section described how the usability of the classical Runge-Kutta method could be greatly improved with the addition of a step size controller. Nevertheless, this improvement comes at a cost. The estimation of the local error at any stage l_k through step doubling necessitates 3n operations instead of n, were there no adaptive step size implemented. (n being the number of operations required to evaluate $\boldsymbol{x}_{k+1}$ from $\boldsymbol{x}_k$). In the case of a Runge-Kutta method, this increased cost per step can become penalising as the number of stages grows.

Let's imagine two different methods, that estimate for the step k the approximations $\boldsymbol{x}_k$ and $\hat{bmx}_k$ with accuracies of order p and q respectively. Then

$$\boldsymbol{x}_k = (t_k) + \mathcal{O}(h^{p+1}) \ , \quad \hat{\boldsymbol{x}_k} = (t_k) + \mathcal{O}(h^{q+1}) \tag{119}$$

And therefore , supposing q > p, the difference of the two approximations results in

$$\hat{\boldsymbol{x}_k} - \boldsymbol{x}_k - = x(t_k) - \boldsymbol{x}_k + \mathcal{O}(h^{p+2}) \tag{120}$$

Thus providing as an approximation of the local error for the method that results in the approximation $\boldsymbol{x}_k$.

In the case of a Runge-Kutta method, most operations at each step are executed to determine the expressions of the different stages. One can thus imagine a setting where two Runge-Kutta methods share their intermediate stages, and whose difference is used at every step to estimate the local truncation error, thus providing an efficient way to equip the method of lower order with a step size controller.

That is exactly what a Runge-Kutta method with an embedded error estimator, or embedded Runge-Kutta method does. The Dormand Prince 5(4) method belongs to that class. Its main method is a Runge-Kutta method of order 5, and its embedded method is another Runge-Kutta method, of order 4, that shares the same intermediary stages.

Note here that contrary to the explanation provided earlier, the main method has here a higher order than the embedded method. This is a common practice, called local extrapolation[5], which de-correlates asymptotically the estimated error from the local truncation error, and instead adds the estimated error in the main method's approximation as a correction.

[5]See [8] page 198

The Dormand-Prince 5(4) (DOPRI54) Butcher's tableau is the following:

0							
$\frac{1}{5}$	$\frac{1}{5}$						
$\frac{3}{10}$	$\frac{3}{40}$	$\frac{9}{40}$					
$\frac{4}{5}$	$\frac{44}{45}$	$\frac{-56}{15}$	$\frac{32}{9}$				
$\frac{8}{9}$	$\frac{19372}{6561}$	$\frac{-25360}{2187}$	$\frac{64448}{6561}$	$\frac{-212}{729}$			
1	$\frac{9017}{3168}$	$\frac{-355}{33}$	$\frac{46732}{5247}$	$\frac{49}{176}$	$\frac{-5103}{18656}$		
1	$\frac{35}{384}$	0	$\frac{500}{1113}$	$\frac{125}{192}$	$\frac{-2187}{6784}$	$\frac{11}{84}$	
x	$\frac{5179}{57600}$	0	$\frac{7571}{16695}$	$\frac{393}{640}$	$\frac{-92097}{339200}$	$\frac{187}{2100}$	$\frac{1}{40}$
$\hat{x}$	$\frac{35}{384}$	0	$\frac{500}{1113}$	$\frac{125}{192}$	$\frac{-2187}{6784}$	$\frac{11}{84}$	
e	$\frac{-71}{57600}$	0	$\frac{71}{16695}$	$\frac{-71}{1920}$	$\frac{17253}{339200}$	$\frac{-22}{525}$	$\frac{1}{40}$

Where the line corresponding to $\hat{x}$ corresponds to the coefficients of the embedded method, and the line e to the coefficients of the error estimator, for which the following holds:

$$\begin{aligned} \boldsymbol{x}_{k+1} &= \boldsymbol{x}_k + h \sum_{i=1}^{s} b_i f(t_k + c_i h, \boldsymbol{X}_i) \\ \hat{\boldsymbol{x}}_{k+1} &= \boldsymbol{x}_k + h \sum_{i=1}^{s} \hat{b}_i f(t_k + c_i h, \boldsymbol{X}_i) \end{aligned} \tag{121}$$

$$\boldsymbol{e}_{k+1} = \boldsymbol{x}_{k+1} - \hat{\boldsymbol{x}}_{k+1} = h \sum_{i=1}^{s} e_i f(t_k + c_i h, \boldsymbol{X}_i)$$

$\boldsymbol{e}_{k+1}$ therefore contains for each step an estimated error, for every dimension of the problem.

Furthermore, another step size controller was implemented here for the DOPRI54 method, the PI controller. Based on the acceptance criterion described earlier for each step r_k, the PI controller determines the evolution of the step size with:

- If the computed acceptance criterion for the trial, r verifies $r < 1$

$$\begin{aligned} h' &= h(\frac{\epsilon}{r})^{k_I}(\frac{r_{acc}}{r})^{k_P} \\ k_I &= \frac{0.4}{p+1} \\ k_P &= \frac{0.3}{p+1} \\ r_{acc} &= r \end{aligned} \tag{122}$$

- else, use asymptotic step size controller

$$h' = (\frac{\epsilon}{r})^{\frac{1}{p+1}} h \tag{123}$$

8.2 Implementation

```python
def rk_step(f, t, x, dt, butcher_tableau, **kwargs):
    A = butcher_tableau['A']
    B = butcher_tableau['B']
    C = butcher_tableau['C']

    if 'E' not in butcher_tableau.keys():
        E = np.zeros(B.shape)
    else:
        E = butcher_tableau['E']

    P = len(C)
    K = np.zeros((P, x.shape[0]))

    K[0, :] = f(t, x, **kwargs)

    for p, (a, c) in enumerate(zip(A[1:], C[1:]), start=1):
        t_p = t + dt * c
        x_p = x + dt * (K.T @ a)
        K[p, :] = f(t_p, x_p, **kwargs)

    return x + dt * (K.T @ B), dt * (K.T @ E)
```

Listing 15: Improvement of the rk step method to handle embedded methods

```python
# Butcher Tableau
C = np.array([0, 1/5, 3/10, 4/5, 8/9, 1, 1])
A = np.array([
    [0, 0, 0, 0, 0, 0, 0],
    [1/5, 0, 0, 0, 0, 0, 0],
    [3/40, 9/40, 0, 0, 0, 0, 0],
    [44/45, -56/15, 32/9, 0, 0, 0, 0],
    [19372/6561, -25360/2187, 64448/6561, -212/729, 0, 0, 0],
    [9017/3168, -355/33, 46732/5247, 49/176, -5103/18656, 0, 0],
    [35/384, 0, 500/1113, 125/192, -2187/6784, 11/84, 0],
])
B = np.array([5179/57600, 0, 7571/16695, 393/640, -92097/339200, 187/2100,
              1/40])
B_hat = np.array([35/384, 0, 500/1113, 125/192, -2187/6784, 11/84, 0])
E = np.array([-71/57600, 0, 71/16695, -71/1920, 17253/339200, -22/525,
              1/40])

# Order
P_DOPRI54 = 5


```

```python
def dopri54_step(f, t, x, dt, **kwargs):

    butcher_tableau = {
        'A': A,
        'B': B,
        'C': C,
        'E': E,
    }

    return rk_step(f, t, x, dt, butcher_tableau, **kwargs)
```

Listing 16: Implementation of the DOPRI54 step as a wrapper of the general RK step

```python
def ode_solver(f, J, t0, tf, N, x0, adaptive_step_size=False, **kwargs):

    dt = (tf - t0)/N

    T = [t0]
    X = [x0]
    controllers = {
        'r': [0.01],
        'e': [np.zeros(x0.shape)],
        'dt': [dt],
    }

    kwargs, abstol, reltol, epstol, facmax, facmin =
   parse_adaptive_step_params(kwargs)
    p = P_DOPRI54
    k_p = 0.4/(p+1)
    k_i = 0.3/(p+1)

    t = t0
    x = x0

    while t < tf:
        if (t + dt > tf):
            dt = tf - t

        accept_step = False
        while not accept_step:
            x_hat, e = dopri54_step(f, T[-1], X[-1], dt, **kwargs)
            r = np.max(np.abs(e) / np.maximum(abstol, np.abs(x_hat)*reltol
   ))

            accept_step = (r <= 1)
            if accept_step:
                t = t + dt
                x = x_hat
                dt = dt * (epstol/r)**(k_i) * (controllers['r'][-1]/r)**(
   k_p)

                T.append(t)
                X.append(x)
```

```
38                  controllers['e'].append(e)
39                  controllers['r'].append(r)
40              else:
41                  dt = dt * (epstol / r)**(1/(p+1))
42              controllers['dt'].append(dt)
43
44
45      T = np.array(T)
46      X = np.array(X)
47      controllers['dt'] = np.array(controllers['dt'])
48      controllers['r'] = np.array(controllers['r'])
49      controllers['e'] = np.array(controllers['e'])
50
51      return X, T, controllers
```

Listing 17: Implementation of the DOPRI54 method under the ode solver interface

8.3 Order and Stability

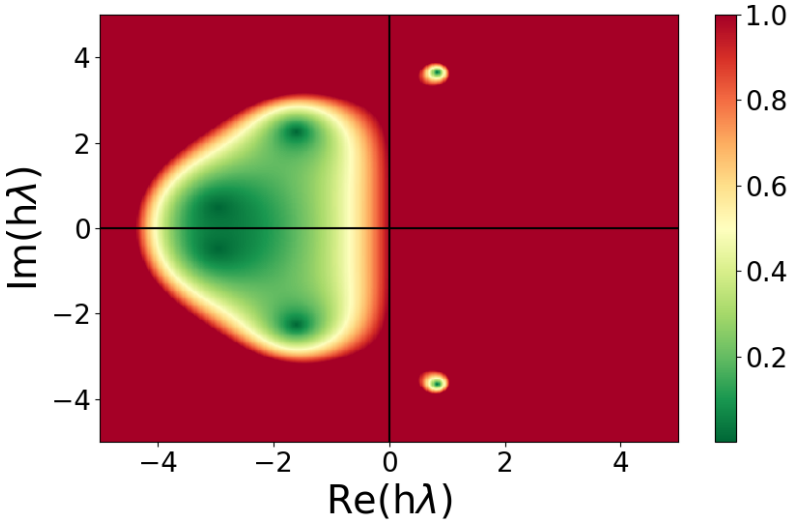


Figure 55: Region of absolute stability (green) for the DOPRI54 method

8.4 Test on the Van der Pol Problem

8.4.1 For $\mu = 2$

8.4.2 For $\mu = 12$

8.5 Test on the A-CSTR Problem

8.5.1 CSTR 1D

8.5.2 CSTR 3D

9 Design your own explicit Runge-Kutta method

9.1 Order Conditions

Designing an explicit Runge-Kutta method requires to determine all coefficients of the Butcher tableau. The desired method here is of order 3 with an embedded error controller of order 2. It has been proven (see for example [1] page 88) that order 3 is attainable with only three stages. Let's therefore write down the desired Butcher tableau as:

$$\begin{array}{c|ccc} 0 & & & \\ c_2 & a_{21} & & \\ c_3 & a_{31} & a_{32} & \\ \hline & b_1 & b_2 & b_3 \\ & \hat{b}_1 & \hat{b}_2 & \hat{b}_3 \end{array}$$

Determining values of the Butcher's tableau coefficients resulting in a Runge-Kutta method of order 3 can be accomplished using the order conditions. Details about such conditions were already presented in 6, resulting in, for order 3:

$$\forall \tau \in \{T_1 \cup ... \cup T_3\} \ , \quad \Phi(\tau) = \frac{1}{\gamma(\tau)} \tag{124}$$

Where the polynomial expressions Φ and the densities γ associated to rooted trees up to order 3 are reminded in Table 5.

τ	τ_1	τ_2	$\tau_{3,1}$	$\tau_{3,2}$
$\mathrm{r}(\tau)$	1	2	3	3
$\sigma(\tau)$	1	1	2	1
$\gamma(\tau)$	1	2	3	6
$\alpha(\tau)$	1	1	1	1
$\Phi(\tau)$	$\sum b_i$	$\sum b_i c_i$	$\sum b_i c_i^2$	$\sum b_i a_{ij} c_j$

Table 5: Reminder of the rooted tree properties

Furthermore, the row sum condition, necessary for any Runge-Kutta of stage order greater than 1 was also exposed in Section 6 and is reminded in the following equation, that must hold for any $i \in [1, s]$

$$\sum_{j=1}^{s} a_{ij} = c_i \tag{125}$$

Combining the previous conditions yields the following system of equations for the Butcher's tableau coefficient

$$\begin{cases} b_1 + b_2 + b_3 = 1 \\ b_2 c_2 + b_3 c_3 = \frac{1}{2} \\ b_2 c_2^2 + b_3 c_3^2 = \frac{1}{3} \\ b_3 a_{32} c_2 = \frac{1}{6} \\ a_{21} = c_2 \\ a_{31} + a_{32} = c_3 \end{cases} \tag{126}$$

Rearranging the terms of such system reveals that the coefficients c_2 and c_3 are free parameters, which determine the values of the rest of the coefficients in the following way:

$$\begin{cases} a_{21} = c_2 \\ a_{31} = \frac{c_3(3c_2^2 - 3c_2 + c_3)}{(3c_2 - 2)c_2} \\ a_{32} = -\frac{c_3^2 - c_2 c_3}{(3c_2 - 2)c_2} \\ b_1 = \frac{(6c_3 - 3)c_2 - 3c_3 + 2}{6c_2 c_3} \\ b_2 = \frac{-3c_3 + 2}{6(c_2 - c_3)c_2} \\ b_3 = \frac{3c_2 - 2}{6c_3(c_2 - c_3)} \end{cases} \tag{127}$$

The choice of the free coefficients c_2 and c_3 must verify $c_2 - c_3 \neq 0$. The following values were chosen for their simplicity.

$$\begin{cases} c_2 = \frac{1}{3} \\ c_3 = \frac{2}{3} \end{cases} \tag{128}$$

Consequently determining the rest of the coefficients:

$$\begin{cases} a_{21} = \frac{1}{3} \\ a_{31} = 0 \\ a_{32} = \frac{2}{3} \\ b_1 = \frac{1}{4} \\ b_2 = 0 \\ b_3 = \frac{3}{4} \end{cases} \tag{129}$$

9.2 Derivation of Coefficients of Error Estimator

The embedded error estimator is desired to be of order 2. The exact same conditions, up to order 2 can be applied on the coefficients $\hat{b}_1$, $\hat{b}_2$ and $\hat{b}_3$. It is also known that only two stages are needed to result in an order 2 method.[6]. Consequently, the third coefficient can be set to $\hat{b}_3 = 0$, resulting in the simple two equations system that follows:

[6] Again, see for example [1] page 88.

$$\begin{cases} \hat{b}_1 + \hat{b}_2 = 1 \\ \hat{b}_2 c_2 = \frac{1}{2} \end{cases} \tag{130}$$

Which has for solution

$$\begin{cases} \hat{b}_1 = -\frac{1}{2} \\ \hat{b}_2 = \frac{3}{2} \end{cases} \tag{131}$$

9.3 Butcher Tableau

As a result, all coefficients of the Butcher's tableau of the custom Runge-Kutta method are determined. The final Butcher's tableau is thus:

$$\begin{array}{c|ccc} 0 & & & \\ \frac{1}{3} & \frac{1}{3} & & \\ \frac{2}{3} & 0 & \frac{2}{3} & \\ \hline & \frac{1}{4} & 0 & \frac{3}{4} \\ & -\frac{1}{2} & \frac{3}{2} & 0 \end{array}$$

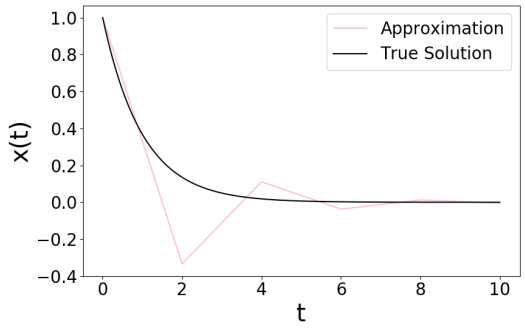
9.4 Test Equation

Again, the test equation is useful to study basic properties of this explicit Runge-Kutta method. As before, the initial point was set to $x_0 = 1$ with parameter $\lambda = -1$. Figure 56 displays on the left the approximated solution by the custom Runge-Kutta method for a very large step size, $h = 2$, against the true solution. On the right is compared the local error estimated by the embedded method against the true local error for each simulation point. The results are encouraging, the method, even though it initially overshoots the trajectory of the true solution, then corrects its path and converges quickly towards the true solution, despite the large step size. The embedded method, similarly, first overestimates drastically the error, but quickly corrects its estimations.

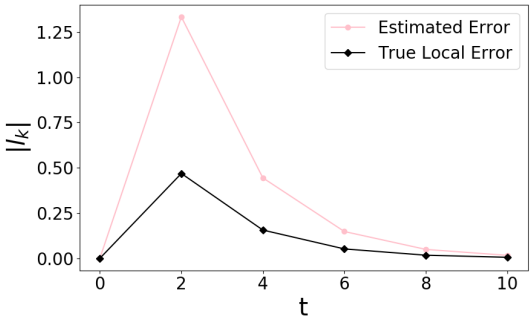
Figure 57 displays the same comparison, but for a step size smaller by an order of magnitude, $h = 0.25$. The approximation is this time clearly more satisfactory and corresponds to what could be expected from an explicit Runge-Kutta method of that order. The embedded method overestimates again the initial local errors, but on a considerably smaller scale, thus suggesting that the decrease in the difference between the estimated and true local error follows the expected dependence in $\mathcal{O}(h^2)$.

9.5 Method Order

Using again the test equation, it is possible to measure the evolution of both the local and global truncation error as a function of the step size h. Figure 58 displays such evolution, with the local truncation error , on the right, clearly following the asymptotic expected

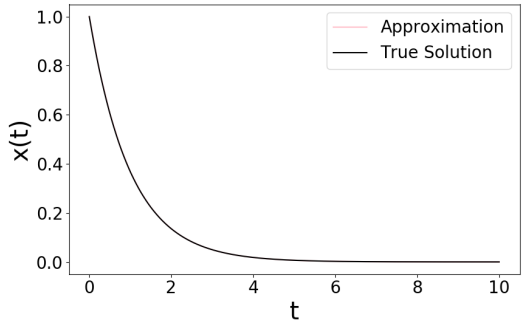


(a) Comparison of the custom RK method against the true solution of the test equation

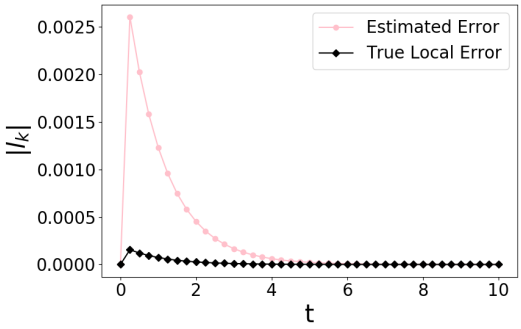


(b) Comparison of the estimated local error from the embedded method with the true local error for each simulation point

Figure 56: Test of the custom RK method on the test equation, for a step size $h = 2$



(a) Comparison of the custom RK method against the true solution of the test equation



(b) Comparison of the estimated local error from the embedded method with the true local error for each simulation point

Figure 57: Test of the custom RK method on the test equation, for a step size $h = 0.25$

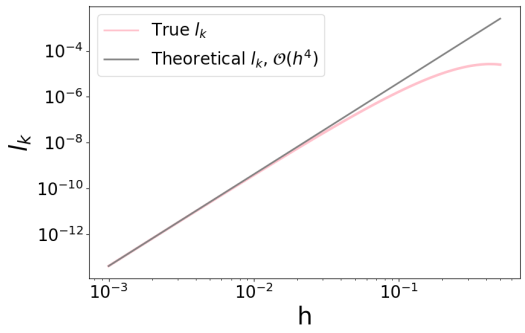
slope of $\mathcal{O}(h^4)$, and with the global truncation error, on the left, also clearly following the asymptotic expected slope of $\mathcal{O}(h^3)$.

This validates the validity of the implementation of the custom Runge-Kutta method.

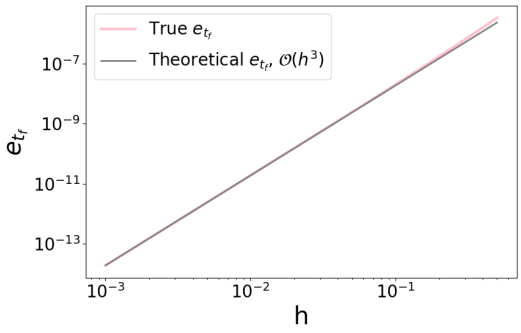
Furthermore, Figure 59 displays the evolution of the estimated local error by the embedded method. It clearly follows the expected asymptotic behaviour of $\mathcal{O}(h^2)$, and is compared to the evolution of the true local truncation error, that follows a slope of $\mathcal{O}(h^4)$.

9.6 Method Stability

Using the previous definition of the stability polynomial for a Runge-Kutta method, expressed as, $R(z) = 1 + z\boldsymbol{B}^T(\boldsymbol{I} - z\boldsymbol{A})^{-1}\mathbf{1}$, it is straightforward to obtain its exact expression for the custom Runge-Kutta method:



(a) Local truncation error for the custom RK method. Logarithmic values



(b) Global truncation error for the custom RK method. Logarithmic values

Figure 58: Truncation errors for the custom RK method

$$R(h\lambda) = 1 + h\lambda + \frac{(h\lambda)^2}{2} + \frac{(h\lambda)^3}{6} \tag{132}$$

Figure 60 displays the stability region of the method. It corresponds very well to the expected shape, as presented in J. Butcher's book [8] page 101, and validates further the derivation of the method.

9.7 3D CSTR Test

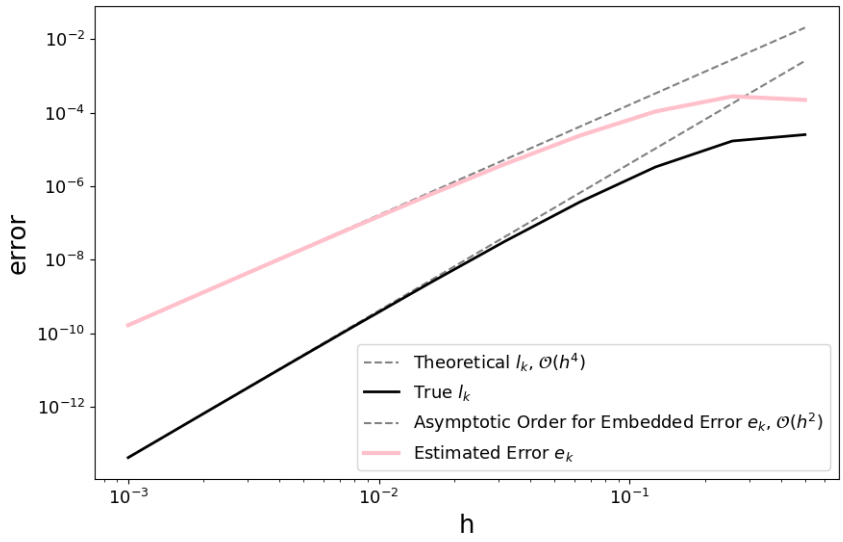


Figure 59: Asymptotic order for the embedded method. (k=10 for measure).

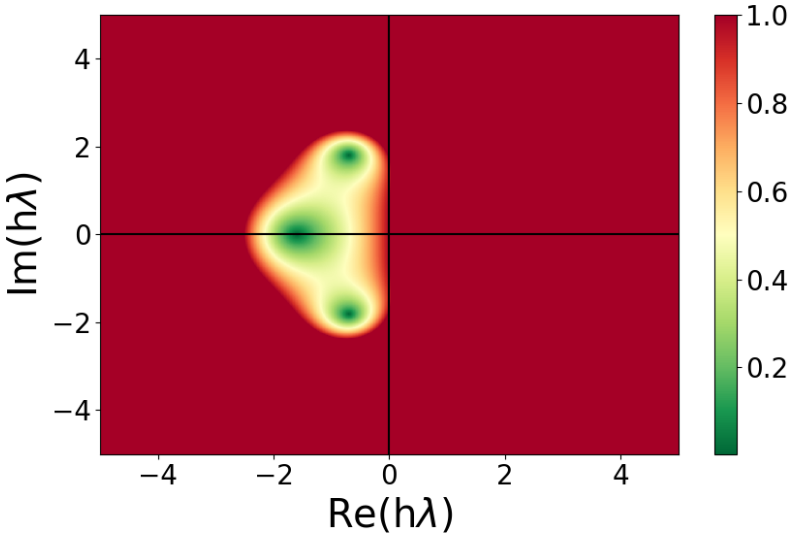


Figure 60: Custom RK method stability region

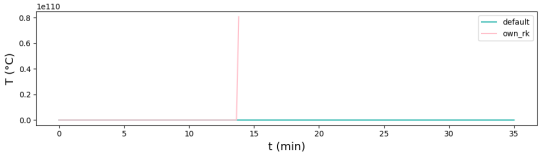
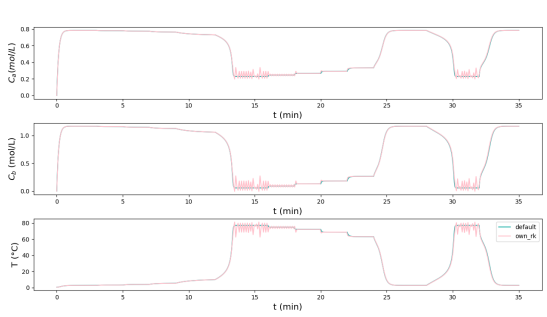
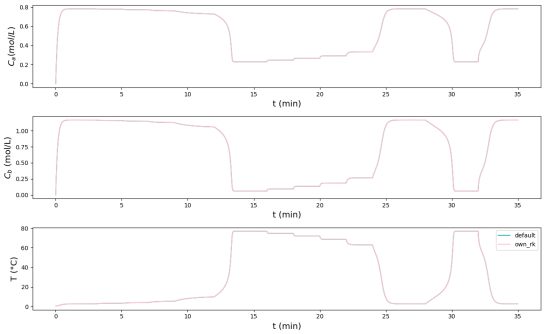


Figure 61: Divergence



(a) Note N = 400 not 200



(b) $h = 0.279$

Figure 62

10 ESDIRK23

10.1 Method Description

As observed earlier, explicit Runge-Kutta methods are not suitable in general for stiff problems as their stability region is bounded. It was seen with the implicit Euler methods that contrary to explicit methods, implicit methods do not suffer from such limitation.

The explicit singly diagonal implicit Runge-Kutta methods (ESDIRK) thus offer a welcome extension of the explicit Runge-Kutta methods for stiff problems. One could think a completely implicit Runge-Kutta method, where all coefficients of the A matrix could be non null. Nevertheless, In practice simplifications can be imposed on the coefficients of the A matrix to make stages computations more efficient, while retaining interesting stability properties.

The family of ESDIRK methods (including the embedded methods) adopts the following Butcher tableau

$$\begin{array}{c|cccc} 0 & 0 & & & \\ c_2 & a_{21} & \gamma & & \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & a_{s1} & a_{s2} & \dots & \gamma \\ \hline & a_{s1} & a_{s2} & \dots & \gamma \\ & \hat{b}_1 & \hat{b}_2 & \dots & \hat{b}_s \\ \hline & e_1 & e_2 & \dots & e_s \end{array}$$

Where γ is the single diagonal element referred to in the term singly diagonal. For such methods, the first stage is always explicit and trivial

$$\begin{aligned} T_1 &= t_k \\ \boldsymbol{X}_1 &= \boldsymbol{x}_k \end{aligned} \tag{133}$$

While the rest of the inner stages are implicit.

$$\begin{aligned} T_i &= t_k + hc_i \\ \boldsymbol{X}_i &= \boldsymbol{x}_k + h\sum_{j=1}^{s} a_{ij} f(T_j, \boldsymbol{X}_j) \end{aligned} \tag{134}$$

To determine the intermediate steps, a numerical approximation, namely the inexact Newton's method is used. It is important to note that the lower triangular definition of A ensures that any stage only depends on the evaluation of f for prior stages, as well as itself, considerably simplifying its determination. As for the standard Newton's method, a residual function is defined for each stage:

$$R_i(\boldsymbol{X}_i) := (\boldsymbol{X}_i - h\gamma f(T_i, \boldsymbol{X}_i)) - (\boldsymbol{x}_k + h\sum_{j=1}^{i-1} a_{ij} f(T_j, \boldsymbol{X}_j)) = 0 \tag{135}$$

Which allows to iteratively update the values of $\boldsymbol{X}_i$ -the nth iteration of the inexact Newton's method resulting in the approximation $\boldsymbol{X}_i^{[n]}$- using

$$\begin{aligned} M\Delta\boldsymbol{X}_i^{[n]} &= -R_i(\boldsymbol{X}_i^{[n]}) \\ \boldsymbol{X}_i^{[n+1]} &= \boldsymbol{X}_i^{[n]} + \Delta\boldsymbol{X}_i^{[n]} \end{aligned} \tag{136}$$

With $M \approx J_i(\boldsymbol{X}_i) = \frac{\partial R_i}{\partial \boldsymbol{X}_i}(\boldsymbol{X}_i)$, and initial step determined with the Euler step

$$\boldsymbol{X}_i^{[0]} = \boldsymbol{x}_k + c_i h f(T_1, \boldsymbol{X}_1) \tag{137}$$

Until the convergence is reached

$$||R_i(\boldsymbol{X}_i^{[n]})|| = \max_{j=1,\dots,D} \frac{|R_i(\boldsymbol{X}_i^{[n]})_j|}{\max\{\text{atol}_j, \text{rtol}_j \boldsymbol{X}_i^{[n]}\}} \tag{138}$$

The approximation $M \approx J_i(\boldsymbol{X}_i)$ can lead to some divergence and thus to a greater number of iterations. Defining

$$\alpha = \frac{||R_i(\boldsymbol{X}_i^{[n+1]})||}{||R_i(\boldsymbol{X}_i^{[n]})||} \tag{139}$$

allows to monitor the convergence rate. If $\alpha \geq 1$, the iteration should be stopped, the step size reduced and M updated. If the convergence is deemed too slow, reducing the step size and/or updating M is also possible.

After these insights into the inner workings of the methods of the ESDIRK family, let's focus on the ESDIRK23 method and derive its stages coefficients using the order and consistency conditions.

The ESDIRK23 method aims to achieve 2nd order accuracy for its main method, with an order 3 embedded method for error estimation. Again, [1], page 88, shows that a stage 3 method can be used here. The Butcher's tableau of the method is as follows:

0	0		
c_2	a_{21}	γ	
1	a_{31}	a_{32}	γ
	a_{31}	a_{32}	γ
	$\hat{b}_1$	$\hat{b}_2$	$\hat{b}_3$
	e_1	e_2	e_3

for the main method, considering the order condition exposed in Equation 107 and the stage order conditions exposed in Equation 104 results in the following system

$$\begin{cases} c_2 = a_{21} + \gamma \\ 1 = a_{31} + a_{32} + \gamma \\ a_{32}c_2 + \gamma = \frac{1}{2} \end{cases} \tag{140}$$

Furthermore, it is also desirable for the main method to be L-Stable. This means that its stability polynomial R should converge to 0 when far away from the origin of the complex plane, $\lim_{z\to\infty} R(z) = 0$. The stability polynomial of the main method computed using Equation 114

$$R(z) = \frac{1}{(1-\gamma z)^2}(1 + (a_{31} + a_{32} - \gamma)z + (a_{21}a_{32} - a_{31}\gamma)z^2) \tag{141}$$

Implying that for L-stability to be verified, the following must hold:

$$a_{21}a_{32} - a_{31}\gamma = 0 \tag{142}$$

Furthermore, it is also wanted to have an order 2 for the second stage $\boldsymbol{X}_2$. Applying the pre-exposed stage order conditions in Equation 103, up to order 2, results in the additional equation[7]

$$\gamma c_2 = \frac{c_2^2}{2} \tag{143}$$

To sum up, the coefficients of the Butcher's tableau for the main method must verify

$$\begin{cases} c_2 = a_{21} + \gamma \\ 1 = a_{31} + a_{32} + \gamma \\ a_{32}c_2 + \gamma = \frac{1}{2} \\ a_{21}a_{32} - a_{31}\gamma = 0 \\ \gamma c_2 = \frac{c_2^2}{2} \end{cases} \tag{144}$$

There are thus 5 equations and 5 unknowns. The solution corresponding to $\gamma = \frac{2-\sqrt{w}}{2}$ will be preferred over $\gamma = \frac{2+\sqrt{w}}{2}$ to keep c_2 bounded between 0 and 1.

On the other hand, applying again the order conditions on the stage coefficients of the embedded method result in

$$\begin{cases} \hat{b}_1 + \hat{b}_2 + \hat{b}_3 = 1 \\ \hat{b}_2 c_2 + \hat{b}_3 = \frac{1}{2} \\ \hat{b}_2 c_2^2 + \hat{b}_3 = \frac{1}{3} \\ 2\hat{b}_2 c_2 \gamma + \hat{b}_3(c_2 a_{32} + \gamma) = \frac{1}{6} \end{cases} \tag{145}$$

The last two equations are dependent, so overall, there are three equations for 3 unknowns, resulting in the now completed Butcher's tableau

[7] The rest of the equation for C(1) and C(2) do not provide further information for the system.

0	0		
2γ	γ	γ	
1	$\frac{1-\gamma}{2}$	$\frac{1-\gamma}{2}$	γ
	$\frac{1-\gamma}{2}$	$\frac{1-\gamma}{2}$	γ
	$\frac{6\gamma-1}{12\gamma}$	$\frac{1}{12\gamma(1-2\gamma)}$	$\frac{1-3\gamma}{3(1-2\gamma)}$
	$\frac{1-6\gamma^2}{12\gamma}$	$\frac{6\gamma(1-2\gamma)(1-\gamma)-1}{12\gamma(1-2\gamma)}$	$\frac{6\gamma(1-\gamma)-1}{3(1-2\gamma)}$

10.2 Method Stability

The stability polynomial of the ESDIRK23 method becomes

$$R(h\lambda) = \frac{1 + (1-2\gamma)h\lambda}{(1-\gamma h\lambda)^2} \tag{146}$$

Which results in the stability region displayed on Figure 63.

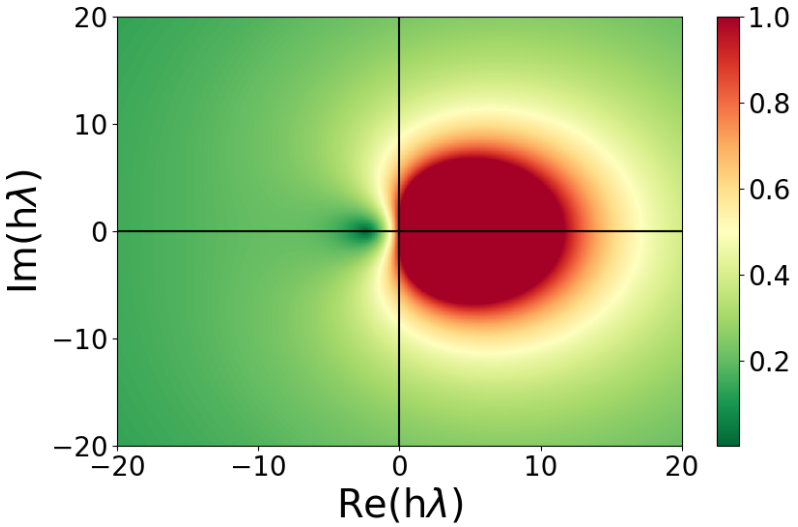


Figure 63: Region of absolute stability (only red is unstable) for the ESDIRK23 method

The ESDIRK23 is thus very clearly A-Stable; its region of stability containing the left half-plane of the complex space. It is also, by construction L-Stable, as $\lim_{z\to\infty} R(z) = 0$.

This implies that the step size can be chosen for the method only for accuracy reasons. The convergence is assured. This means that the method is very well suited to handle stiff problems.

10.3 Implementation

```
# ESDIRK23
# Butcher Tableau
gamma = (2 - np.sqrt(2)) / 2
C = np.array([0, 2 * gamma, 1])
A = np.array([
    [0, 0, 0],
    [gamma, gamma, 0],
    [(1 - gamma) / 2, (1 - gamma) / 2, gamma]
])
B = np.array([(1 - gamma) / 2, (1 - gamma) / 2, gamma])
B_hat = np.array(
    [(6 * gamma - 1) / (12 * gamma), 1 / (12 * gamma * (1 - 2 * gamma)),
   (1 - 3 * gamma) / (3 * (1 - 2 * gamma))])
E = B - B_hat

P_ESDIRK23 = 3


def ode_solver(f, J, t0, tf, N, x0, adaptive_step_size=False, **kwargs):

    dt = (tf - t0) / N
    t = t0
    x = x0

    T = [t0]
    X = [x0]
    controllers = {
        'dt': [dt],
        'r': [0],
    }

    # Parameters needed
    kwargs, newtons_tol, newtons_max_iters = parse_newtons_params(kwargs)
    kwargs, abstol, reltol, epstol, facmax, facmin =
   parse_adaptive_step_params(kwargs)
    kwargs, max_diverged_steps, newtons_tau = parse_inexact_newtons_params
   (kwargs, epstol)
    kwargs, hmax, hmin = parse_adaptive_step_params_esdirk(kwargs)

    p = P_ESDIRK23
    D = x0.shape[0]
    I = np.eye(D)
    T_stages = np.zeros((p,))
    X_stages = np.zeros((p, D))
    F_stages = np.zeros((p, D))

    F_stages[-1, :] = f(t, x, **kwargs)

    n_steps = 1
    n_diverged_steps = 0

    while (t < tf) & (n_diverged_steps < max_diverged_steps):

```

```
        if (t + dt > tf):
            dt = tf - t

        J_eval = J(t, x, **kwargs)
        M = I - dt * gamma * J_eval
        L, U = sp.linalg.lu(M, permute_l=True)

        # Stage 0
        T_stages[0] = t
        X_stages[0, :] = x
        F_stages[0, :] = F_stages[-1, :]

        i = 1
        step_diverged = False
        while (i < p) & (not step_diverged):

            # Inexact Newton's method
            T_stages[i] = t + dt * C[i]
            X_stages[i, :] = x + dt * C[i] * F_stages[i, :]  # Initial EE guess
            sum_i = x + dt * A[i, :i] @ F_stages[:i, :]  # i not included in :i while it is in matlab
            F_stages[i, :] = f(T_stages[i], X_stages[i, :], **kwargs)

            R = X_stages[i, :] - dt * gamma * F_stages[i, :] - sum_i

            # Control convergence of inexact Newton's method
            r_newton = np.amax(np.abs(R) / np.maximum(abstol, np.abs(X_stages[i, :]) * reltol))

            n_iter_in_newton = 0
            r_newton_prev = r_newton
            while (r_newton > newtons_tau) & (not step_diverged):
                # Next iteration
                dX = np.linalg.solve(U, np.linalg.solve(L, -R))
                X_stages[i, :] = X_stages[i, :] + dX

                # Convergence
                F_stages[i, :] = f(T_stages[i], X_stages[i, :], **kwargs)
                R = X_stages[i, :] - (dt * gamma * F_stages[i, :]) - sum_i
                r_newton = np.amax(np.abs(R) / np.maximum(abstol, np.abs(X_stages[i, :]) * reltol))
                n_iter_in_newton += 1

                # Convergence rate
                alpha = r_newton / r_newton_prev
                newton_step_diverged = (alpha >= 1)

                # Nbr iterations
                reached_max_newton_iterations = (n_iter_in_newton >= newtons_max_iters)

                # Composed convergence
```

```
99                step_diverged = newton_step_diverged |
    reached_max_newton_iterations
100
101           # Next stage
102           i += 1
103
104       # Error estimation
105       e = dt * E @ F_stages
106       r_step_control = np.amax(
107           np.abs(e) / np.maximum(abstol, np.abs(X_stages[-1, :]) *
    reltol))
108
109       if step_diverged:
110           accept_step = False
111           dt = dt / 2
112       else:
113           accept_step = (r_step_control <= 1)
114
115           if accept_step:
116               if n_steps == 1:
117                   dt_modified = (epstol / r_step_control) ** (
118                                 1 / (p + 1)) * dt  # First step ->
    asymptotic controller
119               else:
120                   dt_modified = (dt / controllers['dt'][-1]) \
121                                     * ((epstol / r_step_control) ** (1 / (
    p + 1))) \
122                                     * ((controllers['r'][-1] /
    r_step_control) ** (1 / (p + 1))) \
123                                     * dt  # PI controller
124
125               dt = np.minimum(np.maximum(dt_modified, dt * hmin), dt *
    hmax)
126           else:
127               dt = (epstol / r_step_control) ** (1 / (p + 1)) * dt  #
    Rejected step -> asymptotic controller
128
129       controllers['dt'].append(dt)
130       controllers['r'].append(r_step_control)
131
132       if accept_step:
133
134           t = T_stages[-1]
135           x = np.copy(X_stages[-1, :])
136
137           T.append(t)
138           X.append(x)
139
140           n_steps += 1
141           n_diverged_steps = 0
142
143       else:
144           n_diverged_steps += 1
```

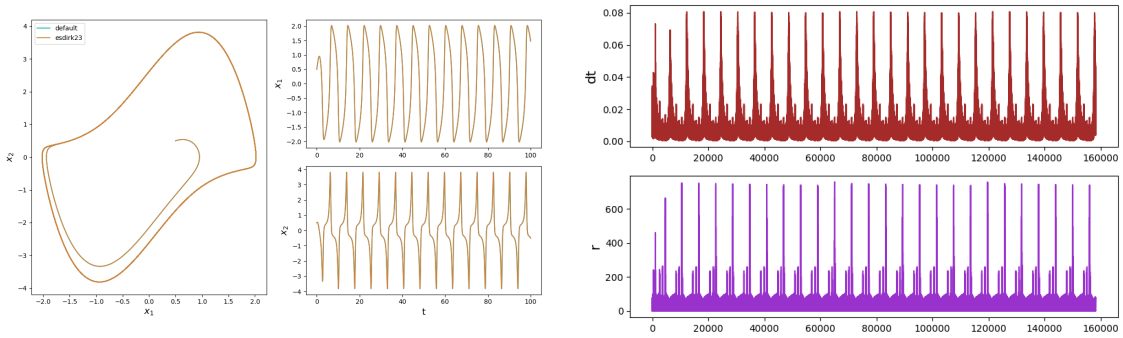
```
145
146     if n_diverged_steps == max_diverged_steps:
147         print(f"/!\\ Max diverged steps {max_diverged_steps} reached, the
    solver stopped.")
148
149     T = np.array(T)
150     X = np.array(X).reshape((-1, x0.shape[0]))
151     controllers['dt'] = np.array(controllers['dt'])
152     controllers['r'] = np.array(controllers['r'])
153
154     return X, T, controllers
```

Listing 18: Implementation of the ESDIRK23 method

10.4 Test on the Van der Pol Problem

10.4.1 For $\mu = 2$

Number of evaluations is very high, divergence in the approximate newton step?



(a) (b)

Figure 64

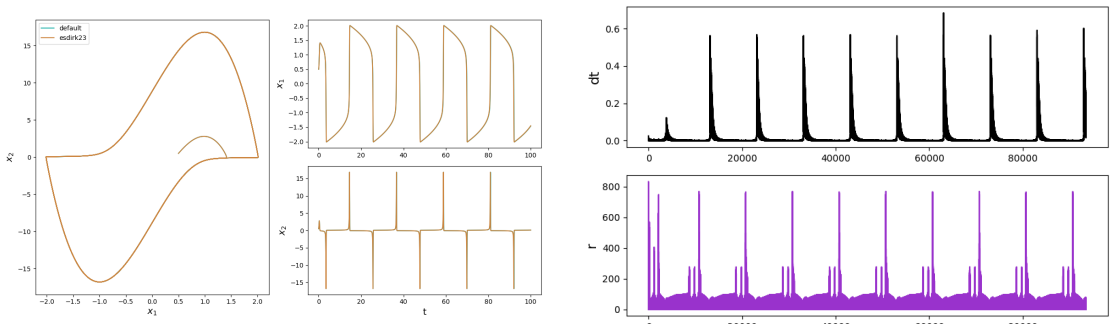
10.4.2 For $\mu = 12$

10.5 Test on the A-CSTR Problem

10.5.1 CSTR 1D

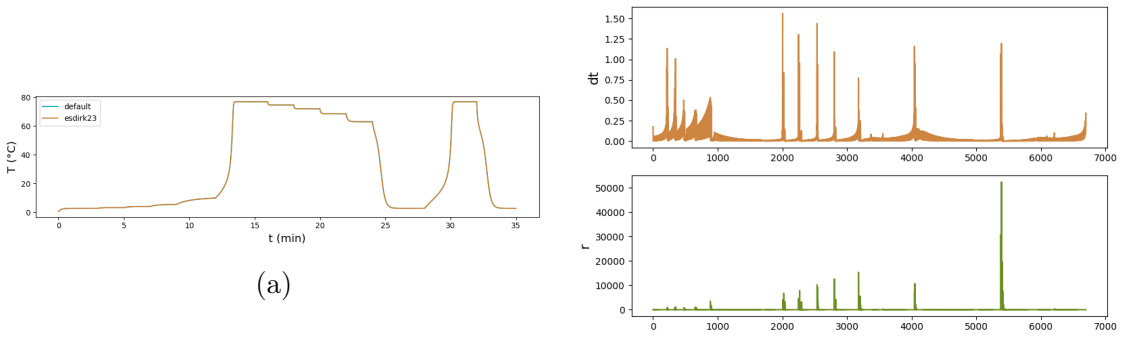
10.5.2 CSTR 3D

10.6 Comparison



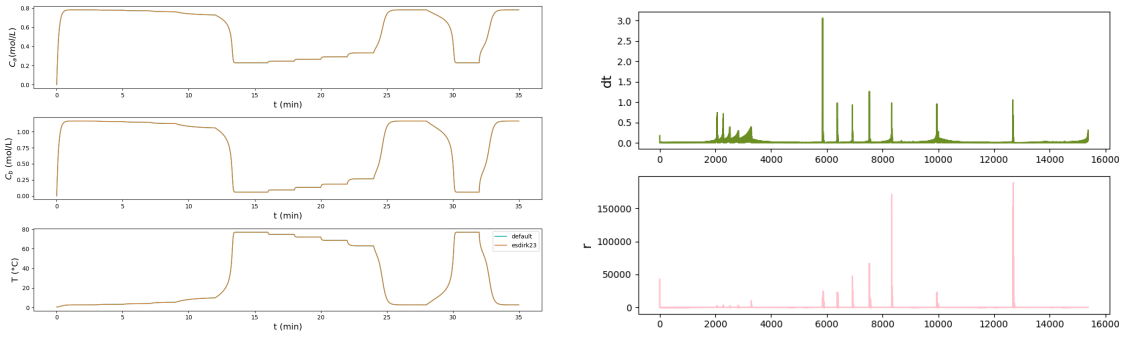
(a) (b)

Figure 65



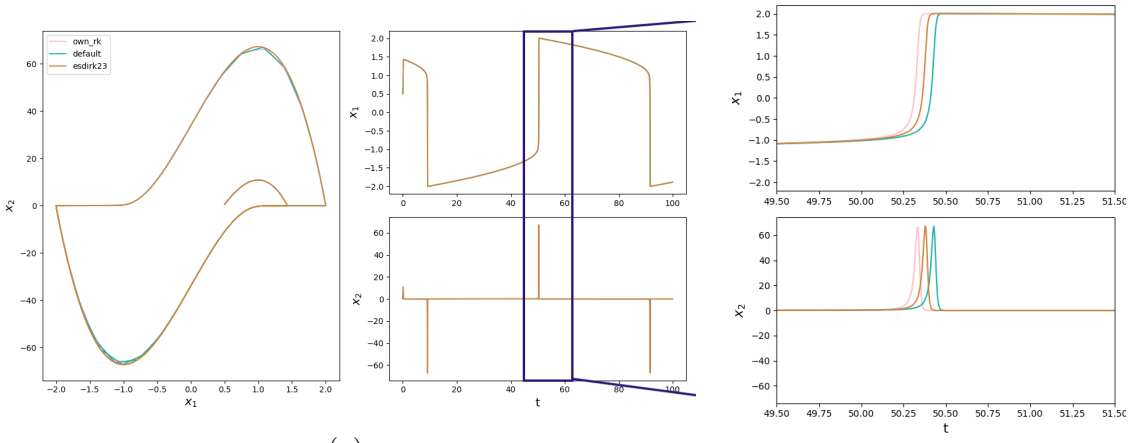
(a) (b)

Figure 66



(a) (b)

Figure 67



(a) (b)

Figure 68: $\mu = 50$

Discussion

List of Figures

List of Tables

Listings

Nomenclature

A-CSTR Adiabatic continuous stirred reactor

DOPRI54 Dormand-Prince 5(4) method

ESDIRK explicit singly diagonal implicit Runge-Kutta method

IVP Initial value problem

ODE Ordinary differential equation

RHS Right hand side

RK4 Classical Runge-Kutta method

SDE Stochastic differential equation

References

[1] U. M. Ascher and L. R. Petzold, *Computer methods for ordinary differential equations and differential-algebraic equations*, vol. 61. Siam, 1998.

[2] V. I. Arnol'd, *Mathematical methods of classical mechanics*, vol. 60. Springer Science & Business Media, 2013.

[3] H. G. Bock, "Numerical treatment of inverse problems in chemical reaction kinetics," in *Modelling of chemical reaction systems*, pp. 102–125, Springer, 1981.

[4] K. Wainwright *et al.*, *Fundamental methods of mathematical economics/Alpha C. Chiang, Kevin Wainwright.* Boston, Mass.: McGraw-Hill/Irwin,, 2005.

[5] E. Hairer, S. P. Nørsett, and G. Wanner, "Solving ordinary differential equations i. nonstiff problems, volume 8 of," 1993.

[6] M. R. Wahlgreen, E. Schroll-Fleischer, D. Boiroux, T. K. S. Ritschel, H. Wu, J. K. Huusom, and J. J. B., "Nonlinear model predictive control for an exothermic reaction in an adiabatic cstr," 2020.

[7] E. Süli and D. F. Mayers, *An introduction to numerical analysis*. Cambridge university press, 2003.

[8] J. Butcher, "Numerical methods for ordinary differential equations," 2008.

[9] A. Iserles, *A first course in the numerical analysis of differential equations*. No. 44, Cambridge university press, 2009.

11 Appendix A