

Danmarks
Tekniske
Universitet



02686

Scientific Computing for Differential Equations

AUTHORS

Pierre Segonne - s182172

May 23, 2020

Contents

1	Explicit ODE solver	8
1.1	Description of the Explicit Euler Method	8
1.2	Python Implementation	8
1.3	Python Implementation with Adaptive Step Size	9
1.4	Test on the Van der Pol Problem	12
1.4.1	For $\mu = 2$	12
1.4.2	For $\mu = 12$	13
1.5	Test on the A-CSTR Problem	13
1.5.1	CSTR 1D	14
1.5.2	CSTR 3D	14
2	Implicit ODE solver	17
2.1	Description of the Implicit Euler algorithm	17
2.2	Python Implementation	18
2.3	Python Implementation with Adaptive Step Size	18
2.4	Test on the Van der Pol Problem	18
2.4.1	For $\mu = 2$	18
2.4.2	For $\mu = 12$	18
2.5	Test on the A-CSTR Problem	18
2.5.1	CSTR 1D	18
2.5.2	CSTR 3D	18
3	Test equation for ODEs	18
3.1	Analytical Solution	18
3.2	Local and Global Truncation Errors	19
3.3	Error Expression for Euler Methods	20
3.3.1	Explicit Euler Method	20
3.3.2	Implicit Euler Method	21
3.4	Plot of Local Error	22
3.5	Plot of Global Error	22
3.6	On Stability	22
4	Solvers for SDEs	25
4.1	Multivariate Standard Wiener Process	25
4.2	Explicit-Explicit Method	26
4.3	Implicit-Explicit Method	27
4.4	Test on the Van der Pol Problem	29
4.4.1	For $\mu = 2$	29
4.4.2	For $\mu = 12$	29
4.5	Test on the A-CSTR Problem	29
4.5.1	CSTR 1D	29
4.5.2	CSTR 3D	29

5	Test equation for SDEs	30
5.1	Analytical Solution	30
5.2	Comparison of the Numerical Solution to the Analytical Solution	31
5.3	Distribution of the Final State of the Numerical Solution	32
5.4	Comparison of Moments	33
5.5	Order of the Explicit-Explicit Method	34
5.6	Order of the Implicit-Explicit Method	34
6	Classical Runge-Kutta method with fixed time step size	36
6.1	Method Description	36
6.2	Implementation	38
6.3	Order and Stability	40
6.4	Test on the Van der Pol Problem	42
6.4.1	For $\mu = 2$	42
6.4.2	For $\mu = 12$	42
6.5	Test on the A-CSTR Problem	42
6.5.1	CSTR 1D	42
6.5.2	CSTR 3D	42
7	Classical Runge-Kutta method with adaptive time step	42
7.1	Method Description	43
7.2	Implementation	43
7.3	Order and Stability	43
7.4	Test on the Van der Pol Problem	43
7.4.1	For $\mu = 2$	43
7.4.2	For $\mu = 12$	43
7.5	Test on the A-CSTR Problem	43
7.5.1	CSTR 1D	43
7.5.2	CSTR 3D	43
8	Dormand-Prince 5(4)	43
8.1	Method Description	43
8.2	Implementation	44
8.3	Order and Stability	44
8.4	Test on the Van der Pol Problem	44
8.4.1	For $\mu = 2$	44
8.4.2	For $\mu = 12$	44
8.5	Test on the A-CSTR Problem	44
8.5.1	CSTR 1D	44
8.5.2	CSTR 3D	44
9	Design your own explicit Runge-Kutta method	44
9.1	Order Conditions	45
9.2	Derivation of Coefficients of Error Estimator	46

9.3	Butcher Tableau	46
9.4	Method Order	46
9.5	Method Stability	47
9.6	3D CSTR Test	48
10	ESDIRK23	48
10.1	Method Description	48
10.2	Method Stability	48
10.3	Implementation	48
10.4	Test on the Van der Pol Problem	48
10.4.1	For $\mu = 2$	48
10.4.2	For $\mu = 12$	48
10.5	Test on the A-CSTR Problem	48
10.5.1	CSTR 1D	48
10.5.2	CSTR 3D	48
10.6	Comparison	48
	List of Figures	I
	List of Tables	III
	List of Listings	IV
	Nomenclature	V
	References	VI
11	Appendix A	VII

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

$$\dot{\mathbf{x}}(t) = f(t, \mathbf{x}(t)) \quad (1)$$

where $\dot{\mathbf{x}} \in \mathbb{R}^n$ is the temporal derivative of $\mathbf{x} \in \mathbb{R}^n$. 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 \mathbf{x} .

$$\dot{\mathbf{x}}(t) = f(t, \mathbf{x}(t)), \quad \mathbf{x}(t_0) = \mathbf{x}_0 \quad (2)$$

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* interface¹. 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:

$$y''(t) = \mu(1 - y(t)^2)y'(t) - y(t) \quad (3)$$

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

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

The function

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

has the following Jacobian matrix:

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

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 $\mu = 2$

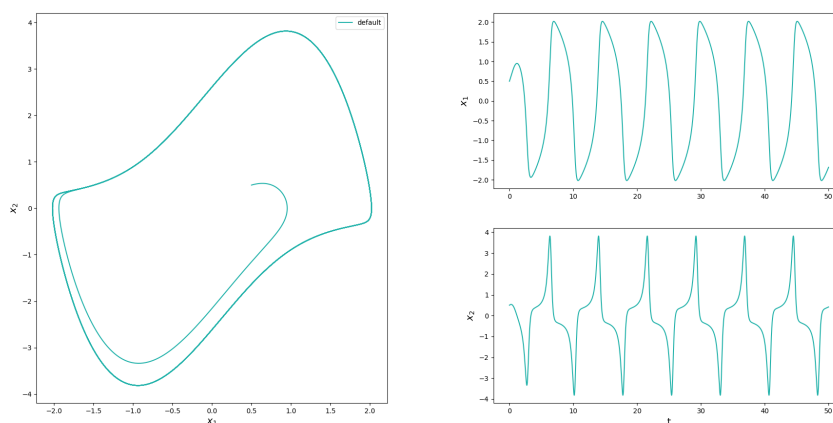


Figure 1: Van der Pol oscillator simulation with default ode solver for parameter $\mu = 2$

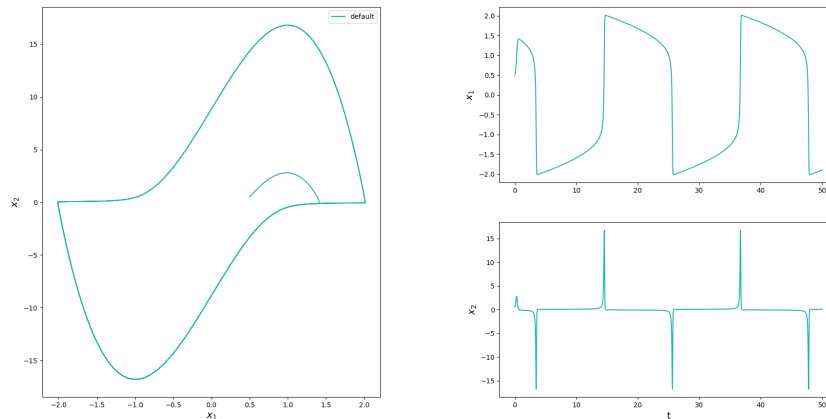
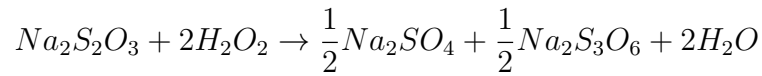


Figure 2: Van der Pol oscillator simulation with default ode solver for parameter $\mu = 12$

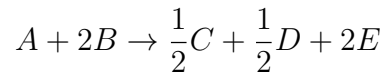
The A-CSTR

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

The reaction is the following



which can be simplified to



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:

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

where

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

and

$$r(C_A, C_B, T) = k(T)C_AC_B \quad (9)$$

and finally

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

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,

$$f\left(t, \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}\right) = \begin{bmatrix} \frac{F}{V}(C_{A,in} - x_1) - k_0 \exp\left(\frac{-E_a}{Rx_3}\right)x_1x_2 \\ \frac{F}{V}(C_{B,in} - x_2) - 2k_0 \exp\left(\frac{-E_a}{Rx_3}\right)x_1x_2 \\ \frac{F}{V}(T_{in} - x_3) + \beta k_0 \exp\left(\frac{-E_a}{Rx_3}\right)x_1x_2 \end{bmatrix} \quad (11)$$

has the following Jacobian:

$$J\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}\right) = \begin{bmatrix} -\frac{F}{V} - k_0 \exp\left(\frac{-E_a}{Rx_3}\right)x_2 & -k_0 \exp\left(\frac{-E_a}{Rx_3}\right)x_1 & -\frac{k_0 E_a}{Rx_3^2} \exp\left(\frac{-E_a}{Rx_3}\right)x_1x_2 \\ -2k_0 \exp\left(\frac{-E_a}{Rx_3}\right)x_2 & -\frac{F}{V} - 2k_0 \exp\left(\frac{-E_a}{Rx_3}\right)x_1 & -2\frac{k_0 E_a}{Rx_3^2} \exp\left(\frac{-E_a}{Rx_3}\right)x_1x_2 \\ \beta k_0 \exp\left(\frac{-E_a}{Rx_3}\right)x_2 & \beta k_0 \exp\left(\frac{-E_a}{Rx_3}\right)x_1 & -\frac{F}{V} + \beta \frac{k_0 E_a}{Rx_3^2} \exp\left(\frac{-E_a}{Rx_3}\right)x_1x_2 \end{bmatrix} \quad (12)$$

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:

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

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

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

has the following Jacobian:

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

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.

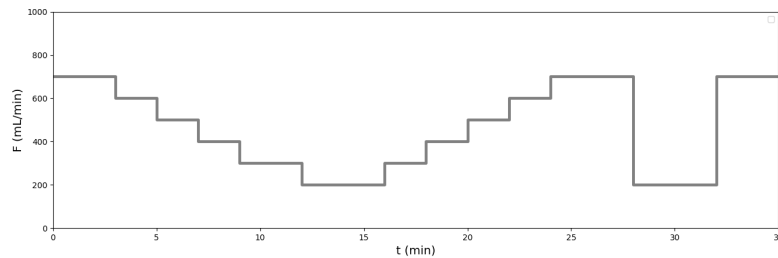


Figure 3: Evolution of the flow F over the integration period

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.

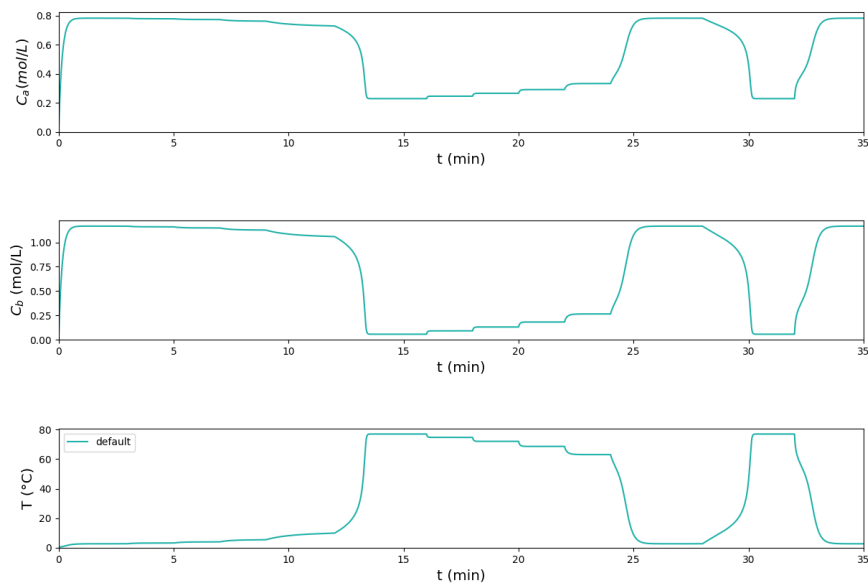


Figure 4: Evolution of the state in the 3D model

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):
```

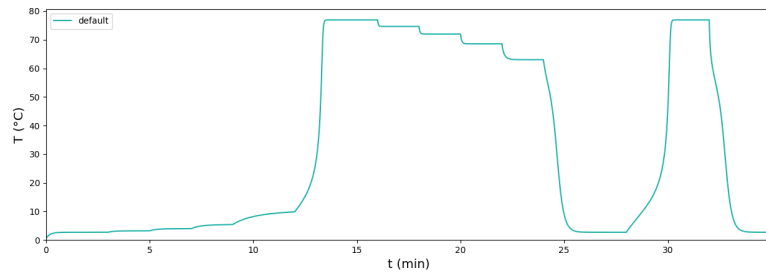


Figure 5: Evolution of the temperature in the 1D model

Where,

- t_0 is the initial time
- t_f 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:

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

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

```
1 def J(t, X, **kwargs):
2     J = ...
3     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, refactor?
- 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 extremely 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:

$$\mathbf{x}'(t) = f(t, \mathbf{x}(t)), \quad \mathbf{x}(t_0) = \mathbf{x}_0 \quad (16)$$

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 $\Delta t = t_{k+1} - t_k = h$:

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

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

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

Therefore, if the evaluation of the RHS of the ODE for all t_k is possible, and if $\mathbf{x}_0 = \mathbf{x}(t_0)$ is known, this approximation then constitutes a valid way to iteratively and numerically solve the ODE. Instead of determining the exact solution $\mathbf{x}(t_k)$ for every t_k , a numerical approximation $\mathbf{x}_k \simeq \mathbf{x}(t_k)$ 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 \mathbf{x}_{k+1} only depends on the previous iteration \mathbf{x}_k and the evaluation of the RHS function at the previous iteration $f(t_k, \mathbf{x}_k)$.

```
1 def ode_solver(f, J, t0, tf, N, x0, **kwargs):
2
3     dt = (tf - t0)/N
4
5     T = [t0]
6     X = [x0]
7
8     for k in range(N):
9         f_eval = f(T[-1], X[-1], **kwargs)
10        X.append(X[-1] + dt*f_eval)
11        T.append(T[-1] + dt)
12
13    T = np.array(T)
14    X = np.array(X)
15
16    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.

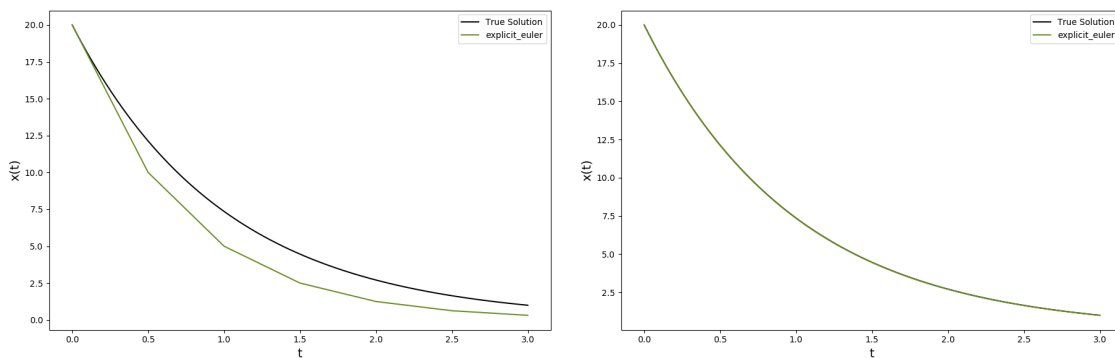


Figure 6: Explicit Euler approximation for large and small step sizes

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.

1.3 Python Implementation with Adaptive Step Size

Including a step size controller in the method is a good way to alleviate the aforementioned short-comings. Controlling the step size during the simulation can be 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.

An efficient way to estimate the error is to perform step doubling.

Concretely, the estimated error will be at each iteration

$$e_{k+1} = |\hat{\mathbf{x}}_{k+1} - \mathbf{x}_{k+1}| \quad (19)$$

where

$$\begin{aligned}\hat{\mathbf{x}}_{k+1} &= \hat{\mathbf{x}}_{k+1/2} + \frac{h}{2}f(t_k, \hat{\mathbf{x}}_{k+1/2}) \\ \hat{\mathbf{x}}_{k+1/2} &= \mathbf{x}_k + \frac{h}{2}f(t_k, \mathbf{x}_k)\end{aligned}\quad (20)$$

and can then be used to compute

$$r_{k+1} = \max\left(\frac{e_{k+1}}{\max(\text{abstol}, \hat{\mathbf{x}}_{k+1}\text{reltol})}\right) \quad (21)$$

which is used to know whether the step should be accepted ($r < 1$) or not. It is also by the asymptotic step size controller to determine the next step size

$$h_{k+1} = \max(\text{min_factor}, \min\left(\sqrt{\frac{\epsilon}{r_{k+1}}}, \text{max_factor}\right))h_k \quad (22)$$

In Figure 7 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 7, is indistinguishable from the true solution.

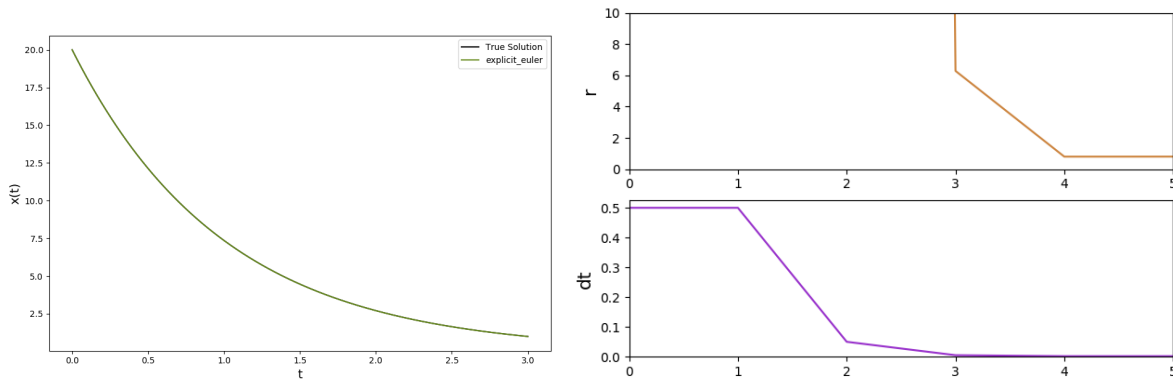


Figure 7: Explicit Euler approximation with adaptive step size

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 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 on equations with great variations, as it will still require an unpractical step size value to stay close to the true solution.

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

```

2     dt = (tf - t0)/N
3
4     T = [t0]
5     X = [x0]
6
7     kwargs, abstol, reltol, epstol, facmax, facmin =
    parse_adaptive_step_params(kwargs)
8
9     t = t0
10    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

```

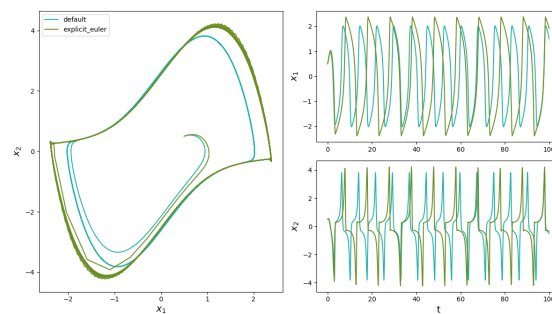
Listing 2: Implementation of the explicit Euler method with adaptive step size

1.4 Test on the Van der Pol Problem

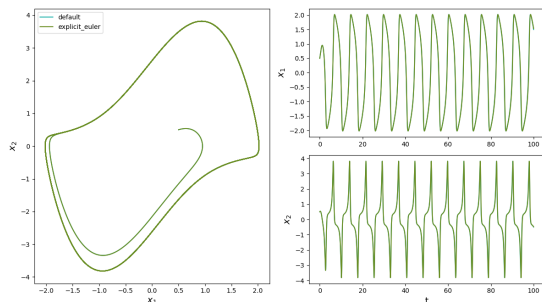
1.4.1 For $\mu = 2$

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.

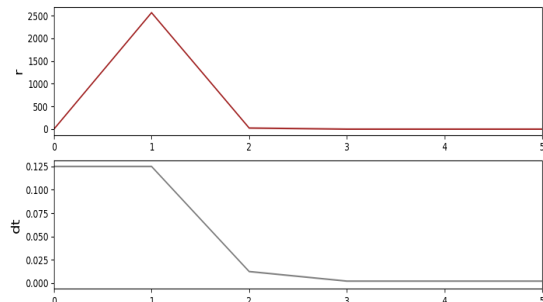
First, the method was applied with a large step size, $h = 0.125$ for a time interval of $[0, 100]$ with initial coordinates $\mathbf{x}_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}$. The following figures display the results of the different simulations



(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 8: 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 8, 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 9

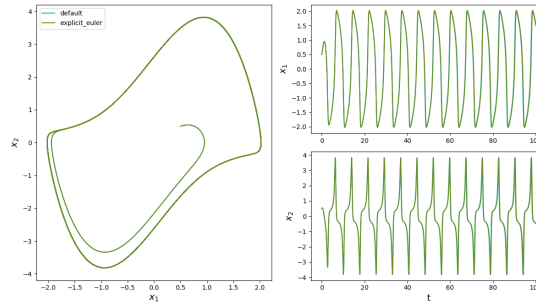


Figure 9: Explicit Euler method on the Van der Pol problem. $\mu = 2$ and $h = 0.001$

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.

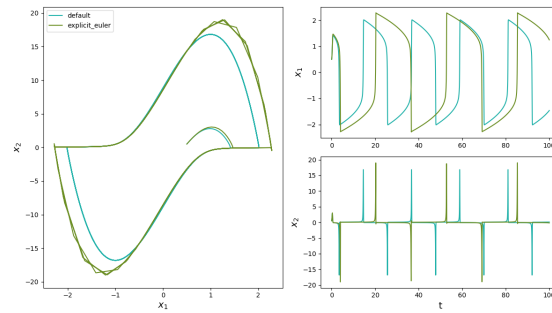
The increased stiffness of the problem is apparent (Figure 10) given the inability of the resulting approximation to handle the peaks of the oscillator, which correspond to areas with more variations in the solution. Here again, introducing the step size controller allows for a much more precise solution. 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 areas 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 11.

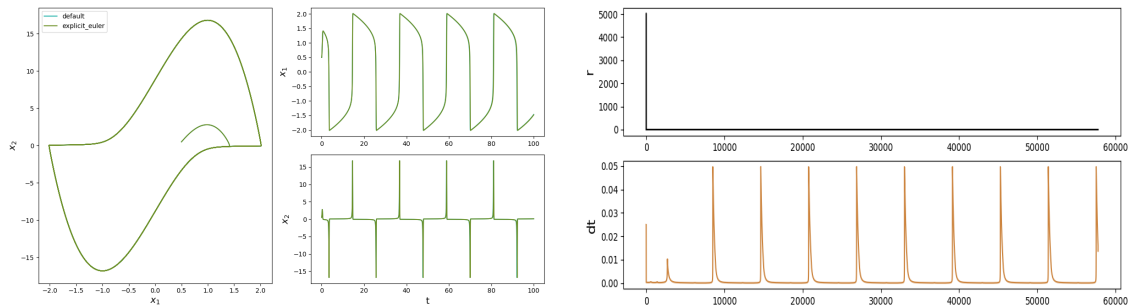
Contrary to what was seen for $\mu = 2$, here decreasing the step size to the same level is not enough to reach a satisfactory approximation. After a few cycles, a clear divergence appears between the default ODE solver's solution and the explicit Euler method's solution. This is a good demonstration 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 10: Simulation results for large step size on the Van der Pol problem. $\mu = 12$ and $h = 0.025$

1.5.1 CSTR 1D

Figure 12 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 13

1.5.2 CSTR 3D

The CSTR 3D model displays the same behaviour than its 1D counterpart, as can be seen in Figure 14 and 15.

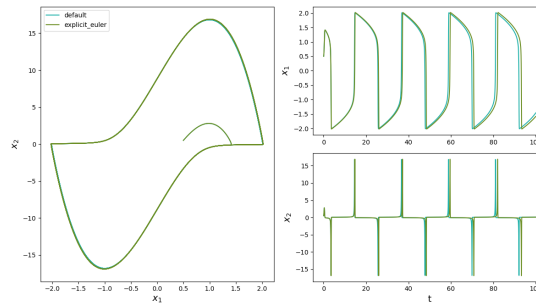
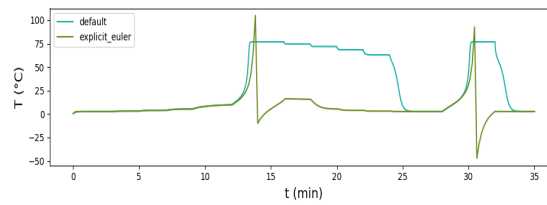
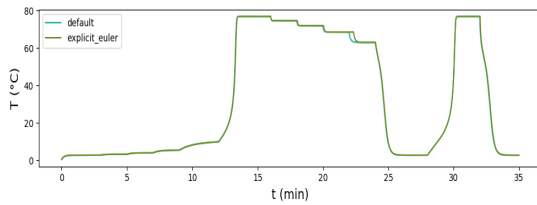


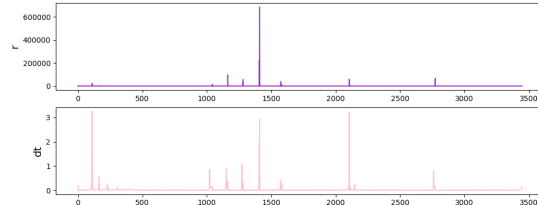
Figure 11: Explicit Euler method on the Van der Pol problem. $\mu = 12$ and $h = 0.001$



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

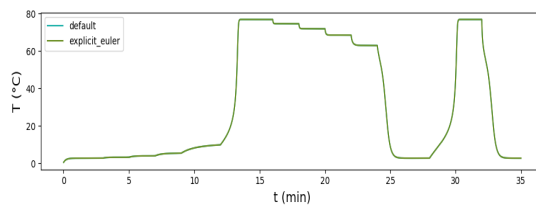
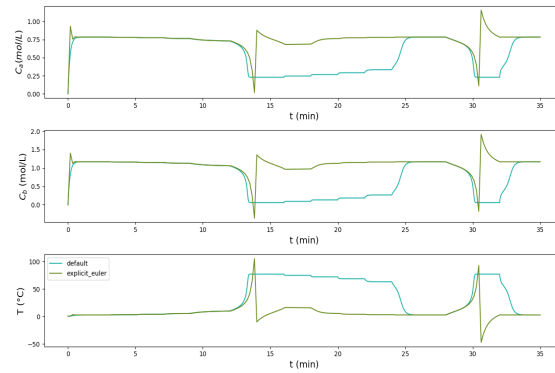
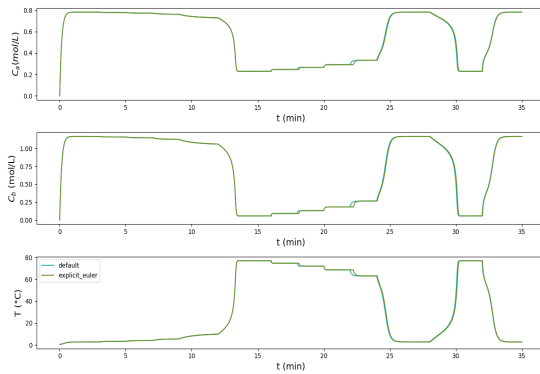


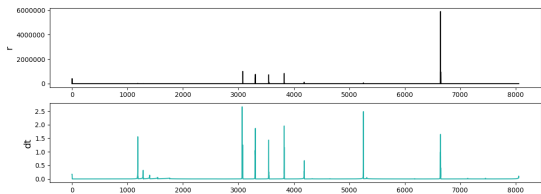
Figure 13: Explicit Euler method on the CSTR 1D problem. $h = 0.00035$



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

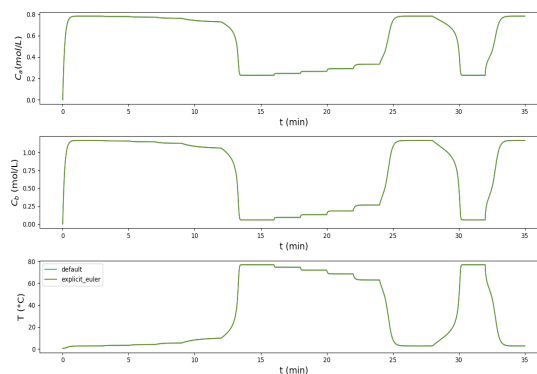


Figure 15: Explicit Euler method on the CSTR 3D problem. $h = 0.00035$

2 Implicit ODE solver

2.1 Description of the Implicit Euler algorithm

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

$$\mathbf{x}'(t) = f(t, \mathbf{x}(t)), \quad \mathbf{x}(t_0) = \mathbf{x}_0 \quad (23)$$

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

$$\mathbf{x}(t_k) = \sum_{n=0}^{\infty} \frac{\mathbf{x}^{(n)}(t_{k+1})}{n!} (t_k - t_{k+1})^n = \sum_{n=0}^{\infty} \frac{\mathbf{x}^{(n)}(t_k)}{n!} (-h)^n \quad (24)$$

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

$$\mathbf{x}(t_k) \simeq \mathbf{x}(t_{k+1}) - h\mathbf{x}'(t_{k+1}) = \mathbf{x}(t_k) - hf(t_{k+1}, \mathbf{x}(t_{k+1})) \quad (25)$$

And therefore

$$\mathbf{x}(t_{k+1}) \simeq \mathbf{x}(t_k) + hf(t_{k+1}, \mathbf{x}(t_{k+1})) \quad (26)$$

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

$$R(\mathbf{x}(t_{k+1})) = \mathbf{x}(t_{k+1}) - \mathbf{x}(t_k) - hf(t_{k+1}, \mathbf{x}(t_{k+1})) = 0 \quad (27)$$

Determining $\mathbf{x}(t_{k+1})$ 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 $\mathbf{x}_{k+1}^{[i]}$ is the i th iteration of the approximation of $\mathbf{x}(t_{k+1})$, the first order Taylor expansion of R in $\mathbf{x}_{k+1}^{[i+1]}$ with respect to $\mathbf{x}_{k+1}^{[i]}$ is:

$$R(\mathbf{x}_{k+1}^{[i+1]}) \simeq R(\mathbf{x}_{k+1}^{[i]}) + \frac{\partial R}{\partial \mathbf{x}}(\mathbf{x}_{k+1}^{[i]})(\mathbf{x}_{k+1}^{[i+1]} - \mathbf{x}_{k+1}^{[i]}) = 0 \quad (28)$$

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

$$-\frac{\partial R}{\partial \mathbf{x}}(\mathbf{x}_{k+1}^{[i]})\Delta \mathbf{x} = R(\mathbf{x}_{k+1}^{[i]}) \quad (29)$$

thus constitutes an update rule for the approximation of $\mathbf{x}(t_{k+1})$. 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 $\mathbf{x}(t_0) = \mathbf{x}_0$ is known, a numerical method to solve the ODE appears. Iteratively,

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

2.2 Python Implementation

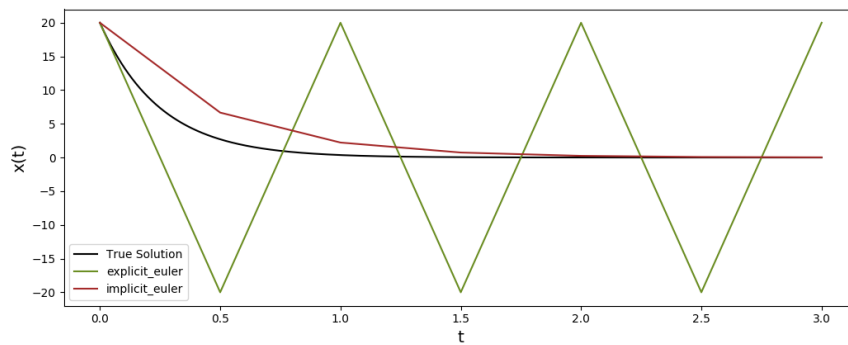


Figure 16: TODO

2.3 Python Implementation 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

3 Test equation for ODEs

3.1 Analytical Solution

Let's now consider the scalar test equation

$$x'(t) = \lambda x(t), \quad x(0) = x_0 \quad (30)$$

In the general case, it is immediately verifiable that

$$x(t) = x_0 e^{\lambda t} \quad (31)$$

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:

$$l_k = x_k - x_{k-1}(t_k) \quad (32)$$

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:

$$e_k = x_k - x(t_k) \quad (33)$$

Figure 17 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.

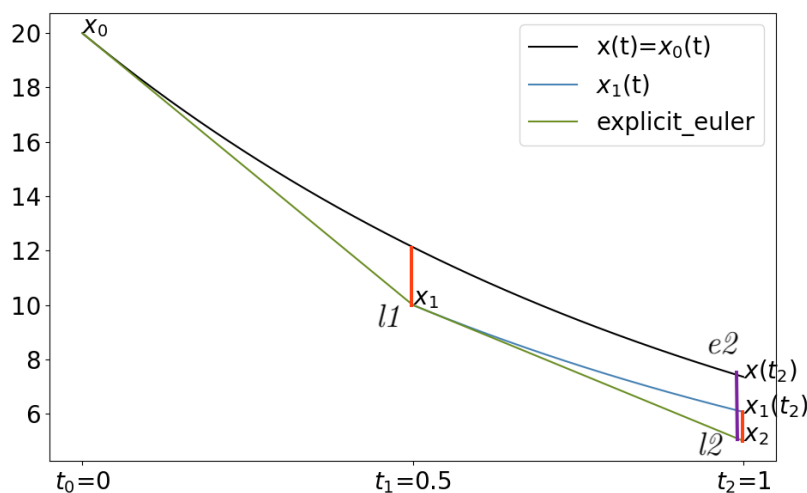


Figure 17: Illustration of the truncation errors for the explicit Euler method

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.

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 express the two terms used x_k and $x_{k-1}(t_k)$:

$$\begin{cases} x_k = (1 + \lambda h)x_{k-1} \\ x_{k-1}(t_k) = x_{k-1}e^{\lambda(t_k - t_{k-1})} = x_{k-1}e^{\lambda h} \simeq x_{k-1}\left(1 + \lambda h + \frac{(\lambda h)^2}{2} + \frac{(\lambda h)^3}{6} + \dots\right) \end{cases} \quad (34)$$

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

$$l_k = x_k - x_{k-1}(t_k) \simeq -x_{k-1}\left(\frac{(\lambda h)^2}{2} + \frac{(\lambda h)^3}{6} + \dots\right) = \mathcal{O}(h^2) \quad (35)$$

Global Truncation Error

true in general case:

Taylor series several variables

$$\begin{cases} x(t) \simeq x(T) + x'(T)(T - t) + x''(T)\frac{(T-t)^2}{2} \\ f(T, x) \simeq f(T, X) + \frac{\partial f}{\partial x}(T, X)(x - X) \end{cases} \quad (36)$$

Apply above with $t = t_k$ and $T = t_{k-1}$, $x = x(t_{k-1})$ and $X = x_{k-1}$

$$\begin{aligned} x(t_k) &= x(t_{k-1}) + x'(t_{k-1})(t_k - t_{k-1}) + x''(t_{k-1})\frac{(t_k - t_{k-1})^2}{2} \\ &= x(t_{k-1}) + f(t_{k-1}, x(t_{k-1}))h + x''(t_{k-1})\frac{h^2}{2} \\ &= x(t_{k-1}) + (f(t_{k-1}, x_{k-1}) + \frac{\partial f}{\partial x}(t_{k-1}, x_{k-1})(x(t_{k-1}) - x_{k-1}))h + x''(t_{k-1})\frac{h^2}{2} \\ &= x(t_{k-1}) + (f(t_{k-1}, x_{k-1}) - \frac{\partial f}{\partial x}(t_{k-1}, x_{k-1})e_{k-1})h + x''(t_{k-1})\frac{h^2}{2} \end{aligned} \quad (37)$$

Thus,

$$\begin{aligned}
e_k &= x_k - x(t_k) \\
&= x_{k-1} - x(t_{k-1}) + \frac{\partial f}{\partial x}(t_{k-1}, x_{k-1})e_{k-1}h - x''(t_{k-1})\frac{h^2}{2} \\
&= e_{k-1}\left(1 + \frac{\partial f}{\partial x}(t_{k-1}, x_{k-1})h\right) - x''(t_{k-1})\frac{h^2}{2}
\end{aligned} \tag{38}$$

Can see the accumulation of errors here!

$$x''(t) = \frac{dx'(t)}{dt} = \frac{df}{dt}(t, x(t)) \tag{39}$$

apply the chain rule:

$$\frac{df}{dx}(g_1(x), g_2(x)) = \frac{dg_1(x)}{dx} \frac{\partial f}{\partial g_1}(g_1(x), g_2(x)) + \frac{dg_2(x)}{dx} \frac{\partial f}{\partial g_2}(g_1(x), g_2(x)) \tag{40}$$

to obtain:

$$\frac{df}{dt}(t, x(t)) = \frac{\partial f}{\partial t}(t, x(t)) + \frac{dx(t)}{dt} \frac{\partial f}{\partial x}(t, x(t)) = \frac{\partial f}{\partial t}(t, x(t)) + f(t, x(t)) \frac{\partial f}{\partial x}(t, x(t)) \tag{41}$$

For continuity reasons

$$\begin{cases} K = \max |\frac{\partial f}{\partial x}(t, x(t))| \\ M = \max |\frac{\partial f}{\partial t}(t, x(t)) + f(t, x(t)) \frac{\partial f}{\partial x}(t, x(t))| \end{cases} \tag{42}$$

$$|e_k| \leq (1 + Kh)|e_{k-1}| + M\frac{h^2}{2} \tag{43}$$

$$|e_k| \leq \frac{M}{2K}(e^{Kt_k} - 1)h \tag{44}$$

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 express the two terms used x_k and $x_{k-1}(t_k)$:

$$\begin{cases} x_k = x_{k-1} + \lambda h x_k \Rightarrow x_k = \frac{1}{1-\lambda h} x_{k-1} \\ x_{k-1}(t_k) \simeq x_{k-1} \left(1 + \lambda h + \frac{(\lambda h)^2}{2} + \frac{(\lambda h)^3}{6} + \dots\right) \end{cases} \tag{45}$$

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

$$l_k = x_k - x_{k-1}(t_k) \simeq x_{k-1} \left(\frac{(\lambda h)^2}{2} + \frac{5(\lambda h)^3}{6} + \dots \right) = \mathcal{O}(h^2) \tag{46}$$

Global Truncation Error

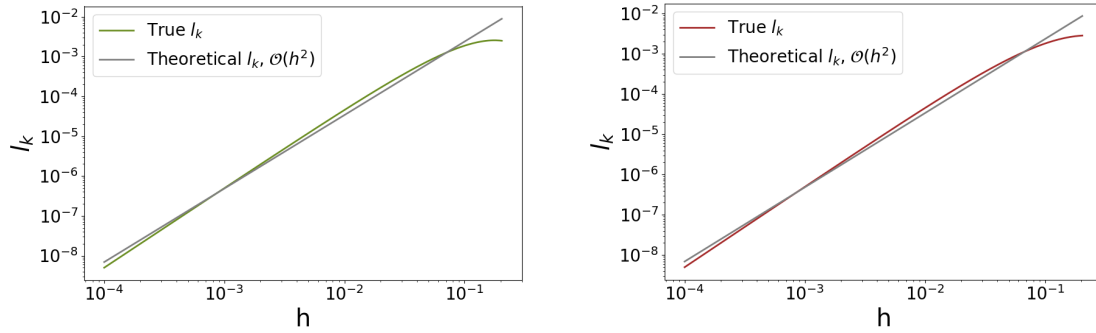
global error TODO

3.4 Plot of Local Error

Figure 18 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 35 and 46 therefore prove that both Euler methods are of order 1.



(a) Local truncation error for the explicit Euler method

(b) Local truncation error for the implicit Euler method

Figure 18: Local truncation errors for Euler methods

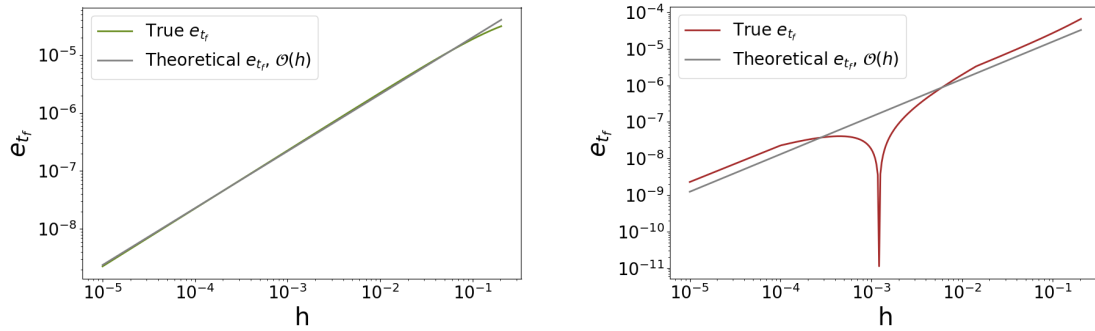
3.5 Plot of Global Error

Figure 19 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 $\mathcal{O}(h)$.

3.6 On Stability

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

²taken at $k = 10$



(a) Global truncation error for the explicit Euler method

(b) Global truncation error for the implicit Euler method

Figure 19: Global truncation errors for Euler methods

$$x'(t) = \lambda x(t), \quad x(0) = 1, \quad \lambda \in \mathbb{C}, \quad \operatorname{Re}(\lambda) < 0 \quad (47)$$

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

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

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

$$\mathcal{D} = \{(h, \lambda) \in \mathbb{R} \times \mathbb{C} \mid \lim_{k \rightarrow \infty} x_k = 0\} \quad (48)$$

Explicit Euler Method

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

$$\begin{aligned} x_{k+1} &= x_k + hf(t_k, x_k) \\ &= x_k(1 + \lambda h) \end{aligned} \quad (49)$$

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

$$x_k = x_0(1 + \lambda h)^k = (1 + \lambda h)^k \quad (50)$$

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

$$\mathcal{D}_{ee} = \{(h, \lambda) \in \mathbb{R} \times \mathbb{C} \mid |1 + \lambda h| < 1\} \quad (51)$$

Implicit Euler Method

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

$$\begin{aligned} x_{k+1} &= x_k + hf(t_k, x_{k+1}) \\ &= x_k + \lambda h x_{k+1} \\ &= x_k \left(\frac{1}{1 - \lambda h} \right) \end{aligned} \quad (52)$$

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

$$x_k = x_0 \left(\frac{1}{1-\lambda h} \right)^k = \left(\frac{1}{1-\lambda h} \right)^k \quad (53)$$

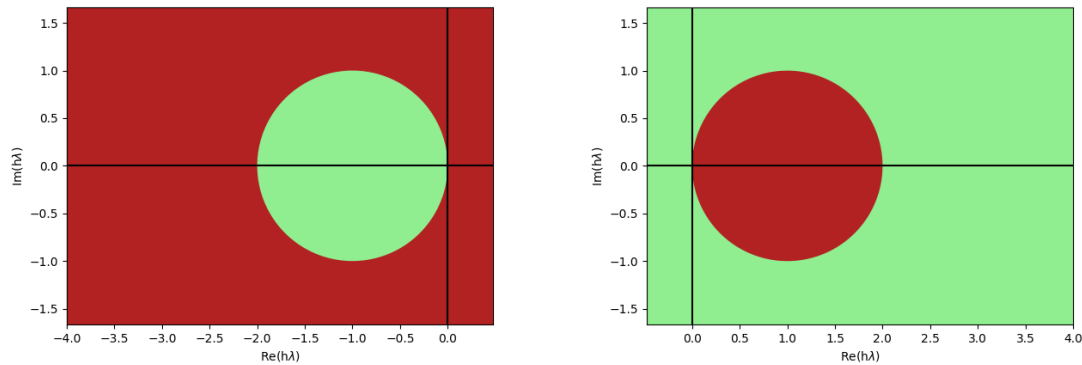
which implies that the region of stability for the implicit Euler method is:

$$\mathcal{D}_{ie} = \{(h, \lambda) \in \mathbb{R} \times \mathbb{C} \mid |1 - \lambda h| > 1\} \quad (54)$$

Figure 20 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** if its region of absolute stability \mathcal{D} contains the entire left complex half plane, i.e. $\mathcal{D} \in \{(h, \lambda) \in \mathbb{R} \times \mathbb{C} \mid \operatorname{Re}(\lambda h) < 0\}$

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 16, 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 20: 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:

$$d\mathbf{x}(t) = f(t, \mathbf{x}(t))dt + g(\mathbf{x}(t))d\mathbf{w}(t) \quad (55)$$

Where g is a diffusion function and \mathbf{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, \dots, t_{N-1} of constant length $h = t_{k+1} - t_k$ the following holds:

$$\begin{aligned} \int_a^b f(t)dt &= \lim_{N \rightarrow \infty} \sum_{i=0}^{N-1} h f(t_i) \\ \int_a^b f(t)d\mathbf{w}(t) &= \lim_{N \rightarrow \infty} \sum_{i=0}^{N-1} f(t_i)\Delta\mathbf{w}_i, \quad \Delta\mathbf{w}_i \sim \mathcal{N}(0, h) \quad (\text{Ito's Integral}) \end{aligned} \quad (56)$$

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:

$$\begin{aligned} \int_{t_{k+1}}^{t_k} f(t)dt &\simeq h f(t_c), \quad t_c \in [t_k, t_{k+1}] \\ \int_{t_{k+1}}^{t_k} f(t)d\mathbf{w}(t) &\simeq f(t_k)\Delta\mathbf{w}_k, \quad \Delta\mathbf{w}_k \sim \mathcal{N}(0, h) \end{aligned} \quad (57)$$

Therefore providing a direct way to iteratively approximate solution of such SDEs, as the integration of equation 55 between t_k and t_{k+1} yields:

$$\mathbf{x}_{k+1} - \mathbf{x}_k = h f(t_c, \mathbf{x}(t_c)) + g(\mathbf{x}_k)\Delta\mathbf{w}_k, \quad t_c \in [t_k, t_{k+1}], \quad \Delta\mathbf{w}_k \sim \mathcal{N}(0, h) \quad (58)$$

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 3

```
1 def sde_solver(f, J, g, dW, t0, tf, N, x0, **kwargs):
2     ...
```

Listing 3: Interface for SDE solver

4.1 Multivariate Standard Wiener Process

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

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

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

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

It is therefore pretty straightforward to implement such process. A random generator can directly provide $N \times k$ gaussian samples³, and a cumulative sum can then directly generate $\mathbf{W}_0, \mathbf{W}_1, \dots, \mathbf{W}_N$. Such implementation can be found in Listing 4

```

1 def wiener_process(T, N, dims=2):
2     W = np.zeros((N, dims))
3     dt = T / N
4     dW = np.random.normal(loc=0, scale=dt, size=(N-1, dims))
5     W[1:, :] = np.cumsum(dW, axis=0)
6
7     # adapt size of dW array
8     dW = np.vstack((np.zeros((1, dims)), dW))
9
10    return dW, W

```

Listing 4: Implementation of a multivariate Wiener process

The previous implementation allows the generation of Figure 21.

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 58 is applied for $t_c = t_k$. For $\Delta w_k \sim \mathcal{N}(0, h)$, the iteration equation can therefore be defined as:

$$\mathbf{x}_{k+1} - \mathbf{x}_k = hf(t_k, \mathbf{x}_k) + g(\mathbf{x}_k)\Delta w_k \quad (59)$$

The implementation of such method can be directly drawn from the implementation of the explicit Euler method. Only the stochastic terms $g(\mathbf{x}_k)\Delta w_k$ to each iteration, as can be seen in Listing 5

```

1 def sde_solver(f, J, g, dW, t0, tf, N, x0, **kwargs):
2     dt = (tf - t0) / N
3
4     T = [t0]
5     X = [x0]

```

³ k is here the dimension of the process

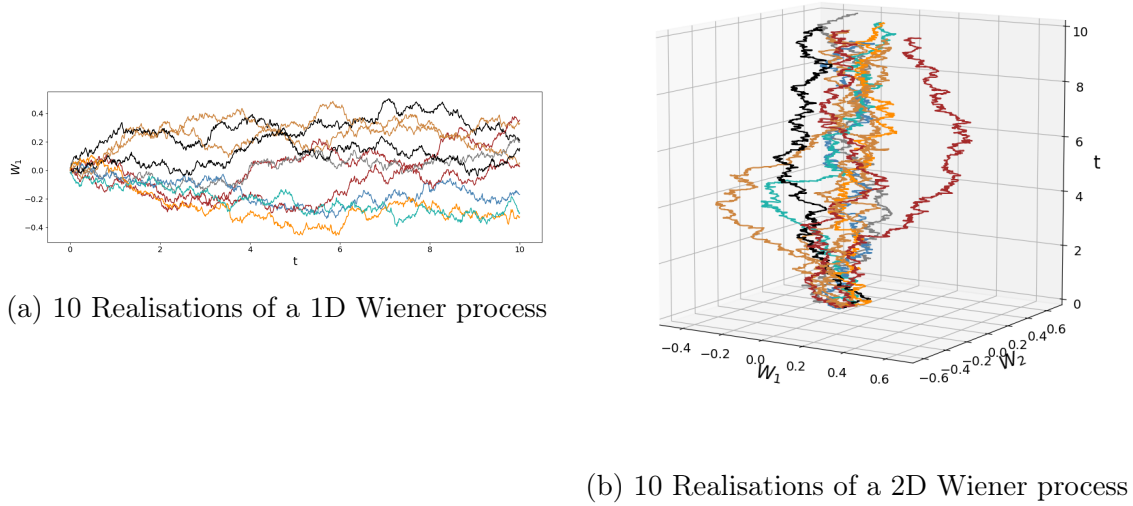


Figure 21: Realisations of Wiener processes

```

6
7   for k in range(N):
8       f_eval = f(T[-1], X[-1], **kwargs)
9       g_eval = g(T[-1], X[-1], **kwargs)
10      X.append(X[-1] + dt * f_eval + dW[k] * g_eval)
11      T.append(T[-1] + dt)
12
13  T = np.array(T)
14  X = np.array(X)
15
16  return X, T

```

Listing 5: Implementation of the explicit-explicit SDE solver

4.3 Implicit-Explicit Method

On the other hand, when equation 58 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:

$$\mathbf{x}_{k+1} - \mathbf{x}_k = hf(t_{k+1}, \mathbf{x}_{k+1}) + g(\mathbf{x}_k)\Delta w_k \quad (60)$$

Similarly as for the implicit Euler equation, the dependency between \mathbf{x}_{k+1} and $f(\mathbf{x}_{k+1})$ 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

$$R_k(\mathbf{x}_{k+1}) = \mathbf{x}_{k+1} - \mathbf{x}_k - hf(t_{k+1}, \mathbf{x}_{k+1}) - g(\mathbf{x}_k)\Delta w_k = 0 \quad (61)$$

For this reason, the newton's method implementation was slightly modified to accept g and Δw_k as arguments. Listing 6 details such implementation.

```

1 def sde_newtons_method(f, J, psi, t, x, dt, x_init, tol, max_iters, **
  kwargs):
2     k = 0
3     x_iter = x_init
4     t_iter = t + dt
5     f_eval = f(t_iter, x_iter, **kwargs)
6     J_eval = J(t_iter, x_iter, **kwargs)
7
8     R = x_iter - dt*f_eval - psi
9     I = np.eye(x.shape[0])
10    while ((k < max_iters) & (norm(R, np.inf) > tol)):
11        k += 1
12        M = I - dt*J_eval
13        dx_iter = np.linalg.solve(M,R)
14        x_iter -= dx_iter
15        f_eval = f(t_iter, x_iter, **kwargs)
16        J_eval = J(t_iter, x_iter, **kwargs)
17        R = x_iter - dt*f_eval - psi
18
19    return x_iter

```

Listing 6: 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 7

```

1 def sde_solver(f, J, g, dW, t0, tf, N, x0, **kwargs):
2     dt = (tf - t0) / N
3
4     T = [t0]
5     X = [x0]
6
7     kwargs, newtons_tol, newtons_max_iters = parse_newtons_params(kwargs)
8
9     for k in range(N):
10        # Use explicit form to start off newton
11        f_eval = f(T[-1], X[-1], **kwargs)
12        g_eval = g(T[-1], X[-1], **kwargs)
13        psi = X[-1] + g_eval*dW[k]
14        x_init = dt * f_eval + psi
15        X.append(sde_newtons_method(f, J, psi, T[-1], X[-1], dt, x_init,
newtons_tol, newtons_max_iters, **kwargs))
16        T.append(T[-1] + dt)
17
18    T = np.array(T)
19    X = np.array(X)
20
21    return X, T

```

Listing 7: 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

$$dx(t) = \lambda x(t)dt + \sigma x(t)d\omega(t) = x(t)(d\omega(t) + \lambda dt) \quad (62)$$

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

$$df = \frac{\partial f(t, x)}{\partial t}dt + \frac{\partial f(t, x)}{\partial x}dx + \frac{1}{2} \frac{\partial^2 f(t, x)}{\partial x^2}dx^2 \quad (63)$$

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:

$$df = \partial_1 f(z_1, z_2)dz_1 + \partial_2 f(z_1, z_2)dz_2 + \frac{1}{2} \partial_{2,2} f(z_1, z_2)dz_2^2 \quad (64)$$

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

$$df(t, x(t)) = \partial_1 f(t, x(t))dt + \partial_2 f(t, x(t))dx(t) + \frac{1}{2} \partial_{2,2} f(t, x(t))dx(t)dx(t) \quad (65)$$

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

$$df(t, x(t)) = \partial_2 f(t, x(t))dx(t) + \frac{1}{2} \partial_{2,2} f(t, x(t))dx(t)dx(t) \quad (66)$$

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 62 can be inserted into 66 to yield:

$$\begin{aligned} d\log(x(t)) &= \frac{dx(t)}{x(t)} - \frac{1}{2} \frac{dx(t)dx(t)}{x(t)^2} \\ &= (\sigma d\omega(t) + \lambda dt) - \frac{1}{2} (\sigma d\omega(t) + \lambda dt)^2 \end{aligned} \quad (67)$$

$(\sigma d\omega(t) + \lambda dt)^2$ can be developed into $(\sigma^2 d\omega(t)^2 + 2\sigma\lambda d\omega(t)dt + \lambda^2 dt^2)$. In the limit $dt \rightarrow 0$, 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:

$$d\log(x(t)) = \sigma d\omega(t) + (\lambda - \frac{\sigma^2}{2})dt \quad (68)$$

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

$$\int_{x_0}^{x(t)} d\log(x(t)) = \int_{\omega_0}^{\omega(t)} \sigma d\omega(t) + \int_0^t \left(\lambda - \frac{\sigma^2}{2}\right) dt \quad (69)$$

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

$$\log(x(t)) = \log(x_0) + \sigma\omega(t) + \left(\lambda - \frac{\sigma^2}{2}\right)t \quad (70)$$

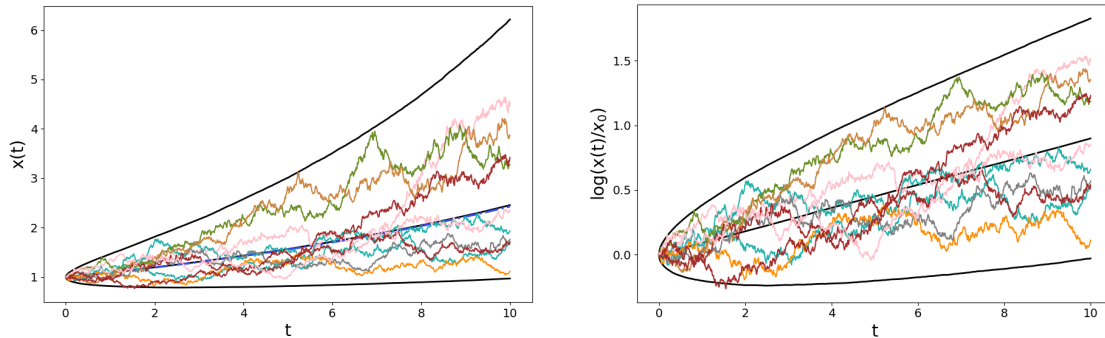
Finally providing an analytical solution to the geometric Brownian motion equation.

$$x(t) = x_0 e^{(\lambda - \frac{\sigma^2}{2})t + \sigma\omega(t)} \quad (71)$$

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 22 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.



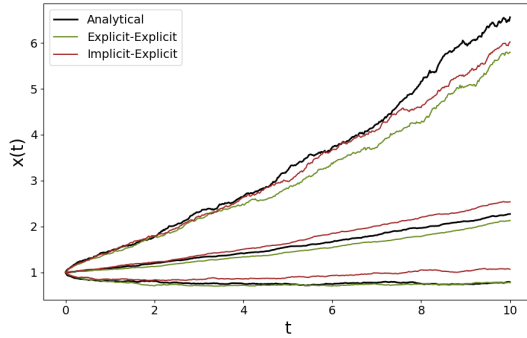
(a) 10 Realisations of the analytical solution of the test equation for SDEs. The blue central dotted line is the solution of the equation without any stochastic term

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

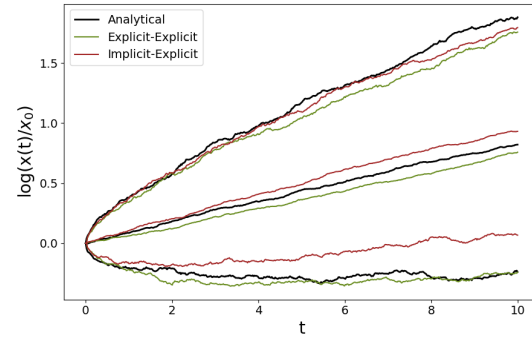
Figure 22: 10 Realisations of the analytical solution of the test equation for SDEs. The black lines correspond to the mean averaged over 10000 realisations, and the 95% confidence interval around it

Figure 23, 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

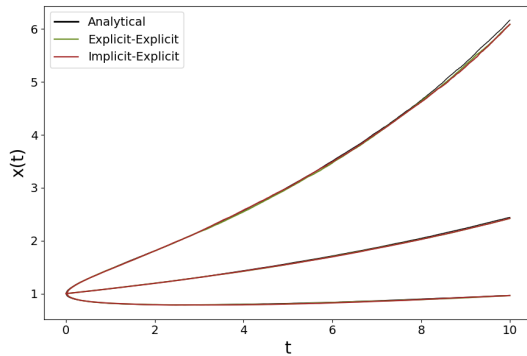
uses 10 000 realisations. For the bottom row, the estimates are so close from each other that the resulting trajectories are almost indistinguishable from each other, thus indicating good performance from the numerical methods.



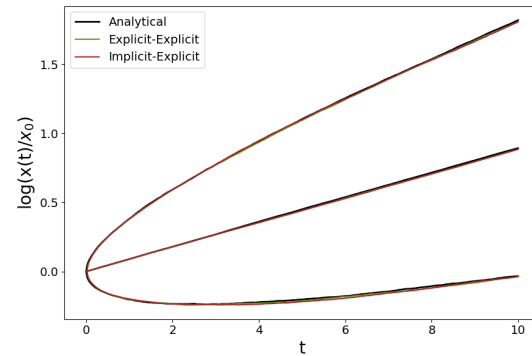
(a) Comparison of the analytical solution against the numerical methods for 100 realisations



(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 23

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 24 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.

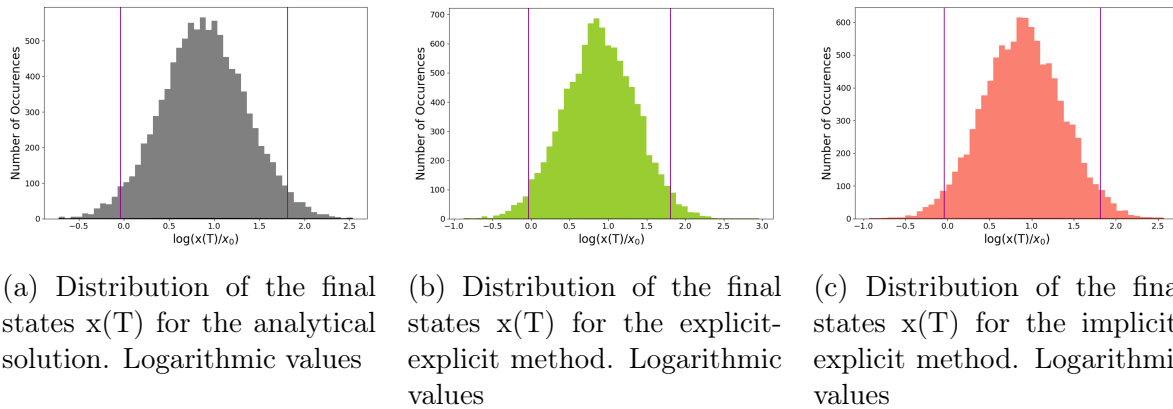


Figure 24: Comparison of the distribution of the final state $x(T)$ between the analytical solution and numerical methods. The logarithmic values are used. The magenta vertical lines are the boundaries determined by the mean ± 1.96 standard deviation, which correspond to the 95% confidence interval

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 23.

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 25 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 $\{e_N\}$

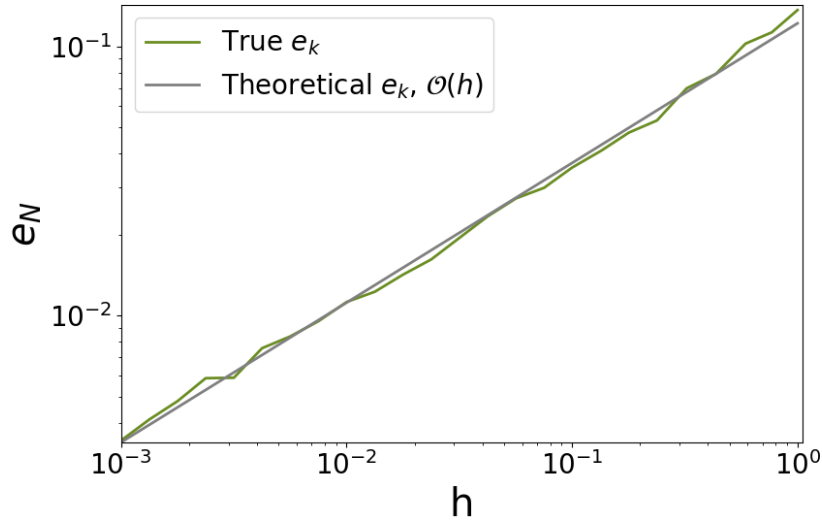


Figure 25: Error of the explicit-explicit method as a function of the step size h . Logarithmic scale

5.6 Order of the Implicit-Explicit Method

Figure 25 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.

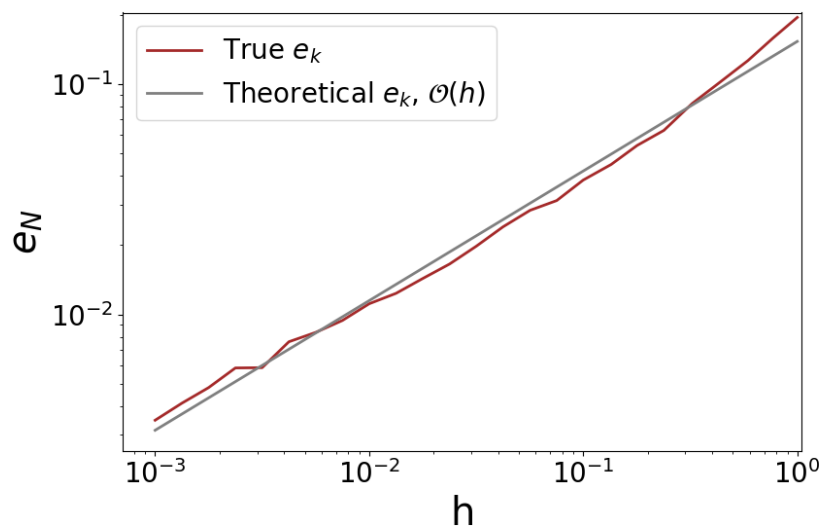


Figure 26: Error of the implicit-explicit method as a function of the step size h . Logarithmic scale

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:

$$\mathbf{x}'(t) = f(t, \mathbf{x}(t)), \quad \mathbf{x}(t_0) = \mathbf{x}_0 \quad (72)$$

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

$$\mathbf{x}(t_{k+1}) - \mathbf{x}(t_k) = \int_{t_k}^{t_{k+1}} f(t, \mathbf{x}(t)) dt \quad (73)$$

Approximating $f(t, \mathbf{x}(t))$ by $f(t_k, \mathbf{x}_k)$ results in the explicit Euler method and with $f(t_{k+1}, \mathbf{x}_{k+1})$ 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.

$$\begin{aligned} \hat{\mathbf{x}}_{k+\frac{1}{2}} &= \mathbf{x}_k + \frac{h}{2} f(t_k, \mathbf{x}_k) \\ \mathbf{x}_{k+1} &= \mathbf{x}_k + h f(t_k + \frac{h}{2}, \hat{\mathbf{x}}_{k+\frac{1}{2}}) \end{aligned} \quad (74)$$

Figure 27 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,

$$\begin{aligned} \hat{x}_{k+\frac{1}{2}} &= x_k + \frac{h}{2} \lambda x_k = x_k(1 + \frac{h}{2} \lambda) \\ x_{k+1} &= x_k + h \lambda \hat{x}_{k+\frac{1}{2}} = x_k(1 + h \lambda + \frac{(h \lambda)^2}{2}) \end{aligned} \quad (75)$$

And therefore,

$$l_{k+1} = x_{k+1} - x_k(t_{k+1}) = x_k(1 + h \lambda + \frac{(h \lambda)^2}{2} - e^{h \lambda}) = \mathcal{O}(h^3) \quad (76)$$

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

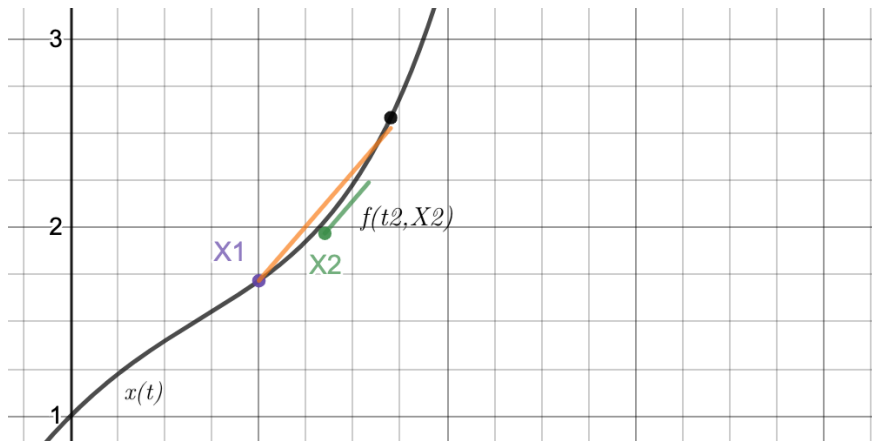


Figure 27: Illustration of basic explicit Runge Kutta method.

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:

$$\begin{aligned} \mathbf{x}_{k+1} &= \mathbf{x}_k + h \sum_{i=1}^s b_i f(t_k + c_i h, \mathbf{X}_i) \\ \mathbf{X}_i &= \mathbf{x}_k + h \sum_{j=1}^s a_{ij} f(t_k + c_j h, \mathbf{X}_j) \end{aligned} \quad (77)$$

Where the different \mathbf{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

$$\begin{cases} T_1 = t_k & X_1 = x_k \\ T_2 = t_k + \frac{1}{2}h & X_2 = x_k + h\frac{1}{2}f(T_1, X_1) \\ T_3 = t_k + \frac{1}{2}h & X_3 = x_k + h\frac{1}{2}f(T_2, X_2) \\ T_4 = t_k + h & X_4 = x_k + hf(T_3, X_3) \end{cases} \quad (78)$$

Resulting in the following update rule for x_k

$$\begin{cases} t_{k+1} = t_k + h \\ x_{k+1} = x_k + h(\frac{1}{6}f(T_1, X_1) + \frac{1}{3}f(T_2, X_2) + \frac{1}{3}f(T_3, X_3) + \frac{1}{6}f(T_4, X_4)) \end{cases} \quad (79)$$

It is interesting to visualise the different intermediary steps for the method. Figure 28 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 28 thus provides an intuitive representation how accurate the RK4 method is.

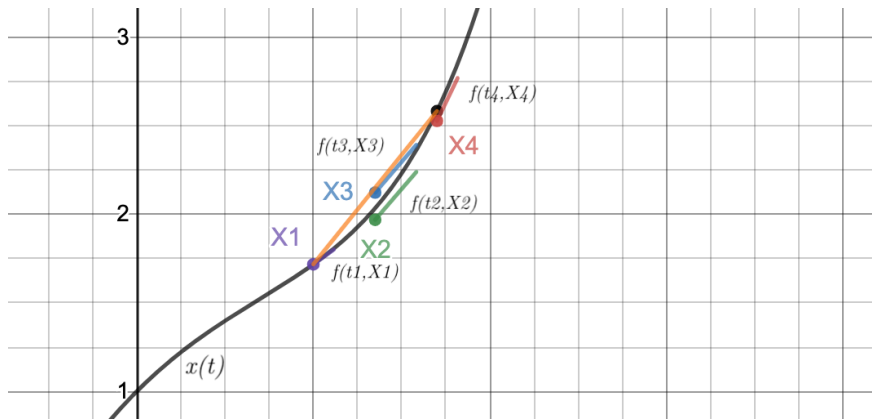


Figure 28: Illustration of the stages of the classical Runge Kutta method. In purple, green, blue and red are the intermediary stages to produce the orange estimation

6.2 Implementation

The general form of a Runge-Kutta step, as presented in Equation 77 can be simplified using matrix notations. Noting $\mathbf{K} = \begin{bmatrix} f(t_k + c_1h, \mathbf{X}_1) \\ \vdots \\ f(t_k + c_sh, \mathbf{X}_s) \end{bmatrix}$, and $\mathbf{B} = \begin{bmatrix} b_1 \\ \vdots \\ b_s \end{bmatrix}$ the upper equation becomes

$$\mathbf{x}_{k+1} = \mathbf{x}_k + h\mathbf{B}^T \mathbf{K} \quad (80)$$

Furthermore, assuming that the matrix \mathbf{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 \mathbf{K} can be computed using the previous rows using,

$$\mathbf{X}_i = \mathbf{x}_k + h\mathbf{A}_{i,:i} \begin{bmatrix} f(t_1, \mathbf{X}_1) \\ \vdots \\ f(t_{i-1}, \mathbf{X}_{i-1}) \end{bmatrix} = \mathbf{x}_k + h\mathbf{A}_{i,:i}\mathbf{K}_{:i} \quad (81)$$

$$\mathbf{K}_i = f(t_k + c_i h, \mathbf{X}_i) = f(t_k + c_i h, \mathbf{x}_k + h\mathbf{A}_{i,:i}\mathbf{K}_{:i})$$

Where \mathbf{K}_i is the i th row of \mathbf{K} , $\mathbf{K}_{:i}$ the $(i-1)$ th first rows of \mathbf{K} and $\mathbf{A}_{i,:i}$ the i th row of \mathbf{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 8 details its *Python* implementation.

```

1 def rk_step(f, t, x, dt, butcher_tableau, **kwargs):
2     A = butcher_tableau['A']
3     B = butcher_tableau['B']
4     C = butcher_tableau['C']
5
6     P = len(C)
7     K = np.zeros((P, x.shape[0]))
8
9     K[0, :] = f(t, x, **kwargs)
10
11     for p, (a, c) in enumerate(zip(A[1:], C[1:]), start=1):
12         t_p = t + dt * c
13         x_p = x + dt * (K.T @ a)
14         K[p, :] = f(t_p, x_p, **kwargs)
15
16     return x + dt * (K.T @ B)

```

Listing 8: 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 9 shows how the `rk4_step` function provides a wrapper for the generic `rk_step` function.

```

1 # Butcher Tableau
2 C = np.array([0, 1 / 2, 1 / 2, 1])
3 A = np.array([
4     [0, 0, 0, 0],
5     [1 / 2, 0, 0, 0],
6     [0, 1 / 2, 0, 0],
7     [0, 0, 1, 0],
8 ])
9 B = np.array([1 / 6, 1 / 3, 1 / 3, 1 / 6])
10
11
12 def rk4_step(f, t, x, dt, **kwargs):
13     butcher_tableau = {
14         'A': A,
15         'B': B,

```

```

16         'C': C,
17     }
18
19     return rk_step(f, t, x, dt, butcher_tableau, **kwargs)[0] # no error
    estimation.

```

Listing 9: 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 10 displays them.

```

1 def ode_solver(f, J, t0, tf, N, x0, adaptive_step_size=False, **kwargs):
2     dt = (tf - t0) / N
3
4     T = [t0]
5     X = [x0]
6     for k in range(N):
7         X.append(rk4_step(f, T[-1], X[-1], dt, **kwargs))
8         T.append(T[-1] + dt)
9
10    T = np.array(T)
11    X = np.array(X)
12
13    return X, T

```

Listing 10: Interface for the RK4 solver

6.3 Order and Stability

TODO prove order 4 for RK4 ?

Figure 29 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 correspond to the theoretical order of accuracy, 4.

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

$$x'(t) = \lambda x(t), \quad x(0) = 1, \quad \lambda \in \mathbb{C}, \quad \operatorname{Re}(\lambda) < 0 \quad (82)$$

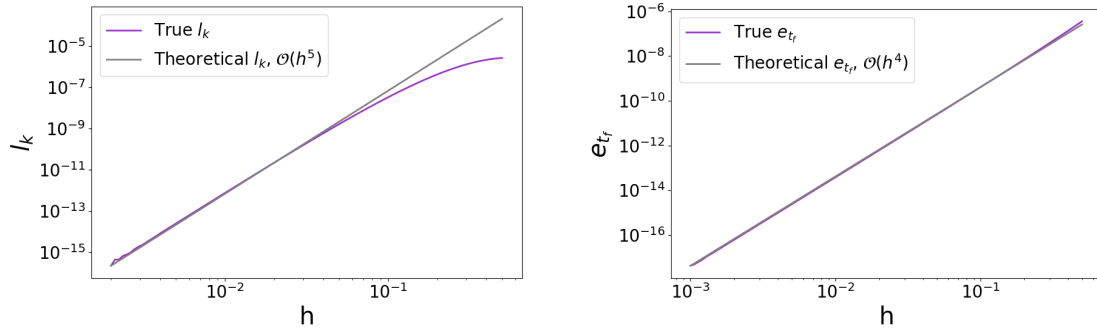
Using $f(t, x(t)) = \lambda x(t)$ in Equation 77, yields that for any $i \in [1, s]$,

$$X_i = x_k + h \sum_{j=1}^s a_{ij} \lambda X_j \quad (83)$$

That expression can be rearranged into matrix form

$$\mathbf{X} = x_k \mathbf{1} + h \lambda \mathbf{A} \mathbf{X} \quad (84)$$

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



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

Figure 29: Truncation errors for the RK4 method

$$\mathbf{X} = x_k(\mathbf{I} - h\lambda\mathbf{A})^{-1}\mathbf{1} \quad (85)$$

The first equation of 77 can also be expressed in a matrix form,

$$x_{k+1} = x_k + h\lambda\mathbf{B}^T\mathbf{X} \quad (86)$$

Which, becomes, when the RHS of 85 is inserted into \mathbf{X} ,

$$x_{k+1} = x_k(1 + h\lambda\mathbf{B}^T(\mathbf{I} - h\lambda\mathbf{A})^{-1}\mathbf{1}) = x_k R(h\lambda) \quad (87)$$

Where R is defined as the following polynomial on \mathbb{C} , $R(z) = 1 + z\mathbf{B}^T(\mathbf{I} - z\mathbf{A})^{-1}\mathbf{1}$. 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,

$$x_k = R(h\lambda)^k x_0 \quad (88)$$

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

$$\mathcal{D}_{RK} = \{(h, \lambda) \in \mathbb{R} \times \mathbb{C} \mid |R(h\lambda)| < 1\} \quad (89)$$

Specifically, for the RK4 method, Figure 30 displays the stability region of its stability polynomial.

$$R(h\lambda) = 1 + h\lambda + \frac{(h\lambda)^2}{2} + \frac{(h\lambda)^3}{6} + \frac{(h\lambda)^4}{24} \quad (90)$$

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 .

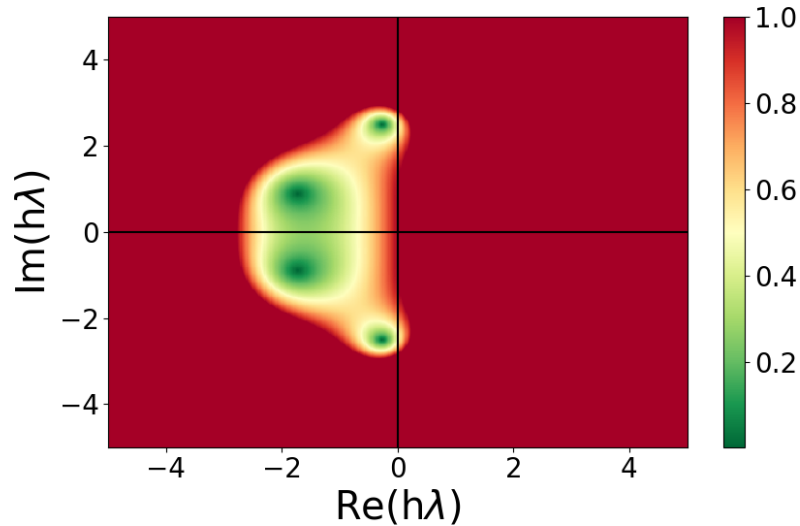


Figure 30: Region of absolute stability (green) for the RK4 method

6.4 Test on the Van der Pol Problem

6.4.1 For $\mu = 2$

6.4.2 For $\mu = 12$

6.5 Test on the A-CSTR Problem

6.5.1 CSTR 1D

6.5.2 CSTR 3D

7 Classical Runge-Kutta method with adaptive time step

Stability requirement is same as in 6.

7.4 we are welcome to use $\mu=2$

7.1 Method Description

7.2 Implementation

7.3 Order and Stability

7.4 Test on the Van der Pol Problem

7.4.1 For $\mu = 2$

7.4.2 For $\mu = 12$

7.5 Test on the A-CSTR Problem

7.5.1 CSTR 1D

7.5.2 CSTR 3D

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 e_k vs true local error l_k .

8.1 Method Description

Butcher's Tableau

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	$\frac{35}{384}$	0	$\frac{500}{1113}$	$\frac{125}{192}$	$\frac{-2187}{6784}$	$\frac{11}{84}$	
\hat{x}	$\frac{5179}{57600}$	0	$\frac{7571}{16695}$	$\frac{393}{640}$	$\frac{-92097}{339200}$	$\frac{187}{2100}$	$\frac{1}{40}$
e	$\frac{71}{57600}$	0	$\frac{-71}{16695}$	$\frac{71}{1920}$	$\frac{-17253}{339200}$	$\frac{22}{525}$	$\frac{-1}{40}$

8.2 Implementation

8.3 Order and Stability

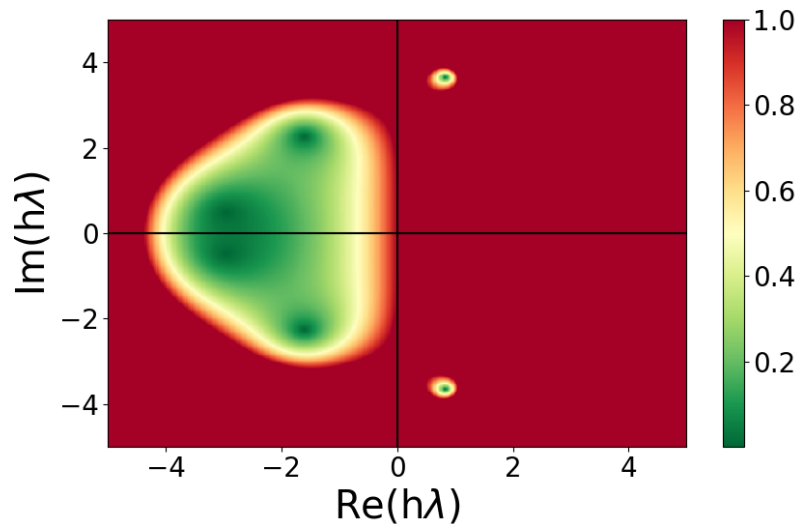


Figure 31: Region of absolute stability (green) for the DOPRI54 method

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

consistency equation: method consistent iif

$$\sum b_i = 1 \quad (91)$$

which is pretty easy to show but is redundant with the order condition at level 1.

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 ([1] page 88) that order 3 is attainable with only three stages. Let's therefore write down the desired Butcher tableau as:

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

$$\Phi(t) = \frac{1}{\gamma(t)} \tag{92}$$





t				
r(t)	1	2	3	3
$\gamma(t)$	1	2	3	6
$\Phi(t)$	$\sum b_i$	$\sum b_i c_i$	$\sum b_i c_i^2$	$\sum b_i a_{ij} c_j$

Table 3: Rooted tree functions

and additional equation, required to achieve at least order one, as explained in [8] page 39,

$$\sum a_{ij} = c_i \tag{93}$$

$$\begin{cases}
 b_1 + b_2 + b_3 = 1 \\
 b_2 c_2 + b_3 c_3 = \frac{1}{2} \\
 b_2 c_2^2 + b_3 c_3^2 = \frac{1}{3} \\
 b_3 a_{32} c_2 = \frac{1}{6} \\
 a_{21} = c_2 \\
 a_{31} + a_{32} = c_3
 \end{cases} \tag{94}$$

$$\begin{cases} a_{21} = c_2 \\ a_{31} = \frac{c_3(3c_2^2 - 3c_2 + c_3)}{(3c_2 - 2)c_2} \\ a_{32} = -\frac{c_3^2 - c_2c_3}{(3c_2 - 2)c_2} \\ b_1 = \frac{(6c_3 - 3)c_2 - 3c_3 + 2}{6c_2c_3} \\ b_2 = \frac{-3c_3 + 2}{6(c_2 - c_3)c_2} \\ b_3 = \frac{3c_2 - 2}{6c_3(c_2 - c_3)} \end{cases} \quad (95)$$

$$\begin{cases} c_2 = \frac{1}{3} \\ c_3 = \frac{2}{3} \end{cases} \quad (96)$$

$$\begin{cases} a_{21} = \frac{1}{3} \\ a_{31} = 0 \\ a_{32} = \frac{2}{3} \\ b_1 = \frac{1}{4} \\ b_2 = 0 \\ b_3 = \frac{3}{4} \end{cases} \quad (97)$$

9.2 Derivation of Coefficients of Error Estimator

$$\begin{cases} \hat{b}_1 + \hat{b}_2 = 1 \\ \hat{b}_2 c_2 = \frac{1}{2} \end{cases} \quad (98)$$

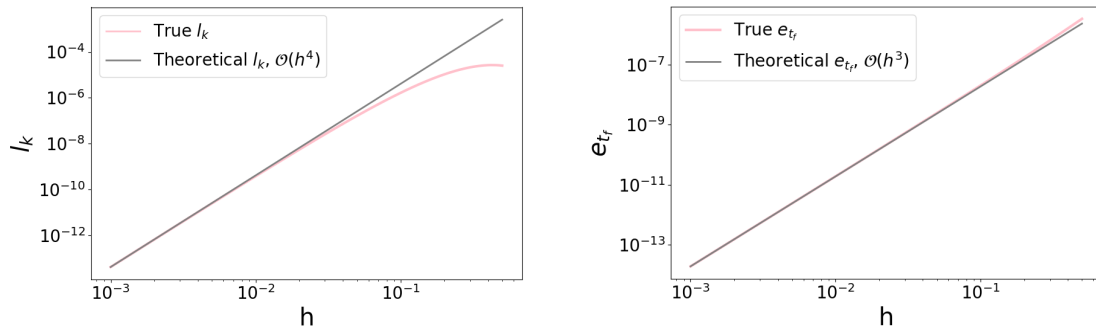
$$\begin{cases} \hat{b}_1 = -\frac{1}{2} \\ \hat{b}_2 = \frac{3}{2} \end{cases} \quad (99)$$

9.3 Butcher Tableau

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

9.4 Method Order

to check order of error embedding use $E(N)$ from the controller, should follow an order 2 decrease (and compare with true error)



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

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

Figure 32: Truncation errors for the custom RK method

9.5 Method Stability

$$R(h\lambda) = 1 + h\lambda + \frac{(h\lambda)^2}{2} + \frac{(h\lambda)^3}{6} \quad (100)$$

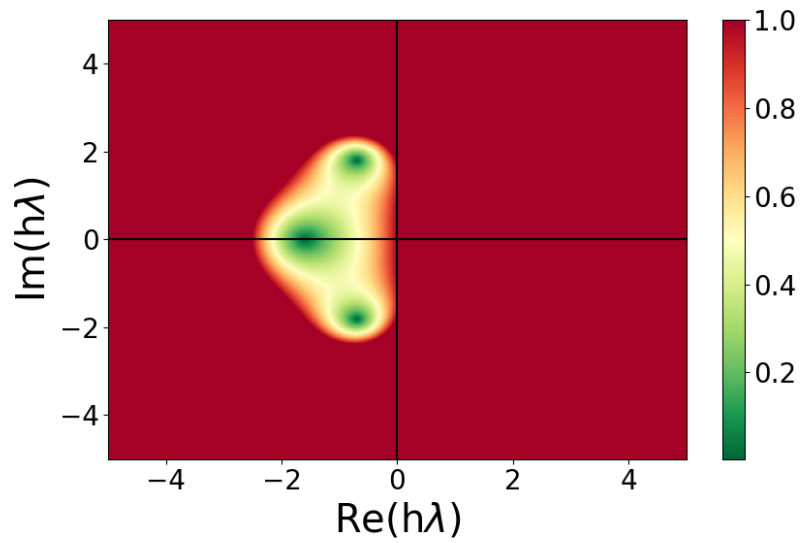


Figure 33: Custom RK method stability region

9.6 3D CSTR Test

10 ESDIRK23

10.1 Method Description

10.2 Method Stability

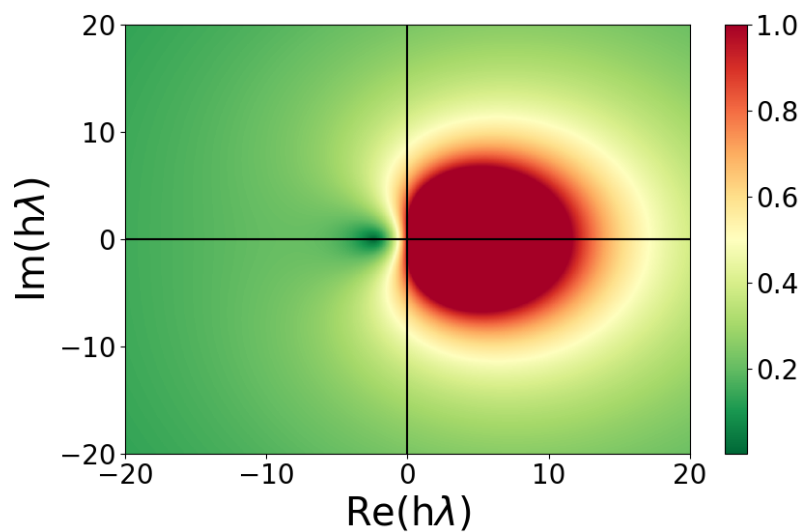


Figure 34: Region of absolute stability (green) for the ESDIRK23 method

10.3 Implementation

10.4 Test on the Van der Pol Problem

10.4.1 For $\mu = 2$

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

List of Figures

1	Van der Pol oscillator simulation with default ode solver for parameter $\mu = 2$	2
2	Van der Pol oscillator simulation with default ode solver for parameter $\mu = 12$	3
3	Evolution of the flow F over the integration period	5
4	Evolution of the state in the 3D model	5
5	Evolution of the temperature in the 1D model	6
6	Explicit Euler approximation for large and small step sizes	9
7	Explicit Euler approximation with adaptive step size	10
8	Simulation results for large step size on the Van der Pol problem. $\mu = 2$ and $h = 0.125$	12
9	Explicit Euler method on the Van der Pol problem. $\mu = 2$ and $h = 0.001$. .	13
10	Simulation results for large step size on the Van der Pol problem. $\mu = 12$ and $h = 0.025$	14
11	Explicit Euler method on the Van der Pol problem. $\mu = 12$ and $h = 0.001$. .	15
12	Simulation results for large step size on the CSTR 1D problem. $h = 0.175$. .	15
13	Explicit Euler method on the CSTR 1D problem. $h = 0.00035$	15
14	Simulation results for large step size on the CSTR 3D problem. $h = 0.175$. .	16
15	Explicit Euler method on the CSTR 3D problem. $h = 0.00035$	16
16	TODO	18
17	Illustration of the truncation errors for the explicit Euler method	19
18	Local truncation errors for Euler methods	22
19	Global truncation errors for Euler methods	23
20	Stability regions for Euler methods	24
21	Realisations of Wiener processes	27
22	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	31
23	32
24	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	33
25	Error of the explicit-explicit method as a function of the step size h . Logarithmic scale	34
26	Error of the implicit-explicit method as a function of the step size h . Logarithmic scale	35
27	Illustration of basic explicit Runge Kutta method.	37
28	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	38
29	Truncation errors for the RK4 method	41
30	Region of absolute stability (green) for the RK4 method	42
31	Region of absolute stability (green) for the DOPRI54 method	44
32	Truncation errors for the custom RK method	47

33	Custom RK method stability region	47
34	Region of absolute stability (green) for the ESDIRK23 method	48

List of Tables

1	Moments of the distribution of $x(T)$ for both the analytical solution and numerical methods	33
2	Moments of the distribution of $x(T)$ for both the analytical solution and numerical methods. Logarithmic values	33
3	Rooted tree functions	45

Listings

1	Implementation of the explicit Euler method	8
2	Implementation of the explicit Euler method with adaptive step size	10
3	Interface for SDE solver	25
4	Implementation of a multivariate Wiener process	26
5	Implementation of the explicit-explicit SDE solver	26
6	Implementation of Newton's method for the implicit-explicit SDE solver	28
7	Implementation of the implicit-explicit SDE solver	28
8	Implementation of the method agnostic rk step	39
9	Implementation of the RK4 step as a wrapper of the general RK step	39
10	Interface for the RK4 solver	40

Nomenclature

A-CSTR Adiabatic continuous stirred reactor

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

11 Appendix A