DEPARTMENT OF APPLIED MATHEMATICS
AND COMPUTER SCIENCE

02686 - Scientific Computing for Differential Equations

Take Home Exam

Author:
Jorge Sintes Fernández

Course responsible:
John Bagterp Jørgensen

June 6, 2021

Contents

List of Figures

List of Tables

Part 1: Test equation for ODEs

Consider the test equation

$$\dot{x}(t) = \lambda x(t), \quad x(0) = x_0, \tag{1}$$

for $\lambda = -1$ and $x_0 = 1$.

1.1. Provide the analytical solution to the test equation.

We can solve the equation analytically as a separable equation:

$$\frac{dx(t)}{dt} = \lambda x(t), \quad \frac{dx(t)}{x(t)} = \lambda dt,$$
$$\int_{x_0}^{x(t)} \frac{dx(t)}{x(t)} = \int_{t_0}^{t} \lambda dt, \quad \Big[\log(x(t))\Big]_{x_0}^{x(t)} = \lambda \Big[t\Big]_{t_0}^{t},$$
$$\log\left(\frac{x(t)}{x_0}\right) = \lambda(t - t_0), \quad x(t) = x_0 \mathrm{e}^{\lambda(t-t_0)}.$$

Substituting λ and x_0, the analytical solution for this specific case will be:

$$x(t) = \mathrm{e}^{-t}.$$

1.2. Explain and provide definitions of the local and the global truncation error.

The definition of local and global truncation errors given in the course slides is the same definition as local and global errors in [1]. Note that the local truncation error is defined differently in this book and thus, we'll stick to the latter definition and will refer to them as local and global errors from now on.

Both the local and global errors are measures of how off the solver method is from the "real" analytical solution. The difference between them comes in how the analytical solution is calculated. Considering x_{n+1} the numerical solution at step $n+1$ and $\hat{x}(t; t_0, x_0)$ the analytical solution at time t with initial conditions t_0 and x_0 ($\hat{x}(t; t_0, x_0) = x_0 \exp(\lambda(t - t_0))$ for the test equation), the local error at step $n + 1$ is calculated as:

$$e^l_{n+1} = \left| x_{n+1} - \hat{x}(t_{n+1}; t_n, x_n) \right|.$$

Notice that in the analytical solution inputs we are treating the last step as it was the initial condition of the solution. This error can be understood as how off this individual step was.

The global error, on the other hand, is calculated as the difference between the method's solution and the real one, starting at the initial conditions. It's calculated as:

$$e^g_{n+1} = \left| x_{n+1} - \hat{x}(t_{n+1}; t_0, x_0) \right|.$$

1.3. Compute the local and global truncation errors for the test equation when solved with a) the explicit Euler method (fixed step size) and b) the implicit Euler method (fixed step size).

We use the following script for computing the solution for the test equation using the explicit Euler method:

```
function [T,X,X_real,e_l,e_g] = EulerExplicit(fun, anal_sol, tspan, h, x0, args)

t0 = tspan(1);
tf = tspan(end);
T = t0:h:tf;
N = size(T,2);
X = zeros(size(x0,1), N);
X_real = zeros(size(x0,1), N);
e_g = zeros(size(X));
e_l = zeros(size(X));
X(:,1) = x0;
```

```
12  X_real(:,1) = x0;
13
14  for k = 1:N-1
15      f = feval(fun, T(k), X(:,k), args{:});
16      X(:,k+1) = X(:,k) + h * f;
17      X_real(:,k+1) = feval(anal_sol, T(k+1), X(:,1), T(1), args{:});
18
19      e_l(:,k+1) = abs(X(:,k+1) - feval(anal_sol, T(k+1), X(:,k), T(k), args{:}));
20      e_g(:,k+1) = abs(X(:,k+1) - X_real(:,k+1));
21  end
22  end
```

Listing 1: Explicit Euler Method Solver

This function takes as inputs `fun`, a pointer to a function where f is defined, `anal_sol`, a pointer to a function where the analytical solution is defined ($e^{\lambda t}$ in our case), the time span `tspan`, the fixed step size `h`, the initial condition `x0` and an array `args` where lambda will be stored. The function also computes the local and global error at every point.

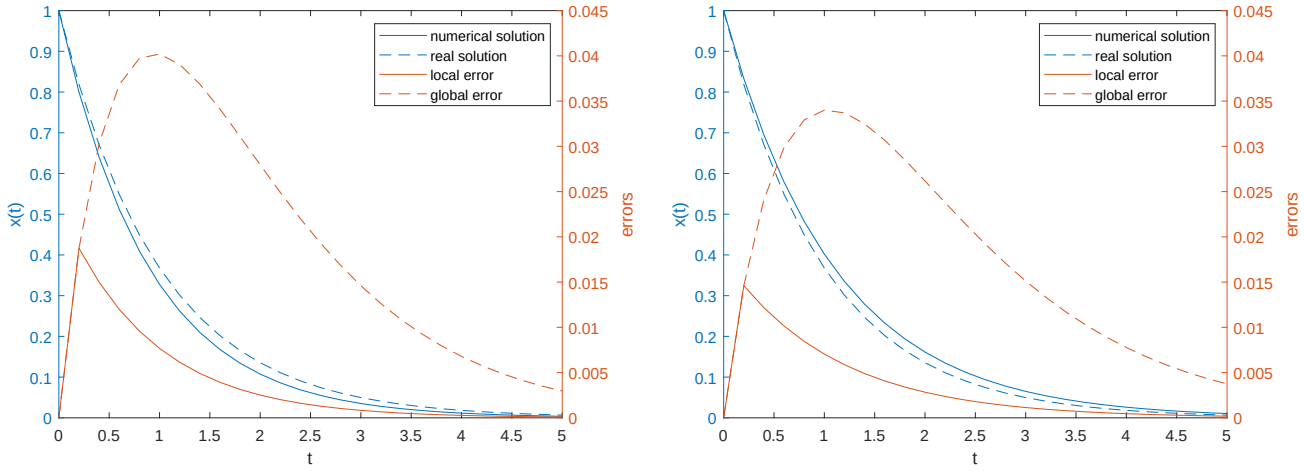
For the implicit Euler method, we just need to change the loop definition to:

```
14  for k = 1:N-1
15      X(:,k+1) = (eye(size(x0,1))-lambda * h)^(-1) * X(:,k);
16      X_real(:,k+1) = feval(anal_sol, T(k+1), X(:,1), T(1), lambda);
17
18      e_l(:,k+1) = abs(X(:,k+1) - feval(anal_sol, T(k+1), X(:,k), T(k), lambda));
19      e_g(:,k+1) = abs(X(:,k+1) - X_real(:,k+1));
20  end
```

Listing 2: Implicit Euler Method Solver

Note that here, because of the test equation, we can solve the implicit step directly. In other implementations we might need to call another method in order to solve for the step, f.e. Newton's method. Both explicit and implicit Euler are explained more thoroughly in part 2 and part 3.

We make a test run for showing the errors with a $\lambda = -1$, $x_0 = 1$ and a time step `h` $= 0.2$. The results are shown below:



(a) Explicit Euler

(b) Implicit Euler

Figure 1: Local and global errors of the test equation

In these plots we can observe the two methods in action, and how they lend us a different result. For this particular time step size both the local and global error in the implicit Euler are lower than in the explicit one.

1.4. Plot the local error vs the time step for the explicit Euler method and the implicit Euler method. Does the plot behave as you would expect. Explain what we mean by order. What is the order of the explicit Euler method and the implicit Euler method, respectively. You should base your answer on your numerical simulations. Is this as you would expect using asymptotic theoretical considerations?

In Ascher-Petzold chapter 3.2 [1] we find a definition of consistency of the method. We can find the order of consistency of these methods through the *local truncation error*. In the book there's a derivation for the explicit Euler method that yields:

$$\mathbf{d}_n = \frac{h_n}{2}\mathbf{y}''(t_{n-1}) + \mathcal{O}(h_n^2).$$

Therefore, we can conclude the explicit Euler method is consistent of order 1. Using Dahlquist theorem [2], as the method is also zero-stable we know the method is convergent of order 1. Later in the chapter, the local error (here written as $\mathbf{l}_n$) is defined. Equation (3.14) in [1] reads:

$$h_n|\mathcal{N}_h\hat{\mathbf{y}}(t_n)| = |\mathbf{l}_n|(1 + \mathcal{O}(h_n)).$$

We can therefore conclude that the local error will have order $1 + 1 = 2$.

We can follow a similar reasoning for obtaining the orders of the implicit Euler. We define the difference operator of the implicit Euler as:

$$\mathcal{N}_h u(t_n) = \frac{u(t_n) - u(t_{n-1})}{h_n} - f(t_n, u(t_n)).$$

Using Taylor's expansion $y(t_{n-1}) = y(t_n) - h_n y'(t_n) + \frac{h_n^2}{2}y''(t_n) + \dots$ we get a local truncation error:

$$\begin{aligned}
\mathbf{d}_n = \mathcal{N}_h y(t_n) &= \frac{y(t_n) - y(t_{n-1})}{h_n} - f(t_n, y(t_n)) \\
&= \frac{y(t_n) - \left[y(t_n) - h_n y'(t_n) + \frac{h_n^2}{2}y''(t_n) + \mathcal{O}(h_n^3)\right]}{h_n} - y'(t_n) \\
&= -\frac{h_n}{2}\mathbf{y}''(t_n) + \mathcal{O}(h_n^2).
\end{aligned}$$

Therefore, we can conclude that the Implicit Euler is consistent of order 1. Notice that we get a different sign as in the explicit Euler, and this is completely consistent with what we see in figure 1. As the implicit Euler is also zero-stable we know that the method is also convergent of order 1. Just as before, the local error should also scale with order 2 with the time step size.

Computing the mean of the local and the global error for the test equation with $\lambda = -1$, $x_0 = 1$, `tspan` $= [0, 20]$ and different time step sizes, ranging from $h = 0.01$ to 0.5 we finally obtain the results shown in Figure 2. The local error escalates as a function of h^2 for both methods, just as what was expected from the mathematical derivation.

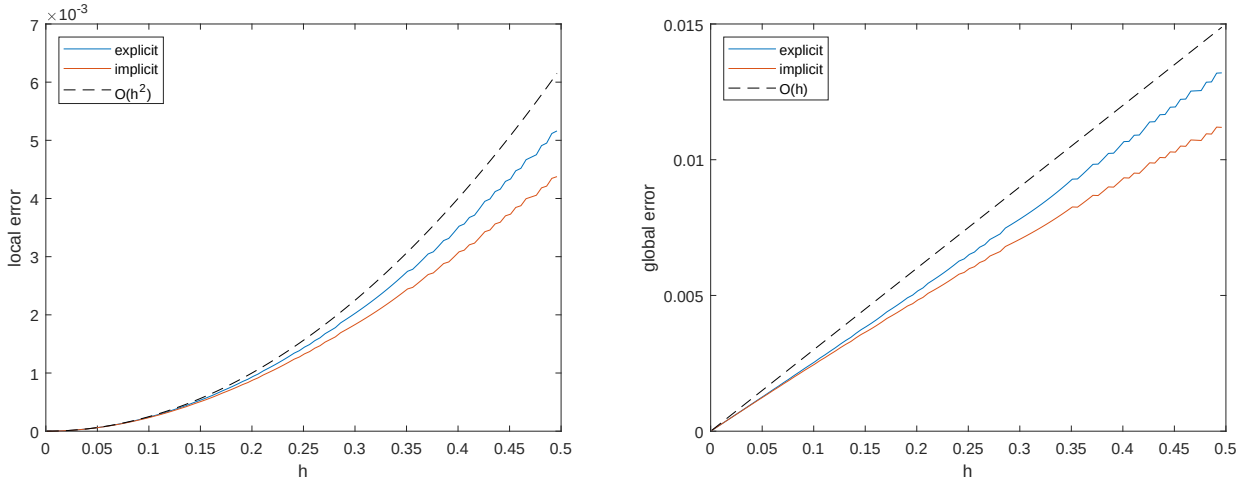
1.5. Plot the global error vs the time step for the explicit Euler method and the implicit Euler method. Does the plot behave as you would expect.

Following what we got in the previous exercise, as both methods have order 1 the global error should scale linearly with h. Taking a look at Figure 2, we can see that the global error behaves as expected for both methods. It's also worth mentioning that the Implicit method has a lower local and global error for the test equation. This makes total sense if we analyze the expression of the local truncation error obtained in the previous exercise.

1.6. Explain stability of a method and derive the expressions for the stability of the explicit Euler method and the implicit Euler method. Plot the stability regions. Explain A-stability. Is the explicit Euler-method A-stable? Is the implicit Euler method A-stable?

The concept of stability of a method comes from the concept of stability of a given problem. When looking at the Test Equation system in 1 (with $\lambda \in \mathbb{C}$), we can distinguish three different cases:

- $\mathrm{Re}(\lambda) > 0$, the solution grows exponentially with t and the problem is *unstable*. The distance between solution curves increases in time. In linear problems, the space of the solution (in this case $\mathbb{R}$) is then referred as the *unstable space*



(a) Local Error *(b) Global Error*

Figure 2: Errors of the Test Equation

- $\mathrm{Re}(\lambda) = 0$, the solution is either oscillating or 0, making the distance between solution curves to stay constant over time. This is referred as the *center space*
- $\mathrm{Re}(\lambda) < 0$, then the exponential decreases over time. The distance between curves decreases and the problem is *asymptotically stable*. This yields an additional *absolute stability* requirement:

$$|x_n| \leq |x_{n-1}|, \quad \forall n. \tag{2}$$

If we now take a look at how the Explicit Euler method is calculated iteratively for the Test Equation:

$$x_{n+1} = x_n + h\lambda x_n = (1 + h\lambda)x_n = (1 + h\lambda)^2 x_{n-1} = \cdots = (1 + h\lambda)^{n+1} x_0.$$

The only way 2 is going to be met is if:

$$|1 + h\lambda| \leq 1.$$

This restricts the stability of the Explicit Euler to certain values of $h\lambda$. This region is depicted coloured in Figure 3.

For the Implicit Euler:

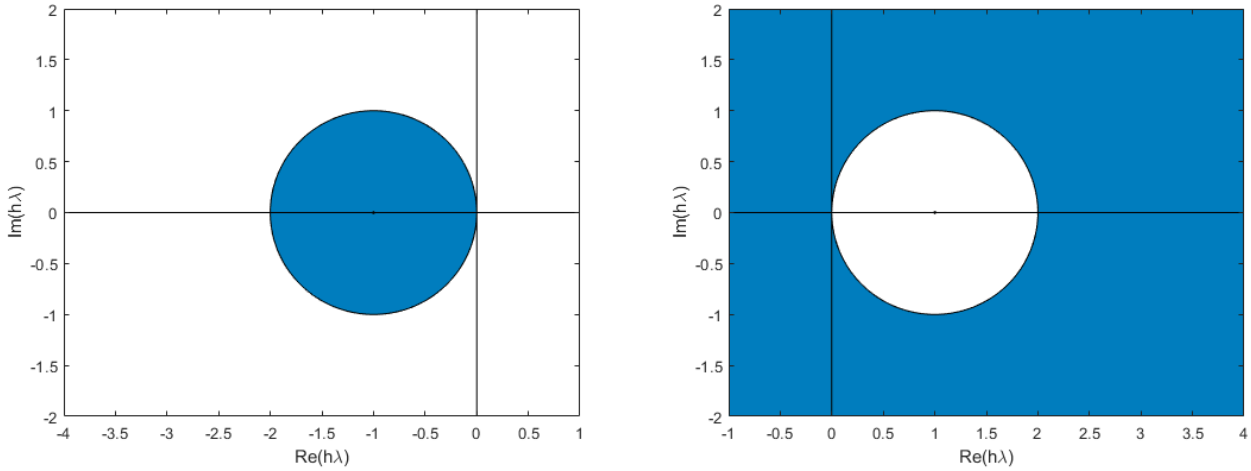
$$x_{n+1} = x_n + h\lambda x_{n+1} = (1 - h\lambda)^{-1} x_n = (1 - h\lambda)^{-2} x_{n-1} = \cdots = (1 - h\lambda)^{-(n+1)} x_0.$$

The stability condition in this case is:

$$\frac{1}{|1 - h\lambda|} \leq 1.$$

This region is also depicted in Figure 3.

This leads us to the concept of A-Stability. A method is considered *A-stable* if its region of absolute stability contains the entire left half-plane of $z = h\lambda$, i.e. $\mathrm{Re}(h\lambda) < 0$. From Figure 3 we can conclude that the Explicit Euler is not A-stable, while the Implicit Euler is.



(a) Explicit Euler *(b) Implicit Euler*

Figure 3: Absolute stability regions

Part 2: Explicit ODE solver

We consider the initial value problem

$$\dot{x}(t) = f(t, x(t), p), \quad x(t_0) = x_0, \tag{3}$$

where $x \in \mathbb{R}^{n_x}$ and $p \in \mathbb{R}^{n_p}$.

2.1. Describe the explicit Euler algorithm (i.e. provide an algorithm for it in your report and explain how you get from the differential equations to the numerical formulas).

Taking a solution $x(t)$ of the IVP (3), we consider its Taylor expansion around time t_0:

$$\begin{aligned} x(t_0 + h) &= x(t_0) + hx'(t_0) + \frac{h^2}{2}x''(t_0) + \mathcal{O}(h^3) \\ &\approx x(t_0) + hx'(t_0). \end{aligned}$$

By ignoring quadratic terms and noticing that $x'(t) = f(t, x(t), p)$ we get the expression of the Explicit Euler method. If we discretize time as $t_n = t_0 + n \cdot h \ : 0 \leq n \leq N$ we can calculate a mesh function x_n at each point in the following way:

$$\begin{aligned} x_0 &= x(t_0), \\ x_n &= x_{n-1} + hf(t_{n-1}, x_{n-1}, p) \quad \text{for } n = 1, 2, \ldots, N. \end{aligned}$$

The numerical solution $x_n \to x(t_n)$ as $h \to 0$. This is proven by the derivation of convergence and global error order of the method in exercise 1.4.

2.2. Implement an algorithm in Matlab for the explicit Euler method with fixed time-step and provide this in your report. Use a format that enables syntax highlighting.

```
function [T,X] = EulerExplicit_fixed(fun, tspan, h, x0, args)

t0 = tspan(1);
tf = tspan(end);
T = t0:h:tf;
N = size(T,2);
X = zeros(size(x0,1), N);
X(:,1) = x0;
```

```
9
10 for k = 1:N-1
11     f = feval(fun, T(k), X(:,k), args{:});
12     X(:,k+1) = X(:,k) + h * f;
13 end
14 end
```

Listing 3: Explicit Euler method with fixed time step size

2.3. Implement an algorithm in Matlab for the explicit Euler method with adaptive time step and error estimation using step doubling.

In this implementation we use an asymptotic time step controller. As we are going to model problems without an analytical solution, we'll use step doubling to make an estimate of the error.

```
1  function [T,X,r_out,h_out,info] = EulerExplicit_adaptive(fun,tspan,h0,x0,abstol,reltol,args)
2
3  epstol = 0.8;
4  facmin = 0.1;
5  facmax = 5.0;
6
7  t0 = tspan(1);
8  tf = tspan(end);
9  t = t0;
10 h = h0;
11 x = x0;
12
13 T = t0;
14 X = x0;
15 r_out = [];
16 h_out = [];
17 info = zeros(1,4);
18
19 nfun = 0;
20 nstep = 0;
21 naccept = 0;
22
23 while t < tf
24     if (t+h > tf)
25         h = tf-t;
26     end
27     f = feval(fun,t,x,args{:});
28     nfun = nfun + 1;
29     AcceptStep = false;
30     while ~AcceptStep
31         x1 = x + h*f;
32
33         hm = 0.5*h;
34         tm = t + hm;
35         xm = x + hm*f;
36         fm = feval(fun,tm,xm,args{:});
37         nfun = nfun + 1;
38         x1hat = xm + hm*fm;
39
40         nstep = nstep + 1;
41
42         e = abs(x1hat-x1);
43         r = max(e./max(abstol, abs(x1hat) .* reltol));
44         AcceptStep = (r <= 1.0);
45
46         if AcceptStep
47             t = t+h;
48             x = x1hat;
49
50             T = [T,t];
51             X = [X,x];
52             r_out = [r_out, r];
53             h_out = [h_out, h];
```

```
54              naccept = naccept + 1;
55          end
56
57          h = max(facmin, min(sqrt(epstol/r), facmax)) * h;
58      end
59  end
60
61  info(1) = nfun;
62  info(2) = nstep;
63  info(3) = naccept;
64  info(4) = nstep - naccept;
65
66  end
```

Listing 4: Explicit Euler method with adaptive time step size

2.4. Test your algorithms on the Van der Pol problem ($\mu = 1.5$ and $\mu = 15$, $\mathbf{x_0} = [1.0; 1.0]$)

The IVP (3) for the Van der Pol problem will be used in the entire assignment by calling the following function:

```
1  function xdot = VanderPol(t,x,mu)
2  xdot = zeros(2,1);
3  xdot(1) = x(2);
4  xdot(2) = mu*(1-x(1)^2)*x(2) - x(1);
5  end
```

Listing 5: Van der Pol equation

Both methods will be called simply by setting the parameters and respectively:

```
1  [T,X] = EulerExplicit_fixed(@VanderPol, tspan, h, x0, args);
2  [T,X,r_out,h_out,info] = EulerExplicit_adaptive(@VanderPol, tspan, h0, x0, abstol, reltol, args);
```

In the following plots we can observe the results of the Explicit Euler method with fixed and adaptive step size, both solving the Van der Pol problem for a μ of 1.5 and 15. The solutions are plotted using different step sizes for the fixed method and different tolerances for the adaptive method. The tolerance value sets both the values of `abstol` and `reltol`.

Taking a look at the results for the fixed method in Figures 4 and 5, we can spot a curious behaviour. Even though the shape of the solution for high step sizes looks alike to the real solution, there is a considerable frequency difference among them. We can also see how for $\mu = 15$ the problem becomes more *stiff*, thus requiring a smaller step size (values of $h \sim 0.2$ made it explode).

The same behaviours can also be observed in the results for the adaptive method in Figures 6 and 7. These plots also show the final chosen step size per every time iteration and the measure of the relative error. Of course, when the tolerance is very fine $h \to 0$ and $r \to 0.8$, which is the chosen value of `epstol`. For the case when $\mu = 15$, we can also observe the increase in stiffness in these two plots. Also, we can observe how the no. of function calls and steps varies when we decrease the tolerance value in Tables 1 and 2.

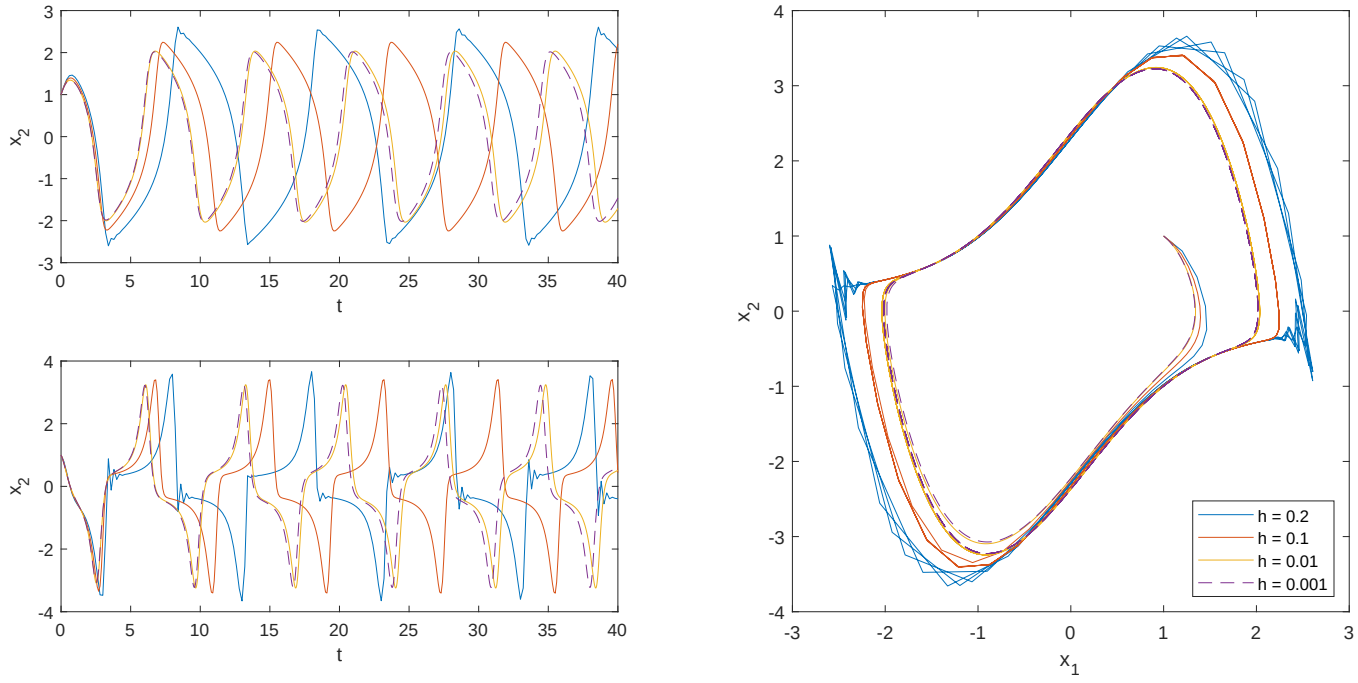


Figure 4: Solution for the Van der Pol problem ($\mu = 1.5$) using Explicit Euler with fixed step size

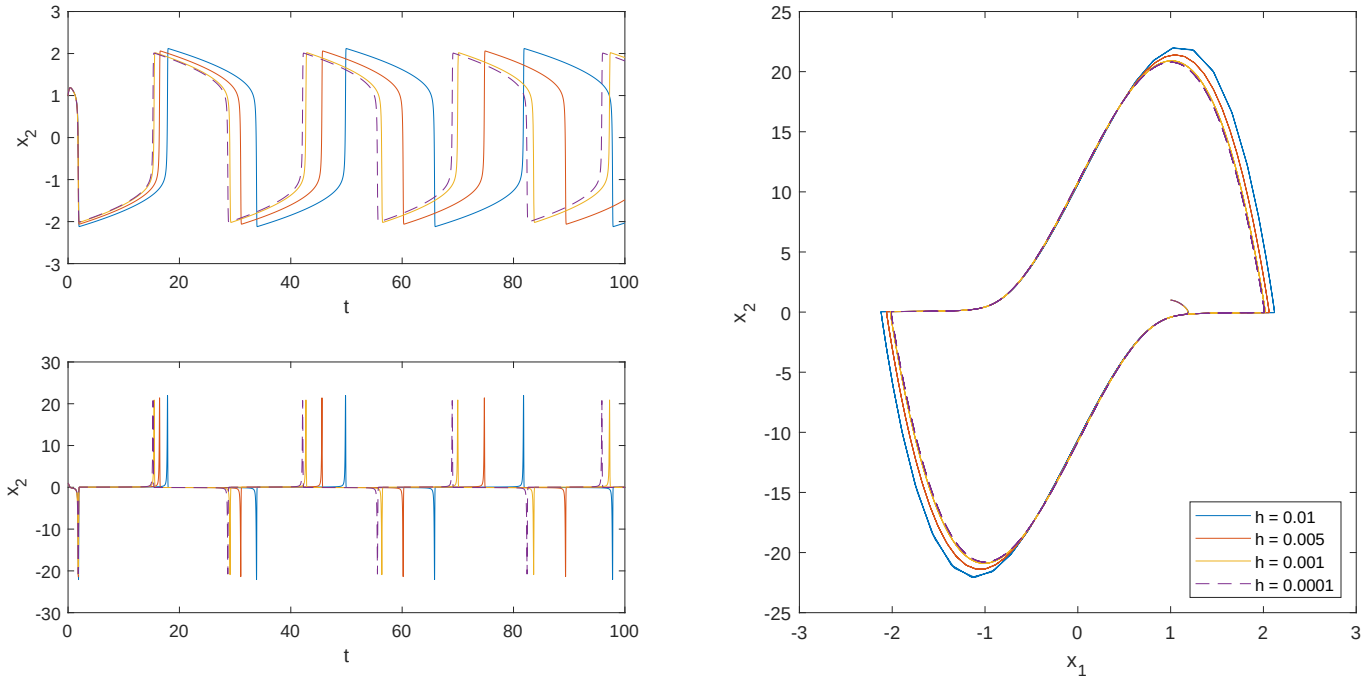


Figure 5: Solution for the Van der Pol problem ($\mu = 15$) using Explicit Euler with fixed step size

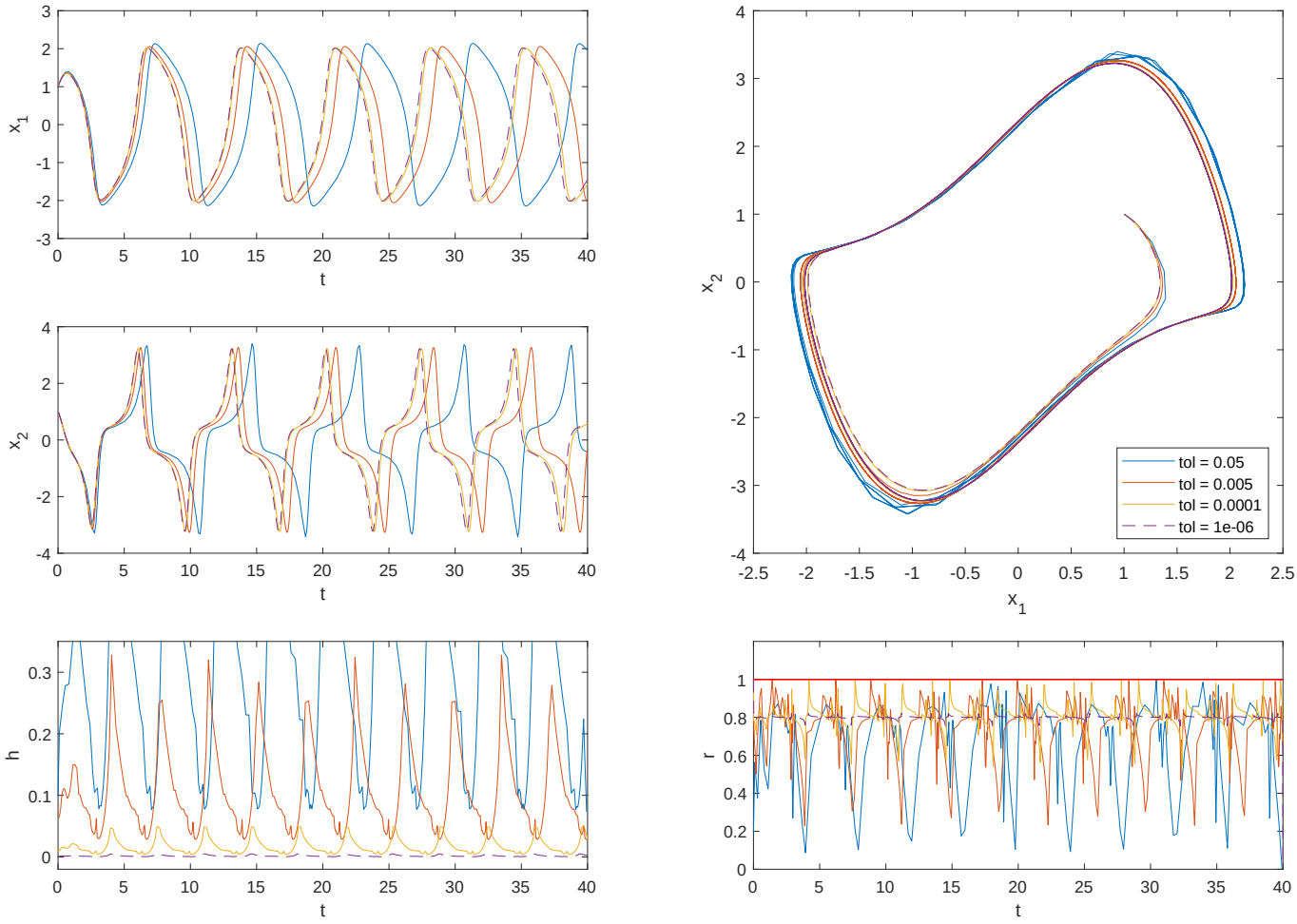


Figure 6: Solution for the Van der Pol problem ($\mu = 1.5$) using Explicit Euler with adaptive step size

Tolerances	0.05	0.005	0.0001	1e-06
Function evaluations	445	1200	7383	73678
Calculated steps	263	660	3695	36840
Accepted steps	182	540	3688	36838
Rejected steps	81	120	7	2

Table 1: Parameters of the Explicit Euler with adaptive step size for the Van der Pol problem ($\mu = 1.5$)

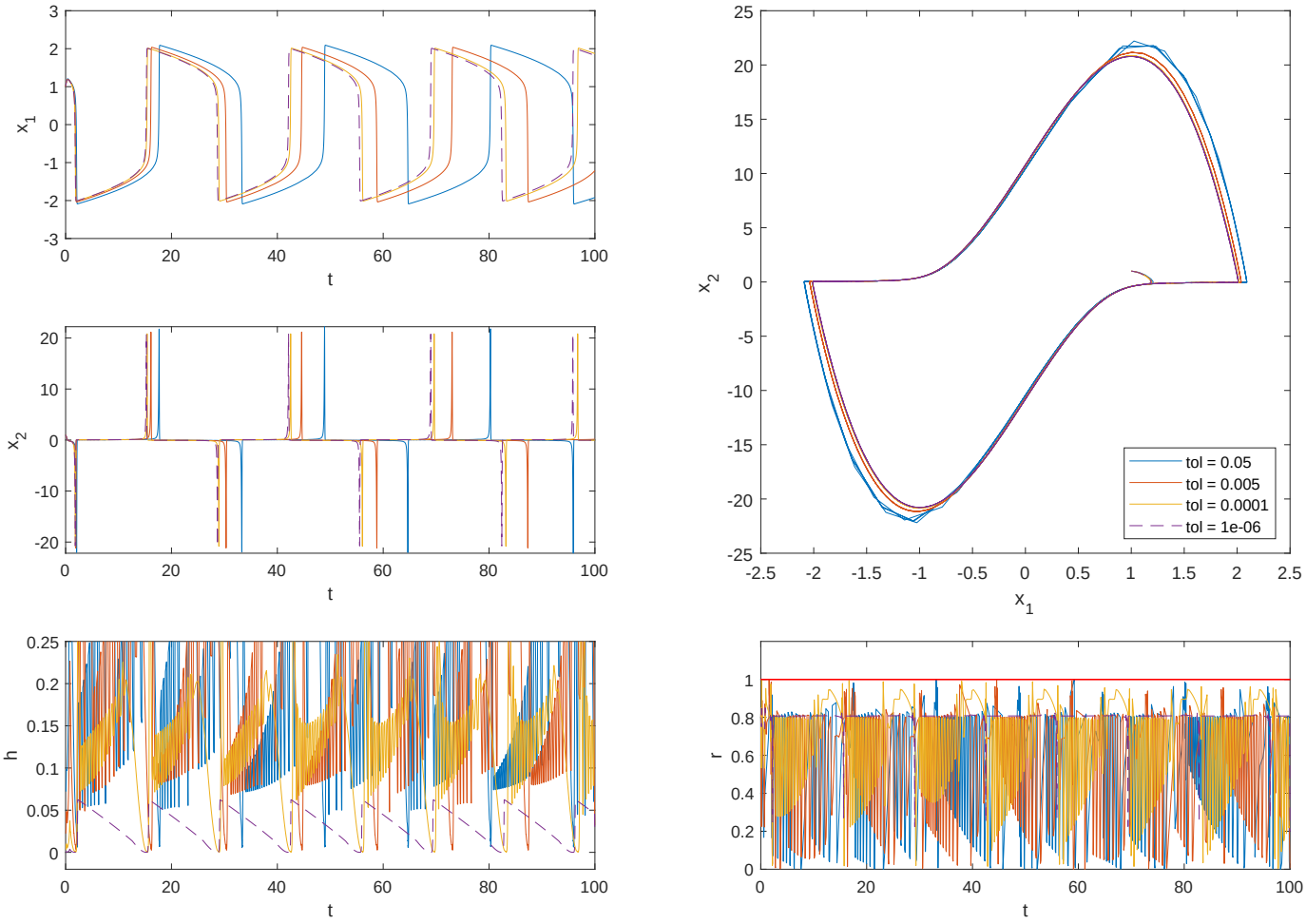


Figure 7: Solution for the Van der Pol problem ($\mu = 15$) using Explicit Euler with adaptive step size

Tolerances	0.05	0.005	0.0001	1e-06
Function evaluations	1583	2485	11246	1.0404e+05
Calculated steps	954	1418	5740	52022
Accepted steps	629	1067	5506	52019
Rejected steps	325	351	234	3

Table 2: Parameters of the Explicit Euler with adaptive step size for the Van der Pol problem ($\mu = 15$)

2.5. Test your algorithms on the adiabatic CSTR problem described in the papers uploaded to Learn (3D-version and 1D-version).

The IVPs (3) for both versions of the CSTR problem, obtained from [4] and [5] are included in the following functions:

```
function xdot = CSTR_3D(t,x,F,params)
% F to IS units
F = F/60000;

beta = params(1);
k0 = params(2);
EaR = params(3);
CAin = params(4);
CBin = params(5);
Tin = params(6);
V = params(7);

CA = x(1);
CB = x(2);
T = x(3);
xdot = zeros(3,1);

% Rate of reaction
r = k0 * exp(-EaR/T) * CA * CB;
% Production rates and rate of temperature
RA = -r;
RB = -2*r;
RT = beta*r;
% Final ODE
xdot(1) = F/V * (CAin - CA) + RA;
xdot(2) = F/V * (CBin - CB) + RB;
xdot(3) = F/V * (Tin - T) + RT;
end
```

Listing 6: CSTR 3D equation

```
function Tdot = CSTR_1D(t, T, F, params)
% F to IS units
F = F/60000;

beta = params(1);
k0 = params(2);
EaR = params(3);
CAin = params(4);
CBin = params(5);
Tin = params(6);
V = params(7);

% Rate of reaction
r = k0 * exp(-EaR/T) * (CAin + 1/beta * (Tin - T)) * (CBin + 2/beta * (Tin - T));
% Rate of temperature
RT = beta * r;
% Final ODE
Tdot = F/V*(Tin - T) + RT;
end
```

Listing 7: CSTR 1D equation

Just as in [4] and [5], the flow will be a function of time. Its shape can be seen in Figure 8, together with a solution for the 3D-CSTR problem. The way to call the method having this flow function over time is slightly trickier than in the Van der Pol problem. I tried first creating F as a piecewise function and substituting the time in every CSTR function call. However, this was proven to be very time consuming as it implied working with symbolic functions. The best way of achieving the same output was by:

```
% Piecewise flow definition
tspans = [[0,3];[3,5];[5,7];[7,9];[9,12];[12,16];[16,18];...
    [18,20];[20,22];[22,24];[24,28];[28,32];[32,35]];
Fs = [700,600,500,400,300,200,300,400,500,600,700,200,700];

T = [];
X = [];

for i=1:size(Fs,2)
    tspan = tspans(i,:)*60;
    F = Fs(i);
    args = {F, [beta,k0,EaR,CAin,CBin,Tin,V]};
    [T_local,X_local] = EulerExplicit_fixed(@CSTR_3D, tspan, h, x0, args);

    T = [T, T_local(1:end-1)];
    X = [X, X_local(:,1:end-1)];

    x0 = X_local(:,end);
end
% Add last point and normalize
T = [T,T_local(end)]/60;
X = [X, X_local(:,end)]-273;
```

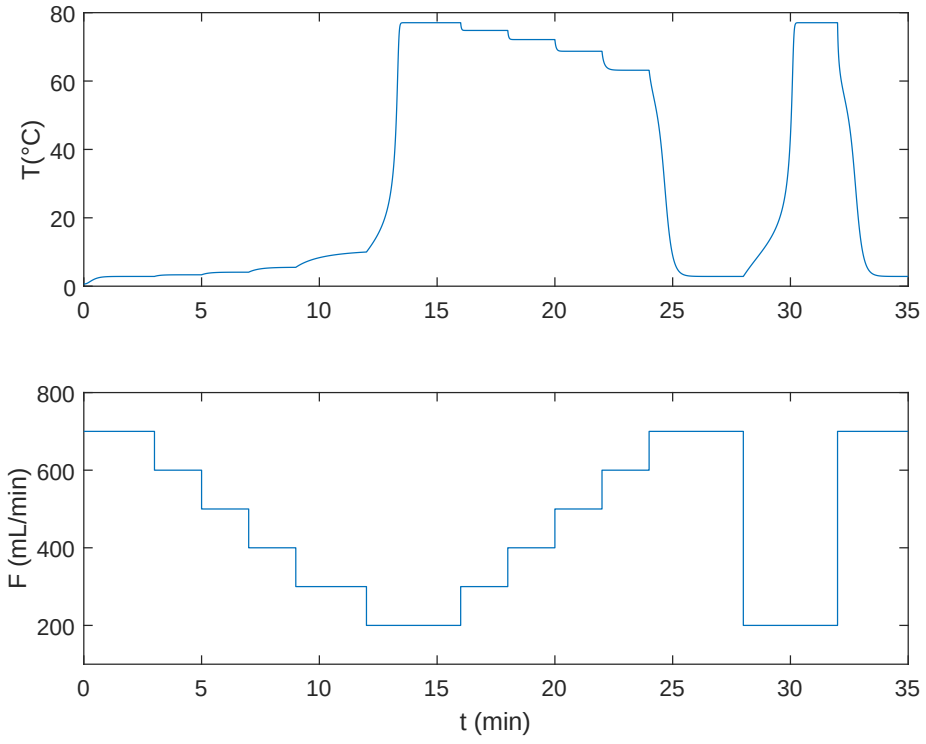


Figure 8: Solution and value of the flow over time for the CSTR 3D problem

The results for the 3D problem and 1D problem are very similar. This can be seen in Figure 9, where a closeup of the interval $t \in [0, 1.5]$ minutes is shown. This was the region where we can find the greatest difference between the two problems. The following plots show the results of different runs of the CSTR problems, for the fixed and adaptive Explicit Euler with different values of steps and tolerances respectively.

Taking a look at the results using the fixed method (Figure 10), it's very difficult to spot a difference between the 3D and 1D case. The plot behaves the same for every chosen time step. However, the results from the adaptive in Figures 11 and 12 and Tables 3 and 12 are much more enlightening. In the 3D problem, the solution is pretty close even for the biggest tolerance of 0.05. Moreover, we can observe some spike behaviour in the value of r that

coincides with every step change of the flow function. Also the problem becomes more stiff when the value of F goes down (the r value and therefore change in h shifts a lot in these regions). This also explains the spike behaviour observed in the fixed method only at these points in time. Finally, we can observe that it takes a tighter tolerance for the 1D problem to converge. However, the number of function evaluations is a lot smaller than the 3D problem for the same tolerances and lowering down enough the tolerance yields pretty good results.

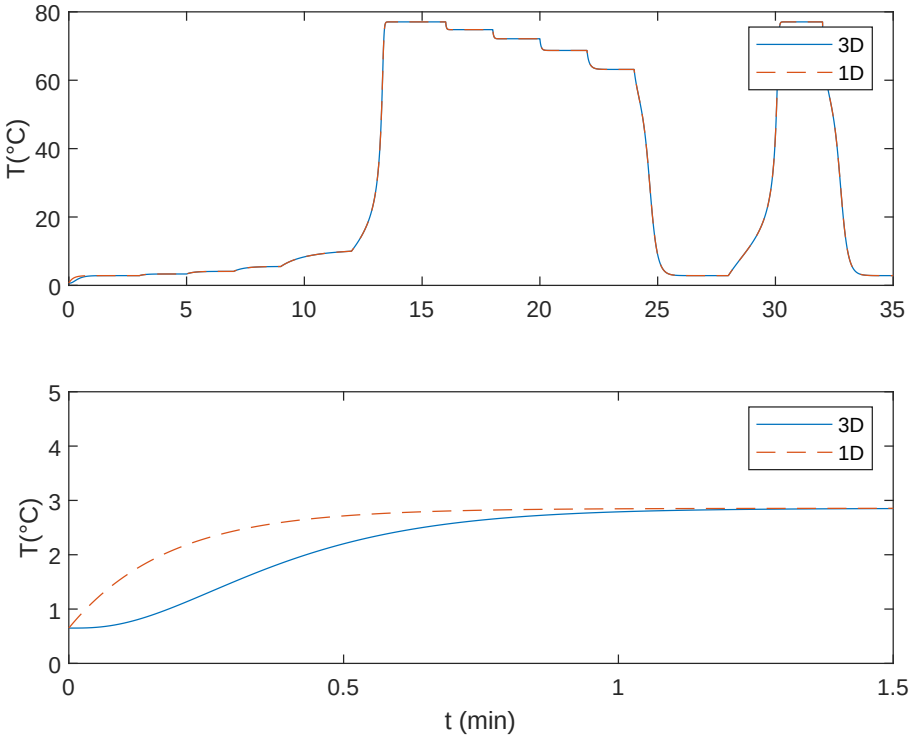
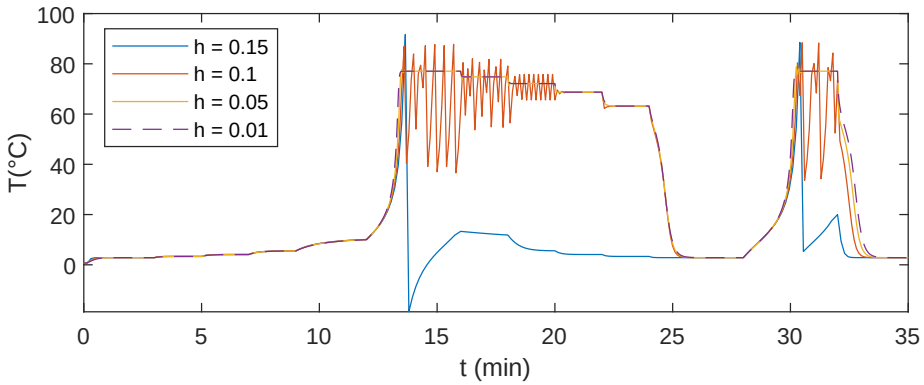
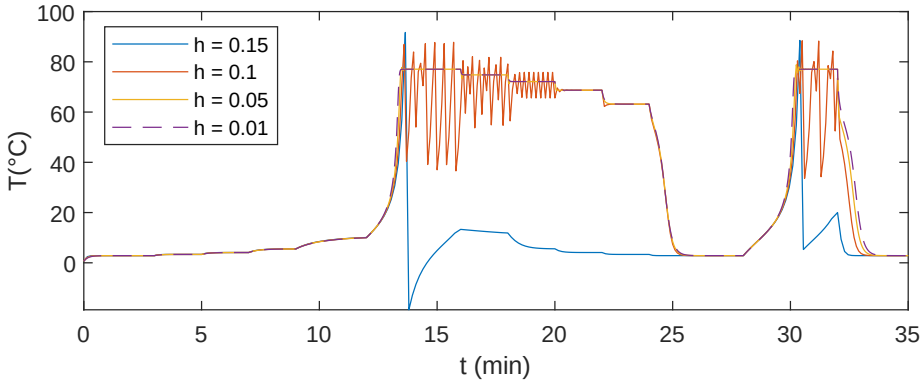


Figure 9: Comparison of the solutions for the CSTR 3D and 1D



(a) CSTR 3D problem



(b) CSTR 1D problem

Figure 10: Solution for the CSTR problem using Explicit Euler with fixed step size

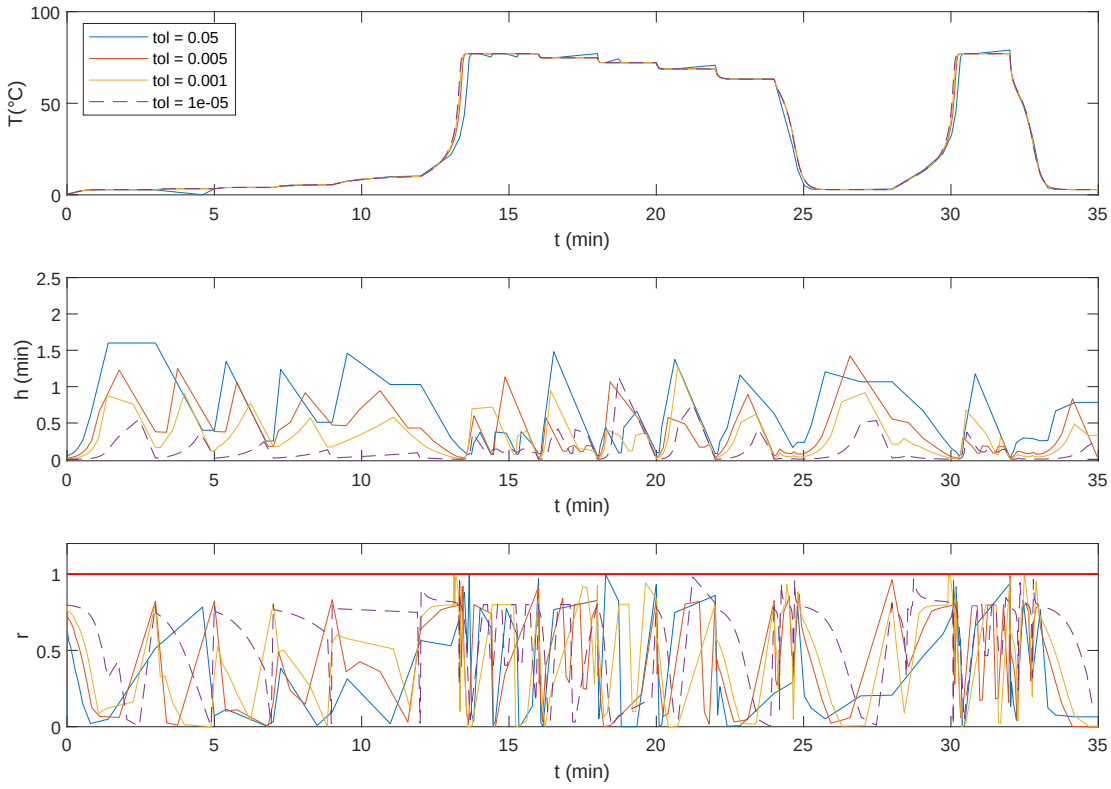


Figure 11: Solution for the CSTR 3D problem using Explicit Euler with adaptive step size

Tolerances	0.05	0.005	0.001	1e-05
Function evaluations	207	431	726	5250
Calculated steps	120	245	400	2657
Accepted steps	87	186	326	2593
Rejected steps	33	59	74	64

Table 3: Parameters of the Explicit Euler with adaptive step size for the CSTR 3D problem

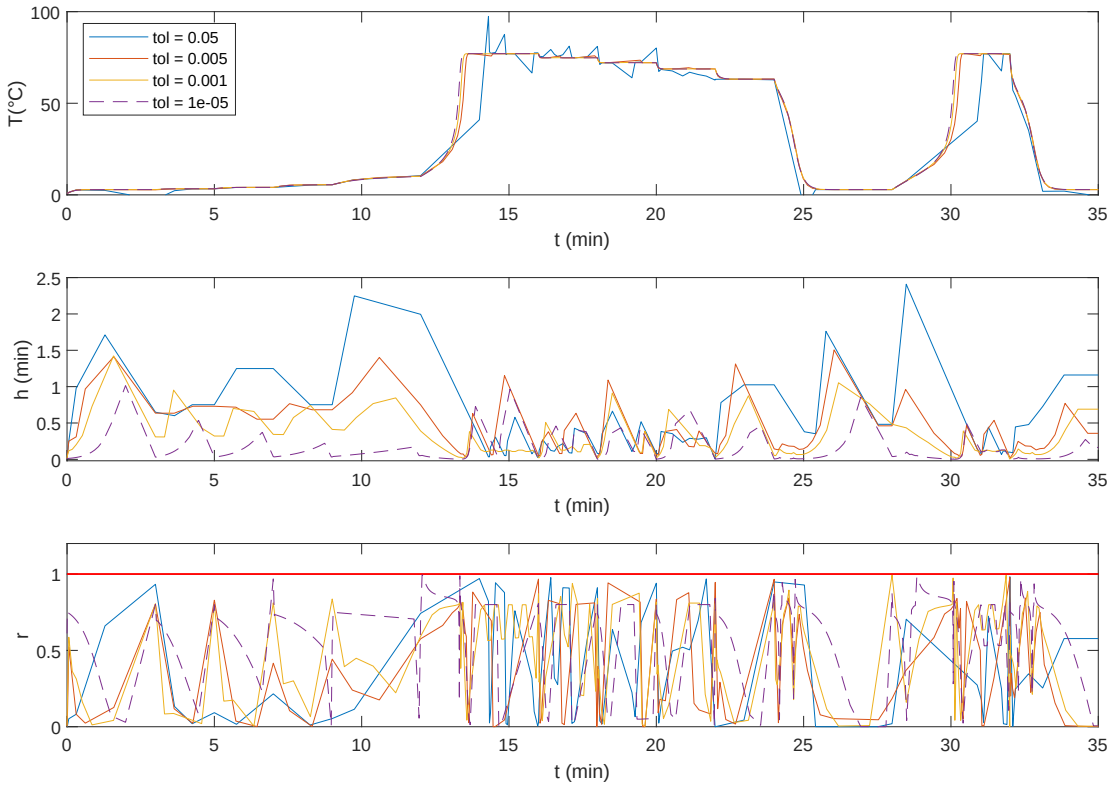


Figure 12: Solution for the CSTR 1D problem using Explicit Euler with adaptive step size

Tolerances	0.05	0.005	0.001	1e-05
Function evaluations	194	256	455	2285
Calculated steps	115	148	261	1171
Accepted steps	79	108	194	1114
Rejected steps	36	40	67	57

Table 4: Parameters of the Explicit Euler with adaptive step size for the CSTR 1D problem

2.6. Compare the results from your algorithms with the results you get using some of Matlab's ODE solvers

We'll compare the solution obtained with the explicit Euler to two ODE solvers from Matlab: `ode45` and `ode15s`. The first one is an implementation of the Dormand-Prince 5(4), a method from the Explicit Runge-Kutta family. Being an explicit method, it's not very good when dealing with stiff problems. `ode15s`, on the other hand, is designed to work specifically with these types of problems. Both of them are also of adaptive step size. For that reason, we will compare the Explicit Euler with adaptive step size to these two: see how they behave for a given tolerance and problem and analyse their increase in the number of function evaluations when narrowing down the tolerance.

For the Van der Pol problem, we tested Explicit Euler against `ode45` for the non-stiff case ($\mu = 1.5$), and against `ode15s` for the stiff case ($\mu = 15$). The obtained results for the Van der Pol problem (Figures 13 and 14 and Tables 5 and 6) show what we discussed previously. The Explicit Euler works fine in the non-stiff case, even though the `ode45` achieves better accuracy with a fewer no. of function evaluations. In the stiff case, however, the no. of evaluations is ridiculously high compared to the `ode15s`.

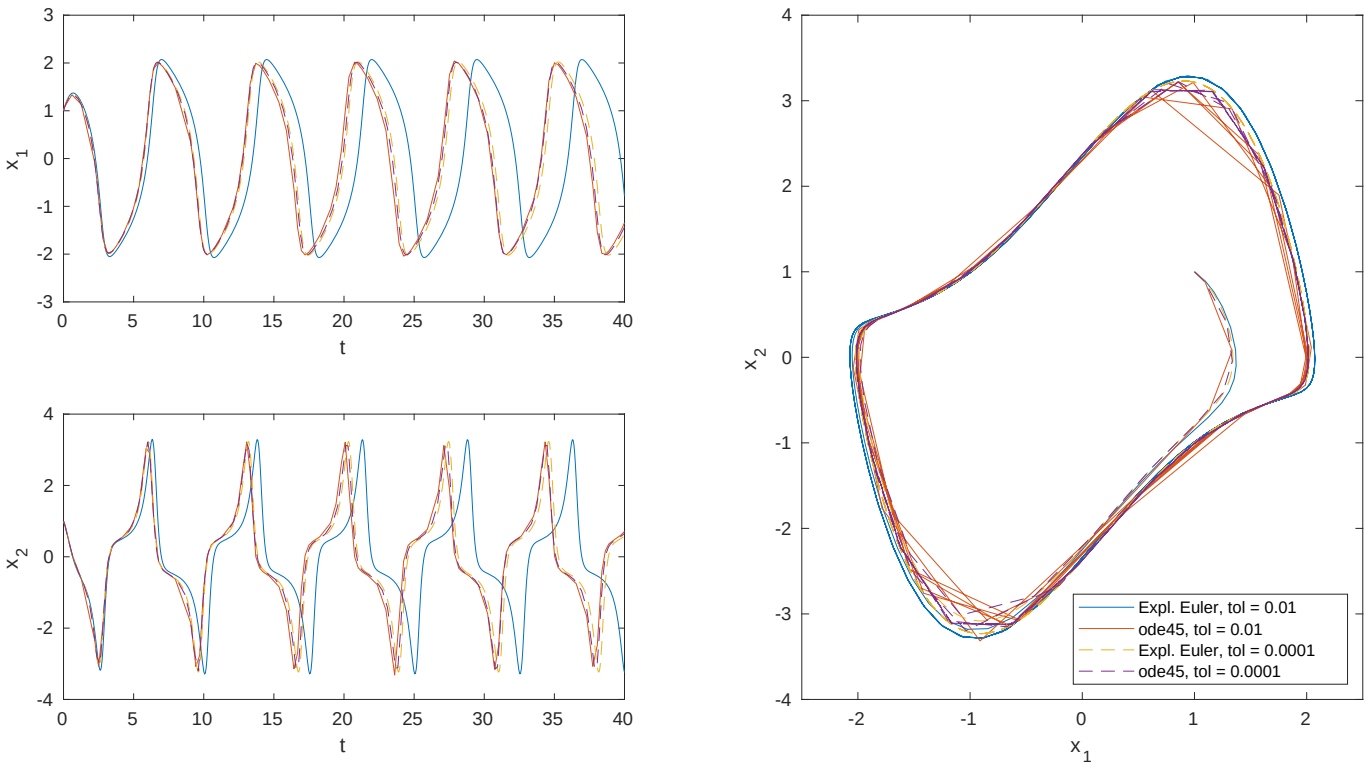


Figure 13: Solution for the Van der Pol problem ($\mu = 1.5$) using Explicit Euler vs. `ode45`

Method	**Expl. Euler**		**ode45**	
Tolerances	0.01	0.0001	0.01	0.0001
Function evaluations	853	7383	787	1357
Calculated steps	473	3695	131	226
Accepted steps	380	3688	100	180
Rejected steps	93	7	31	46

Table 5: Parameters of the Explicit Euler vs. `ode45` for the Van der Pol problem ($\mu = 1.5$)

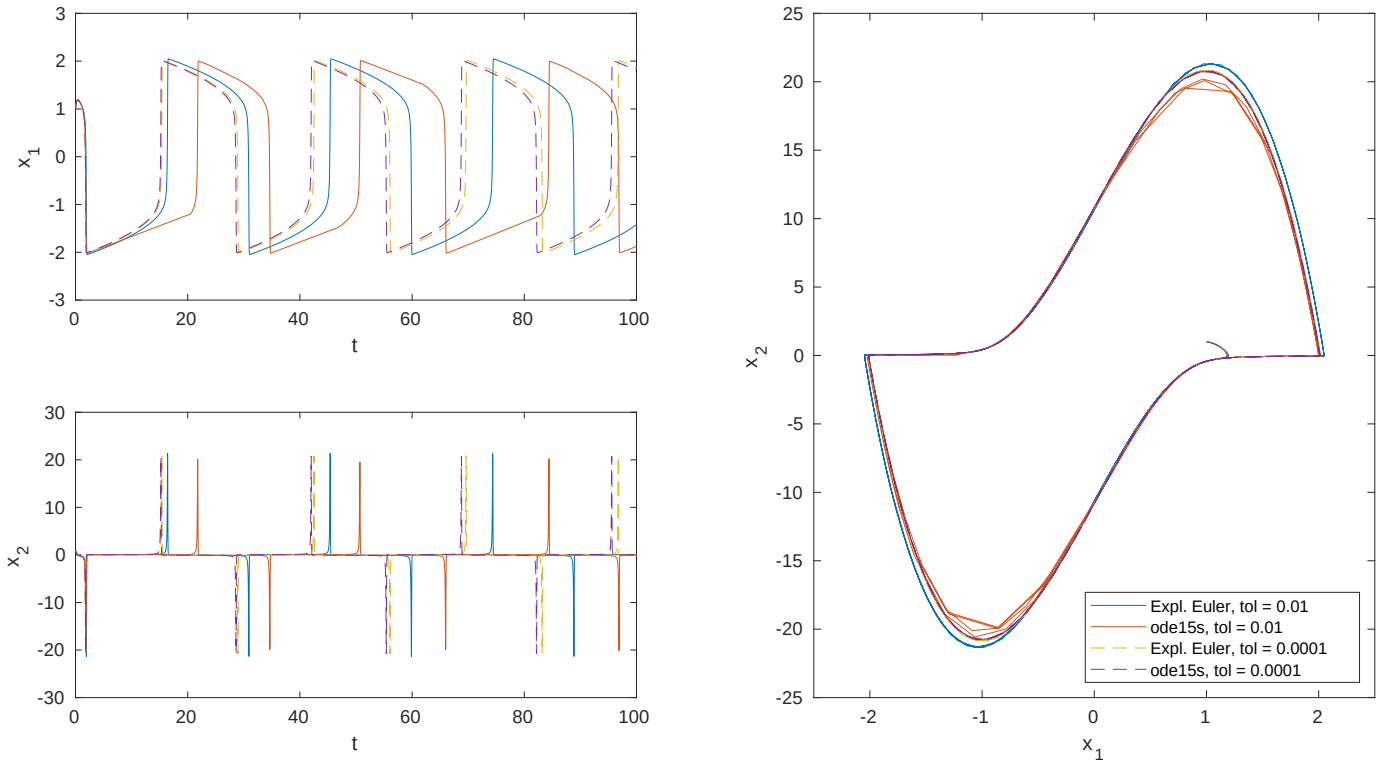
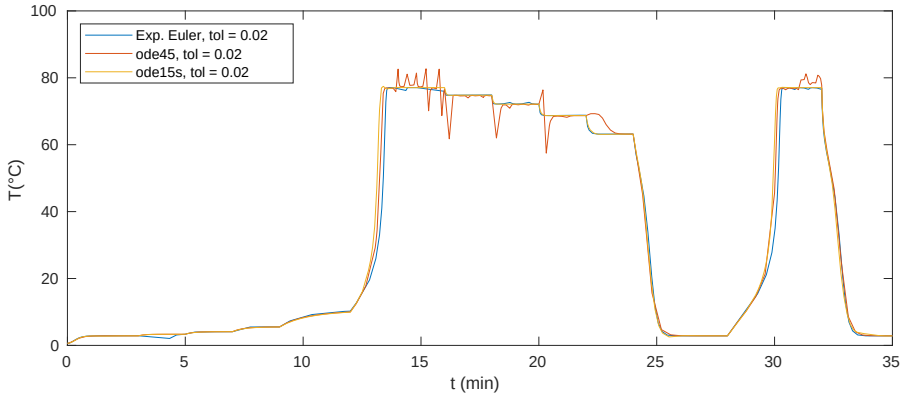


Figure 14: Solution for the Van der Pol problem ($\mu = 15$) using Explicit Euler vs. `ode15s`

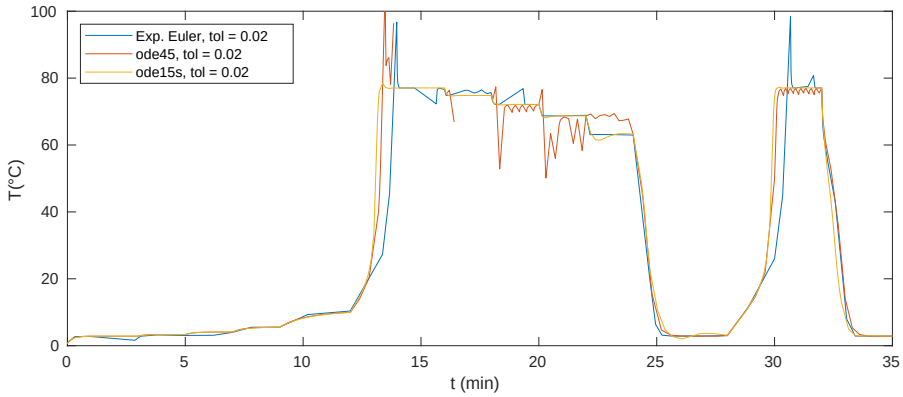
Method	**Expl. Euler**		**ode15s**	
Tolerances	0.01	0.0001	0.01	0.0001
Function evaluations	2108	11246	1273	2780
Calculated steps	1219	5740	558	1327
Accepted steps	889	5506	411	1094
Rejected steps	330	234	147	233

Table 6: Parameters of the Explicit Euler vs. `ode15s` *for the Van der Pol problem ($\mu = 15$)*

For the CSTR problem, we also see how both the Explicit Euler and `ode45` fail where the problem turns more stiff. The `ode15s` achieves very good performance in this case.



(a) CSTR 3D problem



(b) CSTR 1D problem

Figure 15: Solution for the CSTR problem using Explicit Euler vs. `ode45` *and* `ode15s`

Method	**Expl. Euler**	**ode45**	**ode15s**
Tolerances	0.02	0.02	0.02
Function evaluations	291	1477	480
Calculated steps	168	1555	1423
Accepted steps	123	211	203
Rejected steps	45	33	27

Table 7: Parameters of the Explicit Euler vs. `ode45` *and* `ode15s` *for the CSTR-3D problem*

Method	**Expl. Euler**	**ode45**	**ode15s**
Tolerances	0.02	0.02	0.02
Function evaluations	201	1297	375
Calculated steps	117	1270	1275
Accepted steps	84	195	180
Rejected steps	33	19	27

Table 8: Parameters of the Explicit Euler vs. `ode45` *and* `ode15s` *for the CSTR-1D problem*

Part 3: Implicit ODE solver

We consider again the initial value problem in (3).

3.1. Describe the implicit Euler algorithm (i.e. provide an algorithm for it in your report and explain how you get from the differential equations to the numerical formulas).

The idea behind the Implicit Euler is that, instead of using the function evaluation at the previous point, it uses the one at the future point. A more mathematical derivation would be to approximate the integral form of the problem by the right-hand rectangle method, as in

$$x(t_{n+1}) - x(t_n) = \int_{t_n}^{t_{n+1}} f(t, x(t), p)dt \approx hf(t+1, x(t+1), p).$$

As seen in part 1, this improves the absolute region of stability of the Explicit Euler considerably. However, the bad news is that a nonlinear system of equations must be solved in each time step. To achieve this we'll use Newton's method to approximate the solution. The Implicit Euler is therefore calculated as:

$$\begin{aligned} x_0 &= x(t_0), \\ x_n &= x_{n-1} + hf(t_n, x_n, p), \quad \text{solve } x_n \text{ for } n = 1, 2, \ldots, N. \end{aligned}$$

3.2. Implement an algorithm in Matlab for the implicit Euler method with fixed time-step and provide this in your report. Use a format that enables syntax highlighting.

```
function [x,k] = NewtonsMethod(FunJac, tk, xk, h, xinit, tol, maxit, args)
k = 0;
t = tk + h;
x = xinit;
[f,J] = feval(FunJac,t,x,args{:});
k = k + 1;
R = x - h*f - xk;
I = eye(length(xk));

while( (k<maxit) && (norm(R,'inf')>tol) )
    dRdx = I - J*h;
    dx = dRdx\R;
    x = x - dx;
    [f,J] = feval(FunJac,t,x,args{:});
    k = k+1;
    R = x - h*f - xk;
end
```

Listing 8: Newton's method

```
function [T,X] = EulerImplicit_fixed(funJac, tspan, h, x0, args)

t0 = tspan(1);
tf = tspan(end);
T = t0:h:tf;
N = size(T,2);
X = zeros(size(x0,1), N);
X(:,1) = x0;

tol = 1.0e-8;
maxit = 100;

for k=1:N-1
    f = feval(funJac,T(k),X(:,k),args{:});
    xinit = X(:,k) + f*h;
    [X(:,k+1),~] = NewtonsMethod(funJac, T(:,k), X(:,k), h, xinit, tol, maxit, args);
end
end
```

Listing 9: Implicit Euler method with fixed time step size

3.3. Implement an algorithm in Matlab for the implicit Euler method with adaptive time step and error estimation using step doubling.

The Implicit Euler with adaptive time step will also call Newton's method shown in Listing 8

```
function [T,X,r_out,h_out,info] = EulerImplicit_adaptive(funJac,tspan,h0,x0,abstol,reltol,args)

epstol = 0.8;
facmin = 0.1;
facmax = 5.0;

newtontol = 1.0e-8;
maxit = 100;

t0 = tspan(1);
tf = tspan(end);
t = t0;
h = h0;
x = x0;

T = t0;
X = x0;
r_out = [];
h_out = [h0];
info = zeros(1,4);

nfun = 0;
nstep = 0;
naccept = 0;

while t < tf
    if (t+h > tf)
        h = tf-t;
    end
    [f,J] = feval(funJac,t,x,args{:});
    nfun = nfun + 1;

    AcceptStep = false;
    while ~AcceptStep
        % Single step size
        xinit1 = x + h*f;
        [x1,nfun_local] = NewtonsMethod(funJac, t, x, h, xinit1, newtontol, maxit, args);
        nfun = nfun + nfun_local;

        hm = 0.5*h;
        tm = t + hm;
        xinitm = x + hm*f;
        [xm,nfun_local] = NewtonsMethod(funJac, t, x, hm, xinitm, newtontol, maxit, args);
        nfun = nfun + nfun_local;

        [fm,Jm] = feval(funJac,tm,xm,args{:});
        nfun = nfun + 1;
        xinit1hat = xm + hm*f;
        [x1hat,nfun_local] = NewtonsMethod(funJac, tm, xm, hm, xinit1hat, newtontol, maxit, args);
        nfun = nfun + nfun_local;

        nstep = nstep + 1;

        e = abs(x1hat-x1);
        r = max(e./max(abstol, abs(x1hat) .* reltol));
        AcceptStep = (r <= 1.0);

        if AcceptStep
            t = t+h;
            x = x1hat;

            T = [T,t];
            X = [X,x];
            r_out = [r_out, r];
            h_out = [h_out, h];
```

```
            naccept = naccept + 1;
        end

        h = max(facmin, min(sqrt(epstol/r), facmax)) * h;
    end
end

info(1) = nfun;
info(2) = nstep;
info(3) = naccept;
info(4) = nstep - naccept;

end
```

Listing 10: Implicit Euler method with fixed time step size

3.4. Test your algorithms on the Van der Pol problem ($\mu = 2$ and $\mu = 12$, $\mathbf{x_0} = [0.5; 0.5]$).

The way of calling the ODE solvers is similar as in Exercise 2.4. Results are shown in the following Figures and Tables. First thing we can notice when looking at the plot is that the Implicit, contrary to the Explicit, approximates the error from the inside of the steady-state curve: worst approximations of the solution make the frequency of the oscillation larger and the amplitude smaller.

We can also observe the stiffness of $\mu = 12$ is handled a lot better. Specially in Figure 19, while there's still some oscillations when x_2 shifts suddenly, it is considerably better than the Explicit. It's also worth noticing, looking at Tables 9 and 10, the huge increase in the number of function evaluations. This is obviously due to the added call to Newton's method to solve the non linear equation.

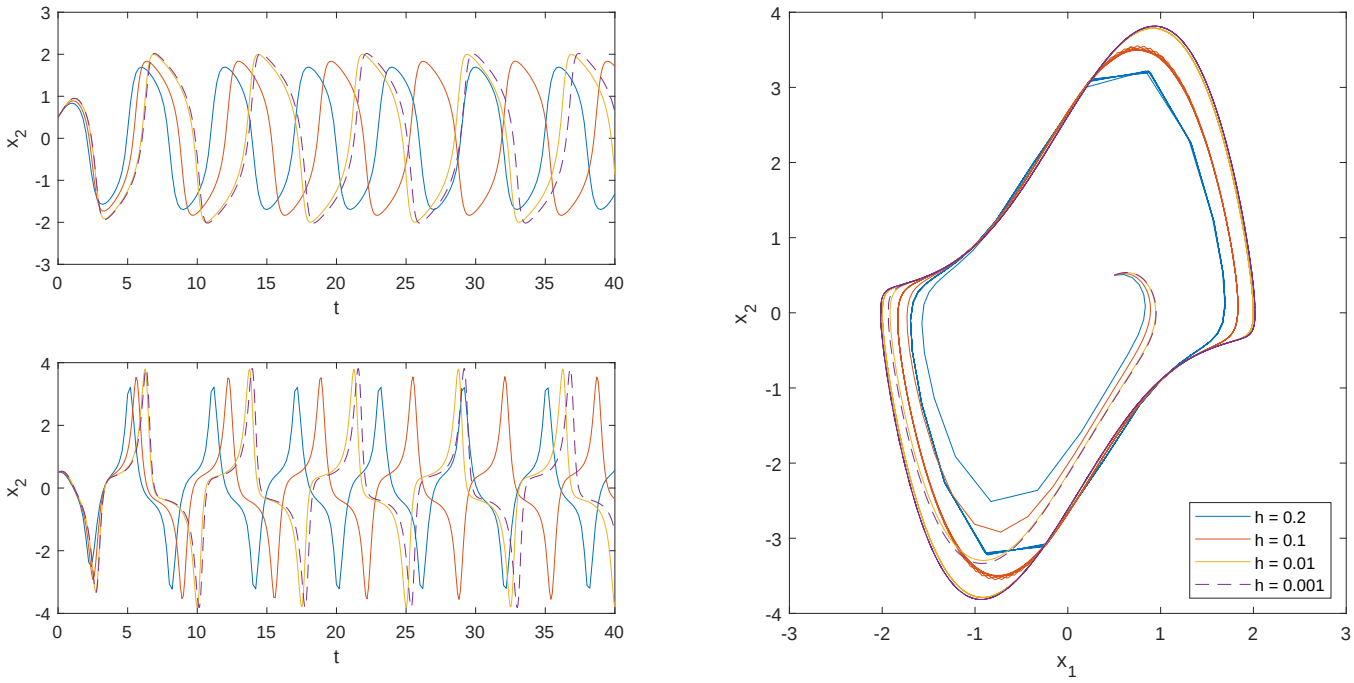


Figure 16: Solution for the Van der Pol problem ($\mu = 2$) using Implicit Euler with fixed step size

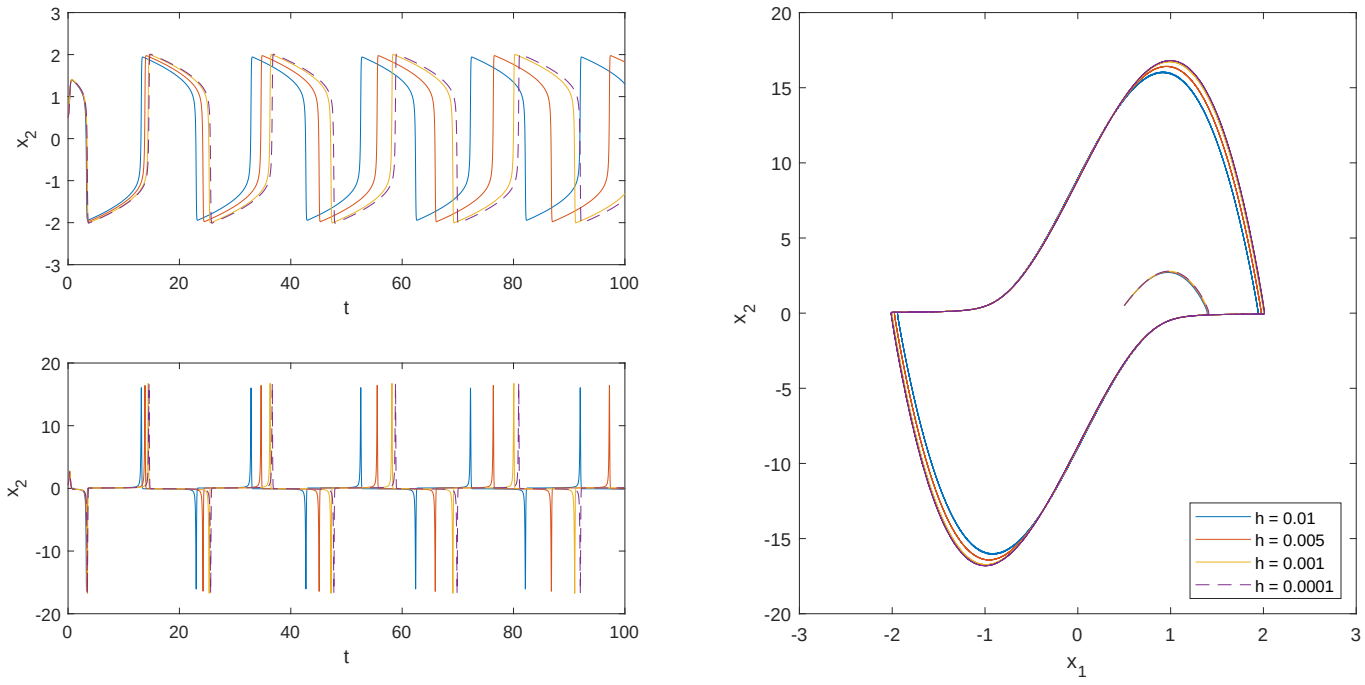


Figure 17: Solution for the Van der Pol problem ($\mu = 12$) using Implicit Euler with fixed step size

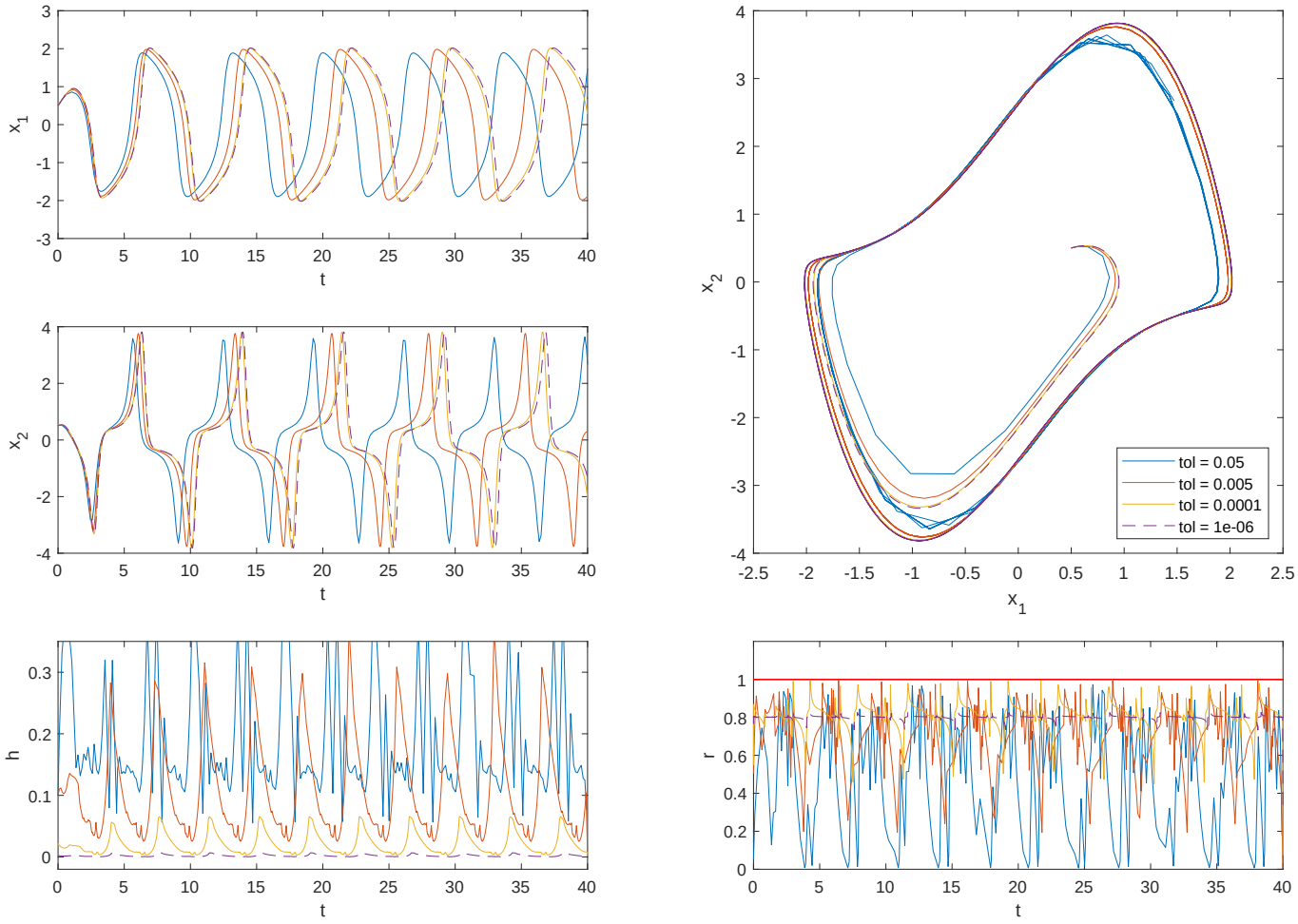


Figure 18: Solution for the Van der Pol problem ($\mu = 2$) using Implicit Euler with adaptive step size

Tolerances	0.05	0.005	0.0001	1e-06
Function evaluations	5732	8162	30764	3.0379e+05
Calculated steps	353	754	3799	37974
Accepted steps	237	589	3797	37972
Rejected steps	116	165	2	2

Table 9: Parameters of the Implicit Euler with adaptive step size for the Van der Pol problem ($\mu = 2$)

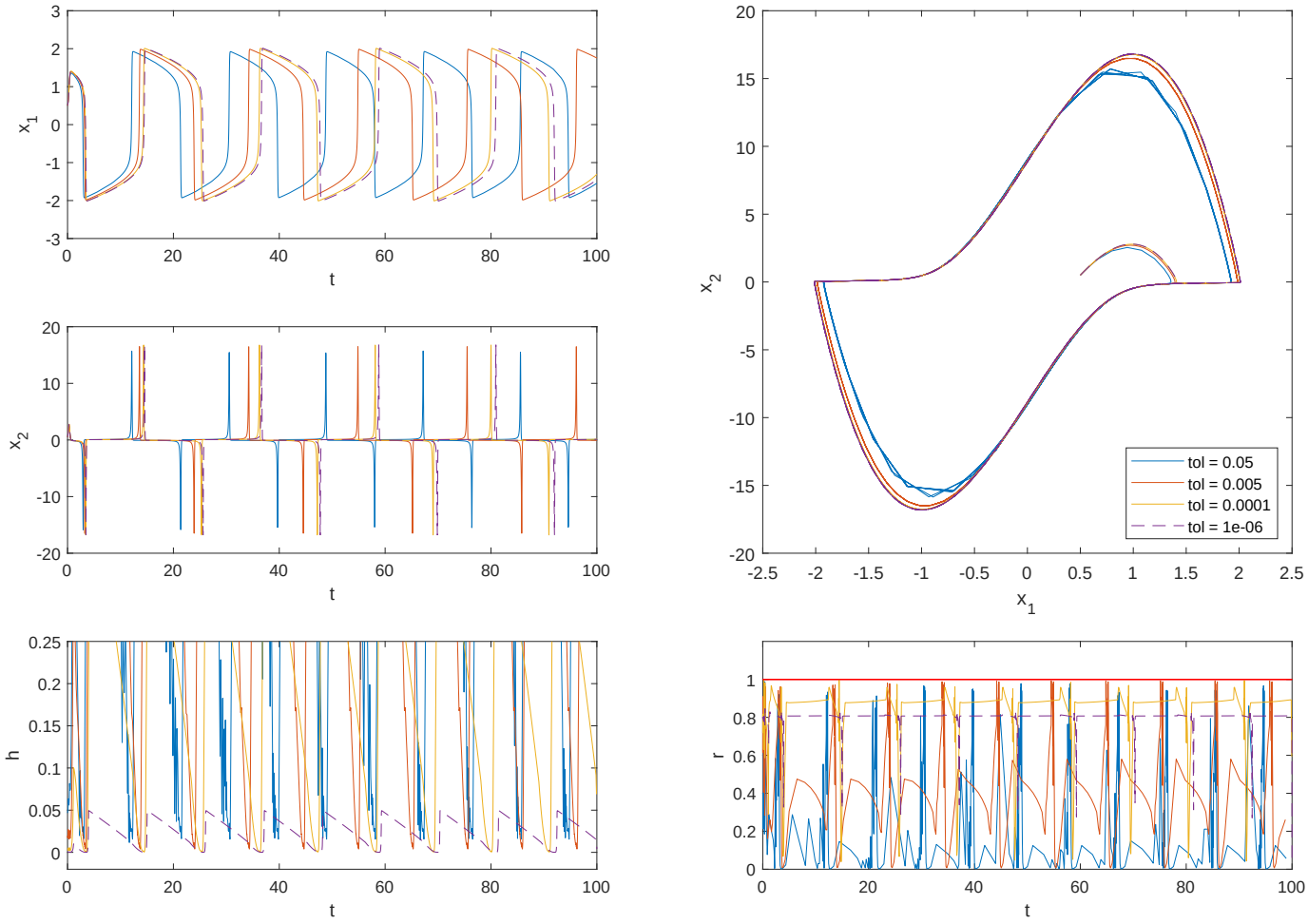


Figure 19: Solution for the Van der Pol problem ($\mu = 12$) using Implicit Euler with adaptive step size

Tolerances	0.05	0.005	0.0001	1e-06
Function evaluations	24307	14368	48124	4.6157e+05
Calculated steps	776	1288	5746	57695
Accepted steps	478	982	5739	57692
Rejected steps	298	306	7	3

Table 10: Parameters of the Implicit Euler with adaptive step size for the Van der Pol problem ($\mu = 12$)

3.5. Test your algorithms on the adiabatic CSTR problem described in the papers uploaded to Learn (3D-version and 1D-version).

Here are shown the results of the Implicit Euler for the CSTR problem. Just as in last exercise, we can't observe any significant difference between the 3D and 1D problem (Figure 20). The results when working with a fixed time step size (Figure 21) are the same for both results. We can observe how the implicit manages to control the *stiffness* of the problem when the values of F go lower, we can't find no longer the oscillations. The highest step size misses completely the values though. However, it converges quicker than the explicit to the real solution.

For the adaptive method, the value `facmax` was set to be 1.5, otherwise the asymptotic step controller was too sensible and diverged a lot. Here, we can see that the step size controller is working and the results converge quickly to the real solution. The 1D solution misses the larger values of T (when F is lower) for the largest tolerance.

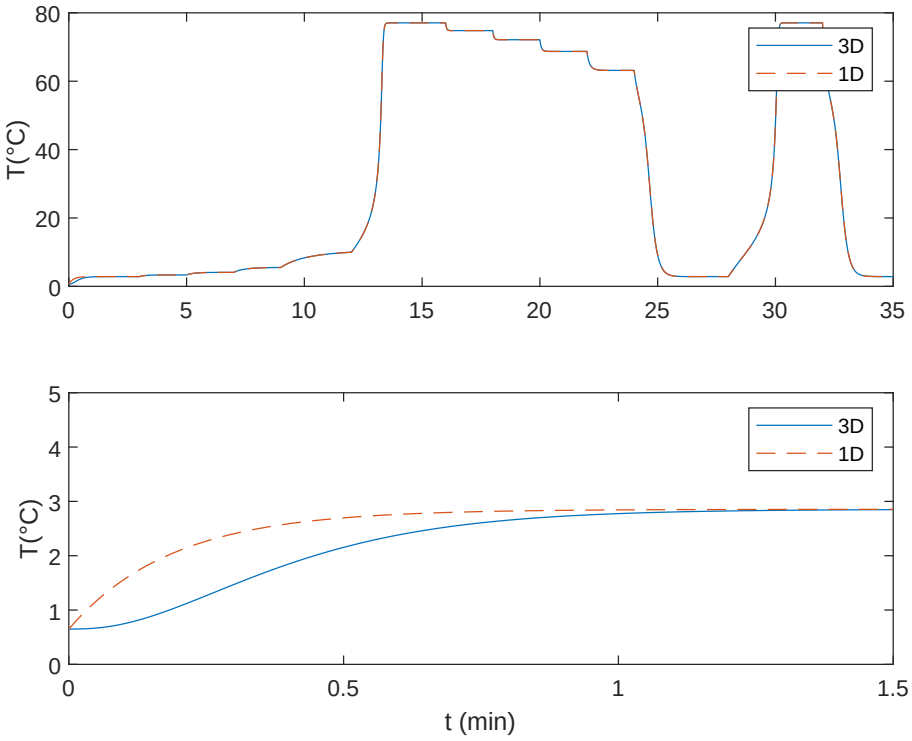
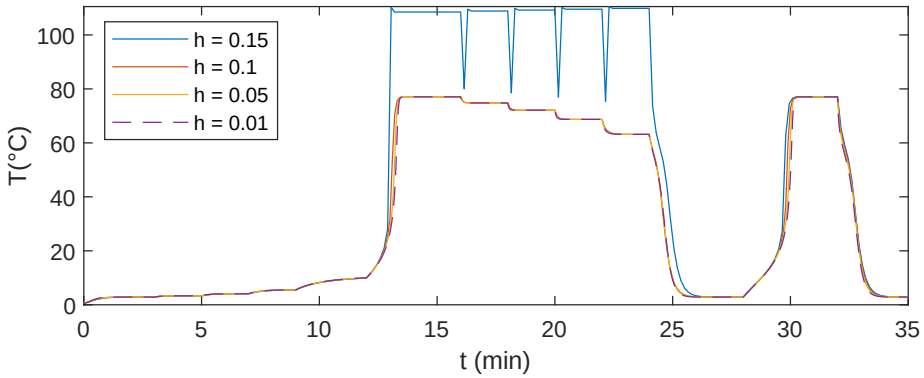
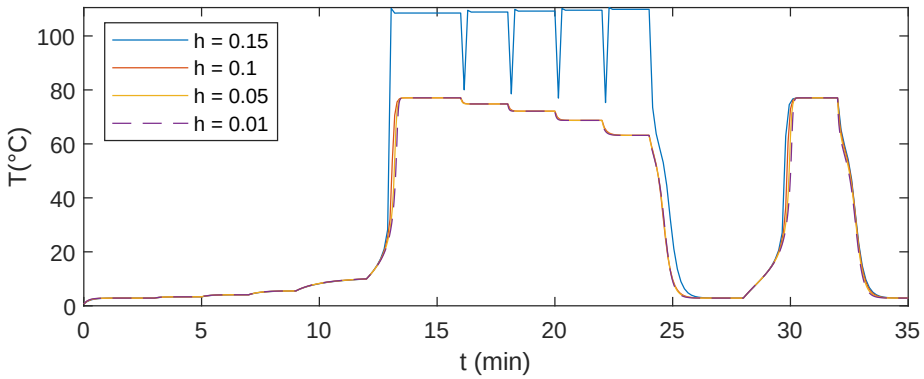


Figure 20: Comparison of the solutions for the CSTR 3D and 1D with Implicit Euler



(a) CSTR 3D problem



(b) CSTR 1D problem

Figure 21: Solution for the CSTR problem using Implicit Euler with fixed step size

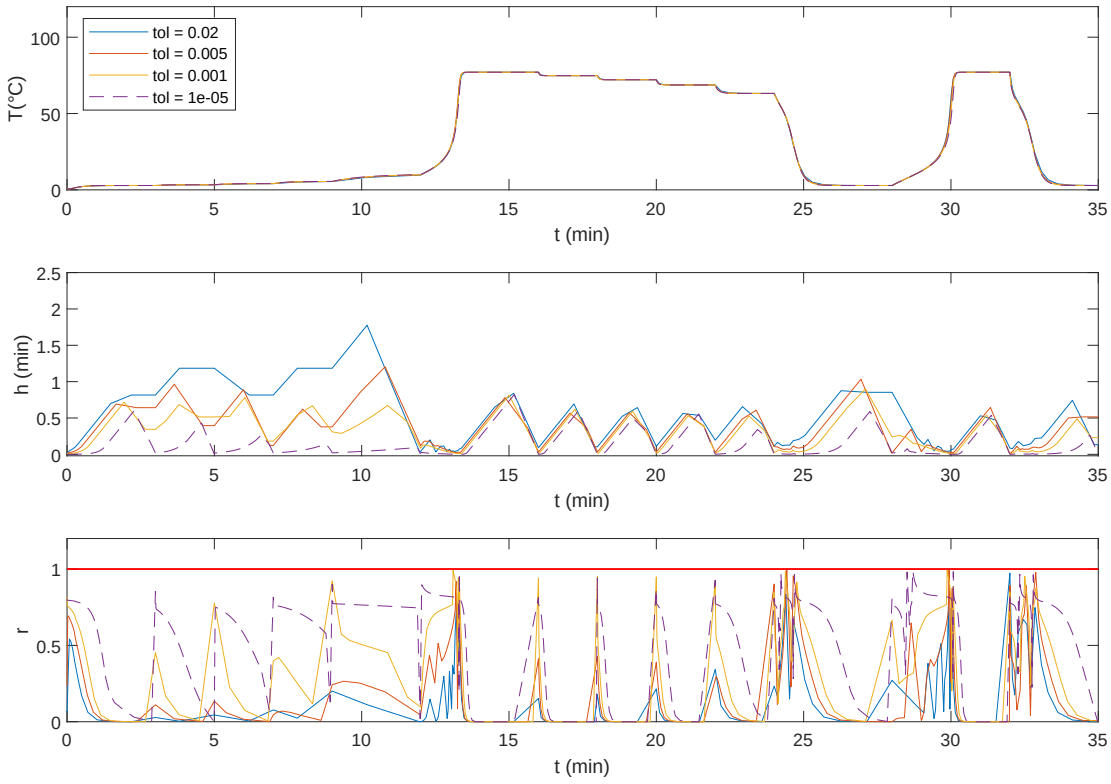


Figure 22: Solution for the CSTR 3D problem using Implicit Euler with adaptive step size

Tolerances	0.02	0.005	0.001	1e-05
Function evaluations	2386	3171	4325	21476
Calculated steps	146	232	388	2589
Accepted steps	121	188	307	2537
Rejected steps	25	44	81	52

Table 11: Parameters of the Implicit Euler with adaptive step size for the CSTR 3D problem

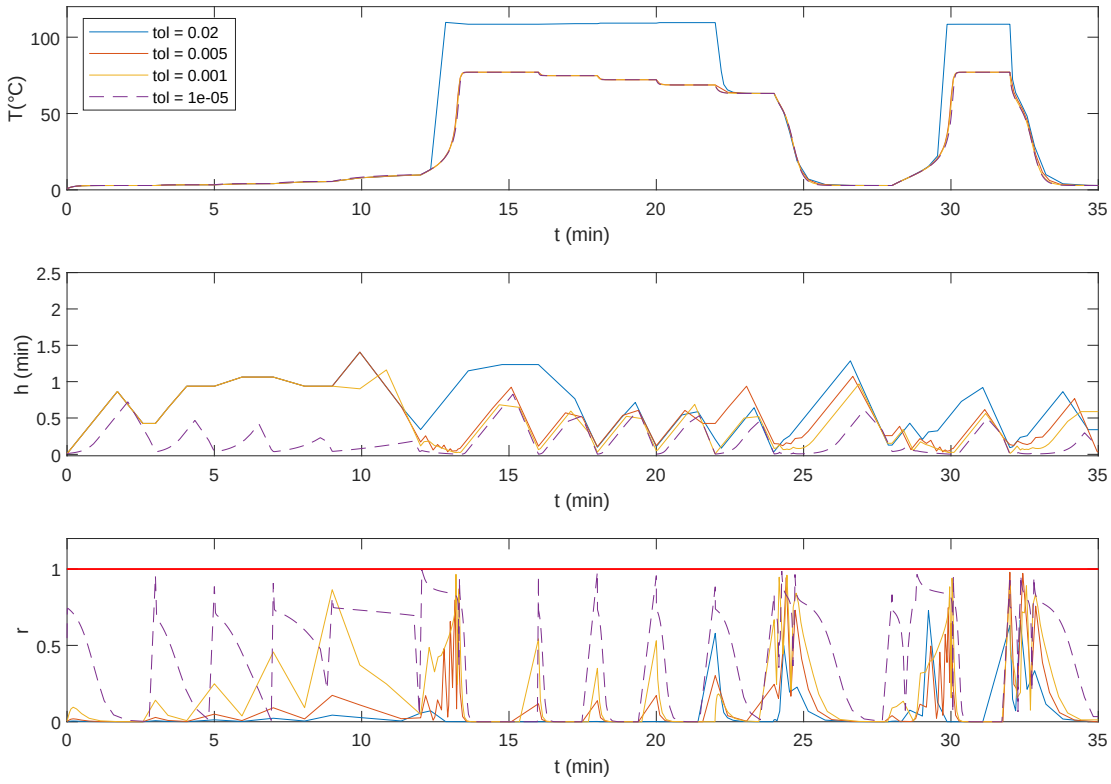


Figure 23: Solution for the CSTR 1D problem using Implicit Euler with adaptive step size

Tolerances	0.02	0.005	0.001	1e-05
Function evaluations	2311	2592	3144	11390
Calculated steps	88	141	222	1126
Accepted steps	80	115	173	1073
Rejected steps	8	26	49	53

Table 12: Parameters of the Implicit Euler with adaptive step size for the CSTR 1D problem

3.6. Compare the results from your algorithms with the results you get using some of Matlab's ODE solvers The report should contain figures and a discussion of your algorithm for different tolerances and step sizes.

For the Van der Pol problem, we tested Implicit Euler against `ode45` for the non-stiff case ($\mu = 1.5$), and against `ode15s` for the stiff case ($\mu = 15$). Contrary to what we obtained using explicit Euler, the Implicit Euler works fine in the stiff case, even though the `ode15s` achieves better accuracy with a fewer no. of function evaluations. This is also due to the fact that we're using step doubling to approximate the error. In the non-stiff case, however, the no. of evaluations is ridiculously high compared to the `ode45` , which makes sense given that it's an implicit method vs. an explicit one in a non-stiff problem.

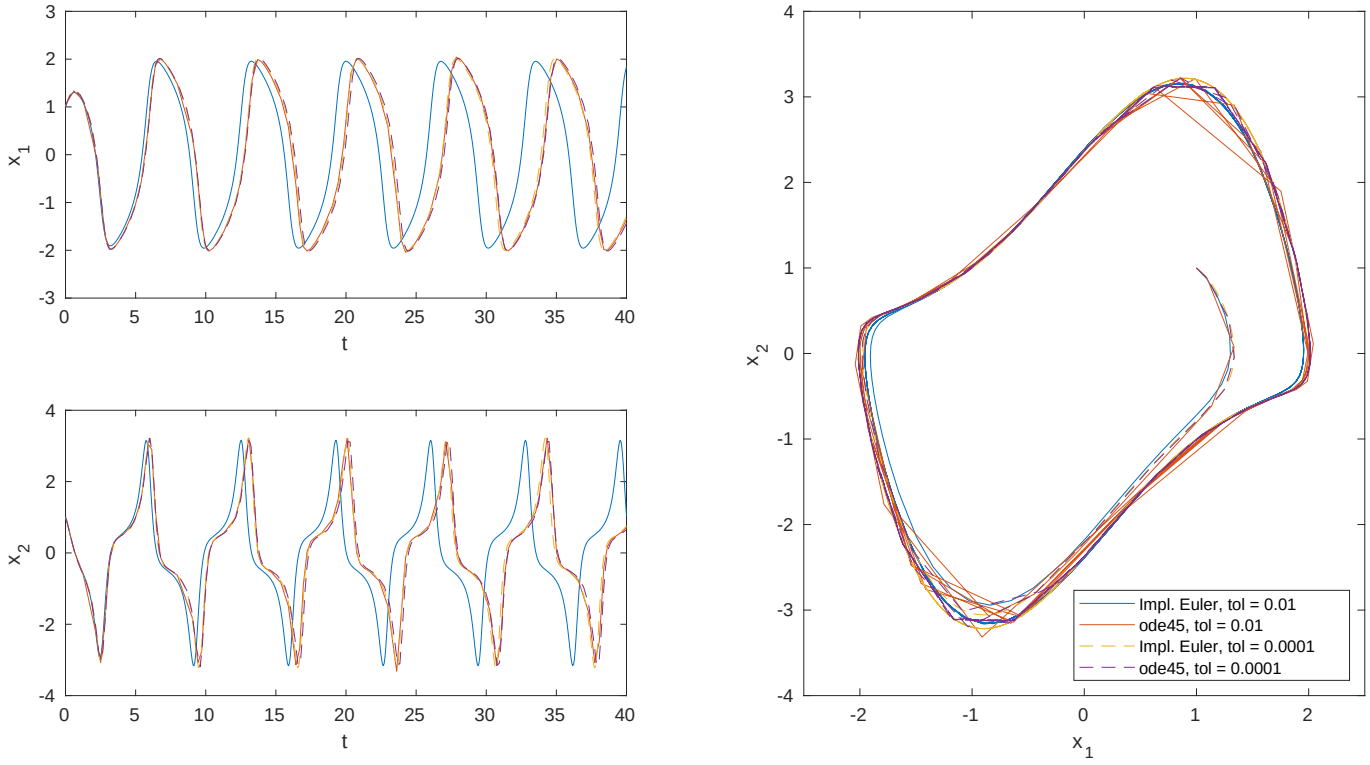


Figure 24: Solution for the Van der Pol problem ($\mu = 1.5$) using Implicit Euler vs. `ode45`

Method	**Impl. Euler**		**ode45**	
Tolerances	0.01	0.0001	0.01	0.0001
Function evaluations	5713	29600	787	1357
Calculated steps	522	3666	131	226
Accepted steps	402	3664	100	180
Rejected steps	120	2	31	46

Table 13: Parameters of the Implicit Euler vs. `ode45` *for the Van der Pol problem ($\mu = 1.5$)*

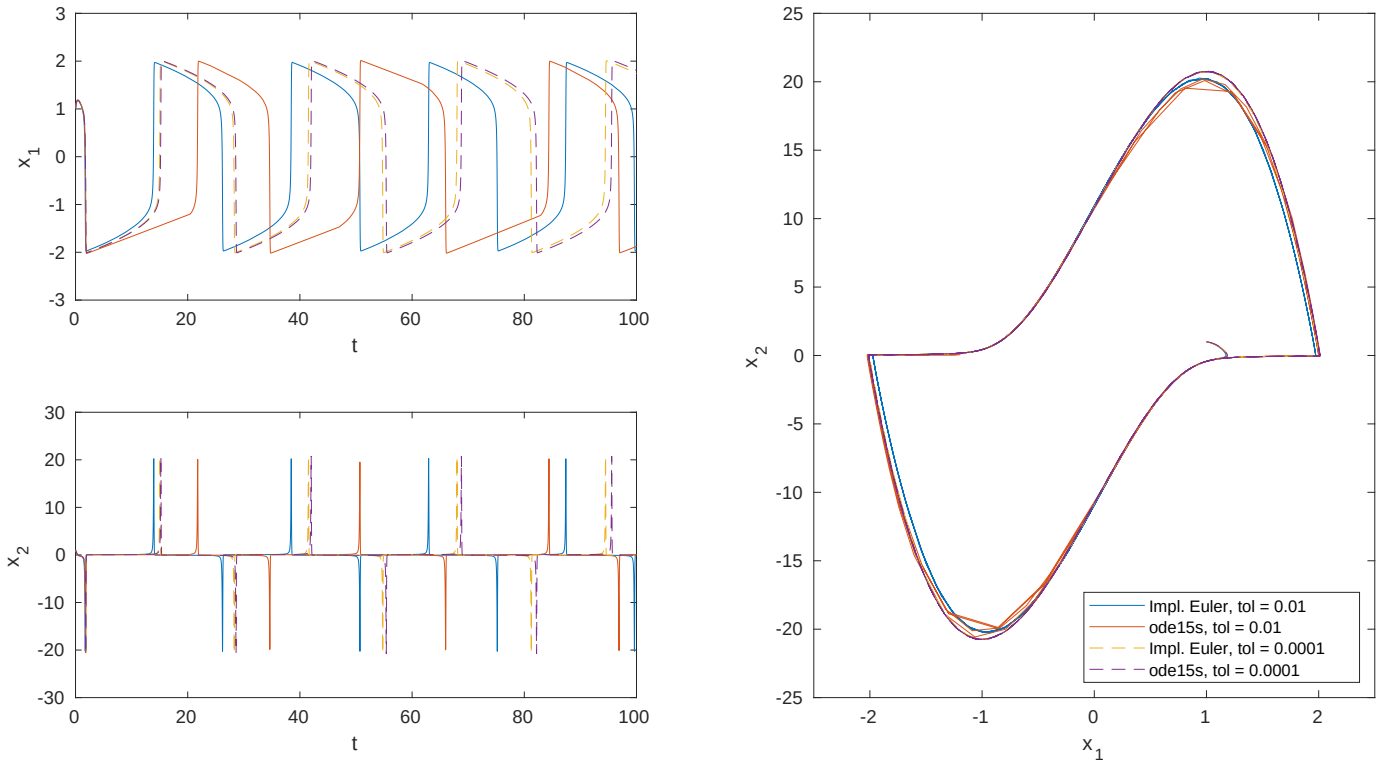
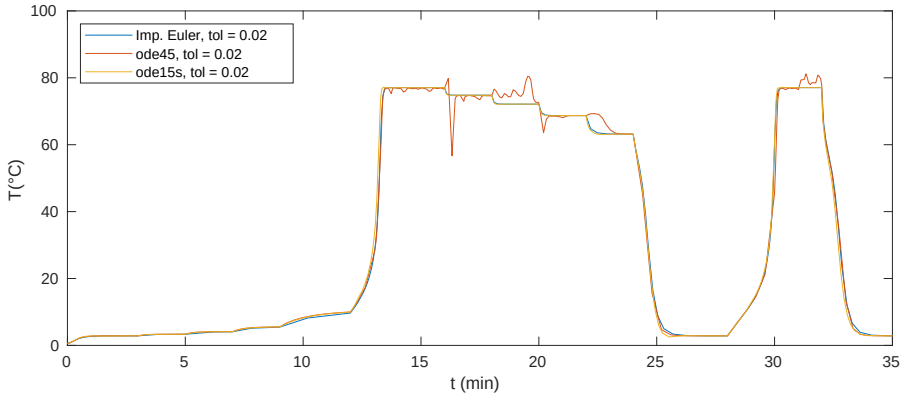


Figure 25: Solution for the Van der Pol problem ($\mu = 15$) using Implicit Euler vs. `ode15s`

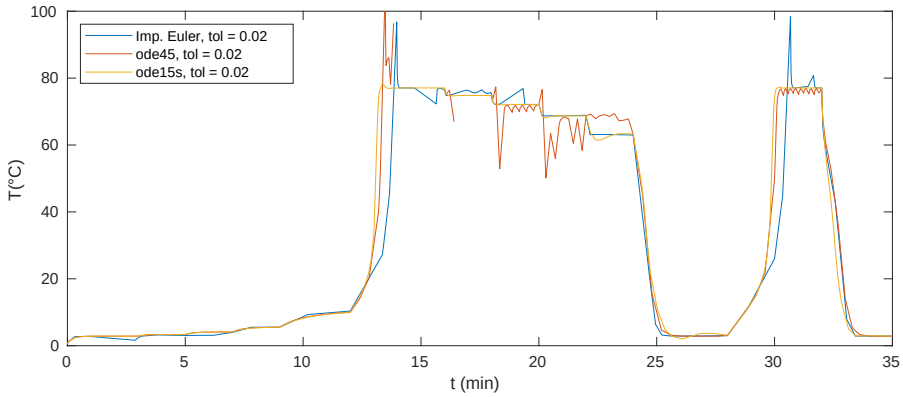
Method	**Impl. Euler**		**ode15s**	
Tolerances	0.01	0.0001	0.01	0.0001
Function evaluations	12508	43567	1273	2780
Calculated steps	937	5198	558	1327
Accepted steps	743	5189	411	1094
Rejected steps	120	2	147	233

Table 14: Parameters of the Implicit Euler vs. `ode15s` *for the Van der Pol problem ($\mu = 15$)*

For the CSTR problem, we also see how both the Implicit Euler and `ode15s` achieve good performance for the 3D case. For the 1D case we see the Implicit deviating a bit, but this could be caused by the tolerance being too wide. The `ode45` achieves worse performance in both cases.



(a) CSTR 3D problem



(b) CSTR 1D problem

Figure 26: Solution for the CSTR problem using Implicit Euler vs. `ode45` *and* `ode15s`

Method	**Impl. Euler**	**ode45**	**ode15s**
Tolerances	0.02	0.02	0.02
Function evaluations	2386	1513	477
Calculated steps	146	1586	1408
Accepted steps	121	220	201
Rejected steps	25	30	29

Table 15: Parameters of the Implicit Euler vs. `ode45` *and* `ode15s` *for the CSTR-3D problem*

Method	**Impl. Euler**	**ode45**	**ode15s**
Tolerances	0.02	0.02	0.02
Function evaluations	201	1297	375
Calculated steps	117	1270	1275
Accepted steps	84	195	180
Rejected steps	33	19	27

Table 16: Parameters of the Implicit Euler vs. `ode45` *and* `ode15s` *for the CSTR-1D problem*

3.7. Discuss when one should use the implicit Euler method rather than the explicit Euler method and illustrate that using an example.

As previously discussed, the Implicit Euler method is a lot more robust than the Explicit, this is, it handles *stiff problems* a lot better. The reason behind this behaviour is connected to the *region of stability* of the method. We showed the region of stability for the Explicit and Implicit Euler in Figure 3. The problem can require an extremely low h for the Explicit Method in order to satisfy $|y_{n-1}| \leq |y_n|$. The implicit, being A-stable, doesn't have this problem.

If we solve the Van der Pol problem for $\mu = 100$ with Explicit and Implicit method with adaptive step we will observe this behaviour. Both will be solved with a tolerance of 10^{-3}. Just by looking at the h vs. time and r vs. time we can notice the difference. Notice that the time step with the Explicit must be really small even though the solution is really smooth. Otherwise, it would explode. The Implicit manages the stiffness a lot better, getting to values of time step of more than 8 units of time. The solution is equally valid in both of them. Looking at table 17, we notice that the number of steps calculated is more than 10 times smaller on the implicit. In this case, the Implicit proves to be computationally better, even though we are calling the Newton method at every time step.

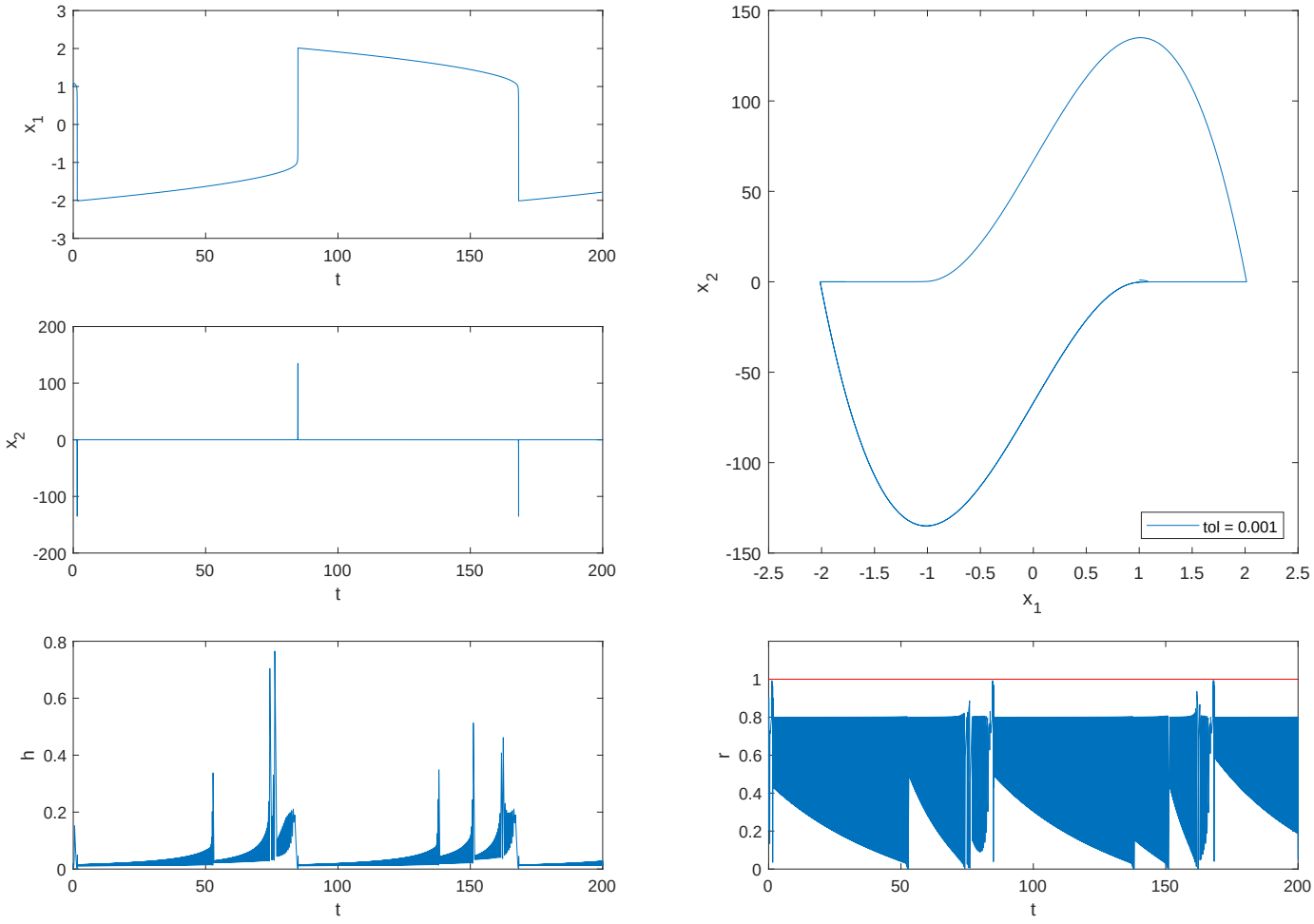


Figure 27: Solution for the Van der Pol problem ($\mu = 100$) using Explicit Euler with adaptive step size

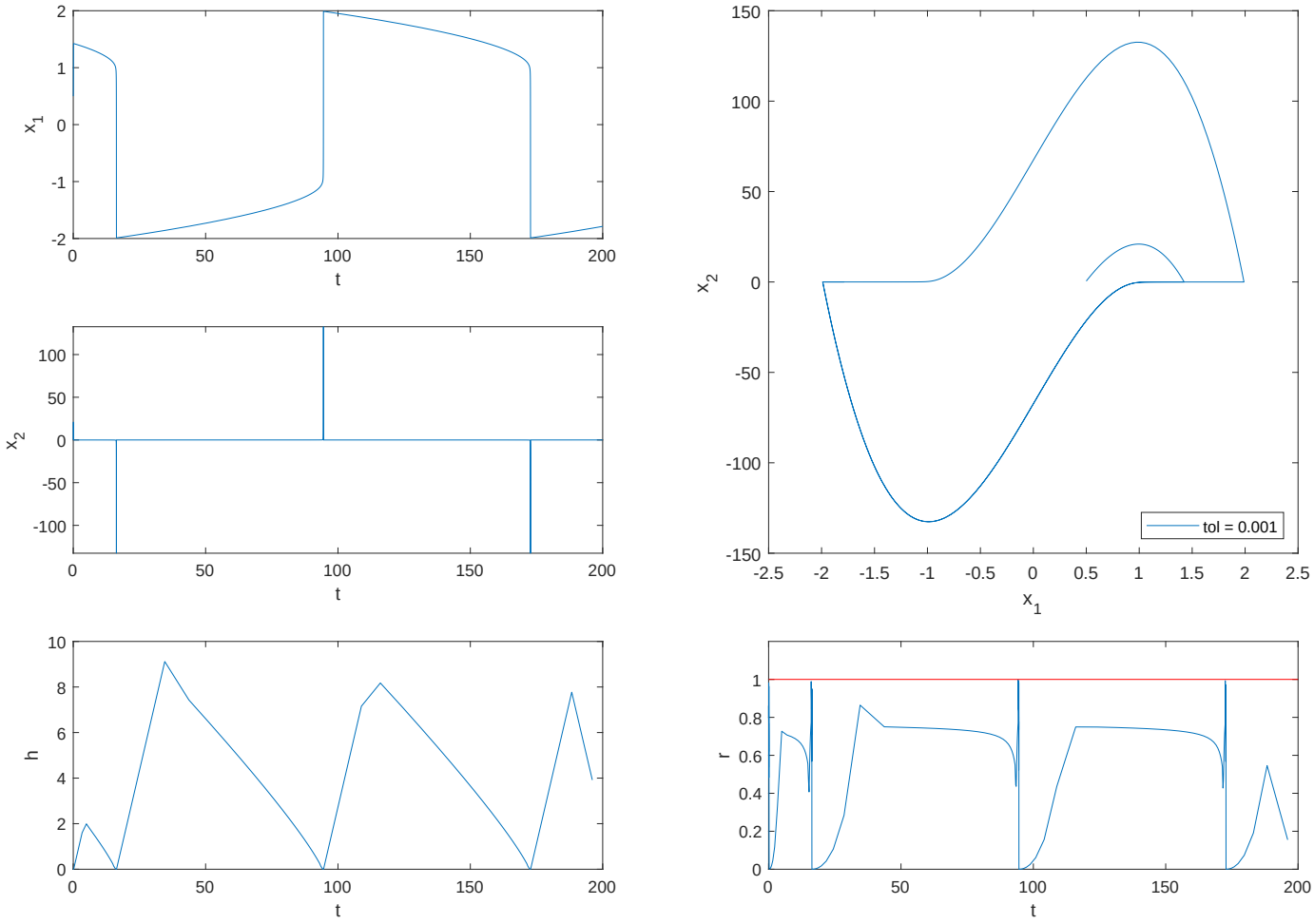


Figure 28: Solution for the Van der Pol problem ($\mu = 100$) using Implicit Euler with adaptive step size

Tolerances	Explicit	Implicit
Function evaluations	22654	12407
Calculated steps	13450	1207
Accepted steps	9204	1076
Rejected steps	4246	131

Table 17: Parameters of the Explicit and Implicit methods for the Van der Pol problem ($\mu = 100$)

Part 4: Solvers for SDEs

We consider now stochastic differential equations in the form

$$dx(t) = f(t, x(t), p_f)dt + g(t, x(t), p_g)d\omega(t), \quad d\omega(t) \sim \mathcal{N}_{iid}(0, Idt) \tag{4}$$

where $x \in \mathbb{R}^{n_x}$ and ω is a stochastic variable with dimension n_w, p_f and p_g are parameters for $f : \mathbb{R} \times \mathbb{R}^{n_x} \times \mathbb{R}^{n_{p_f}} \to \mathbb{R}^{n_x}$ and $g : \mathbb{R} \times \mathbb{R}^{n_x} \times \mathbb{R}^{n_{p_g}} \to \mathbb{R}^{n_x \times n_w}$ (i.e. the result of g is a matrix of size $n_x \times n_w$).

4.1. Make a function in Matlab that can realize a multivariate standard Wiener process.

We implement the Wiener process in the following function. This generates `Ns` realizations of a Wiener process of size `nW`.

```
function [Tw,W,dW] = StdWienerProcess(tspan,h,nW,Ns,seed)

if nargin == 4
    rng(seed);
end

t0 = tspan(1);
tf = tspan(2);

Tw = t0:h:tf;
N = size(Tw,2);

dW = sqrt(h)*randn(nW,N,Ns);
W  = [zeros(nW,1,Ns) cumsum(dW,2)];
W = W(:,1:end-1,:);
end
```

Listing 11: Multivariate standard Wiener process

4.2. Implement the explicit-explicit method with fixed step size.

```
function [T,X] = EulerMaruyama(ffun,gfun,tspan,h,x0,nW,args)
t0 = tspan(1);
tf = tspan(end);
T = t0:h:tf;
N  = size(T,2);

X  = zeros(size(x0,1),N);
X(:,1) = x0;

[~,~,dW] = StdWienerProcess(tspan,h,nW,1);

for k=1:N-1
    f = feval(ffun,T(k),X(:,k),args{:});
    g = feval(gfun,T(k),X(:,k),args{:});

    psi = X(:,k) + g*dW(:,k);

    X(:,k+1) = psi + h*f;
end
```

Listing 12: Euler-Maruyama Method

4.3. Implement the implicit-explicit method with fixed step size.

```
function [x,f,J] = NewtonsMethodSDE(ffun,t,h,psi,xinit,tol,maxit,args)
I = eye(length(xinit));
x = xinit;

[f,J] = feval(ffun,t,x,args{:});
R = x - h*f - psi;
it = 1;

while ( (norm(R,'inf')>tol) &&(it<= maxit) )
    dRdx = I - J*h;
    mdx = dRdx\R;

    x   = x - mdx;
    [f,J] = feval(ffun,t,x,args{:});
    R = x - h*f - psi;
    it = it + 1;
end
end
```

Listing 13: Newton's Method for SDEs

```
function [T,X] = ImplicitExplicit(ffun,gfun,tspan,h,x0,nW,args)

newtontol = 1e-8;
maxit = 100;

t0 = tspan(1);
tf = tspan(end);
T = t0:h:tf;
N  = size(T,2);

X  = zeros(size(x0,1),N);
X(:,1) = x0;

[~,~,dW] = StdWienerProcess(tspan,h,nW,1);
k = 1;

f = feval(ffun,T(k),X(:,k),args{:});
for k=1:N-1
    g = feval(gfun,T(k),X(:,k),args{:});

    psi = X(:,k) + g*dW(:,k);

    xinit = psi + h*f;
    [X(:,k+1),f,~] = NewtonsMethodSDE(ffun,T(:,k+1),h,psi,xinit,newtontol,maxit,args);
end
end
```

Listing 14: Implicit-Explicit Method

4.4. Describe your implementations and the idea you use in deriving the methods. Provide Matlab code for your implementations.

The idea behind the Euler-Maruyama and Implicit-Explicit is the same as in the Explicit and Implicit Euler already described in Parts 2 and 3, for the function f. The problem is that now we also have a function g associated to the Wiener process and, as in order to calculate it we need to discretize it, we just use the Explicit Euler for g both in Euler-Maruyama and Implicit-Explicit. Notice that the Newton's method for SDEs is slightly different to the one already presented in Listing 8. The Matlab code is already provided in the last two questions.

4.5. Test your implementations using SDE versions of the Van der Pol problem.

We will test the implementations in two SDE versions of the Van der Pol problem. One with *state independent diffusion*:

$$\begin{aligned} dx_1(t) &= x_2(t)dt, \\ dx_2(t) &= \left[\mu(1 - x_1(t)^2)x_2(t) - x_1(t)\right] dt + \sigma d\omega(t), \end{aligned}$$

and another one with *state dependent diffusion*:

$$\begin{aligned} dx_1(t) &= x_2(t)dt, \\ dx_2(t) &= \left[\mu(1 - x_1(t)^2)x_2(t) - x_1(t)\right] dt + \sigma(1 + x_1(t)^2)d\omega(t). \end{aligned}$$

We implement f and the two gs in the following scripts:

```
function [f,J] = VanderPolDrift(t,x,mu,sigma)
f = zeros(2, 1);

f(1) = x(2);
f(2) = mu * (1 - x(1)^2)*x(2) - x(1);

if nargout > 1
    J = zeros(2, 2);

    J(1,2) = 1;
    J(2,1) = -2 * mu * x(1) * x(2) - 1;
    J(2,2) = mu * (1-x(1)^2);
end
end

function g = VanderPolDiffusion1(t,x,mu,sigma)
g = [0; sigma];
end

function g = VanderPolDiffusion2(t,x,mu,sigma)
g = [0; sigma*(1.0+x(1)^2)];
end
```

In the following plots we show 5 realisations of the Euler-Maruyama and the Implicit-Explicit methods for the state independent diffusion case ($\sigma = 1$) and for the dependent ($\sigma = 0.5$). The runs were obtained setting $\mu = 3$, $h = 0.1$, `tspan` $= [0, 40]$ and $x_0 = [0.5; 0.5]$.

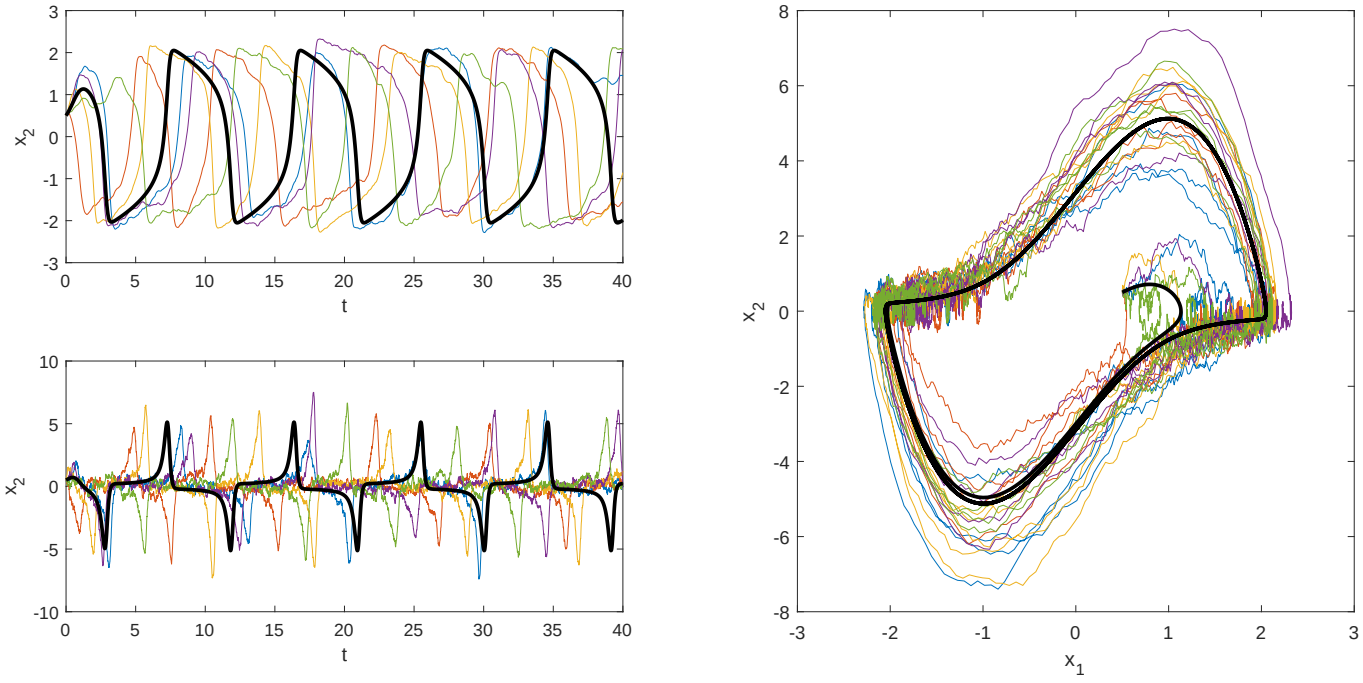


Figure 29: 5 realisations of the solution for the Van der Pol SDE problem with state independent diffusion ($\mu = 3, \sigma = 1$) using Euler-Maruyama with fixed step size

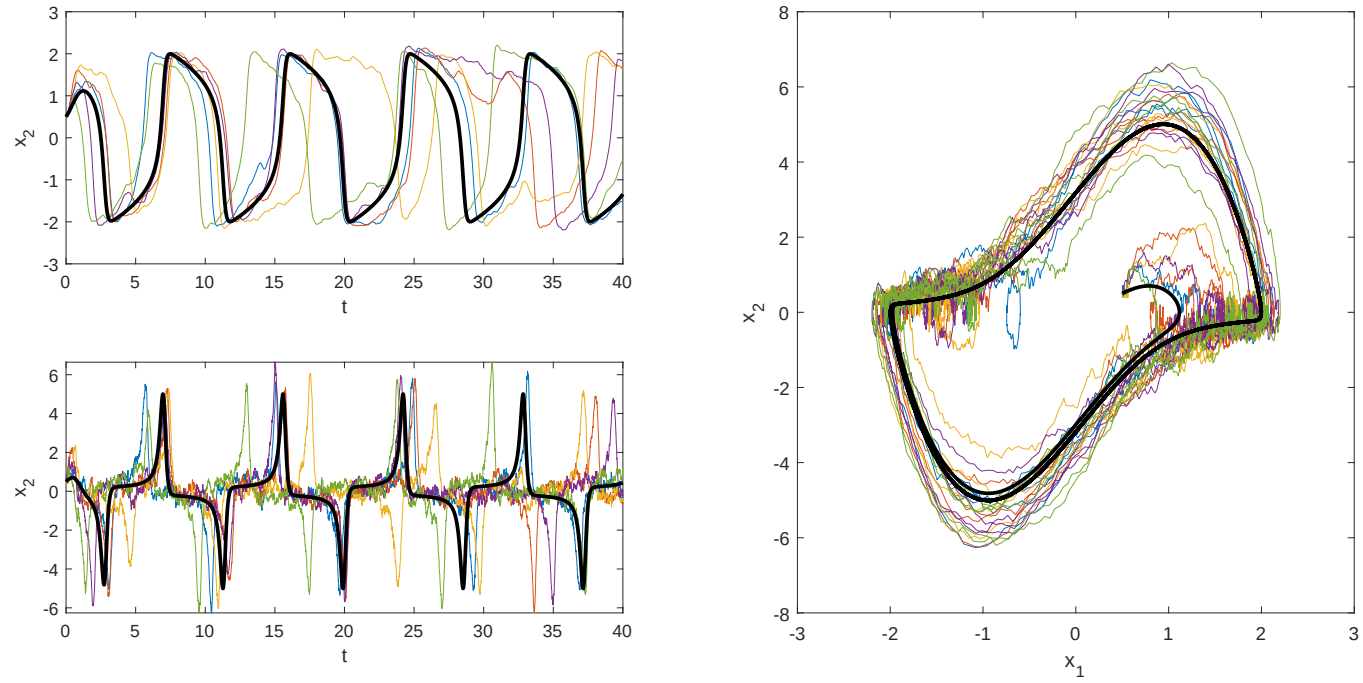


Figure 30: 5 realisations of the solution for the Van der Pol SDE problem with state independent diffusion ($\mu = 3, \sigma = 1$) using Implicit-Explicit with fixed step size

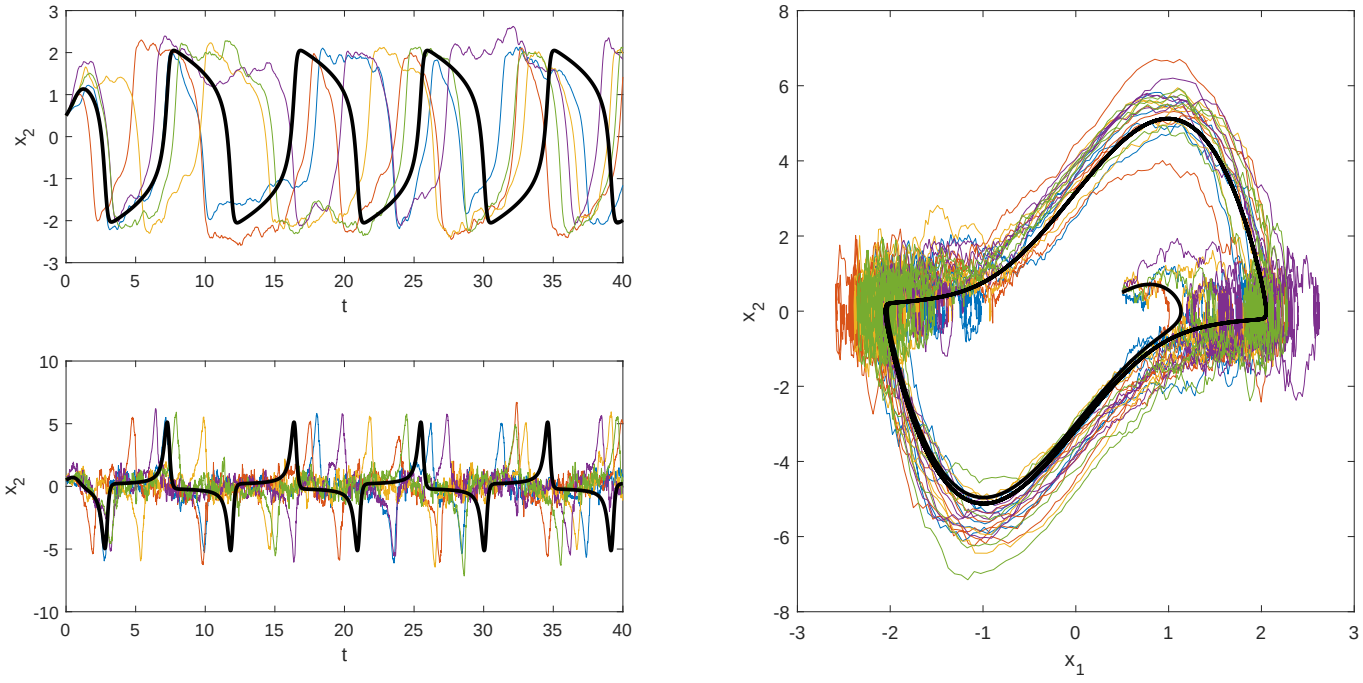


Figure 31: 5 realisations of the solution for the Van der Pol SDE problem with state dependent diffusion ($\mu = 3, \sigma = 0.5$) using Euler-Maruyama with fixed step size

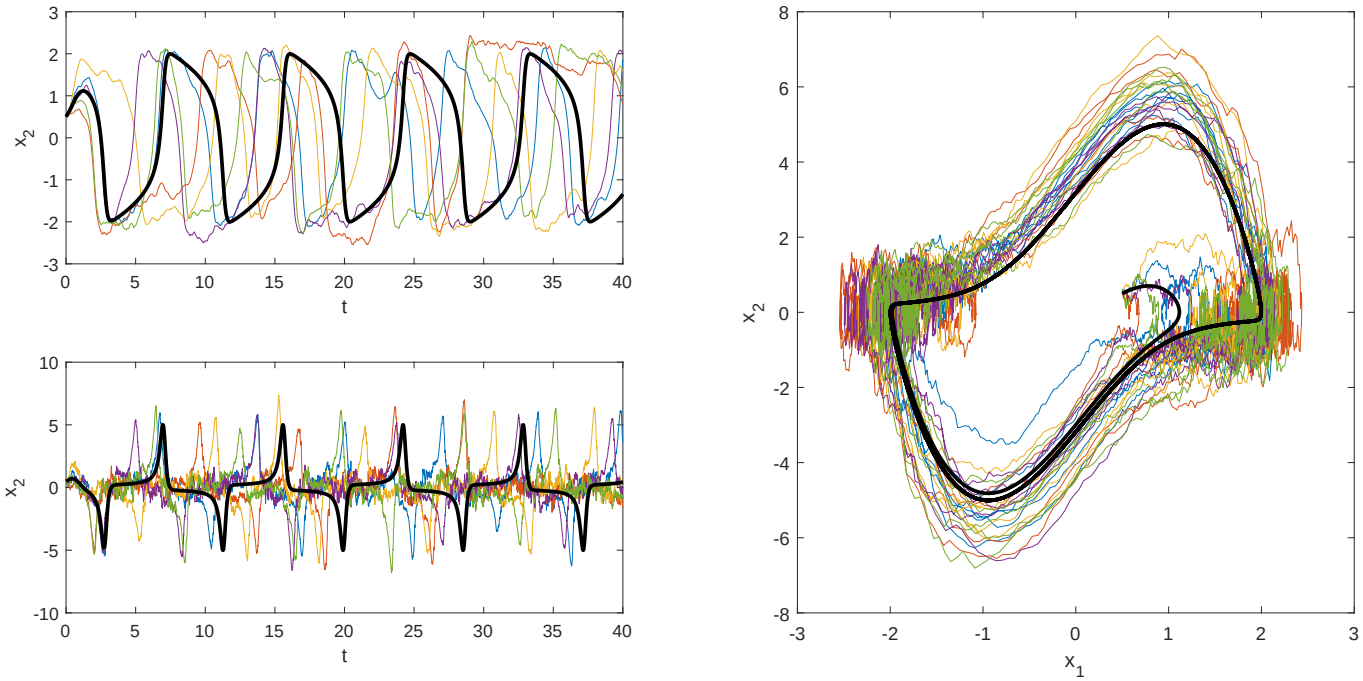
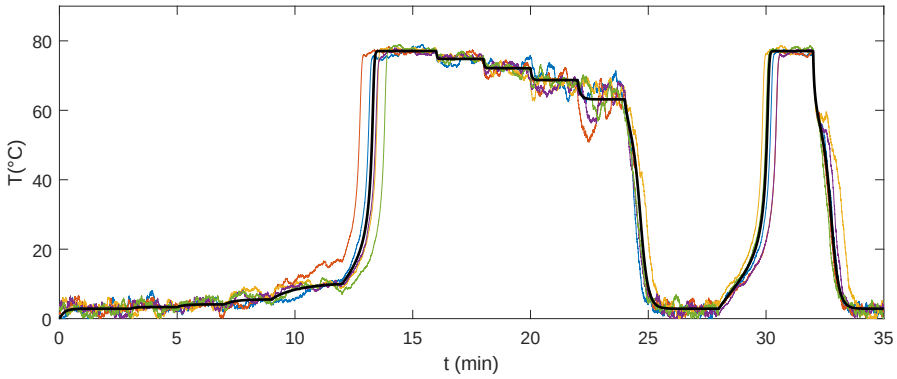


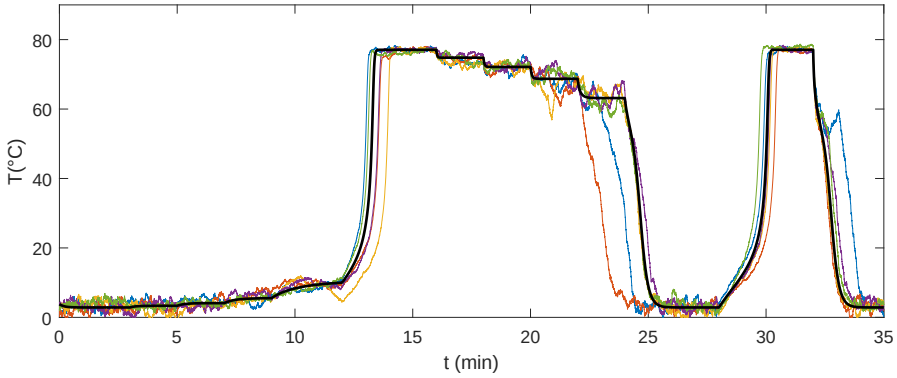
Figure 32: 5 realisations of the solution for the Van der Pol SDE problem with state dependent diffusion ($\mu = 3, \sigma = 0.5$) using Implicit-Explicit with fixed step size

4.6. Test your algorithms on the adiabatic CSTR problem described in the papers uploaded to Learn (3D-version and 1D-version).

For the CSTR problem, we'll also compute 5 realisations of the Euler-Maruyama and the Implicit-Explicit methods for the 3D and the 1D case. These runs are obtained setting all the parameters as in [4] and [5]. σ was set to 0.5 for all runs. It's worth noticing how the noise decreases for the 1D problem even though they are run with the same σ.

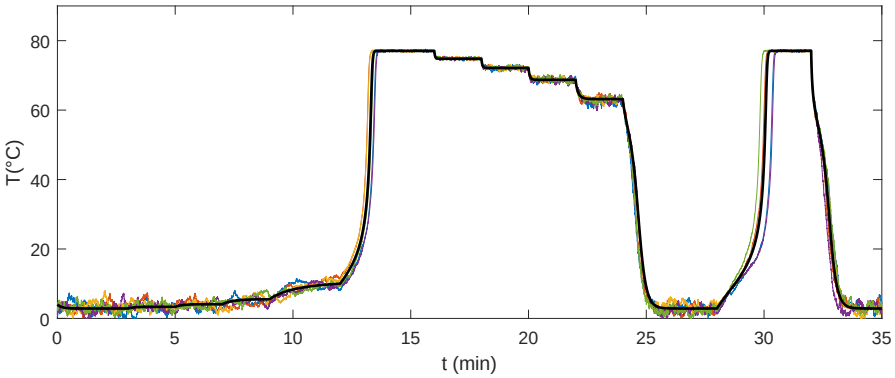


(a) Euler-Maruyama method

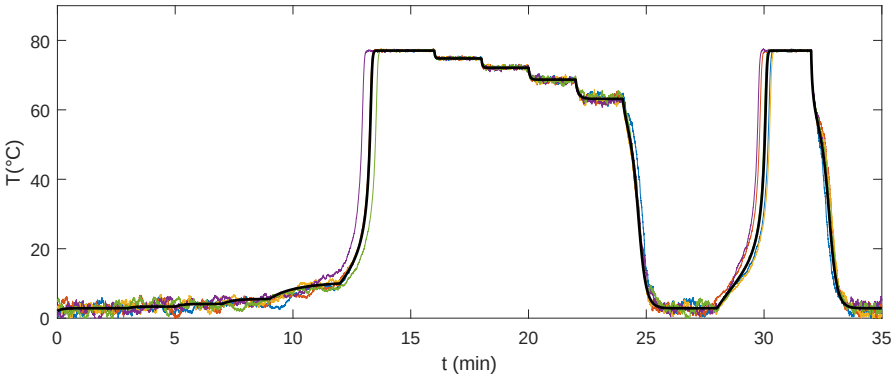


(b) Implicit-Explicit method

Figure 33: 5 realisations of the solution for the 3D CSTR SDE problem ($\sigma = 5$) using fixed step size



(a) Euler-Maruyama method



(b) Implicit-Explicit method

Figure 34: 5 realisations of the solution for the 1D CSTR SDE problem ($\sigma = 5$) using fixed step size

Part 5: Classical Runge-Kutta method with fixed time step size

We consider again the initial value problem in (3).

5.1. Describe the classical Runge-Kutta method with fixed step size

The classical Runge-Kutta is an iterative method used to calculate numerical approximations to the solution of initial value problems. It works by taking a time step size h, where we will perform the following calculations:

$$
\begin{aligned}
T_1 &= t_n & X_1 &= x_n \\
T_2 &= t_n + \frac{1}{2}h & X_2 &= x_n + h\frac{1}{2}f(T_1, X_1) \\
T_3 &= t_n + \frac{1}{2}h & X_3 &= x_n + h\frac{1}{2}f(T_2, X_2) \\
T_4 &= t_n + h & X_4 &= x_n + hf(T_3, X_3)
\end{aligned}
$$

Once this is calculated, we can get the actual step that we will store as part of our solution as follows:

$$
\begin{aligned}
t_{n+1} &= t_n + h \\
x_{n+1} &= x_n + h\left(\frac{1}{6}f(T_1, X_1) + \frac{1}{3}f(T_2, X_2) + \frac{1}{3}f(T_3, X_3) + \frac{1}{6}f(T_4, X_4)\right)
\end{aligned}
$$

Note that the previous 8 calculations must be performed in every single step. To solve the initial value problem we just start from x_0, pick a step size h, and iterate over the algorithm.

0				
1/2	1/2			
1/2	0	1/2		
1	0	0	1	
x	1/6	1/3	1/3	1/6

Table 18: Butcher Tableau of the classical Runge-Kutta method

5.2. Implement an algorithm in Matlab for the classical Runge-Kutta method with fixed time step size. Provide the code in your report. Use a format that enables syntax highlighting. Comment on the code

We implement the classical Runge-Kutta method with fixed time step size with the following code:

```
function [T, X] = ClassicRK4_fixed(fun,tspan,h,x0,args)
% time interval
t0 = tspan(1);
tf = tspan(end);
T = t0:h:tf;
N = size(T,2);
X = zeros(size(x0,1), N);
X(:,1) = x0;

for k = 1:N-1
    t = T(k);
    x = X(:,k);

    % Stage 1
    T1 = t;
    X1 = x;
    F1 = feval(fun,T1,X1,args{:});
    % Stage 2
    T2 = t + h/2;
```

```
20      X2 = x + h/2*F1;
21      F2 = feval(fun,T2,X2,args{:});
22      % Stage 3
23      T3 = T2;
24      X3 = x + h/2*F2;
25      F3 = feval(fun,T3,X3,args{:});
26      % Stage 4
27      T4 = t + h;
28      X4 = x + h*F3;
29      F4 = feval(fun,T4,X4,args{:});
30
31      % final solution
32      X(:,k+1) = x + h*(1/6*F1+1/3*F2+1/3*F3+1/6*F4);
33
34  end
35  end
```

Listing 15: Classical Runge-Kutta method with fixed time step size

This implementation uses the following input arguments:

- `fun` is a pointer to a function where our IVP function f is defined.
- `tspan` is a vector of two values stating the initial and final time of the simulation.
- `x0` is the initial condition, i.e. the value of $x(t_0)$.
- `h` is the fixed time step size we want for our simulation.
- `args` is an array that includes constants needed to calculate the IVP function (μ in the Van der Pol problem, λ in the test equation, all the parameters in the CSTR problem).

5.3. Test your problem for test equation. Discuss order and stability of the numerical method

We will proceed testing the classical Runge-Kutta method (RK4) in a similar manner as we did for the Explicit and Implicit Euler methods in part 1. As the RK4 has order 4, we should observe the local error having order 5 and the global error having order 4. Results are shown in the folowing figures, setting $\lambda = -1$, $h = 0.2$ for 35 and `tspan` $= [0, 30]$ for 36.

The shape of the local and global errors vs. time is very similar to the one obtained for the Explicit and Implicit Euler in Figure 1. However, both having the same time step size, the solution of the RK4 is a lot more precise. The scale of the errors is not even comparable.

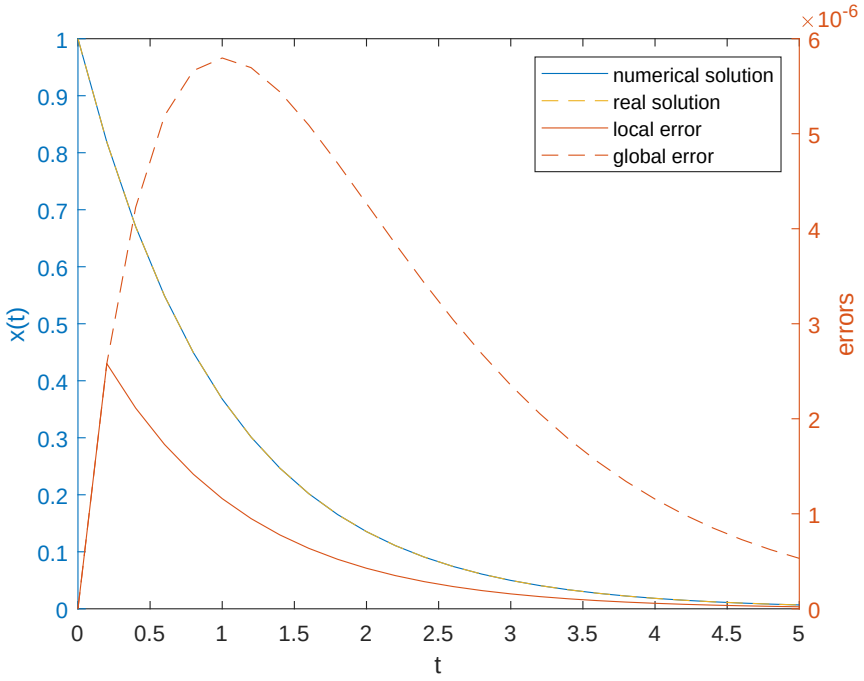
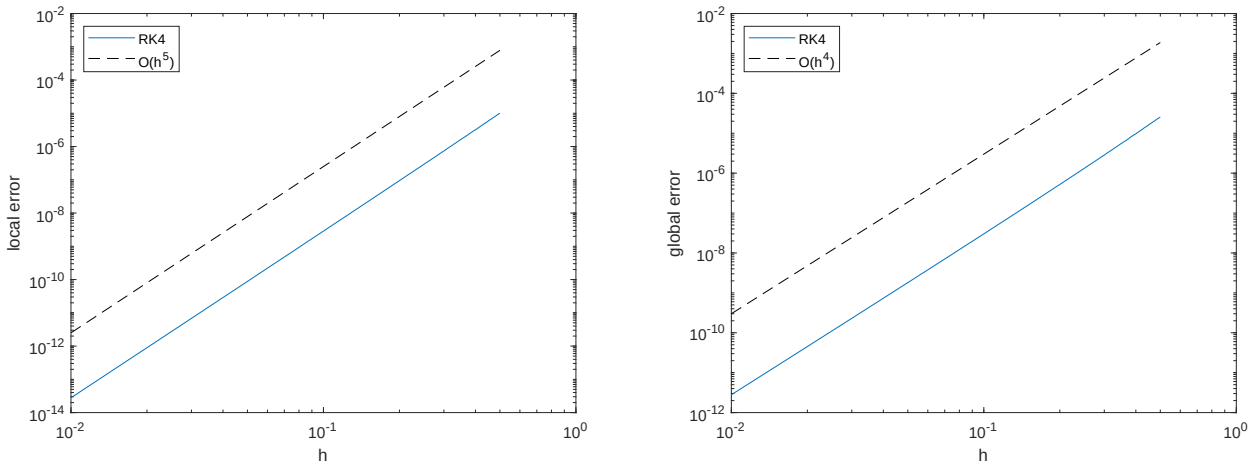


Figure 35: Solution and errors vs. time for the Test equation using fixed classical Runge-Kutta method



(a) Local error *(b) Global error*

Figure 36: Local and global errors vs. time-step size for the Test equation using fixed classical Runge-Kutta method

For the stability of the method, the solution by a Runge-Kutta method for the test equation is given as:

$$x_{n+1} = R(h\lambda)x_n \quad R(z) = 1 + zb'(I - zA)^{(} - 1)e, \tag{5}$$

where, A and b are the top right matrix and the bottom right vector in the Butcher Tableau 18. The stability regions for complex values of $h\lambda$ are shown in Figure 37. Here we can observe that the method is neither A-stable, nor L-stable, given that there are regions on the left side plane that are greater than 1 and that $\lim_{z\to-\infty} |R(z)| \neq 0$.

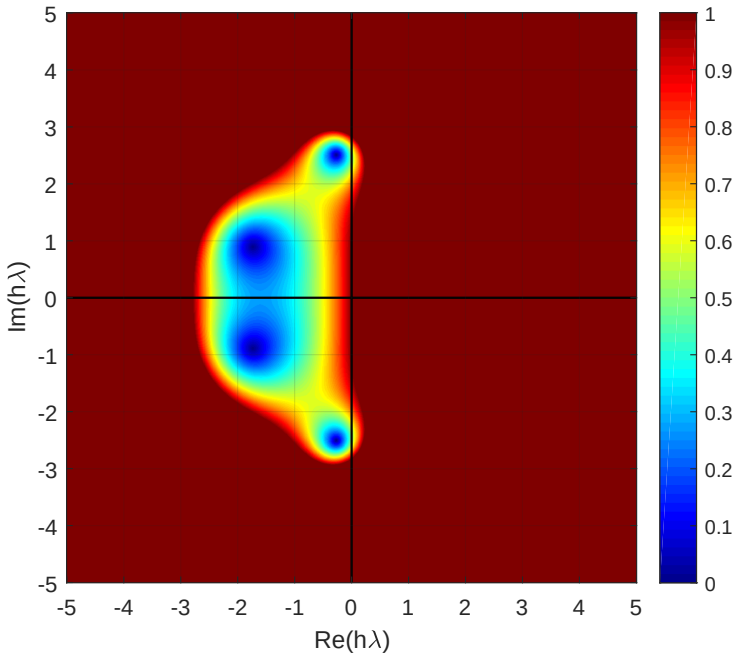


Figure 37: Values of $R(h\lambda)$ for the classical Runge-Kutta method

5.4. Test your algorithms on the Van der Pol problem ($\mu = 1.5$ and $\mu = 15$, $\mathbf{x_0} = [1.0; 1.0]$).

We can observe that the classical Runge-Kutta manages to approach the real solution with a way bigger step size than the Explicit Euler (Figures 4 and 5) in both cases. However, for $\mu = 15$ we need to lower the step size almost by a factor of 10 due to the larger stiffness of the problem. After all, the classical Runge-Kutta is a explicit method and thus suffer from the same problems as the Explicit Euler.

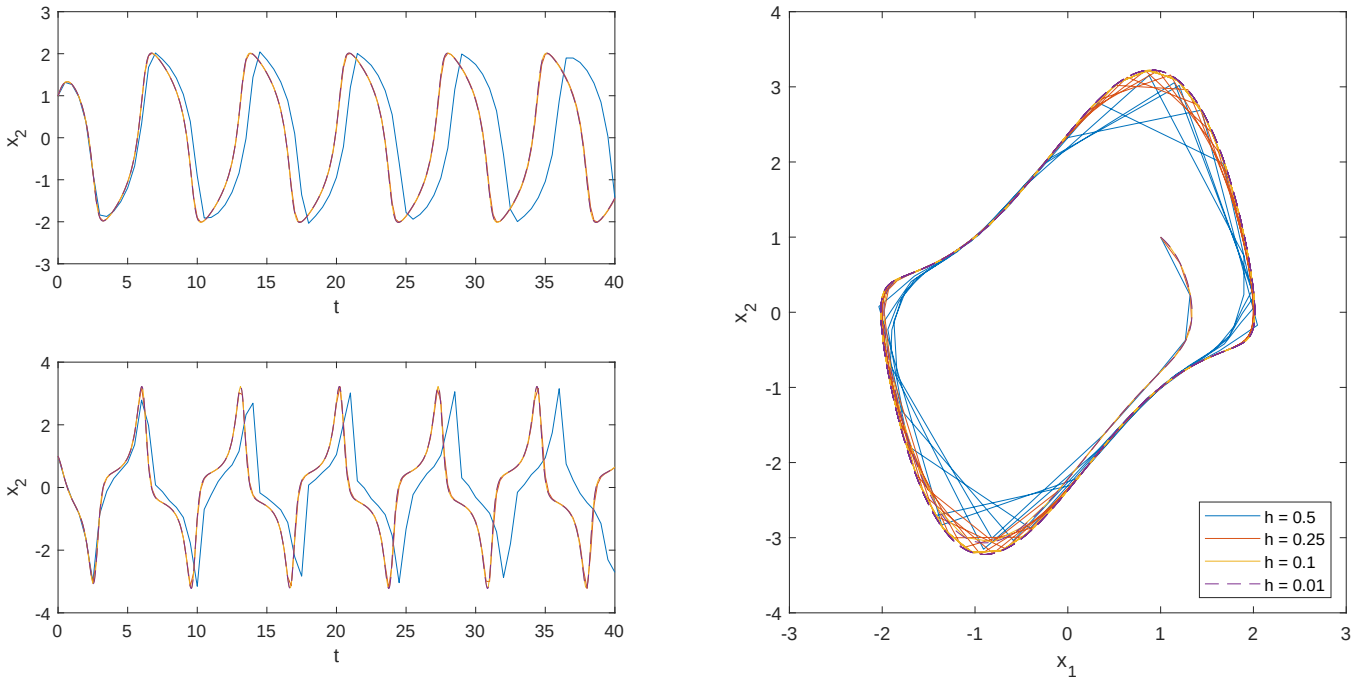


Figure 38: Solution for the Van der Pol problem ($\mu = 1.5$) using classical Runge-Kutta with fixed step size

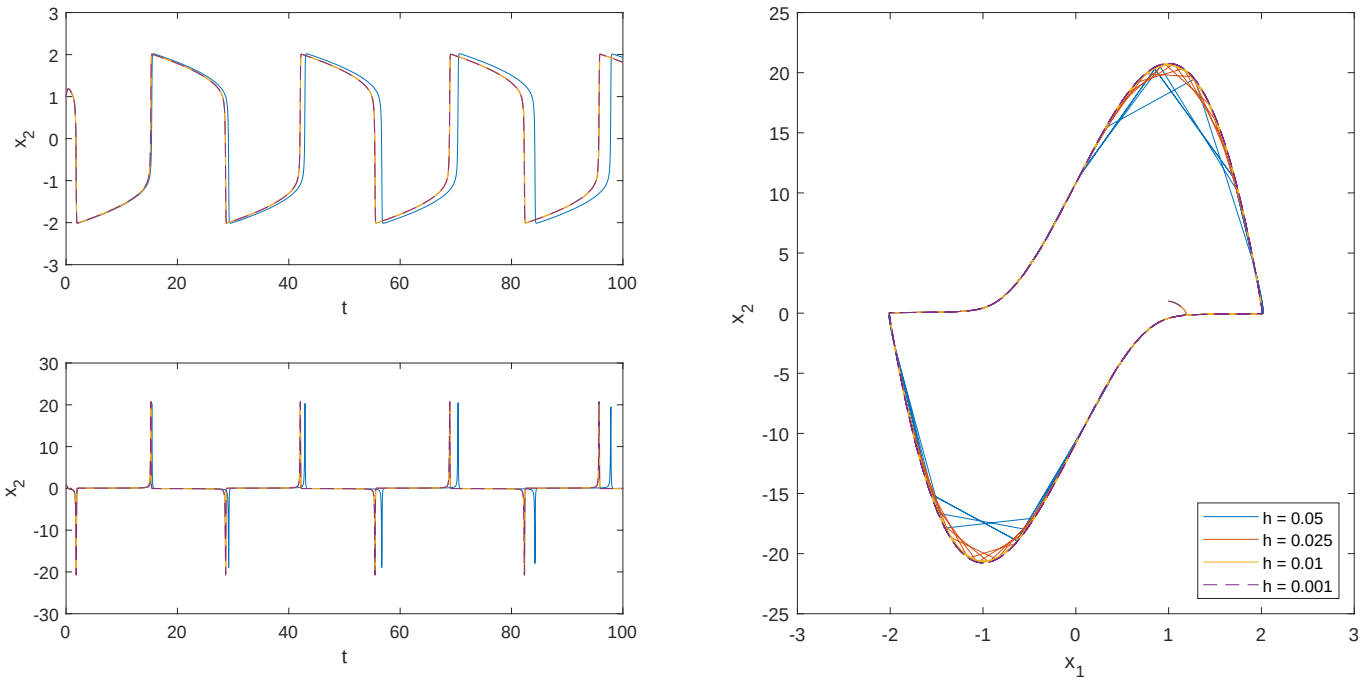
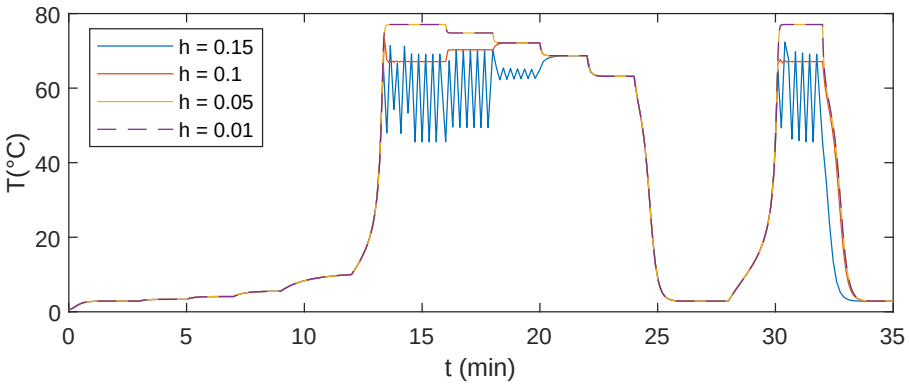


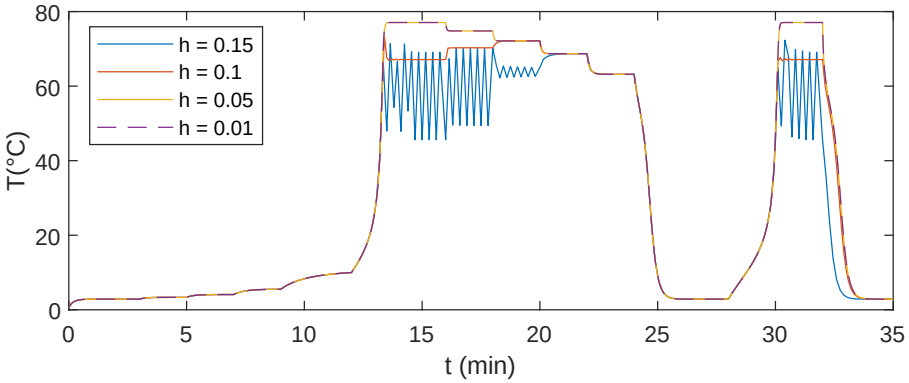
Figure 39: Solution for the Van der Pol problem ($\mu = 15$) using classical Runge-Kutta with fixed step size

5.5. Test your algorithms on the adiabatic CSTR problem described in the papers uploaded to Learn (3D-version and 1D-version).

For the CSTR problem, we can still see an increase in performance compared to the Explicit Euler (Figure 10) but not as significant as in the Van der Pol case. We can still see the spiky behaviour when T reaches its maximum (low F), when the problem is more stiff. The step size of $h = 0.1$ already approaches the solution pretty good, only failing at some values in this stiff area of the problem.



(a) CSTR 3D problem



(b) CSTR 1D problem

Figure 40: Solution for the CSTR problem using classical Runge-Kutta with fixed step size

5.6. Compare the results from your algorithms with the results you get using some of Matlab's ODE solvers

As in this part the classical Runge-Kutta with fixed time step size has been the only one implemented, we cannot directly compare to the chosen Matlab solvers (`ode45` and `ode15s`), as they both use adaptive step size. We'll compare the classical Runge-Kutta in next section, Exercise 6.6, with its adaptive version.

Part 6: Classical Runge-Kutta method with adaptive time step

We consider again the initial value problem in (3).

6.1. Describe the classical Runge-Kutta method with adaptive step size

The classical Runge-Kutta is described in exercise 5.1. The only change introduced in this part will be implementing the error estimation using step doubling and the asymptotic step size controller.

6.2. Implement an algorithm in Matlab for the classical Runge-Kutta method with adaptive time step size. Provide the code in your report. Use a format that enables syntax highlighting. Comment on the code

```
function [T,X,r_out,h_out,info] = ClassicRK4_adaptive(fun,tspan,h0,x0,abstol,reltol,args)

epstol = 0.8;
kpow = 0.2;
facmin = 0.1;
facmax = 5.0;

t0 = tspan(1);
tf = tspan(end);
t = t0;
h = h0;
x = x0;

T = t0;
X = x0;
r_out = [];
h_out = [];
info = zeros(1,4);

nfun = 0;
nstep = 0;
naccept = 0;

while t < tf
    if (t+h > tf)
        h = tf-t;
    end

    nfun = nfun + 1;
    AcceptStep = false;
    while ~AcceptStep
        x1 = ClassicRK4_step(fun,t,h,x,args);
        nfun = nfun + 4;

        hm = 0.5*h;
        tm = t + hm;
        xm = ClassicRK4_step(fun,t,hm,x,args);
        nfun = nfun + 4;
        x1hat = ClassicRK4_step(fun,tm,hm,xm,args);
        nfun = nfun + 4;
        nstep = nstep + 1;

        e = abs(x1hat-x1);
        r = max(e./max(abstol, abs(x1hat) .* reltol));
        AcceptStep = (r <= 1.0);

        if AcceptStep
            t = t+h;
            x = x1hat;

            T = [T,t];
            X = [X,x];
            r_out = [r_out, r];
            h_out = [h_out, h];
```

```
55              naccept = naccept + 1;
56          end
57
58          h = max(facmin, min((epstol/r)^kpow, facmax)) * h;
59      end
60  end
61
62  info(1) = nfun;
63  info(2) = nstep;
64  info(3) = naccept;
65  info(4) = nstep - naccept;
66
67  end
```

Listing 16: Classical Runge-Kutta method with adaptive time step size

6.3. Test your problem for test equation. Discuss order and stability of the numerical method

Implementing the adaptive classical Runge-Kutta just as we did for the classical one leads us to the results shown below. It's curious to see that the relation between the local and global error vs. the tolerance. The relation between them was found empirically.

The stability of the method is already discussed in Exercise 5.3.

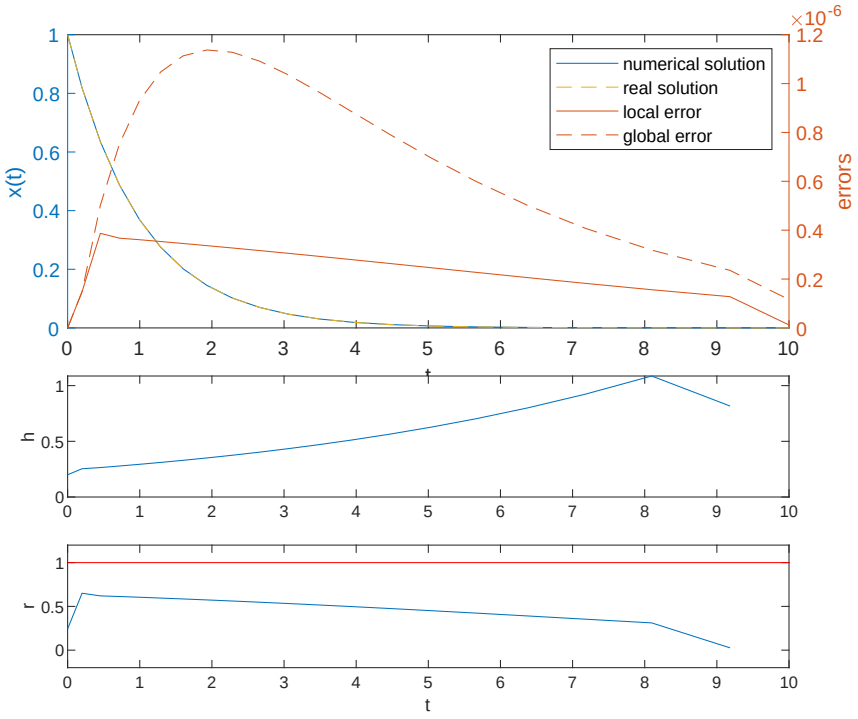
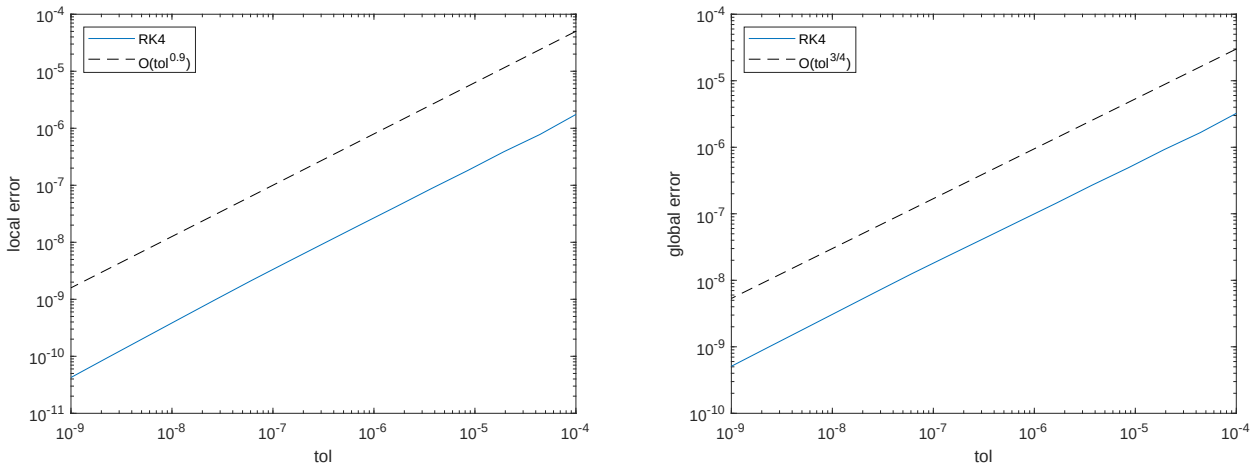


Figure 41: Solution and errors vs. time for the Test equation using adaptive classical Runge-Kutta method



(a) Local error *(b) Global error*

Figure 42: Local and global errors vs. time-step size for the Test equation using adaptive classical Runge-Kutta method

6.4. Test your algorithms on the Van der Pol problem ($\mu = 1.5$ and $\mu = 15$, $\mathbf{x_0} = [1.0; 1.0]$).

The results for the Van der pol problem are shown below. When comparing to the Explicit Euler with adaptive results shown in Figures 6 and 7, we can see that the method achieves better performance when working with same tolerances, or even larger. The change of frequency of the signals has disappeared completely and all tolerances approach the real solution pretty good. However, as it's an explicit method, it's still sensible to the stiffness of the Van der Pol with $\mu = 15$. Specially, comparing Tables 2 and 20, we can see how the number of necessary steps is a lot smaller in the Runge-Kutta. However, for smaller tolerances the number of function evaluations is higher. Thus, the Runge-Kutta method is a better alternative if we are interested in a very precise solution, with a very low tolerance.

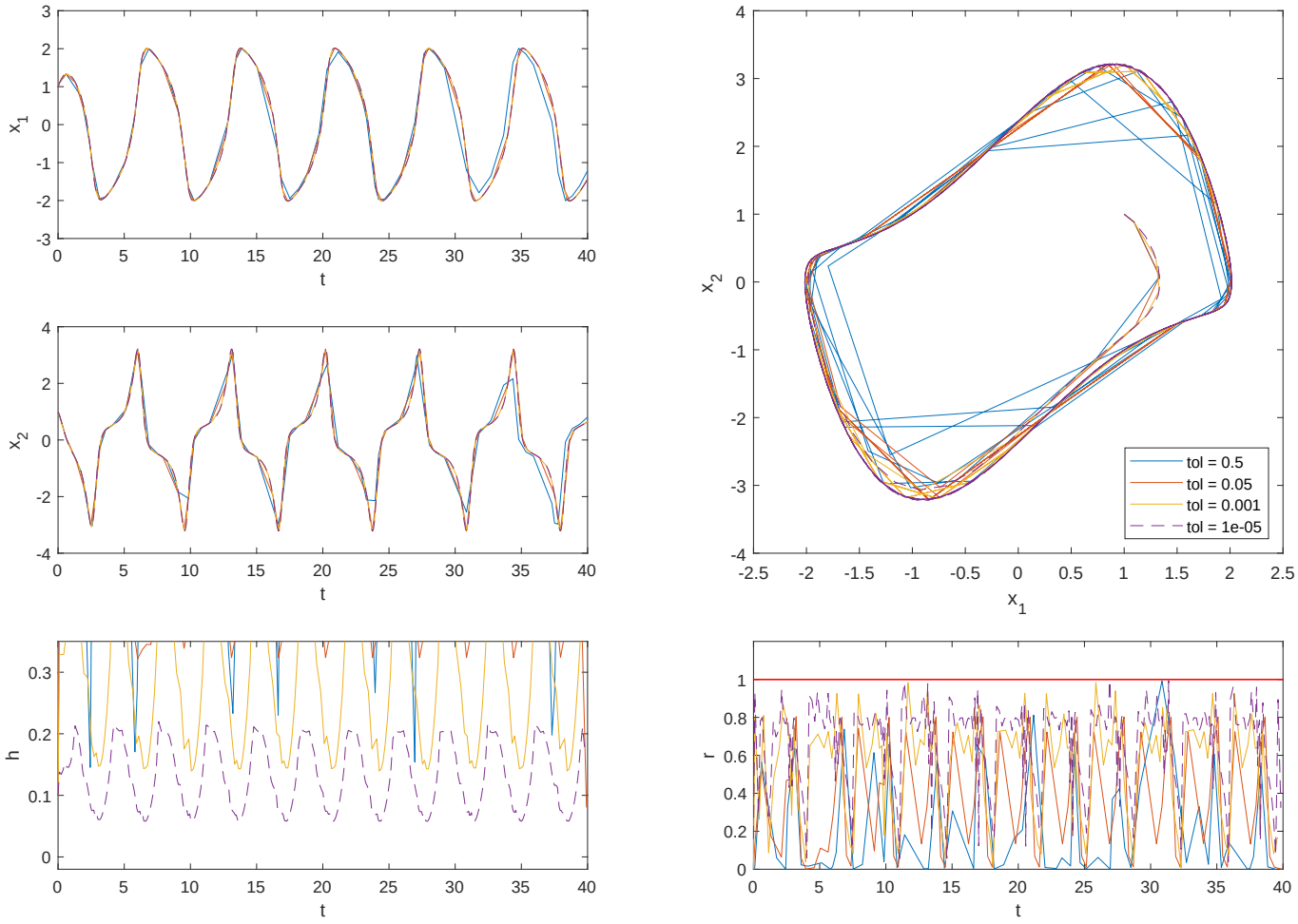


Figure 43: Solution for the Van der Pol problem ($\mu = 1.5$) using classical Runge-Kutta with adaptive step size

Tolerances	0.5	0.05	0.001	1e-05
Function evaluations	1335	1690	2843	6440
Calculated steps	106	134	224	506
Accepted steps	63	82	155	368
Rejected steps	43	52	69	138

Table 19: Parameters of the classical Runge-Kutta with adaptive step size for the Van der Pol problem ($\mu = 1.5$)

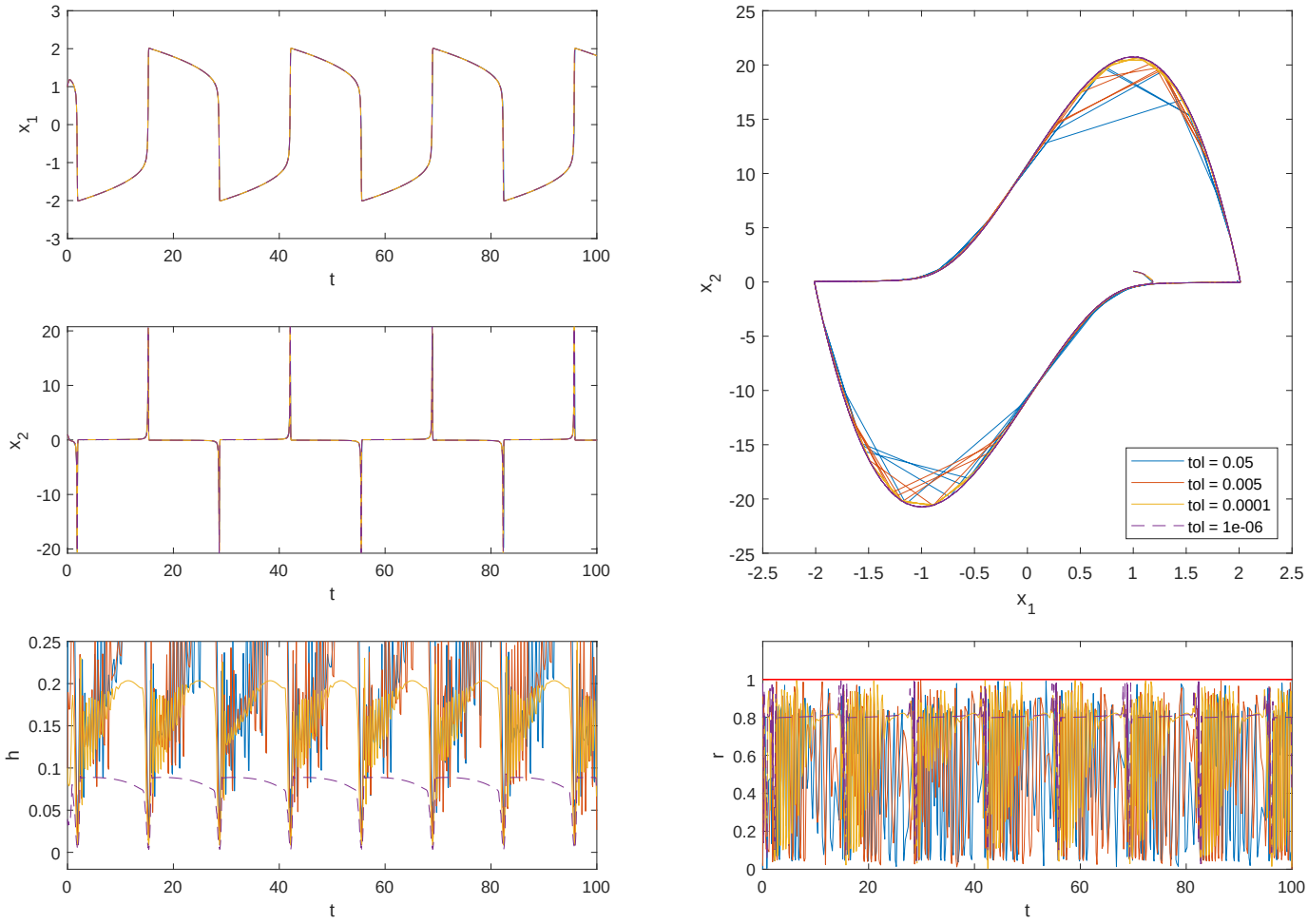


Figure 44: Solution for the Van der Pol problem ($\mu = 15$) using classical Runge-Kutta with adaptive step size

Tolerances	0.05	0.005	0.0001	1e-06
Function evaluations	10951	11526	14814	26382
Calculated steps	867	910	1164	2050
Accepted steps	547	606	846	1782
Rejected steps	320	304	318	268

Table 20: Parameters of the classical Runge-Kutta with adaptive step size for the Van der Pol problem ($\mu = 15$)

6.5. Test your algorithms on the adiabatic CSTR problem described in the papers uploaded to Learn (3D-version and 1D-version).

Just as in the Van der Pol, the classical Runge-Kutta with adaptive step size manages to outperform the convergence achieved by the Explicit Euler. It also deals a lot better with the stiff area of the problem. However, even though it needs less steps to achieve a better solution with the same tolerance, the extra amount of function evaluations per step makes it only worthed when the tolerance is pretty small.

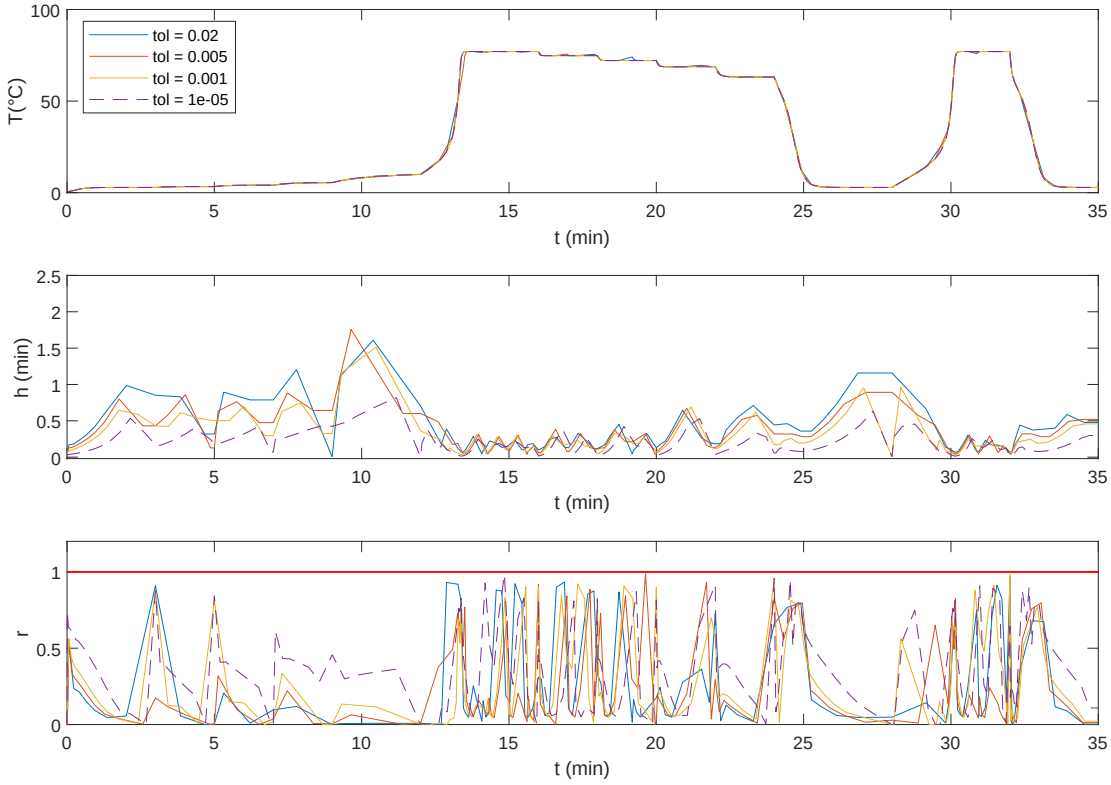


Figure 45: Solution for the CSTR 3D problem using classical Runge-Kutta with adaptive step size

Tolerances	0.02	0.005	0.001	1e-05
Function evaluations	2068	2168	2650	4274
Calculated steps	163	170	208	334
Accepted steps	112	128	154	266
Rejected steps	51	42	54	68

Table 21: Parameters of the classical Runge-Kutta with adaptive step size for the CSTR 3D problem

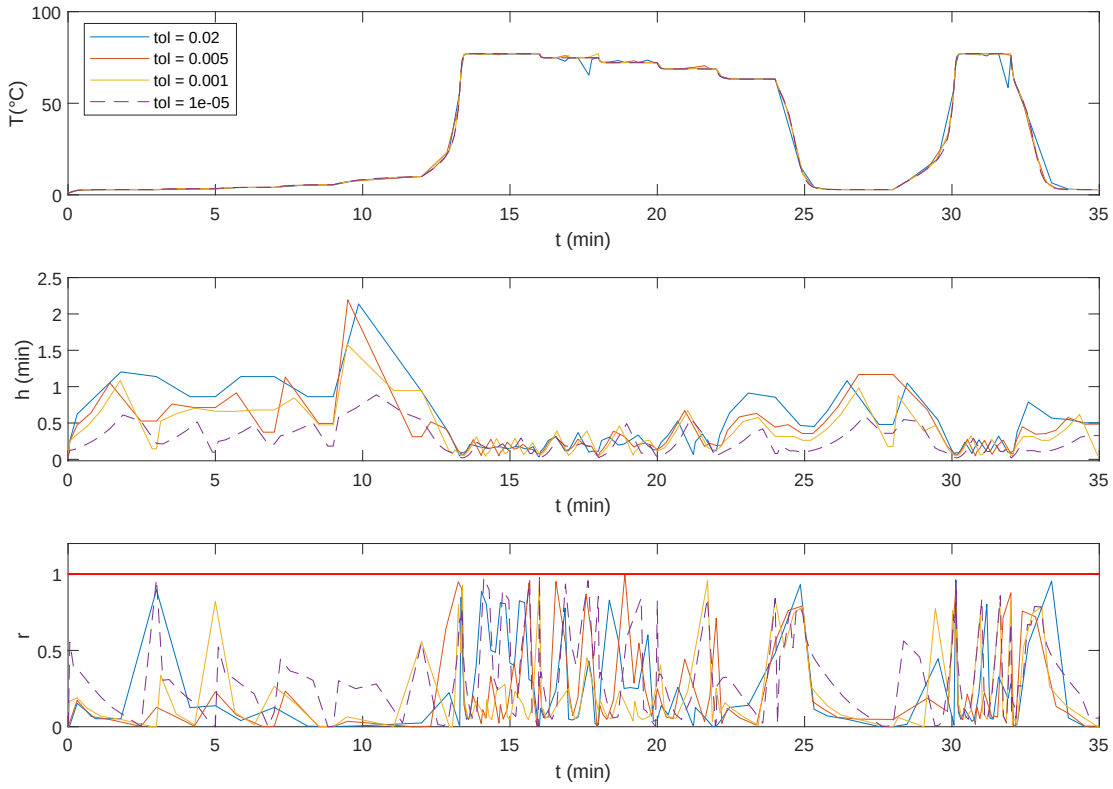


Figure 46: Solution for the CSTR 1D problem using classical Runge-Kutta with adaptive step size

Tolerances	0.02	0.005	0.001	1e-05
Function evaluations	1741	1909	2143	3425
Calculated steps	137	150	168	268
Accepted steps	97	109	127	209
Rejected steps	40	41	41	59

Table 22: Parameters of the classical Runge-Kutta with adaptive step size for the CSTR 1D problem

6.6. Compare the results from your algorithms with the results you get using some of Matlab's ODE solvers

For the Van der Pol problem, we tested the classical Runge-Kutta against `ode45` for the non-stiff case ($\mu = 1.5$), and against `ode15s` for the stiff case ($\mu = 15$). With this method, we are already approaching the "big leagues" of ODE solvers. We can observe directly the effect that having a greater order of the error has on the solution: it's a lot more precise with big tolerances. Actually, the classical Runge-Kutta achieves similar accuracy to the `ode45`, when not better. However, this all comes with a cost, the number of function evaluations is also larger, but it's not as far from it as both Euler methods. As it's an explicit method, it doesn't stand a chance against the `ode15s` in the stiff case when looking at the number of function evaluations. However, the accuracy of the method is still surprisingly good, specially for low tolerances, where the `ode15s` misses the solution.

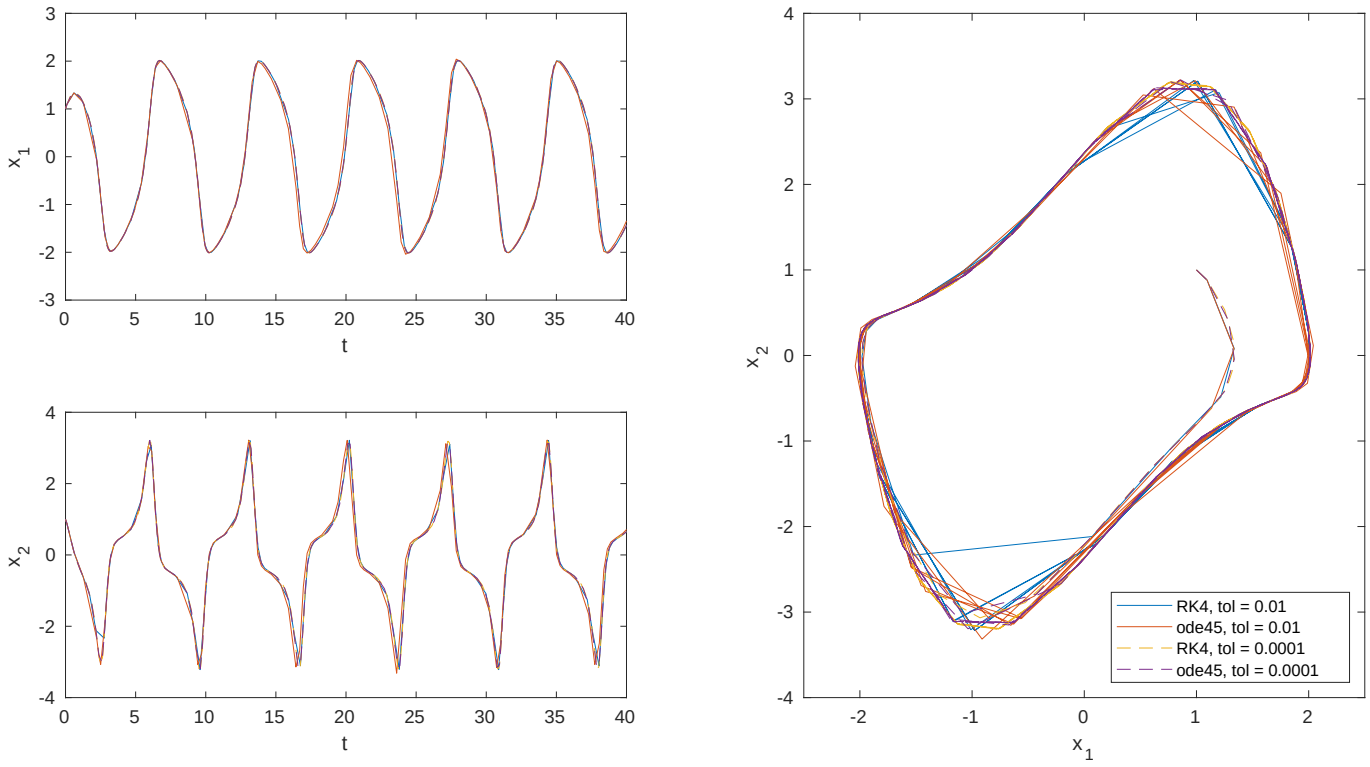


Figure 47: Solution for the Van der Pol problem ($\mu = 1.5$) using Classical Runge-Kutta vs. `ode45`

Method	**RK4**		**ode45**	
Tolerances	0.01	0.0001	0.01	0.0001
Function evaluations	2071	4341	787	1357
Calculated steps	164	342	131	226
Accepted steps	103	237	100	180
Rejected steps	61	105	31	46

Table 23: Parameters of the Classical Runge-Kutta vs. `ode45` *for the Van der Pol problem ($\mu = 1.5$)*

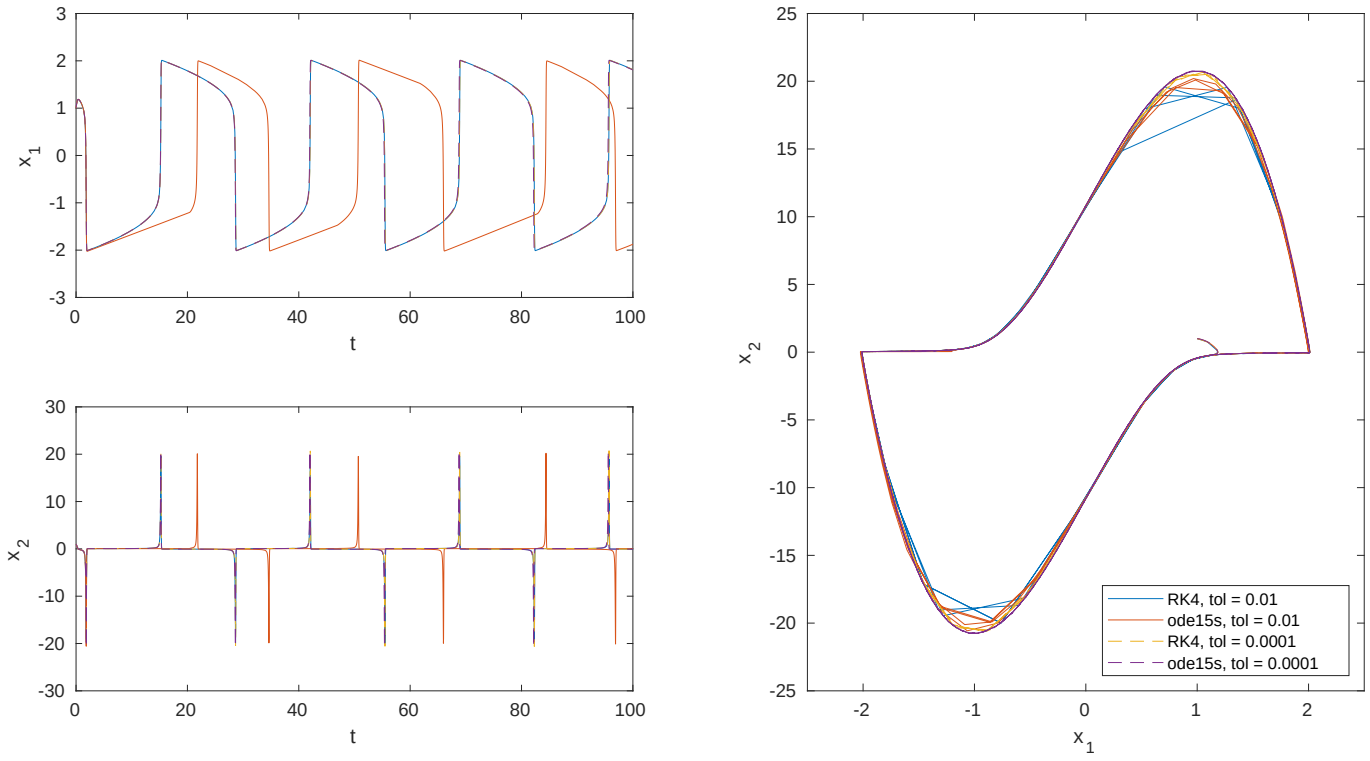
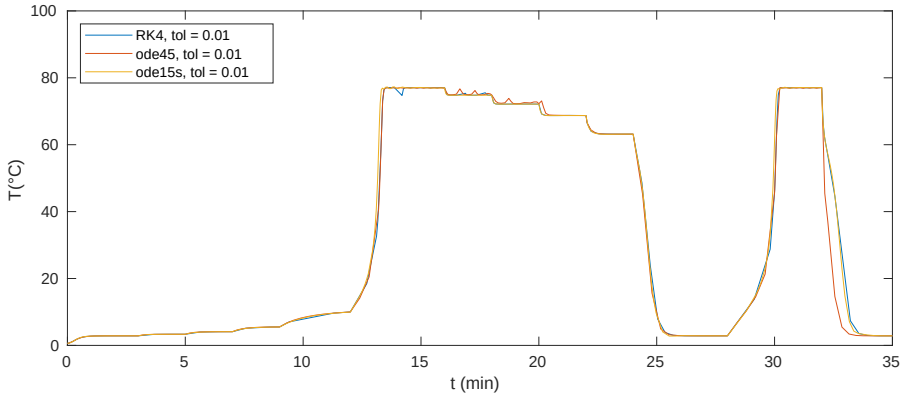


Figure 48: Solution for the Van der Pol problem ($\mu = 15$) using Classical Runge-Kutta vs. `ode15s`

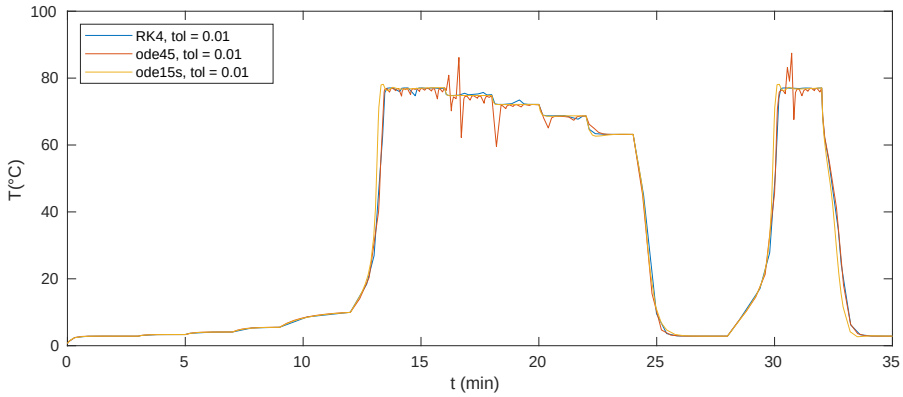
Method	**RK4**		**ode15s**	
Tolerances	0.01	0.0001	0.01	0.0001
Function evaluations	11442	14814	1273	2780
Calculated steps	905	1164	558	1327
Accepted steps	582	846	411	1094
Rejected steps	323	318	147	233

Table 24: Parameters of the Classical Runge-Kutta vs. `ode15s` *for the Van der Pol problem ($\mu = 15$)*

For the CSTR problem, we see how the three of them achieve good performance. The classical Runge-Kutta approaches the solution better that the `ode45` , but having more function evaluations. The `ode15s` continues to be the best one for this problem, specially in the 1D case.



(a) CSTR 3D problem



(b) CSTR 1D problem

Figure 49: Solution for the CSTR problem using Classical Runge-Kutta vs. `ode45` *and* `ode15s`

Method	**RK4**	**ode45**	**ode15s**
Tolerances	0.01	0.01	0.01
Function evaluations	1978	1351	548
Calculated steps	155	1459	1612
Accepted steps	118	196	232
Rejected steps	37	27	36

Table 25: Parameters of the Classical Runge-Kutta vs. `ode45` *and* `ode15s` *for the CSTR-3D problem*

Method	**RK4**	**ode45**	**ode15s**
Tolerances	0.01	0.01	0.01
Function evaluations	1906	1531	408
Calculated steps	150	1651	1316
Accepted steps	106	220	186
Rejected steps	44	33	30

Table 26: Parameters of the Classical Runge-Kutta vs. `ode45` *and* `ode15s` *for the CSTR-1D problem*

Part 7: Dormand-Prince 5(4)

We consider again the initial value problem in (3).

7.1. Describe the Dormand-Prince (DOPRI54) method with adaptive time step size

The DOPRI54 method is an explicit method of the Runge-Kutta family of ODE solvers. The method calculates a fourth- and fifth-order accurate solutions. The difference between them is used as an approximation of the local truncation error. Note that this approximation is obtained by only using an extra function evaluation, avoiding all the computational costs of other ways of approximating it (f.e. the step doubling we used in all the adaptive implementations before). This behaviour makes the DOPRI54 a very solid choice as an adaptive ODE solver.

0							
1/5	1/5						
3/10	3/40	9/40					
4/5	44/45	-56/15	32/9				
8/9	19372/6561	-25360/2187	64448/6561	-212/729			
1	9017/3168	-355/33	46732/5247	49/176	-5103/18656		
1	35/384	0	500/1113	125/192	-2187/6784	11/84	
	35/384	0	500/1113	125/192	-2187/6784	11/84	0
	5179/57600	0	7571/16695	393/640	-92097/339200	187/2100	1/40

Table 27: Butcher Tableau of the Dormand-Prince 5(4) method

7.2. Implement an algorithm in Matlab for the DOPRI54 method with adaptive time step size. Provide the code in your report. Use a format that enables syntax highlighting. Comment on the code

```
function [T_out,X_out,e_out,r_out,h_out,info] = ...
    DOPRI54_adaptive(fun,tspan,h0,x0,abstol,reltol,butcher,args)

% Controller parameters
epstol = 0.8;
kpow = 1/6;
facmin = 0.1;
facmax = 5.0;

t0 = tspan(1);
tf = tspan(end);
t = t0;
h = h0;
x = x0;

nx = size(x0,1);

T_out = t0;
X_out = x0;
e_out = zeros(nx,1);
r_out = [];
h_out = [];
info = zeros(4,1);

nfun = 0;
nstep = 0;
naccept = 0;

% Extract Butcher Tableau
s = butcher.stages;
AT = butcher.AT;
b = butcher.b;
c = butcher.c;
d = butcher.d;

```

```
% Allocate memory for the stages
T = zeros(1,s);
X = zeros(nx,s);
F = zeros(nx,s);

while t < tf

    % Size of last step
    if t+h > tf
        h = tf - t;
    end
    % First stage
    T(1) = t;
    X(:,1) = x;
    F(:,1) = feval(fun,T(1),X(:,1),args{:});
    nfun = nfun + 1;

    AcceptStep = false;

    while ~AcceptStep
        % Precalculated parameters
        hAT = h*AT;
        hb = h*b;
        hc = h*c;
        hd = h*d;

        % Following stages
        for i = 2:s
            T(i) = t + hc(i);
            X(:,i) = x + F(:,1:i-1)*hAT(1:i-1,i);
            F(:,i) = feval(fun,T(i),X(:,i),args{:});
            nfun = nfun + 1;
        end

        % Error estimation
        e = F*hd;
        xhat = x + F*hb;
        r = max(abs(e)./max(abstol,abs(xhat).*reltol));
         % Check conditon
        AcceptStep = r <=1;

        if AcceptStep
            t = t+h;
            x = xhat;

            T_out = [T_out,t];
            X_out = [X_out,x];
            e_out = [e_out,e];
            r_out = [r_out, r];
            h_out = [h_out, h];
            naccept = naccept + 1;
        end

        % step size controller (Asymptotic or second order PI)
        h = max(facmin,min((epstol/r)^kpow,facmax))*h;
        nstep = nstep+1;
    end
end

info(1) = nfun;
info(2) = nstep;
info(3) = naccept;
info(4) = nstep - naccept;

end
```

Listing 17: Dormand-Prince 5(4) method with adaptive time step size

7.3. Test your problem for test equation. Discuss order and stability of the numerical method

We will proceed testing the Dormand-Prince 5(4) method in a similar manner as we did for the classical Runge-Kutta method (RK4) in part 5. The method has order 5 but, as the asked implementation is only for an adaptive method we can't show the same plots as for the fixe Runge-Kutta. The shape of the local and global errors vs. time is similar to the one obtained for the adaptive classical Runge-Kutta in Figure 41. Weirdly, the solution for the Dormand-Prince 5(4) has slightly more error. It's also worth noticing that the approximation of the error of the method is larger than the real local error, which is a nice thing.

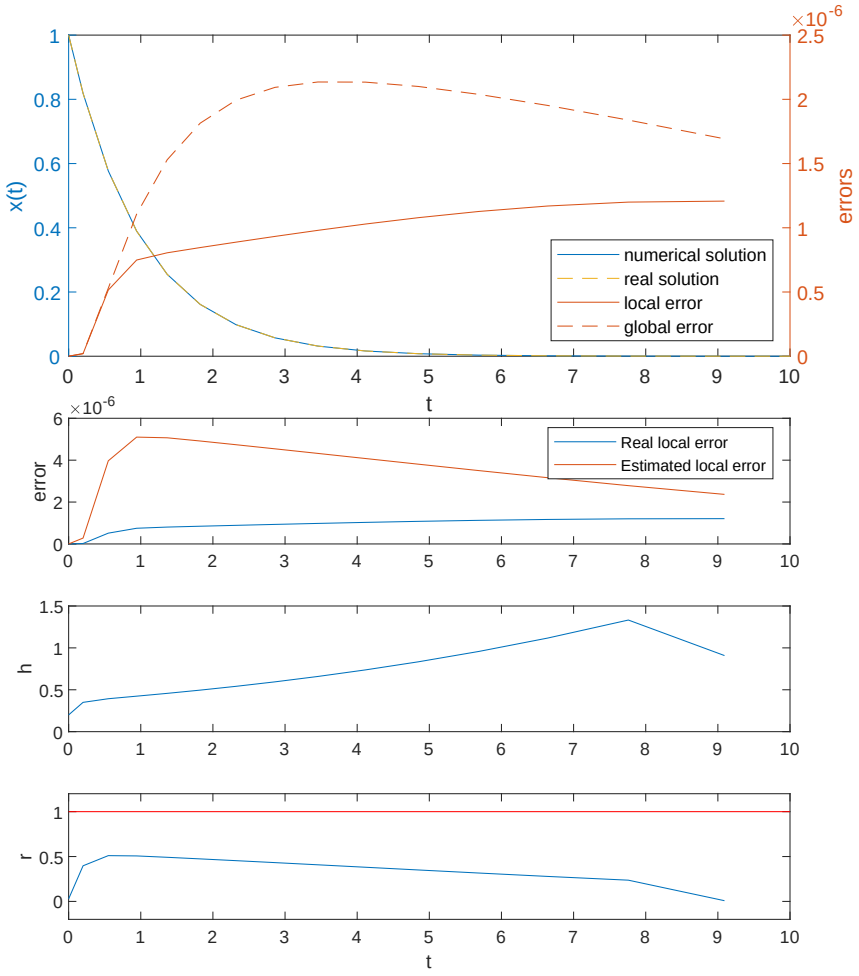
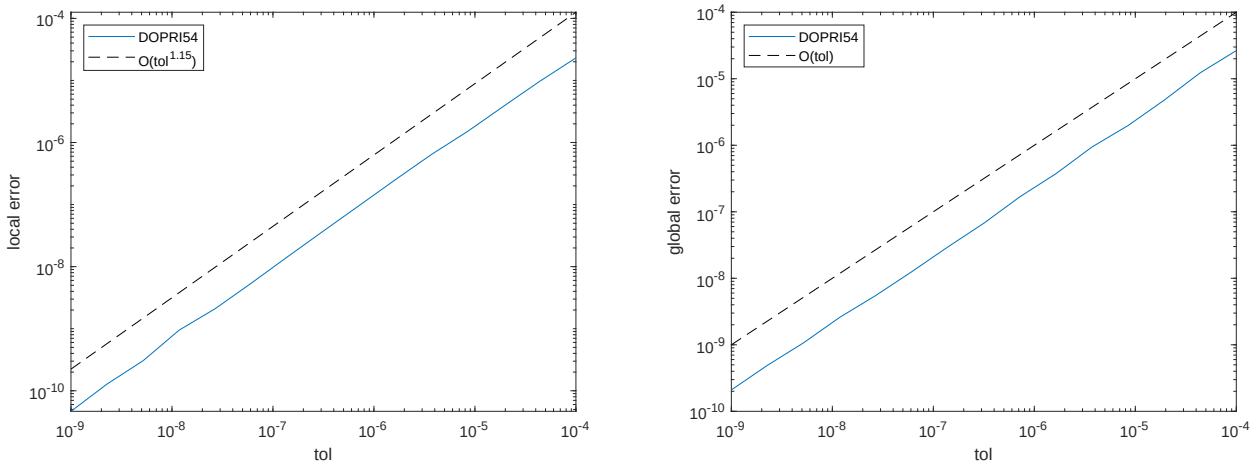


Figure 50: Solution and errors vs. time for the Test equation using adaptive Dormand-Prince 5(4) method



(a) Local error *(b) Global error*

Figure 51: Local and global errors vs. tolerances for the Test equation using adaptive Dormand-Prince 5(4) method

For the stability of the method, the solution of a member of the Runge-Kutta family of ODE solvers for the test equation is given as:

$$x_{n+1} = R(h\lambda)x_n \quad R(z) = 1 + zb'(I - zA)^(- 1)e,$$

where, A and b are the top right matrix and the bottom right vector in the Butcher Tableau 27. The stability regions for complex values of $h\lambda$ are shown in Figure 52. Here we can observe that the method is neither A-stable, nor L-stable, given that there are regions on the left side plane that are greater than 1 and that $\lim_{z\to-\infty} |R(z)| \neq 0$.

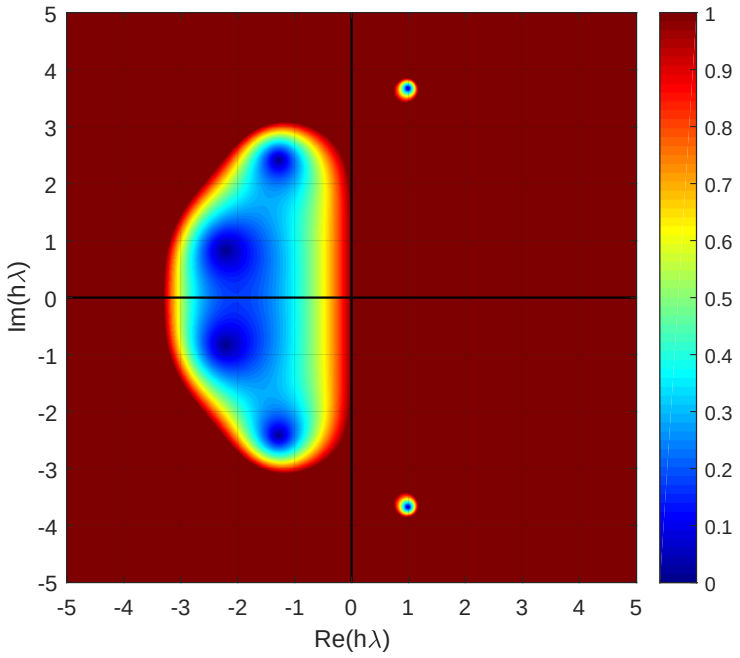


Figure 52: Values of $R(h\lambda)$ for the Dormand-Prince 5(4) method

7.4. Test your algorithms on the Van der Pol problem ($\mu = 1.5$ and $\mu = 15$, $\mathbf{x_0} = [1.0; 1.0]$).

The results for the Van der pol problem are shown below. When comparing to the Explicit Euler with adaptive results shown in Figures 6 and 7, we can see that the method achieves better performance when working with same tolerances, or even larger. Contrary to what happened for the classical Runge-Kutta, the shift of frequency of the signals has reappeared completely and bigger tolerances don't work as good as before. Here we can observe a pretty strange behaviour, the biggest tolerance solution shifts negatively at some points. Also, as it's an explicit method, it's still sensible to the stiffness of the Van der Pol with $\mu = 15$, although it handles the change in step size a lot better than the other ones.

Finally, comparing both tables with the results from the adaptive classical Runge-Kutta (Tables 19 and 20) we observe that, although the number of calculated steps is similar, the number of function evaluations is considerably lower in the Dormand-Prince 5(4), due to the way of calculating an approximate local error and not using step doubling.

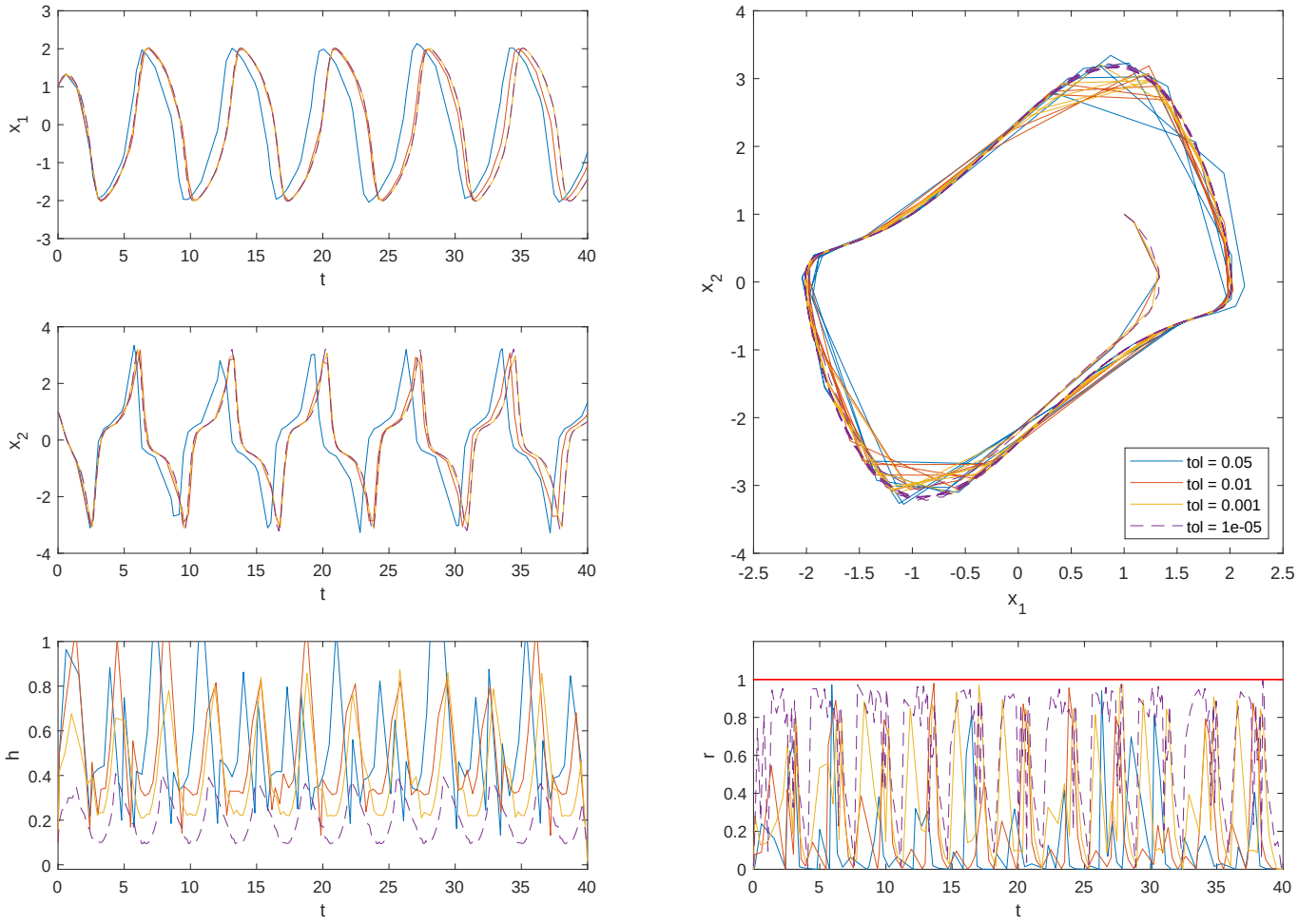


Figure 53: Solution for the Van der Pol problem ($\mu = 1.5$) using Dormand-Prince 5(4) with adaptive step size

Tolerances	0.05	0.01	0.001	1e-05
Function evaluations	867	936	1194	2350
Calculated steps	131	141	180	352
Accepted steps	81	90	114	238
Rejected steps	50	51	66	114

Table 28: Parameters of the Dormand-Prince 5(4) with adaptive step size for the Van der Pol problem ($\mu = 1.5$)

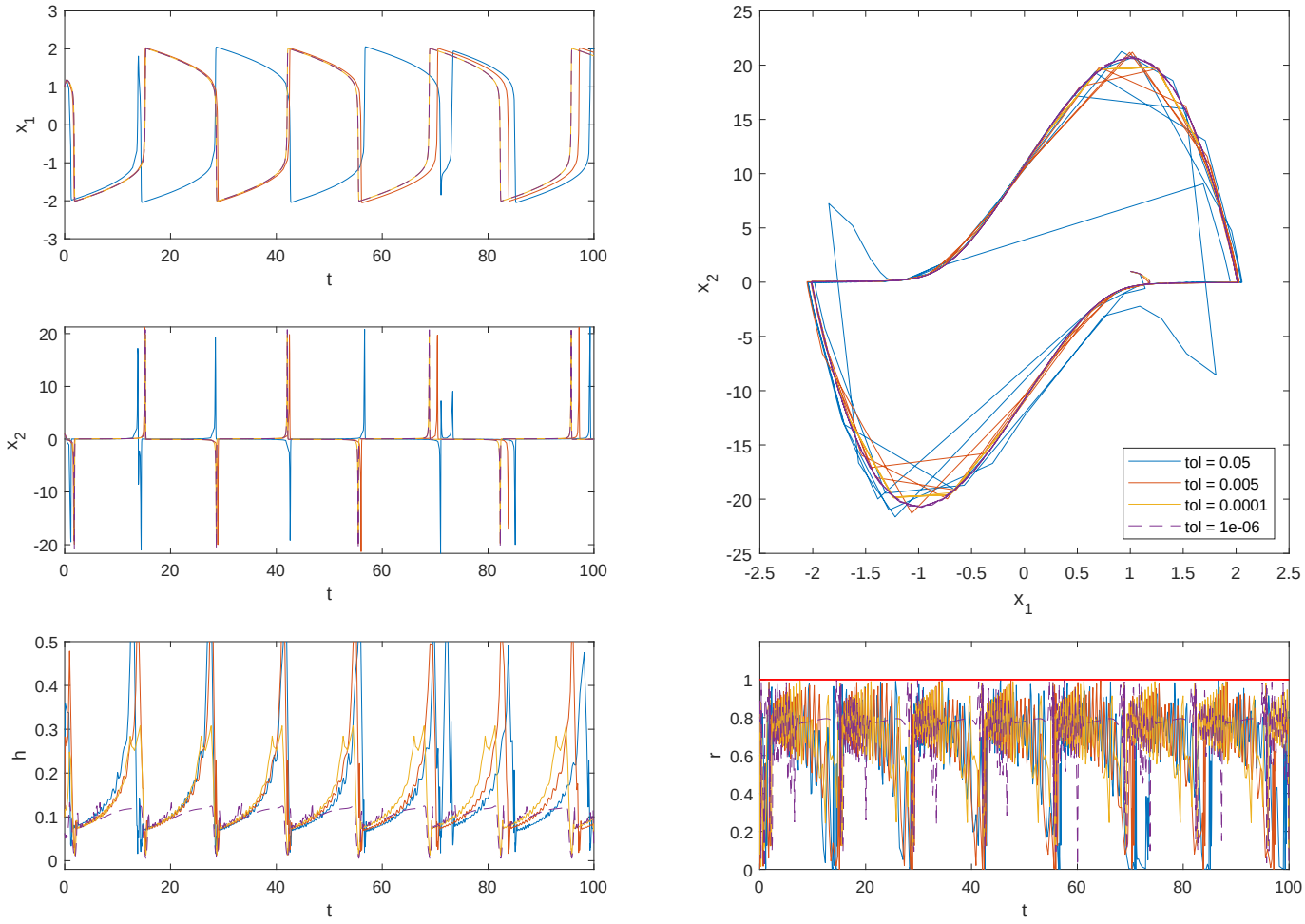


Figure 54: Solution for the Van der Pol problem ($\mu = 15$) using Dormand-Prince 5(4) with adaptive step size

Tolerances	0.05	0.005	0.0001	1e-06
Function evaluations	6729	6624	7590	10998
Calculated steps	979	959	1106	1608
Accepted steps	855	870	954	1350
Rejected steps	124	89	152	258

Table 29: Parameters of the Dormand-Prince 5(4) with adaptive step size for the Van der Pol problem ($\mu = 15$)

7.5. Test your algorithms on the adiabatic CSTR problem described in the papers uploaded to Learn (3D-version and 1D-version).

Again, as in the Van der Pol, we need to set the tolerance a bit lower than in the classical Runge-Kutta for it to converge. However, it deals a lot better with the stiff area of the problem, specially for the 3D case. Again, we observe by comparing the tables that for the same number of calculated steps we need a lot less function evaluations.

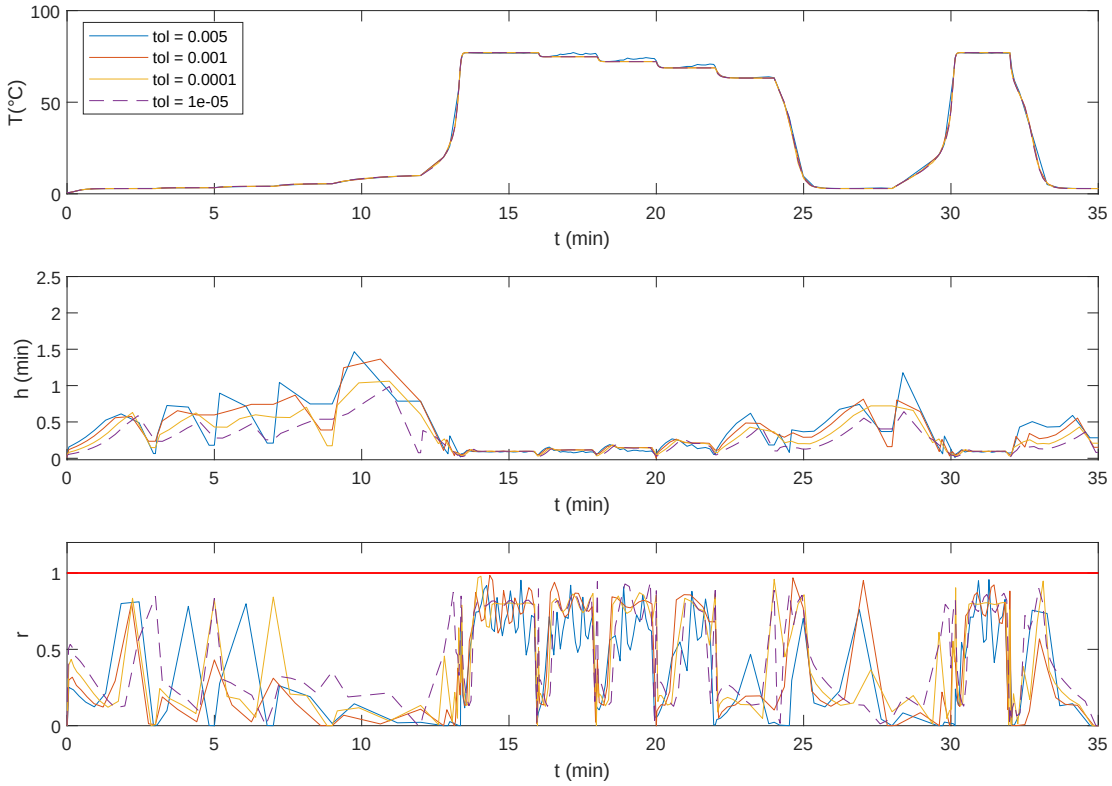


Figure 55: Solution for the CSTR 3D problem using Dormand-Prince 5(4) with adaptive step size

Tolerances	0.005	0.001	0.0001	1e-05
Function evaluations	1507	1423	1538	1918
Calculated steps	223	209	225	281
Accepted steps	169	169	188	232
Rejected steps	54	40	37	49

Table 30: Parameters of the Dormand-Prince 5(4) with adaptive step size for the CSTR 3D problem

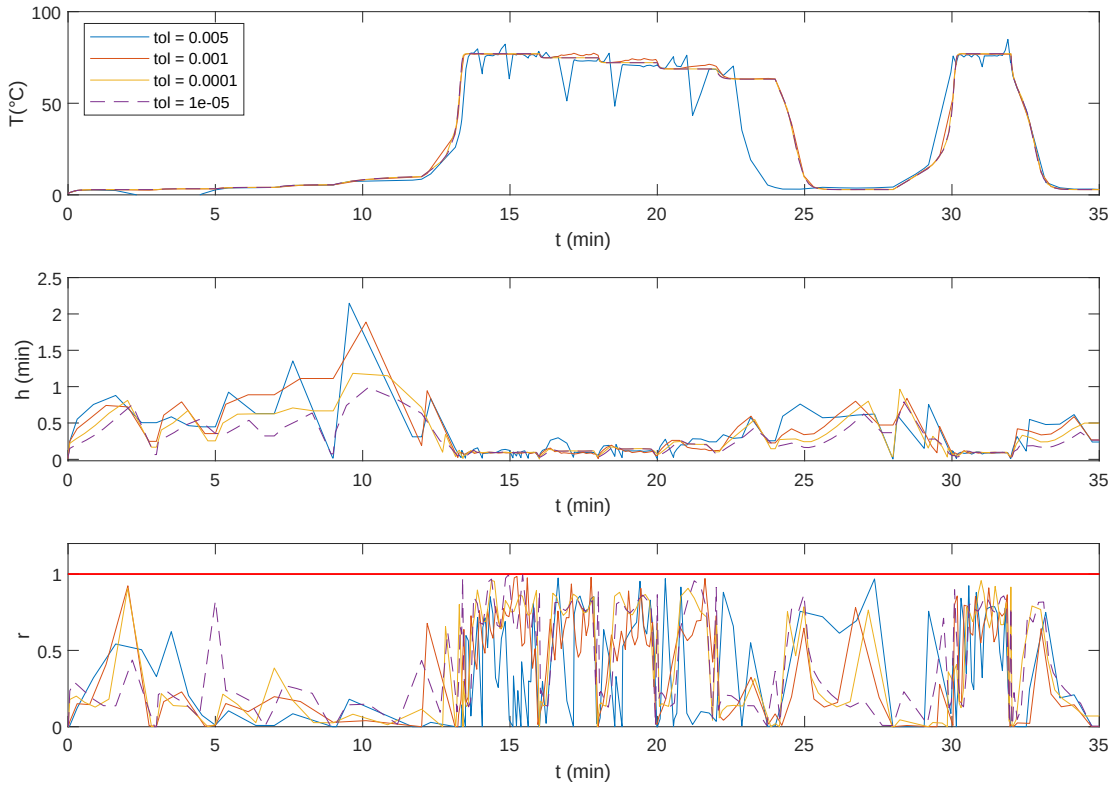


Figure 56: Solution for the CSTR 1D problem using Dormand-Prince 5(4) with adaptive step size

Tolerances	0.005	0.001	0.0001	1e-05
Function evaluations	1397	1441	1408	1637
Calculated steps	206	213	206	240
Accepted steps	161	163	172	197
Rejected steps	45	50	34	43

Table 31: Parameters of the Dormand-Prince 5(4) with adaptive step size for the CSTR 1D problem

7.6. Compare the results from your algorithms with the results you get using some of Matlab's ODE solvers (in particular ode45 which implements the DOPRI54 method)

For the Van der Pol problem, we tested the Dormad-Prince 4(5) method against `ode45` for the non-stiff case ($\mu = 1.5$), and against `ode15s` for the stiff case ($\mu = 15$). With this method, should be expecting a similar performance, if not exact same to the `ode45` , as the Matlab documentation says this is their implemented method. However, the results we obtain are actually different. This could be caused because Matlab implementation uses a different way of calculating the error other than the approximation from the method, or that we forgot to adjust some of the flags other than the actual tolerances.

In the results, the Dormand-Prince achieves same accuracy as the `ode45` . IN the number of function evaluations though, we see a difference. For the Van der Pol it has a little bit more, but in the CSTR it has a lot less function evaluations. Their performance for this problem is also quite similar. As it's an explicit method, `ode15s` still remains the best choice for the stiff case, when looking at the number of function evaluations.

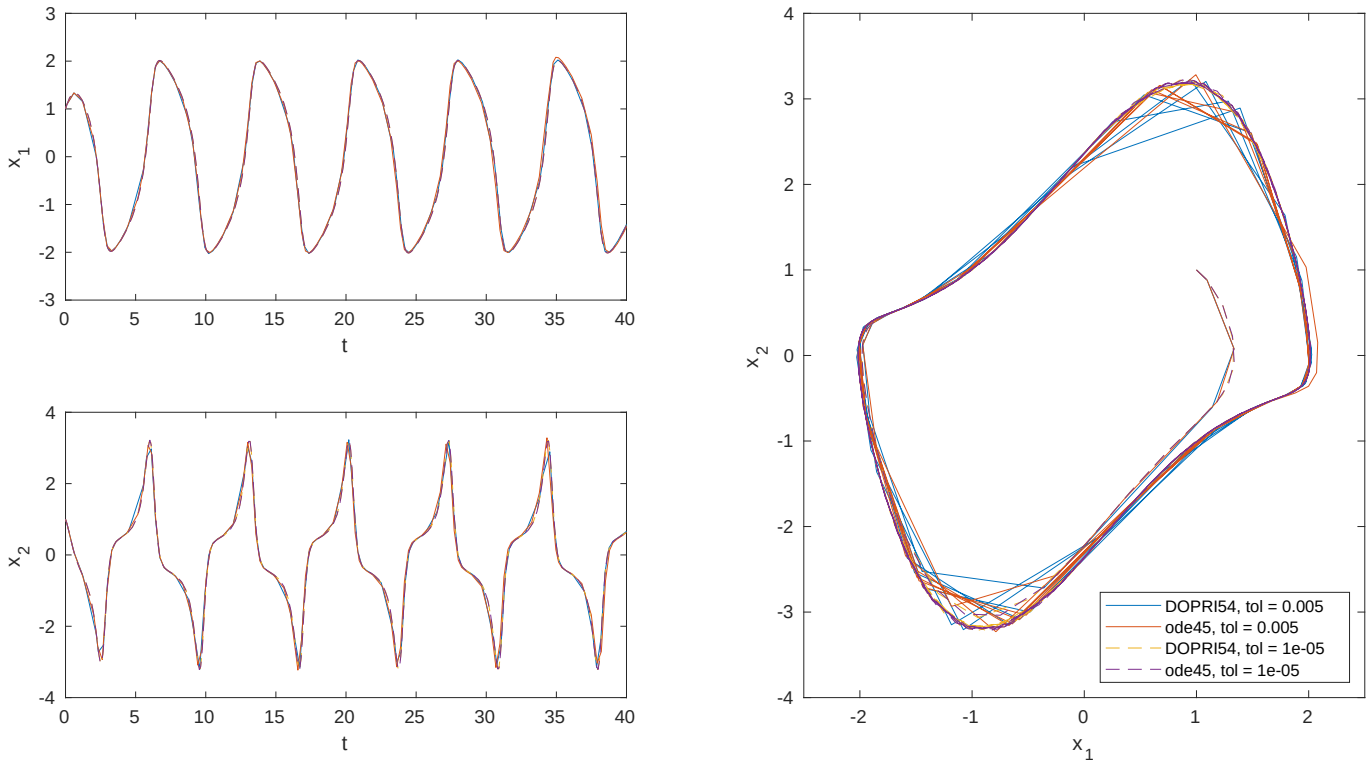


Figure 57: Solution for the Van der Pol problem ($\mu = 1.5$) using Dormand-Prince 5(4) vs. `ode45`

Method	**DOPRI45**		**ode45**	
Tolerances	0.005	1e-05	0.005	1e-05
Function evaluations	960	2350	793	1903
Calculated steps	144	352	132	317
Accepted steps	96	238	103	270
Rejected steps	48	114	29	47

Table 32: Parameters of the Dormand-Prince 5(4) vs. `ode45` *for the Van der Pol problem ($\mu = 1.5$)*

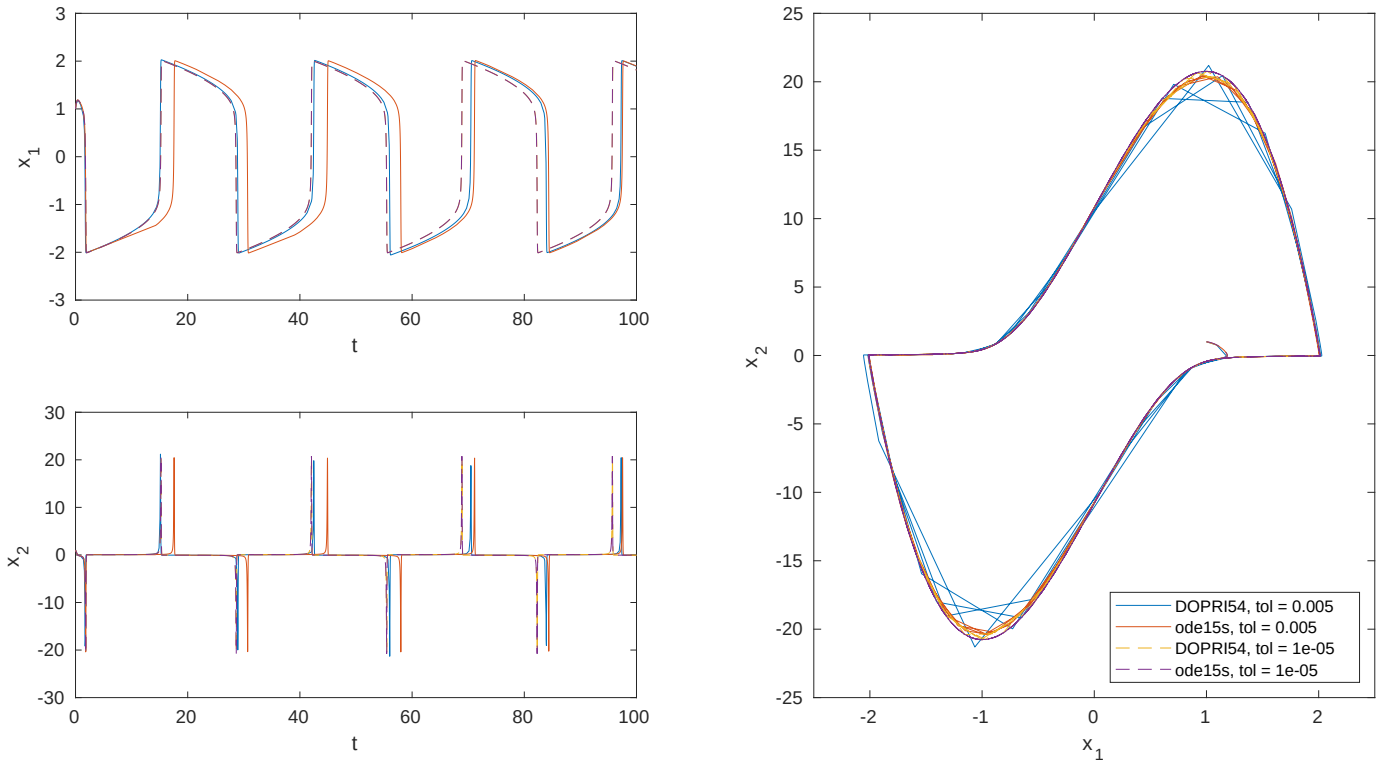
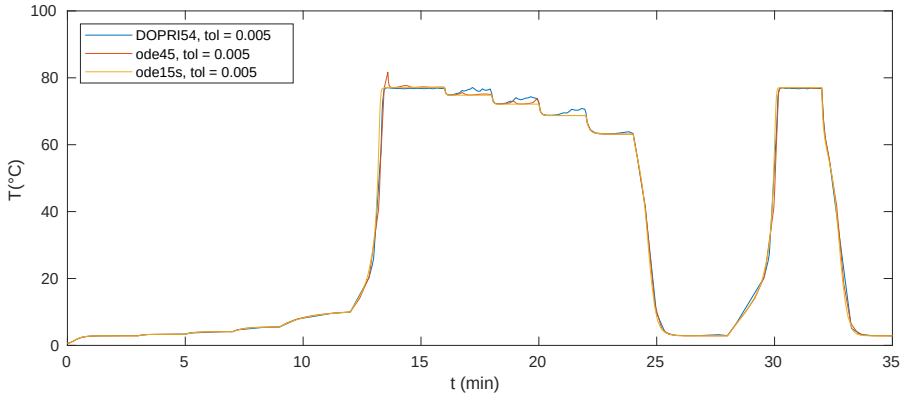


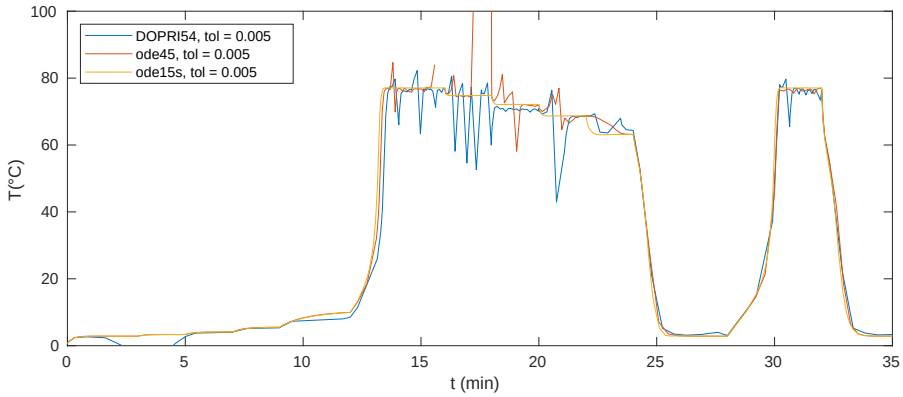
Figure 58: Solution for the Van der Pol problem ($\mu = 15$) using Dormand-Prince 5(4) vs. `ode15s`

Method	**DOPRI45**		**ode15s**	
Tolerances	0.005	1e-05	0.005	1e-05
Function evaluations	6612	8728	1538	3364
Calculated steps	957	1273	670	1699
Accepted steps	870	1090	499	1456
Rejected steps	87	183	171	243

Table 33: Parameters of the Dormand-Prince 5(4) vs. `ode15s` *for the Van der Pol problem ($\mu = 15$)*



(a) CSTR 3D problem



(b) CSTR 1D problem

Figure 59: Solution for the CSTR problem using Dormand-Prince 5(4) vs. `ode45` *and* `ode15s`

Method	**DOPRI45**	**ode45**	**ode15s**
Tolerances	0.005	0.005	0.005
Function evaluations	1507	1285	551
Calculated steps	223	1376	1634
Accepted steps	169	200	245
Rejected steps	54	12	31

Table 34: Parameters of the Dormand-Prince 5(4) vs. `ode45` *and* `ode15s` *for the CSTR-3D problem*

Method	**DOPRI45**	**ode45**	**ode15s**
Tolerances	0.005	0.005	0.005
Function evaluations	1419	1711	416
Calculated steps	210	1874	1398
Accepted steps	159	237	207
Rejected steps	51	46	29

Table 35: Parameters of the Dormand-Prince 5(4) vs. `ode45` *and* `ode15s` *for the CSTR-1D problem*

Part 8: ESDIRK23

8.1. Using the order conditions and other conditions derive the ESDIRK23 method.

We can characterize different classes of Runge-Kutta methods based on their A matrix in the Butcher Tableau. Explicit Runge-Kutta methods (ERK) have a strictly lower triangular A-matrix, implying that the inner calculations in the algorithm to obtain the next step can be done directly. The classical Runge-Kutta (Table 18) and Dormand-Prince 5(4) (Table 27) are some examples of ERK methods. However, just as we saw in previous parts, these methods suffer from stability limitations when applied to stiff problems.

The Fully Implicit Runge-Kutta methods (FIRK) are characterized by excellent stability properties making them a nice choice to solve these kind of problems. Their A-matrix is completely full. However, in each step of the method a system of $n \times s$ non-linear equations must be solved, normally by calling an iterative method and causing a huge computational cost. In the literature we can find different variations of the methods that try to achieve some of the stability properties of FIRK methods but reducing the computational cost, namely Diagonally Implicit Runge-Kutta (DIRK), Singly-Diagonally Implicit Runge-Kutta (SDIRK) and Explicit Singly-Diagonally Implicit Runge-Kutta (ESDIRK). [3]

In the ESDIRK familly of methods (Explicit Singly-Diagonally Implicit Runge-Kutta methods), the first step is explicit ($c_1 = 0$ and $a_{11} = 0$), the internal stages $2, \ldots, s$ are singly diagonally implicit and the last stage is the same as the next iteration first stage. The derivation of the ESDIRK23 method is pretty long and can be found in [3]. The Butcher Tableau of the method for $\gamma = \frac{2-\sqrt{2}}{2}$is:

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

Table 36: Butcher Tableau of the Explicit Singly-Diagonally Implicit Runge-Kutta 23 method

8.2. Plot the stability region of the ESDIRK23 method. Is it A-stable? Is it L-stable? Discuss the practical implications of the stability region of ESDIRK23

The stability region of the ESDIRK23 is calculated for the test equation using expression 5 and substituting its corresponding values of the Butcher Tableau. The result is shown in Figure 60.

Here, we can observe that the method is not only A-stable but also L-stable. This basically means that the method handles really good stiff problems as expected, and without the extra computational cost of Fully Implicit Runge-Kutta methods.

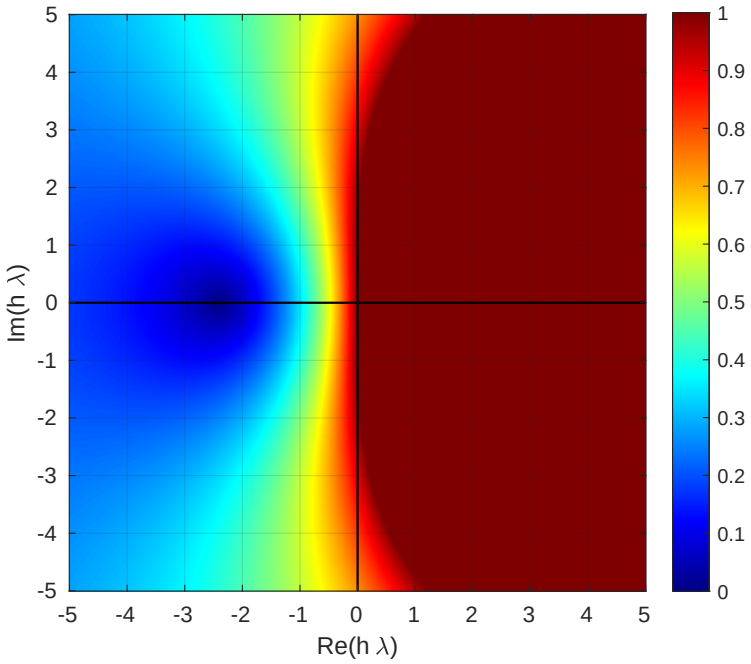


Figure 60: Values of $R(h\lambda)$ for the Explicit Singly-Diagonally Implicit Runge-Kutta 23 method

8.3. Implement ESDIRK23 with variable step size.

For this part, we'll use the ESDIRK library given in the course codes. This library lets you select from different ESDIRK methods and implements them with adaptive step size. THe implementation is shown below:

```
function [Tout,Xout,Gout,info,stats] = ESDIRK(fun,jac,t0,tf,x0,h0,absTol,relTol,Method,varargin)

%% ESDIRK23 Parameters
%=========================================================================
% Runge-Kutta method parameters
switch Method
    case 'ESDIRK12'
        gamma = 1;
        AT = [0 0;0 gamma];
        c  = [0; 1];
        b  = AT(:,2);
        bhat = [1/2; 1/2];
        d  = b-bhat;
        p  = 1;
        phat = 2;
        s = 2;
    case 'ESDIRK23'
        gamma = 1-1/sqrt(2);
        a31 = (1-gamma)/2;
        AT = [0 gamma a31;0 gamma a31;0 0 gamma];
        c  = [0; 2*gamma; 1];
        b  = AT(:,3);
        bhat = [    (6*gamma-1)/(12*gamma); ...
            1/(12*gamma*(1-2*gamma)); ...
            (1-3*gamma)/(3*(1-2*gamma))    ];
        d  = b-bhat;
        p  = 2;
        phat = 3;
        s = 3;
    case 'ESDIRK34'
        gamma = 0.43586652150845899942;
        a31 = 0.14076777472470619619;
        a32 = -0.1083655513813208000;
```

```
        AT  = [0 gamma a31   0.10239940061991099768;
            0 gamma a32   -0.3768784522555561061;
            0 0     gamma 0.83861253012718610911;
            0 0     0     gamma                 ];
        c  = [0; 0.87173304301691799883; 0.46823874485184439565; 1];
        b  = AT(:,4);
        bhat = [0.15702489786032493710;
            0.11733044137043884870;
            0.61667803039212146434;
            0.10896663037711474985];
        d = b-bhat;
        p = 3;
        phat = 4;
        s = 4;
end


% error and convergence controller
epsilon = 0.8;
tau = 0.1*epsilon; %0.005*epsilon;
itermax = 20;
ke0 = 1.0/phat;
ke1 = 1.0/phat;
ke2 = 1.0/phat;
alpharef = 0.3;
alphaJac = -0.2;
alphaLU  = -0.2;
hrmin = 0.01;
hrmax = 10;
%=========================================================================
tspan = [t0 tf]; % carsten
info = struct(...
            'Method',    Method,  ... % carsten
            'nStage',    s,       ... % carsten
            'absTol',    'dummy',  ... % carsten
            'relTol',    'dummy',  ... % carsten
            'iterMax',   itermax, ... % carsten
            'tspan',     tspan,   ... % carsten
            'nFun',      0, ...
            'nJac',      0, ...
            'nLU',       0, ...
            'nBack',     0, ...
            'nStep',     0, ...
            'nAccept',   0, ...
            'nFail',     0, ...
            'nDiverge',  0, ...
            'nSlowConv', 0);



%% Main ESDIRK Integrator
%=========================================================================
nx = size(x0,1);
F = zeros(nx,s);
t = t0;
x = x0;
h = h0;

[F(:,1),g]  = feval(fun,t,x,varargin{:});
info.nFun = info.nFun+1;
[dfdx,dgdx] = feval(jac,t,x,varargin{:});
info.nJac = info.nJac+1;
FreshJacobian = true;
if (t+h)>tf
    h = tf-t;
end
hgamma = h*gamma;
dRdx = dgdx - hgamma*dfdx;
[L,U,pivot] = lu(dRdx,'vector');
info.nLU = info.nLU+1;
hLU = h;
```

```
105
106 FirstStep = true;
107 ConvergenceRestriction = false;
108 PreviousReject = false;
109 iter = zeros(1,s);
110
111 % Output
112 chunk = 100;
113 Tout = zeros(chunk,1);
114 Xout = zeros(chunk,nx);
115 Gout = zeros(chunk,nx);
116
117 Tout(1,1) = t;
118 Xout(1,:) = x.';
119 Gout(1,:) = g.';
120
121 while t<tf
122     info.nStep = info.nStep+1;
123     %=========================================================================
124     % A step in the ESDIRK method
125     i=1;
126     diverging = false;
127     SlowConvergence = false; % carsten
128     alpha = 0.0;
129     Converged = true;
130     while (i<s) && Converged
131         % Stage i=2,...,s of the ESDIRK Method
132         i=i+1;
133         phi = g + F(:,1:i-1)*(h*AT(1:i-1,i));
134
135         % Initial guess for the state
136          if i==2
137              dt = c(i)*h;
138              G = g + dt*F(:,1);
139              X = x + dgdx\(G-g);
140          else
141              dt = c(i)*h;
142              G  = g + dt*F(:,1);
143              X  = x + dgdx\(G-g);
144          end
145         T = t+dt;
146
147         [F(:,i),G] = feval(fun,T,X,varargin{:});
148         info.nFun = info.nFun+1;
149         R = G - hgamma*F(:,i) - phi;
150 %         rNewton = norm(R./(absTol + abs(G).*relTol),2)/sqrt(nx);
151         rNewton = norm(R./(absTol + abs(G).*relTol),inf);
152         Converged = (rNewton < tau);
153         %iter(i) = 0; % original, if uncomment then comment line 154: iter(:) = 0;
154         % Newton Iterations
155         while ~Converged && ~diverging && ~SlowConvergence%iter(i)<itermax
156             iter(i) = iter(i)+1;
157             dX = U\(L\(R(pivot,1)));
158             info.nBack = info.nBack+1;
159             X = X - dX;
160             rNewtonOld = rNewton;
161             [F(:,i),G] = feval(fun,T,X,varargin{:});
162             info.nFun = info.nFun+1;
163             R = G - hgamma*F(:,i) - phi;
164 %             rNewton = norm(R./(absTol + abs(G).*relTol),2)/sqrt(nx);
165             rNewton = norm(R./(absTol + abs(G).*relTol),inf);
166             alpha = max(alpha,rNewton/rNewtonOld);
167             Converged = (rNewton < tau);
168             diverging = (alpha >= 1);
169             SlowConvergence = (iter(i) >= itermax); % carsten
170             %SlowConvergence = (alpha >= 0.5); % carsten
171             %if (iter(i) >= itermax), i, iter(i), Converged, diverging, pause, end % carsten
172         end
173         %diverging = (alpha >= 1); % original, if uncomment then comment line 142: diverging = ...
                (alpha >= 1)*i;
174         diverging = (alpha >= 1)*i; % carsten, recording which stage is diverging
```

```
175     end
176     %if diverging, i, iter, pause, end
177     nstep = info.nStep;
178     stats.t(nstep) = t;
179     stats.h(nstep) = h;
180     stats.r(nstep) = NaN;
181     stats.iter(nstep,:) = iter;
182     stats.Converged(nstep) = Converged;
183     stats.Diverged(nstep)  = diverging;
184     stats.AcceptStep(nstep) = false;
185     stats.SlowConv(nstep)  = SlowConvergence*i; % carsten, recording which stage is converging ...
            to slow (reaching maximum no. of iterations)
186     iter(:) = 0; % carsten
187     %=========================================================================
188     % Error and Convergence Controller
189     if Converged
190         % Error estimation
191         e = F*(h*d);
192 %        r = norm(e./(absTol + abs(G).*relTol),2)/sqrt(nx);
193         r = norm(e./(absTol + abs(G).*relTol),inf);
194         CurrentStepAccept = (r<=1.0);
195         r = max(r,eps);
196         stats.r(nstep) = r;
197         % Step Length Controller
198         if CurrentStepAccept
199             stats.AcceptStep(nstep) = true;
200             info.nAccept = info.nAccept+1;
201             if FirstStep || PreviousReject || ConvergenceRestriction
202                 % Aymptotic step length controller
203                 hr = 0.75*(epsilon/r)^ke0;
204             else
205                 % Predictive controller
206                 s0 = (h/hacc);
207                 s1 = max(hrmin,min(hrmax,(racc/r)^ke1));
208                 s2 = max(hrmin,min(hrmax,(epsilon/r)^ke2));
209                 hr = 0.95*s0*s1*s2;
210             end
211             racc = r;
212             hacc = h;
213             FirstStep = false;
214             PreviousReject = false;
215             ConvergenceRestriction = false;
216
217             % Next Step
218             t = T;
219             x = X;
220             g = G;
221             F(:,1) = F(:,s);
222
223         else % Reject current step
224             info.nFail = info.nFail+1;
225             if PreviousReject
226                 kest = log(r/rrej)/(log(h/hrej));
227                 kest = min(max(0.1,kest),phat);
228                 hr   = max(hrmin,min(hrmax,((epsilon/r)^(1/kest))));
229             else
230                 hr = max(hrmin,min(hrmax,((epsilon/r)^ke0)));
231             end
232             rrej = r;
233             hrej = h;
234             PreviousReject = true;
235         end
236
237         % Convergence control
238         halpha = (alpharef/alpha);
239         if (alpha > alpharef)
240             ConvergenceRestriction = true;
241             if hr < halpha
242                 h = max(hrmin,min(hrmax,hr))*h;
243             else
244                 h = max(hrmin,min(hrmax,halpha))*h;
```

```
245             end
246         else
247             h = max(hrmin,min(hrmax,hr))*h;
248         end
249         h = max(1e-8,h);
250         if (t+h) > tf
251             h = tf-t;
252         end
253 
254         % Jacobian Update Strategy
255         FreshJacobian = false;
256         if alpha > alphaJac
257             [dfdx,dgdx] = feval(jac,t,x,varargin{:});
258             info.nJac = info.nJac+1;
259             FreshJacobian = true;
260             hgamma = h*gamma;
261             dRdx = dgdx - hgamma*dfdx;
262             [L,U,pivot] = lu(dRdx,'vector');
263             info.nLU = info.nLU+1;
264             hLU = h;
265         elseif (abs(h-hLU)/hLU) > alphaLU
266             hgamma = h*gamma;
267             dRdx = dgdx-hgamma*dfdx;
268             [L,U,pivot] = lu(dRdx,'vector');
269             info.nLU = info.nLU+1;
270             hLU = h;
271         end
272     else % not converged
273         info.nFail=info.nFail+1;
274         CurrentStepAccept = false;
275         ConvergenceRestriction = true;
276         if FreshJacobian && diverging
277             h = max(0.5*hrmin,alpharef/alpha)*h;
278             info.nDiverge = info.nDiverge+1;
279         elseif FreshJacobian
280             if alpha > alpharef
281                 h = max(0.5*hrmin,alpharef/alpha)*h;
282             else
283                 h = 0.5*h;
284             end
285         end
286         if ~FreshJacobian
287             [dfdx,dgdx] = feval(jac,t,x,varargin{:});
288             info.nJac = info.nJac+1;
289             FreshJacobian = true;
290         end
291         hgamma = h*gamma;
292         dRdx = dgdx - hgamma*dfdx;
293         [L,U,pivot] = lu(dRdx,'vector');
294         info.nLU = info.nLU+1;
295         hLU = h;
296     end
297 
298     %=========================================================================
299     % Storage of variables for output
300 
301     if CurrentStepAccept
302        nAccept = info.nAccept;
303        if nAccept > length(Tout);
304            Tout = [Tout; zeros(chunk,1)];
305            Xout = [Xout; zeros(chunk,nx)];
306            Gout = [Gout; zeros(chunk,nx)];
307        end
308        Tout(nAccept,1) = t;
309        Xout(nAccept,:) = x.';
310        Gout(nAccept,:) = g.';
311     end
312 end
313 info.nSlowConv = length(find(stats.SlowConv)); % carsten
314 nAccept = info.nAccept;
315 Tout = Tout(1:nAccept,1);
```

```
316  Xout = Xout(1:nAccept,:);
317  Gout = Gout(1:nAccept,:);
318
319  end
```

Listing 18: ESDIRK23 method with adaptive time step size

8.4. Test your algorithms on the Van der Pol problem ($\mu = 1.5$ and $\mu = 15$, $\mathbf{x_0} = [1.0; 1.0]$).

The results for the Van der Pol problem using ESDIRK 23 method are shown below. We can see that the convergence of the solution is pretty good, but what's more interesting is the improvement in the "stiff" version of the problem ($\mu = 15$). Here, the method uses even less function evaluations than in the non-stiff problem, achieving also a great convergence. The only drawback of the method is working with non-stiff problems. Comparing Table 37 with the results for Dormand-Prince 5(4) (Table 28), we can see how the number of evaluations increases considerably. For this type of problems, we can achieve better accuracy using an explicit method.

It's worth mentioning that the values of r shown go past 1 because the algorithm used (18) stores every value of h and r, not only the ones accepted.

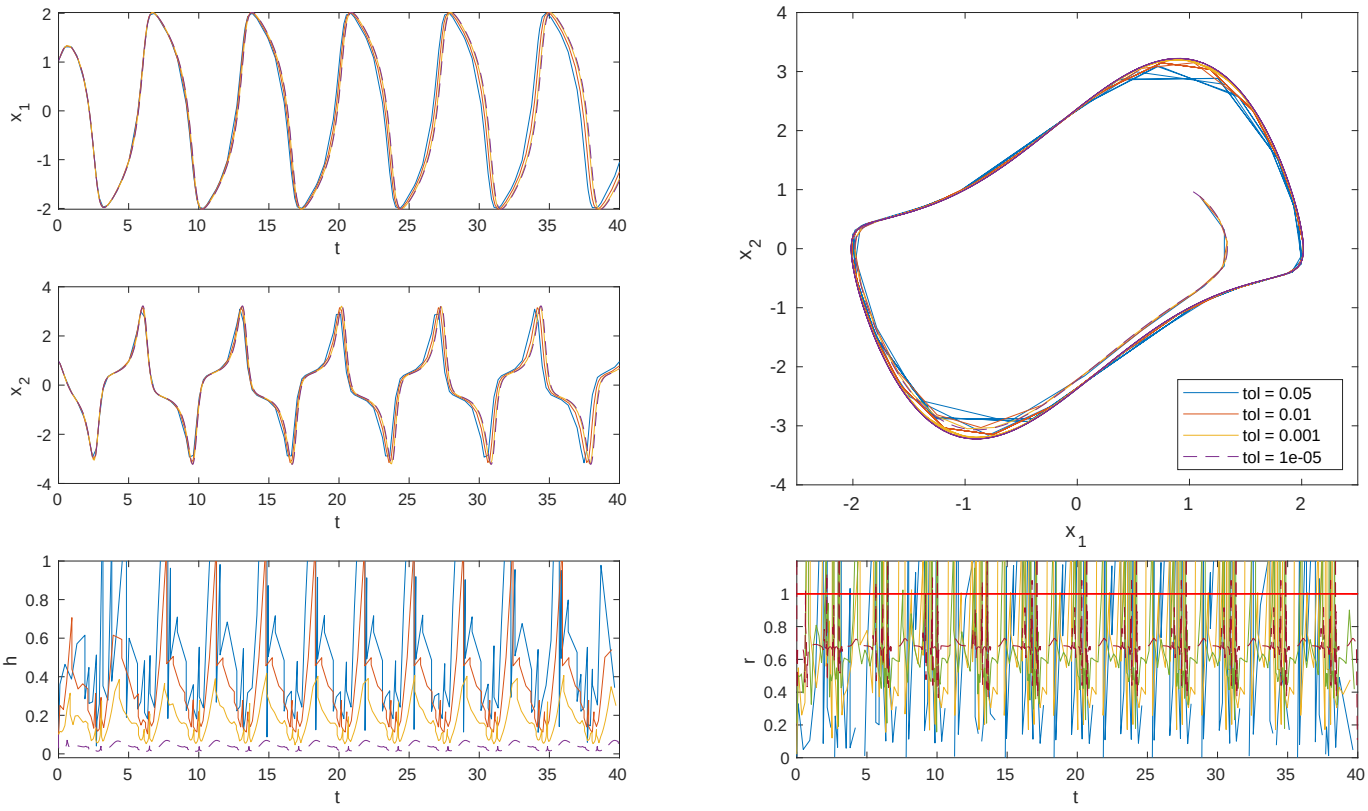


Figure 61: Solution for the Van der Pol problem ($\mu = 1.5$) using ESDIRK23 with adaptive step size

Tolerances	0.05	0.01	0.001	1e-05
Function evaluations	1196	1417	1864	6197
Calculated steps	178	220	356	1397
Accepted steps	117	158	312	1372
Rejected steps	61	62	44	25

Table 37: Parameters of the ESDIRK23 with adaptive step size for the Van der Pol problem ($\mu = 1.5$)

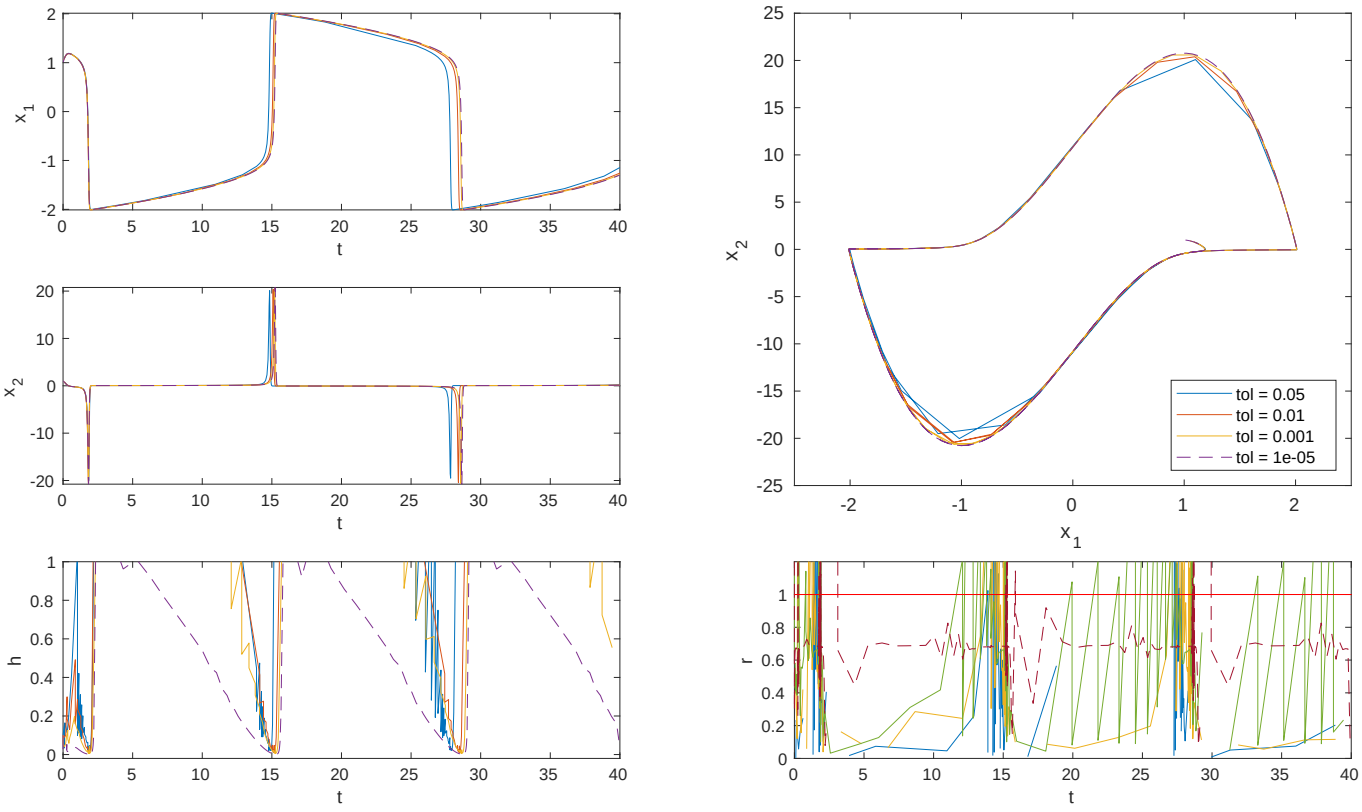


Figure 62: Solution for the Van der Pol problem ($\mu = 15$) using ESDIRK23 with adaptive step size

Tolerances	0.05	0.01	0.001	1e-05
Function evaluations	740	814	1325	3763
Calculated steps	115	127	223	802
Accepted steps	78	98	188	782
Rejected steps	37	29	35	20

Table 38: Parameters of the ESDIRK23 with adaptive step size for the Van der Pol problem ($\mu = 15$)

8.5. Test your algorithms on the adiabatic CSTR problem described in the papers uploaded to Learn (3D-version and 1D-version).

Not surprisingly, the ESDIRK 23 also shows great performance on the CSTR problem. It captures the solution greatly, and handles the stiffness when F is low amazingly.

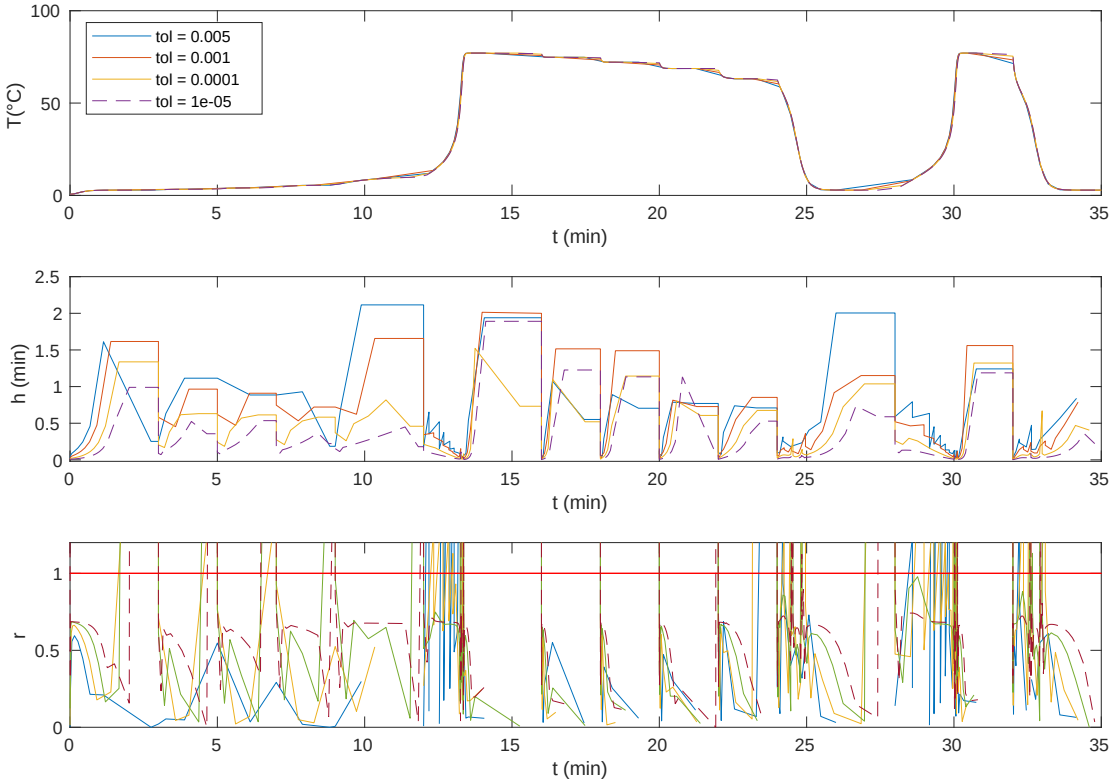


Figure 63: Solution for the CSTR 3D problem using ESDRIK 23 with adaptive step size

Tolerances	0.005	0.001	0.0001	1e-05
Function evaluations	878	1095	1497	2454
Calculated steps	129	171	267	495
Accepted steps	90	120	227	455
Rejected steps	39	51	40	40

Table 39: Parameters of the ESDIRK 23 with adaptive step size for the CSTR 3D problem

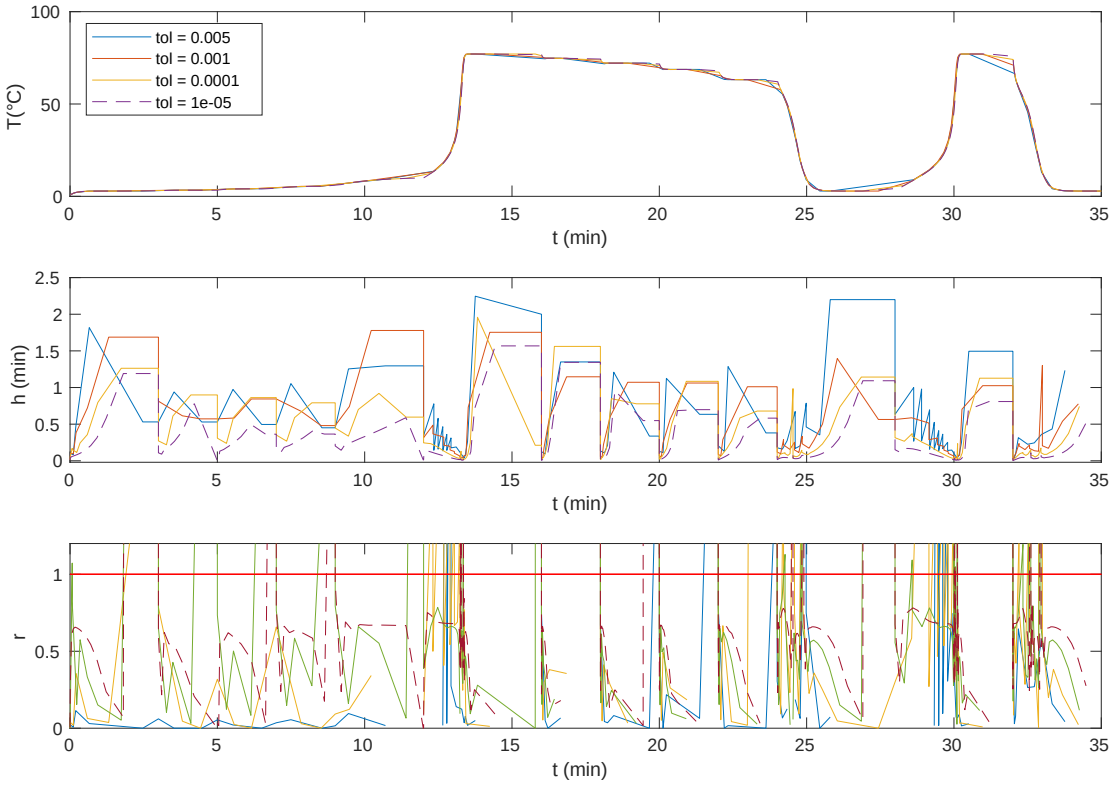


Figure 64: Solution for the CSTR 1D problem using ESDIRK 23 with adaptive step size

Tolerances	0.005	0.001	0.0001	1e-05
Function evaluations	741	888	1226	1802
Calculated steps	107	130	201	339
Accepted steps	78	91	157	305
Rejected steps	29	39	44	34

Table 40: Parameters of the ESDIRK 23 with adaptive step size for the CSTR 1D problem

8.6. Compare the solution and the number of function evaluations with your own explicit Runge-Kutta method to the other solvers that you have used in this exam problem. Discuss when it is appropriate to use ESDIRK23 and illustrate that with an example you choose.

For the sake of comparison, we will compare the ESDIRK23 with the same Matlab solvers we already used for the entirety of the project. We're interested in seeing if the ESDIRK23 could be a good alternative to the `ode15s` for the stiff cases while still being usable for non-stiff problems (compared to `ode45`).

The results are very promising. The ESDIRK23 manages to achieve a convergent solution in the non-stiff problem (Figure 65) without a huge increase in the number of function evaluations, specially if we take into account the huge difference between calculated steps. In the stiff Van der Pol problem it still has some more function evaluations than the `ode15s` , but nothing considering the difference all the other explicit methods had.

It's also curious to see how in the CSTR problem the method performs a lot less steps than the other ODEs, achieving overall good performance and failing a bit on the steps. This behaviour is easily solved by cranking a little bit tighter the tolerance. Overall, we can conclude that the ESDIRK23 achieves a good balance between accuracy and computation for both stiff and non-stiff problems.

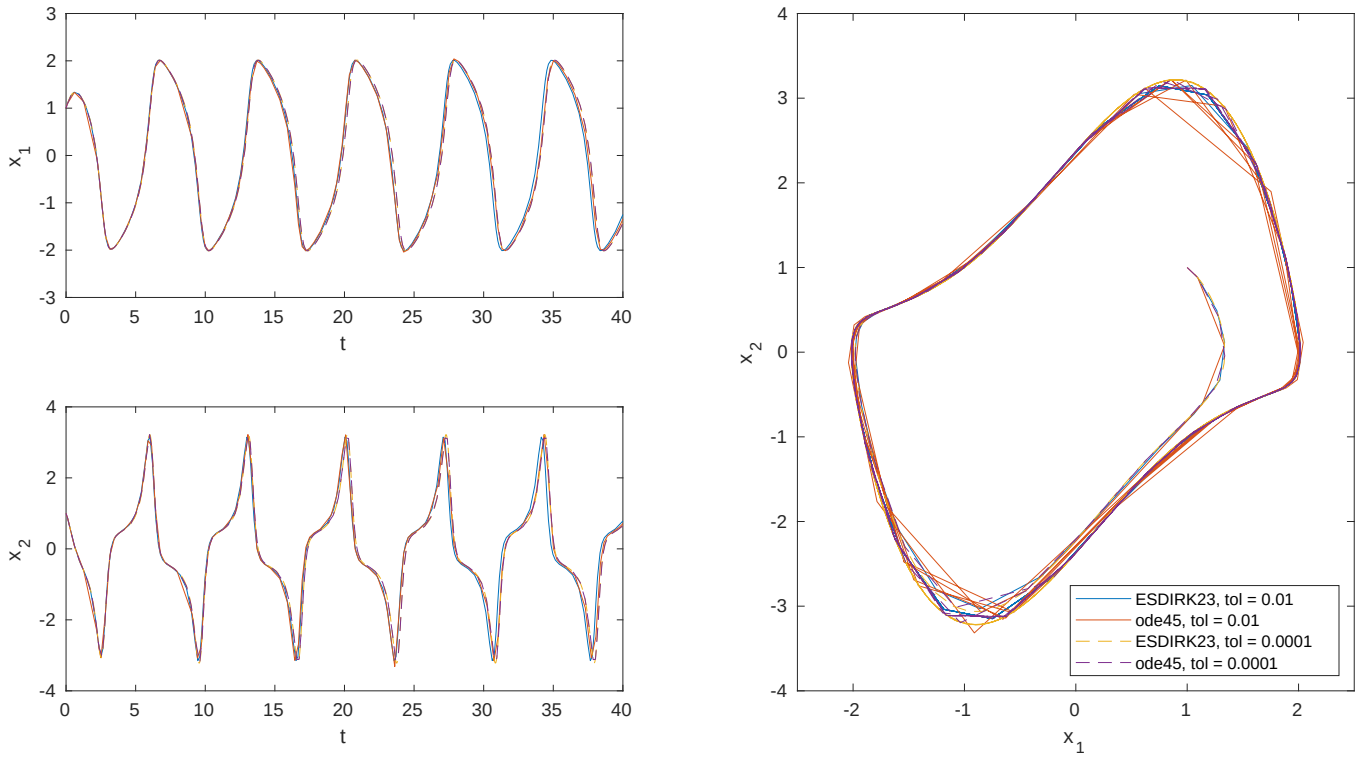


Figure 65: Solution for the Van der Pol problem ($\mu = 1.5$) using ESDIRK23 vs. `ode45`

Method	**ESDIRK23**		**ode45**	
Tolerances	0.01	0.0001	0.01	0.0001
Function evaluations	1417	3250	787	1357
Calculated steps	220	679	131	226
Accepted steps	158	645	100	180
Rejected steps	62	34	31	46

Table 41: Parameters of the ESDIRK23 vs. `ode45` *for the Van der Pol problem ($\mu = 1.5$)*

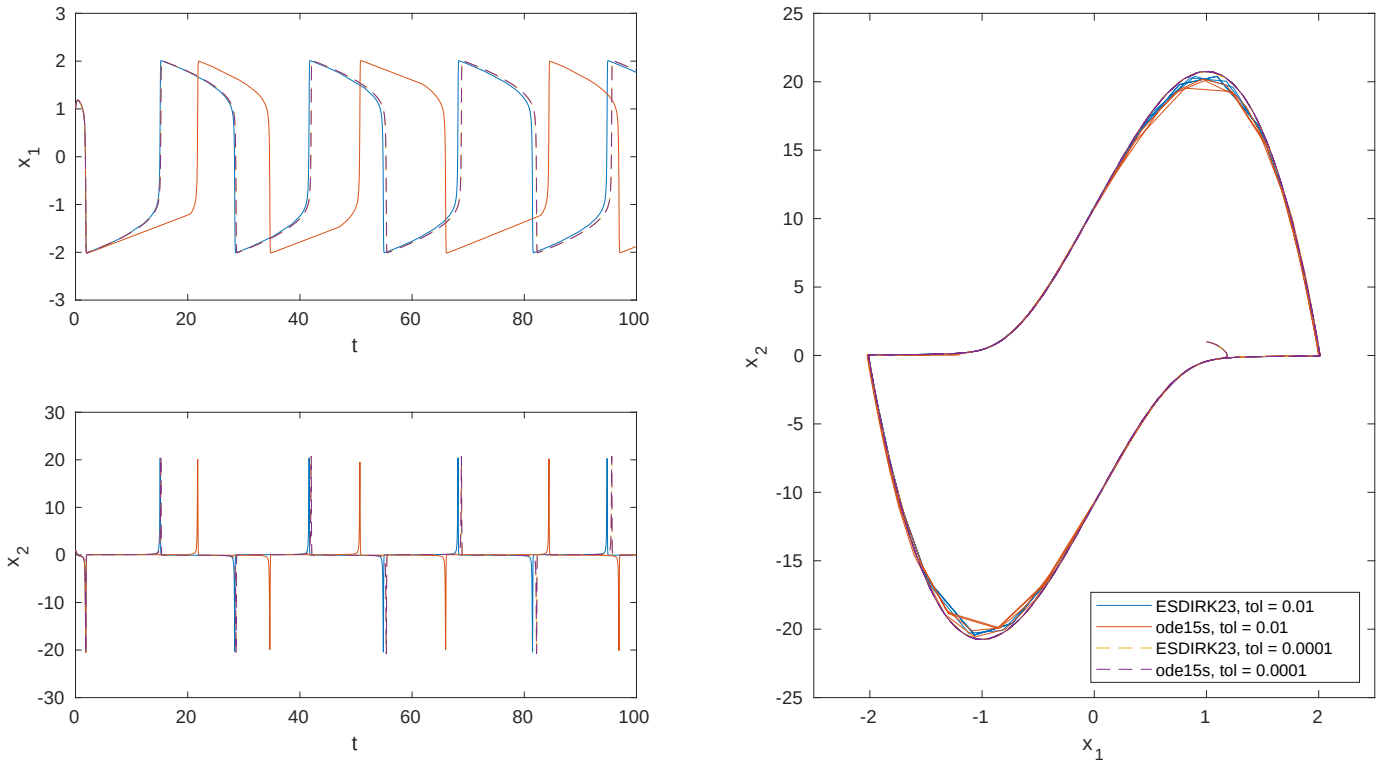
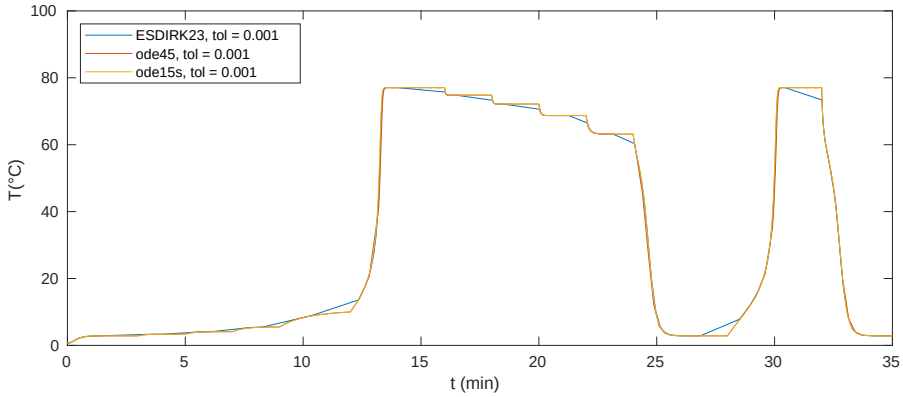


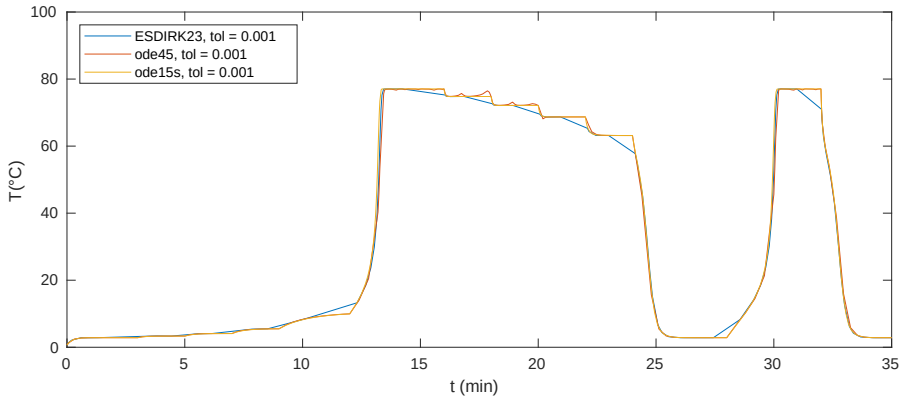
Figure 66: Solution for the Van der Pol problem ($\mu = 15$) using ESDIRK23 vs. `ode15s`

Method	**ESDIRK23**		**ode15s**	
Tolerances	0.01	0.0001	0.01	0.0001
Function evaluations	2112	5111	1273	2780
Calculated steps	327	990	558	1327
Accepted steps	250	947	411	1094
Rejected steps	77	43	147	233

Table 42: Parameters of the ESDIRK23 vs. `ode15s` *for the Van der Pol problem ($\mu = 15$)*



(a) CSTR 3D problem



(b) CSTR 1D problem

Figure 67: Solution for the CSTR problem using ESDIRK23 vs. `ode45` *and* `ode15s`

Method	**ESDIRK23**	**ode45**	**ode15s**
Tolerances	0.001	0.001	0.001
Function evaluations	1095	1447	770
Calculated steps	171	1520	2248
Accepted steps	120	210	340
Rejected steps	51	29	62

Table 43: Parameters of the ESDIRK23 vs. `ode45` *and* `ode15s` *for the CSTR-3D problem*

Method	**ESDIRK23**	**ode45**	**ode15s**
Tolerances	0.001	0.001	0.001
Function evaluations	888	1315	538
Calculated steps	130	1395	1604
Accepted steps	91	195	241
Rejected steps	39	22	48

Table 44: Parameters of the ESDIRK23 vs. `ode45` *and* `ode15s` *for the CSTR-1D problem*

Part 9: Discussion and Conclusions

9.1. Discuss and compare the different solution methods for the test problem used (Van der Pol and CSTR).

We've already commented out the results of each individual method in its corresponding part. We've also compared them indirectly when discussing each particular solution. This project's findings agree with what is stated in the literature: explicit methods have problems of stability when applied to stiff ODE systems and implicit methods can be used to overcome these problems. However, solving a non-linear system of equations is required in the internal steps of implicit methods, usually solved by calling an iterative method which increases the number of function calls, creating an overhead. Thus, implicit methods are only useful when working in stiff problems with really low tolerances, where we are interested in having a very precise solution.

During the project we've looked at different Explicit methods (Explicit Euler, classical Runge-Kutta and Dormand-Prince 5(4)) and we've seen how the convergence of the solution gets better as we chose methods of higher orded. However, all these methods perform poorly when applied to stiff problems so, we tried two different implicit architectures, Implicit Euler and Explicit Simply-Diagonally Implicit Runge-Kutta 2(3) method. Both achieve overall good performance when working with stiff problems. However, the magic of the ESDIRK23 is that it manages to also keep the computational cost pretty low for non-stiff problems.

In this chapter, we are interested in the performance of the models. In order to achieve this, we'll take a look at how the different stats of every model evolve through different tolerances when applied to each particular problem. The results are shown in the following figures.

The first realisation we can make is that the ODE solvers from Matlab achieve very good performance in the Van der Pol problem (low no. of function evaluations), both in the stiff and in the non-stiff case. In the CSTR problem, however, their no. of calculated steps is ridiculously high, which makes them worse in the no. of function evaluations, of course. This behaviour is quite strange, and it could be caused by several reasons. It could be that the solvers from Matlab are optimised for certain, more common, problems. It could also be the fact that, due to the way the flow function has been defined with Matlab, the last hidden state is not passed to the next iteration of the flow function because we cannot access these values. Thus, the initial hidden state is reseted every time there's a change in the flow function. This, however, only occurs 20+ times which is not enough to justify a deviation this huge.

Taking a look at our implemented methods, the ESDIRK23 and DOPRI4(5) are the ones that achieve best performance. This is no surprise and was already observed in their corresponding parts. However, the slope of function evaluations is a little bit higher on the ESDIRK23 than on the DOPRI54, caused by the fact that the latter is fully explicit while the ESDIRK23 is not. In the stiff Van der Pol (Figure 69), as well as in the CSTR cases but less noticeable, we can see how the performance of the ESDIRK outcomes the DOPRI54 for large tolerances. We've already shown that for these tolerances both methods have pretty good convergence.

It's also worth noticing how both Explicit and Implicit Euler have similar slopes in all the problems, and they are the worst performing ones, both in the number of function evaluations and in convergence as shown before. The classical Runge-Kutta, although its simplicity, manages to have a very good performance overall, both in computations and convergence of the solution.

In conclusion, this project has shown the enormous potential of the Runge-Kutta family of methods. It has also shown the difficulty of dealing with stiff problems, and how to adapt to each different problem by using explicit-implicit methods. Among the ODE solvers implemented for the project we can find Explicit Euler, Implicit Euler, Classical Runge-Kutta, Dormand-Prince 5(4) and ESDIRK23, aswell as SDE solvers: Euler-Maruyama and Implicit-Explicit. We analysed their stability regions and tested them on two different problems (Van der Pol and CSTR, each one with two cases). Lastly, we've developed the idea of using a semi-implicit method in order to find the balance between computational cost and better handling stiffness, with overal good results.

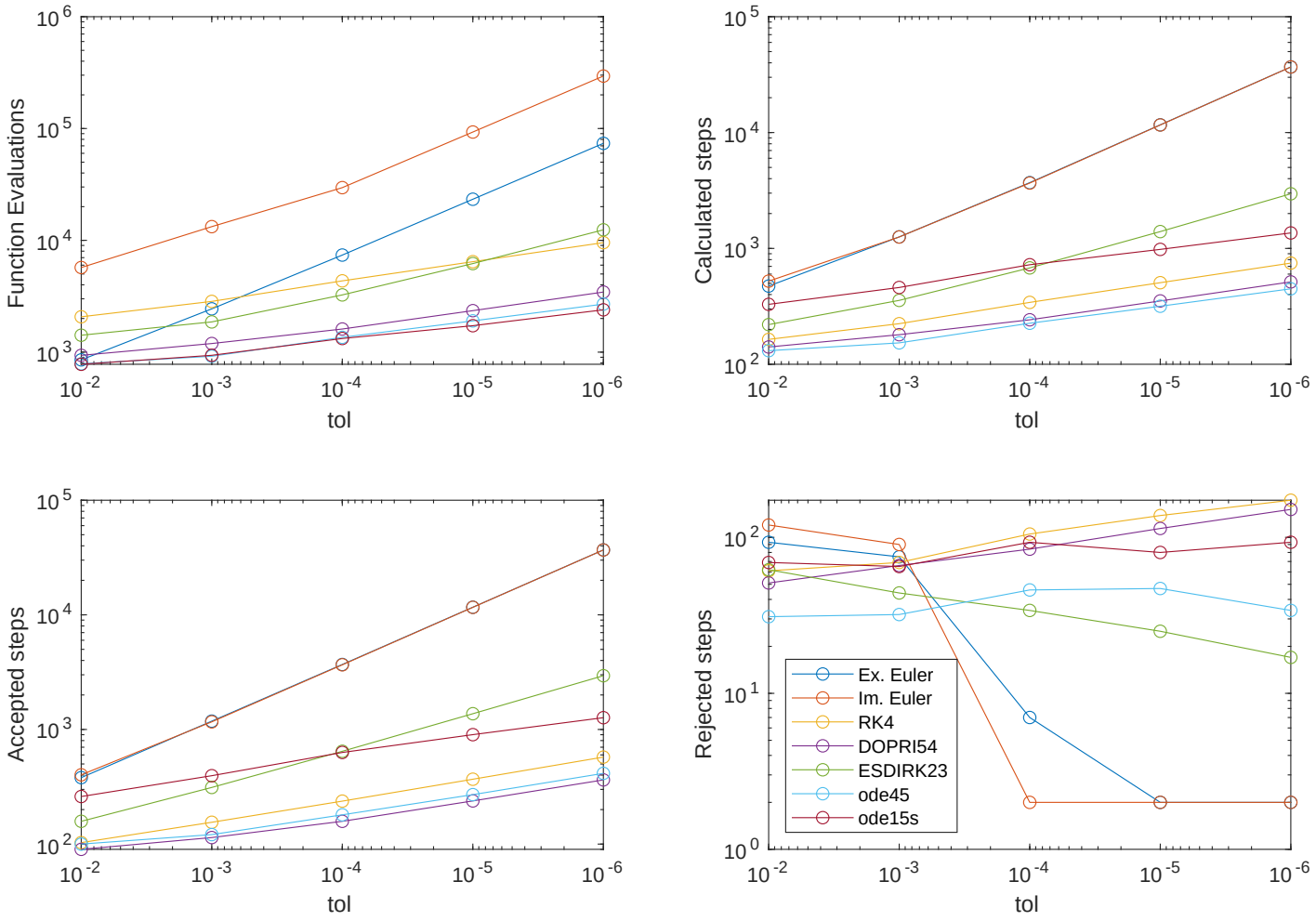


Figure 68: Performance comparison for the Van der Pol problem ($\mu = 1.5$)

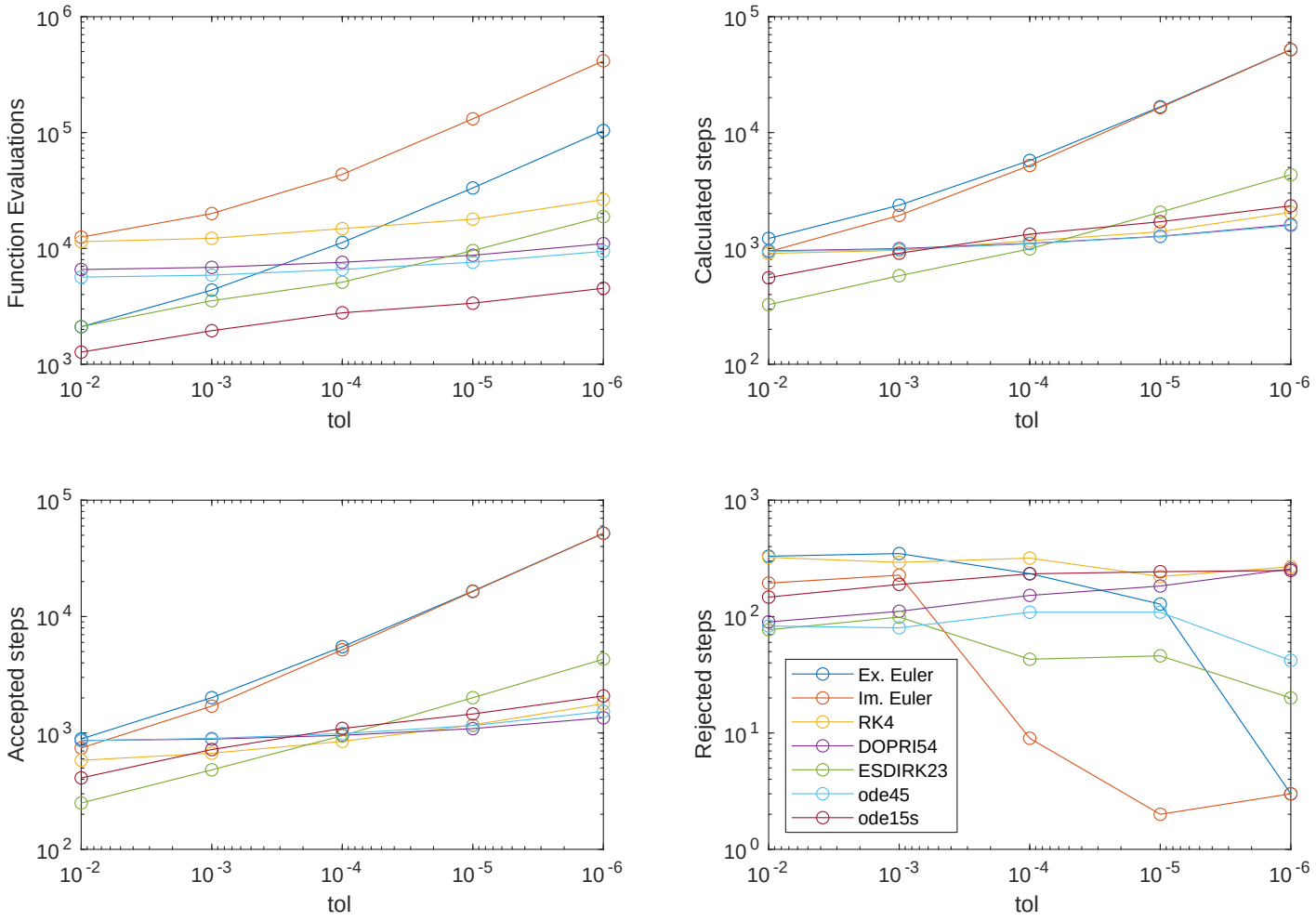


Figure 69: Performance comparison for the Van der Pol problem ($\mu = 15$)

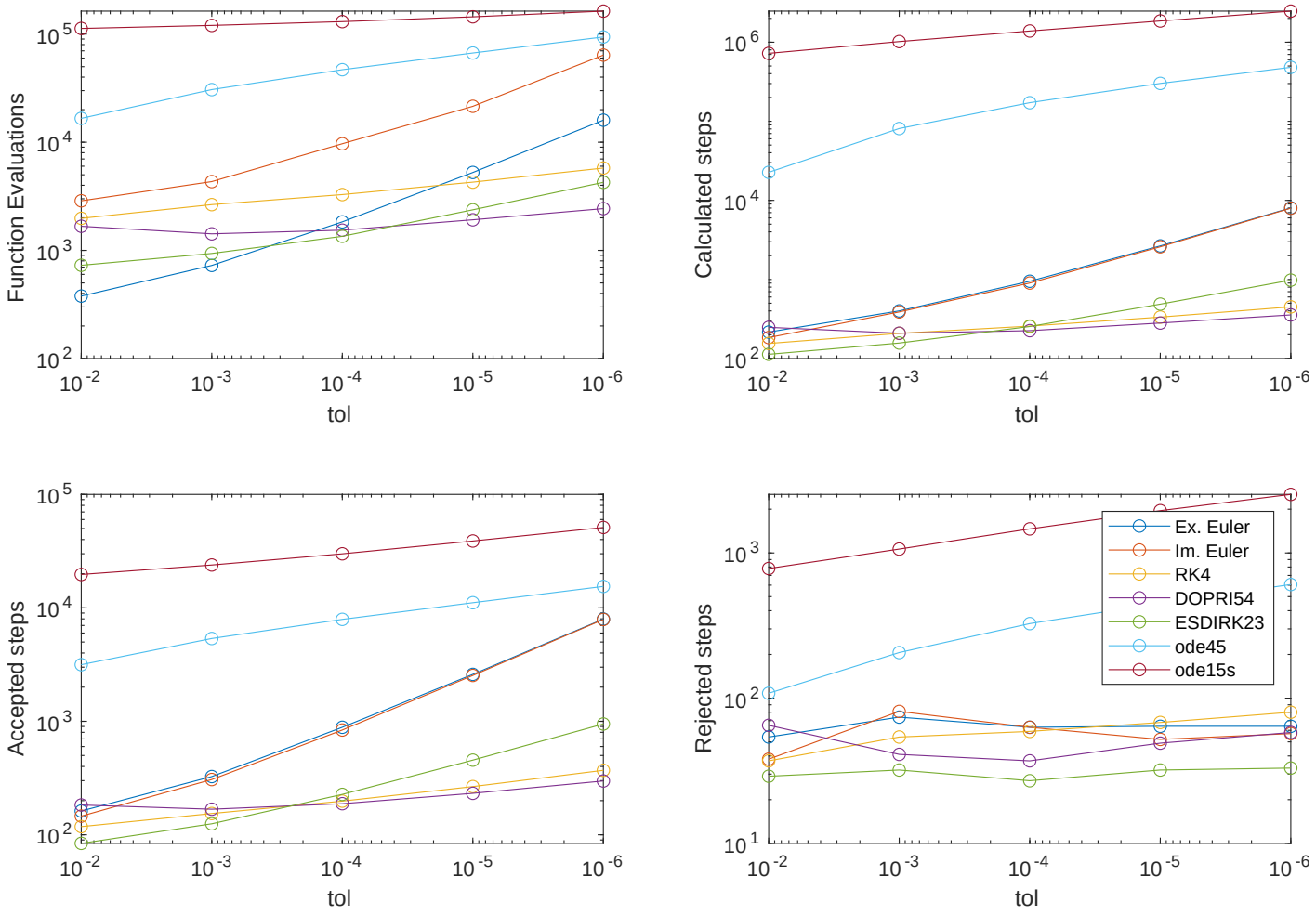


Figure 70: Performance comparison for the CSTR-3D problem

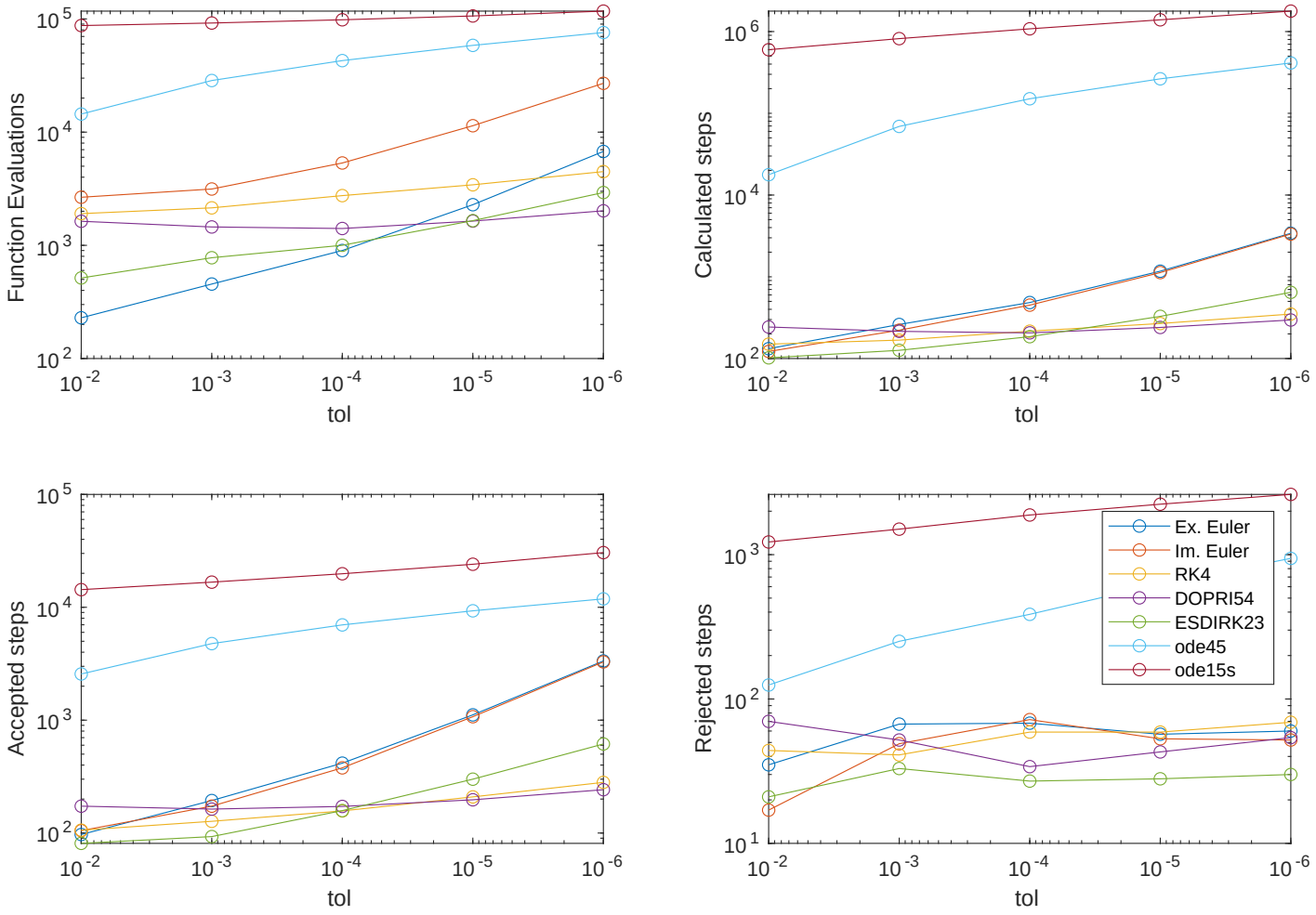


Figure 71: Performance comparison for the CSTR-1D problem

Appendix

This report, as well as all the Matlab codes, figures and results included in this project have been created by Jorge Sintes Fernández and are available in the GitHub repository: `https://github.com/JorgeSintes/scientific_computing`

References

[1] Uri M. Ascher and Linda R. Petzold. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. Society for Industrial and Applied Mathematics, 1998. ISBN: 9780898714128.

[2] Germund Dahlquist. "Convergence and stability in the numerical integration of ordinary differential equations". In: *Mathematica Scandinavica* 4 (Dec. 1956), pp. 33–53. DOI: 10.7146/math.scand.a-10454. URL: https://www.mscand.dk/article/view/10454.

[3] John Bagterp Jørgensen, Morten Rode Kristensen, and Per Grove Thomsen. "A Family of ESDIRK Integration Methods". In: (2018). arXiv: 1803.01613 [math.NA].

[4] John Bagterp Jørgensen et al. "Simulation of NMPC for a Laboratory Adiabatic CSTR with an Exothermic Reaction". In: (2020), pp. 202–207. DOI: 10.23919/ECC51009.2020.9143733.

[5] Morten Ryberg Wahlgreen et al. "Nonlinear Model Predictive Control for an Exothermic Reaction in an Adiabatic CSTR". In: *IFAC-PapersOnLine* 53.1 (2020). 6th Conference on Advances in Control and Optimization of Dynamical Systems ACODS 2020, pp. 500–505. ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2020.06.084. URL: https://www.sciencedirect.com/science/article/pii/S2405896320301038.