02685

Scientific Computing for Differential Equations

Assignment 2

Phillip Brinck Vetter (s144097)
Raja Shan Zaker Mahmood (s144102)
Steffen Sloth (s144101)

Group 1
March 19, 2018
Technical University of Denmark

Mini Project 1

1 Explicit ODE solver

We consider the Initial Value Problem (IVP).

$$\dot{x}(t) = f(t, x(t), p) \qquad x(t_0) = x_0 \tag{1}$$

Where $x \in \mathcal{R}^{n_x}$ and a the parameter $p \in \mathcal{R}^{n_p}$. For the subsequent considerations, we will assume that f is sufficiently smooth. We discretize the time interval, forming a mesh. We call the endpoint t_f and discretize the interval into N points.

$$t_0 < t_1 < ... < t_N = t_f \tag{2}$$

At the k'th point, we therefore have stepsize $h_k = t_k - t_{k-1}$.

At each point in the interval, we construct an approximation x_k to the function value at the point $x(t_k)$. So this gives us an array of values $x_0, ..., x_N$ that constitute an approximation to the solution to the IVP. We want to find an iterative starting at x_0 and moving forward.

1,1)

We can expand the wanted approximation in terms of a Taylor series. Using that the desired point t_k can be written $t_{k-1} + h_k$.

$$x(t_{k-1} + h_k) = x(t_{k-1}) + h_k x'(t_{k-1}) + \frac{1}{2} h_k^2 x''(t_{k-1})... \tag{3}$$

The truncation of this series after the first derivative gives us the Forward Euler Method by using the ODE in 1 to replace the first derivative with the right hand function.

$$x_k = x_{k-1} + h_k f(t_{k-1}, x_{k-1}) \tag{4}$$

This is an explicit method, in that each step is determined by earlier steps. It is also a one step method as it is self starting from the IC. By including more terms in the series in equation 3, we arrive at the Taylor series methods. As this algorithm uses fixed step size, we can specify this in a program, as well as the number of equations to be solved concurrently. The algorithm itself functions as a loop, with each step forward in time, adding the step size times the function evaluated at the previous step. We see directly from equation 3 and Taylors Theorem, that the one step error of Eulers method is $O(h^2)$. This means that the LTE and GTE is $O(h)$ and the method has an order of 1. There are a number of other ways of arriving at this formula. From the first part of the course, we learned to derive it from a finite difference approximation to the derivative. This lends itself well to when we discuss the LTE of the method. A very graphical and intuitive way would be to integrate equation 1 directly, using a rectange approximation to the integral on the right hand side.

1,2)

The Matlab implementation is straightforward. One thing to bear in mind, is that Matlab begins vector indices at 1 instead of 0. We include stats to be able to track how many function evaluations are used when running the algorithm.

```
function [T,X,stat] = ExplicitEuler(fun,tspan,N,x0,varargin)

%Memory Allocation
X = nan(length(x0),N);
T = nan(1,N);

stat.nfun = 0;  %Function evaluations
stat.lerror = []; %local error
stat.gerror = []; %glocal error

%Time step and initial conditions
dt = ( tspan(2)-tspan(1) ) / N;
X(:,1) = x0;
T(1) = tspan(1);

% Explicit Euler Algorithm with stats.
for n = 1:N
    X(:,n+1) = X(:,n) + dt*feval(fun,T(n),X(:,n),varargin{:});
    T(n+1) = T(n) + dt;

    stat.nfun = stat.nfun + 1;
end

end
```

1,3)

The Explicit Euler method can be augmented with an adaptive time step and with step doubling error estimation. This allows the algorithm to adjust the time step to fit the problem at hand. We use two different measure of tolerance, the absolute and the relative tolerance. The definition of them follows from the nomenclature. When we mention tolerance without quantification, we mean that both tolerances are set to the number mentioned.

```
function [T,X,stat] = ExplicitEulerAdaptive(fun,tspan,x0,h0,abstol,reltol,varargin)

%Error Control Parameters
eps = 0.8;
facmin = 0.1;
facmax = 5;

%Initialization
h = h0;
t0 = tspan(1);
tf = tspan(2);

%Start values
x(:,1) = x0;
t = t0;

%Output
X = x;
T = t;
stat.feval = 0;
stat.iter = 0;
stat.naccept = 0;
```

```
stat.nreject = 0;
stat.h = h;
stat.r = 0;

while t < tf
    %Final time step
    if tf < t + h
        h = tf - t;
    end

    %Function Evluation
    f = feval(fun,t,x,varargin{:});
    stat.feval = stat.feval + 1;

    AcceptStep = false;
    while ~AcceptStep
    stat.iter = stat.iter + 1;

    %Entire Step h
    xe = x + h*f;

    %1. Double Step h/2
    hm = 0.5*h;
    xm = x + hm*f;

    %2. Double Step h/2
    tm = t + hm;
    xf = xm + hm * feval(fun,tm,xm,varargin{:});
    stat.feval = stat.feval + 1;

    %Estimate Error : Entire Step vs Double Step
    e = xf - xe;
    r = max( abs(e) ./  max( abstol , abs(xf).*reltol ) );

    %Update if Error O.K otherwise change h and retry
    AcceptStep = ( r <= 1 );
    if AcceptStep
        t = t + h;
        x = xf;

        X = [X,x];
        T = [T,t];
        stat.h = [stat.h h];
        stat.r = [stat.r r];
        stat.naccept = stat.naccept + 1;
    else
        stat.nreject = stat.nreject + 1;
    end

    %Step Size Controller
    h_update = max( facmin , min( sqrt(eps/r) , facmax ) ) * h;
    h = h_update;

    end

end
end
```

1,4)

We now test the Euler Method, both with fixed and adaptive step sizes. The first problem we will look at is the Van der Pol oscillator, with IC's $[0.5; 0.5]$, as well as two different value for the parameter μ. The results are shown in figure 1 and 2.

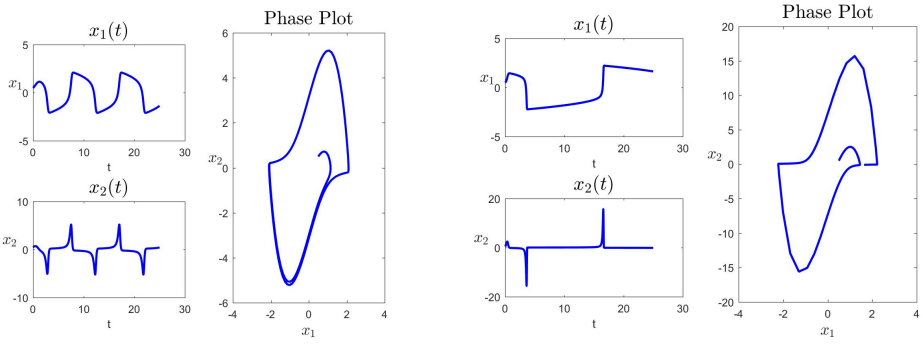


Figure 1: (a) Numerical Solution of the Van der Pol Problem, with μ=3, $N = 10^3$. Using Eulers method with fixed step size. (b) Numerical Solution of the Van der Pol Problem, with μ=10, $N = 10^3$. Using Eulers method with fixed step size.

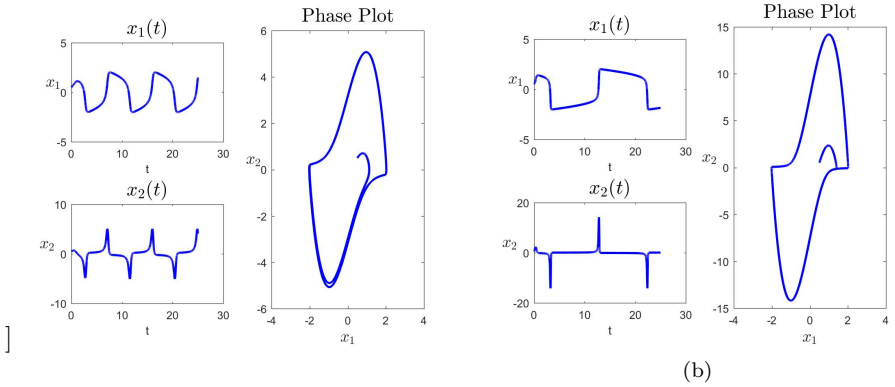
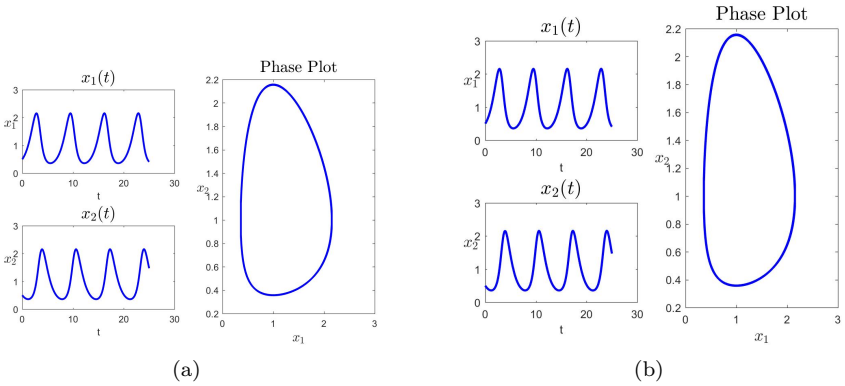


Figure 2: (a) Numerical Solution of the Van der Pol Problem, with μ=3, $h_0 = 10^{-2}$. Using Eulers method with adaptive step size. 7657 steps used. (b) Numerical Solution of the Van der Pol Problem, with μ=10, $h_0 = 10^{-2}$. Using Eulers method with adaptive step size. 6280 steps used at tolerance 10^{-5}

1,5)

The fixed and adaptive methods are used on the Predator Prey problem. The results are shown in figure 3.

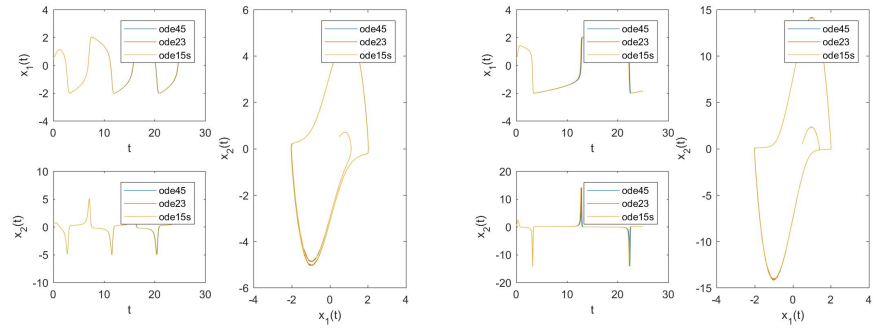


(a) (b)

Figure 3: (a)Numerical Solution of the Prey Predator, with $N = 10^4$. Using the explicit Eulers method with fixed step size. (b)Numerical Solution of the Prey Predator, with $h_0 = 10^{-2}$. Using the explicit Eulers method with adaptive step size. 3565 steps

1,6)

We test some of Matlabs ODE solvers on the same problems that we have considered when applying the explicit Euler method. We are not including them in the previous plots. The reason is that it would clutter too much, to have so many solvers reaching almost identical conclusions. We try out ode45, ode23 and ode15s for both of the test problems considered so far, in figure 4 and 5.



(a) (b)

Figure 4: Matlab numerical solutions of the Van der Pol Problem (a) with μ=3. Steps: ode45, 430 steps, ode23, 256 steps, ode15s, 340 steps (b) with μ=10, Steps: ode45, 741 steps, ode23, 336 steps, ode15s, 327 steps

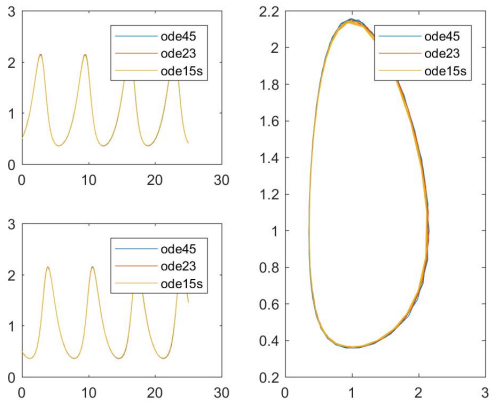


Figure 5: Matlab numerical solutions of the Prey Predator Problem. Steps: ode45, 117 steps, ode23, 91 steps, ode15s, 122 steps

This motivates us in looking at which area it might be interesting to look at the step size and tolerances of our solvers. We will exemplify by the Prey-Predator system, where our own solvers took a lot of steps to produce acceptable solutions as the ones found in figure 3. We try a range of step sizes and tolerances in figure 6 and 7. In figure 8, we plot the tolerance versus the number of steps taken on logarithmic scales. We see that for low tolerances we take an inordinate amount of steps, which makes sense.

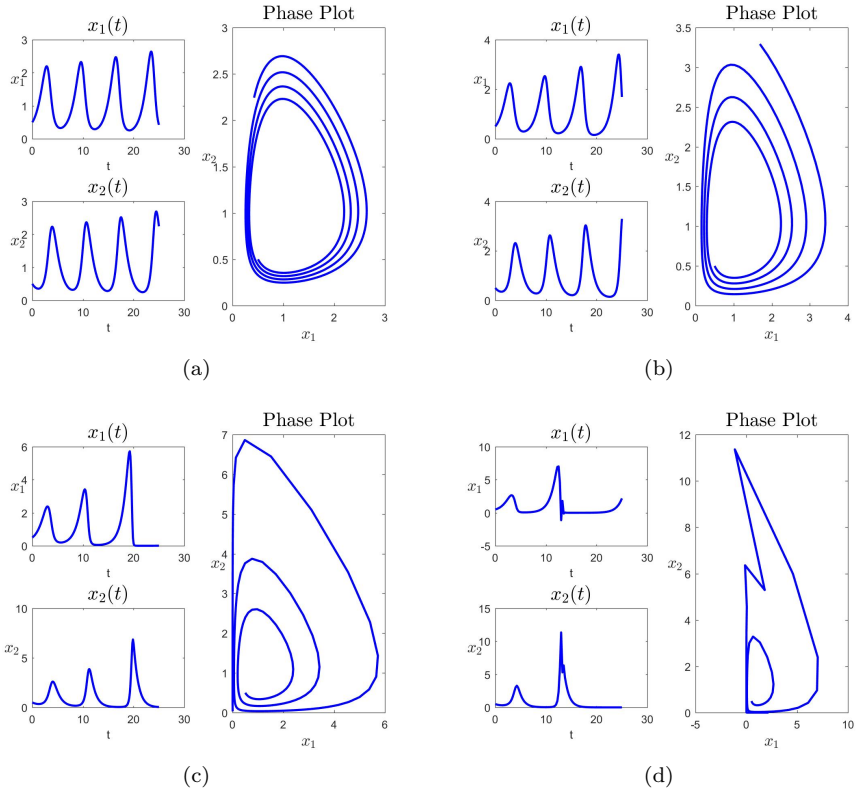


Figure 6: Numerical solution to the Predator-Prey problem for different step sizes, using the Explicit Euler Method with fixed step size. (a) 1000 steps. (b) 500 steps. (c) 200 steps. (d) 100 steps.

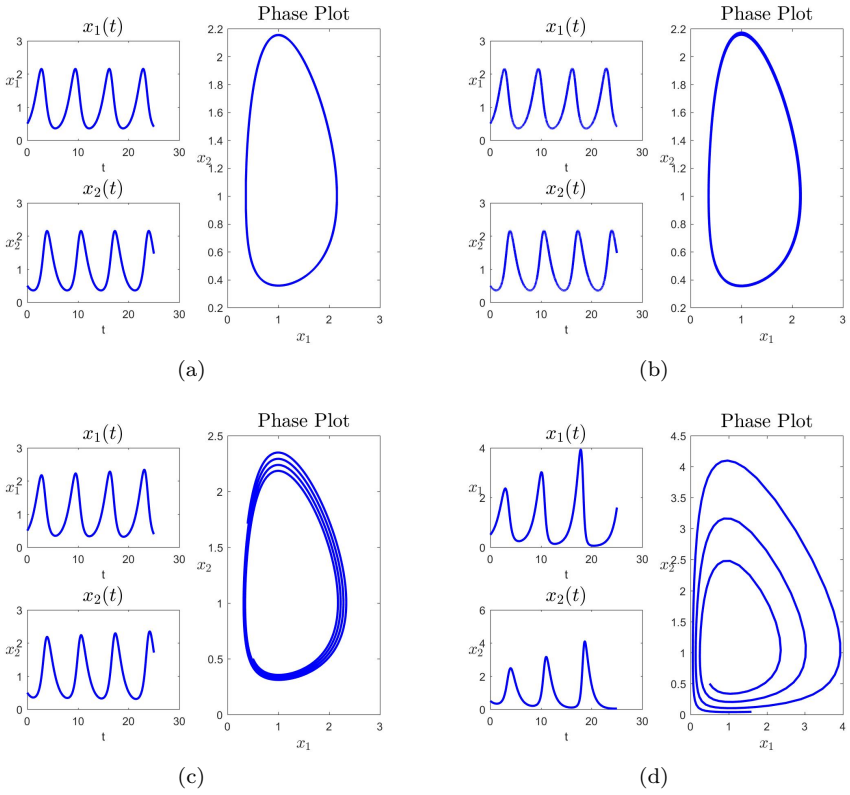


Figure 7: Numerical solution to the Predator-Prey problem for different tolerances, using the Explicit Euler Method with adaptive step size. (a) 111439 steps. $tol = 10^{-8}$ (b) 11182 steps.$tol = 10^{-6}$ (c) 1156 steps. $tol = 10^{-4}$(d) 139 steps.$tol = 10^{-2}$

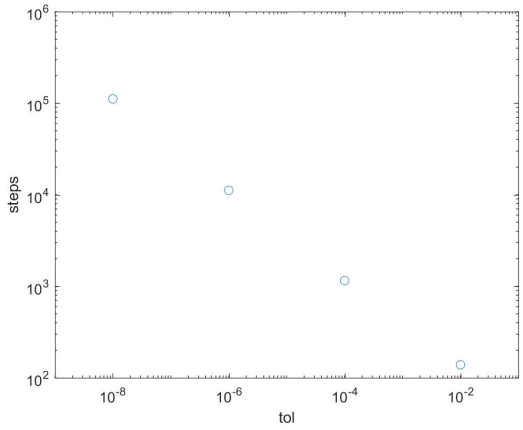


Figure 8: How the step size scales as a function of tolerance for the explicit euler method with adaptive step size.

The way we structure the interface for the ODE solvers is inspired by the syntax used by matlab. We would like to keep (for the most part) the same placement of variables and styles of function calling. This seems to be the standard generally and can minimize the invariable, man-made errors.

2 Implicit ODE solver

2.1)

The derivation of the implicit version of Euler's method is analogous to the one done for the explicit method. We choose to center the Taylor expansion around t_k instead of t_{k-1} and obtain.

$$x(t_{k+1} - h) = x(t_{k+1}) - h_k x'(t_{k+1}) + \frac{1}{2} h_k^2 x''(t_{k+1}) ... \tag{5}$$

As with the explicit version, we see that the one step error is $O(h^2)$. The implicit method is thereby order 1 as well. By rearranging the terms and using the ode in equation 1, we can rewrite the above.

$$x_{k+1} = x_k + h_k f(t_{k+1}, x_{k+1}) \tag{6}$$

This numerical method is implicit, as the calculation of the value at each step in general entails the solution of a nonlinear equation.

$$R(x_{k+1}) = x_{k+1} - x_k - h_k f(t_{k+1}, x_{k+1}) = 0 \tag{7}$$

This equation can be solved be a a root finding algorithm such as Newtons method. An implementation of Newtons method for use in our ODE solvers, can be found in the appendix.

2.2)

In the appendix, you can find the version of Newtons method used inside the code for the Implicit Euler methods considered.

```
function [T,X,stat] = ImplicitEuler(fun,tspan,N,x0,varargin)

%Memory Allocation
X = nan(length(x0),N);
T = nan(1,N);

%Root Finding Parameters
maxit = 100;
tol = 10^(-3);

%Function evaluations and stats
stat.iter = 0;

%Time step and initial conditions
h = ( tspan(2)-tspan(1) ) / N;
X(:,1) = x0;
T(1) = tspan(1);

for n = 1:N
    stat.iter = stat.iter + 1;

    %Function evaluation
    X_init = X(:,n) + h*feval(fun,T(n),X(:,n),varargin{:});

    %Solve next iterate using Newton's Methon
```

```
    X(:,n+1) = NewtonMethodODE(fun,T(n),X(:,n),h,X_init,tol,maxit,varargin{:});
    T(n+1) = T(n) + h;
end
end
```

2.3)

The adaptive step size is more or less a direct incorporation of the version found in the explicit version.

```
function [T,X,stat] = ImplicitEulerAdaptive(fun,tspan,x0,h0,abstol,reltol,varargin)

%Error Control Parameters
eps = 0.8;
facmin = 0.1;
facmax = 5;

%Root-Finding Parameters
maxit = 100;
tol = 10^(-3);

%Initialization
h = h0;
t0 = tspan(1);
tf = tspan(2);

%Start values
x(:,1) = x0;
t = t0;

%Output
X = x;
T = t;
stat.feval = 0;
stat.iter = 0;
stat.naccept = 0;
stat.nreject = 0;
stat.h = h;
stat.r = 0;

while t < tf
    %Final time step
    if tf < t + h
        h = tf - t;
    end

    %Function Evluation
    f = feval(fun,t,x,varargin{:});
    stat.feval = stat.feval + 1;

    AcceptStep = false;
    while ~AcceptStep
     stat.iter = stat.iter + 1;

    %Entire Step Start :::

    %Initial Guess with Forward Euler
    xe_init = x + h*f;
    %Solve for next iterate
    xe = NewtonMethodODE(fun,t,x,h,xe_init,tol,maxit,varargin{:});

    %Entire Step End :::

    %Double Step Start :::
```

```
    %1. Double Step
    hm = 0.5*h;
    %Initial Guess
    xm_init = x + hm*f;
    %Solve for next iterate
    xm = NewtonMethodODE(fun,t,x,hm,xm_init,tol,maxit,varargin{:});

    %2. Double Step
    tm = t + hm;
    %Initial Guess
    xf_init = xm + hm * feval(fun,tm,xm,varargin{:});
    stat.feval = stat.feval + 1;
    %Solve for next iterate
    xf = NewtonMethodODE(fun,tm,xm,hm,xf_init,tol,maxit,varargin{:});

    %Double Step End :::

    %Estimate Error : Entire Step vs Double Step
    e = xf - xe;
    r = max( abs(e) ./  max( abstol , abs(xf).*reltol ) );

    %Update if Error O.K otherwise change h and retry
    AcceptStep = ( r <= 1 );
    if AcceptStep
        t = t + h;
        x = xf;

        X = [X,x];
        T = [T,t];
        stat.h = [stat.h h];
        stat.r = [stat.r r];
        stat.naccept = stat.naccept + 1;
    else
        stat.nreject = stat.nreject + 1;
    end

    %Step Size Controller
    h_update = max( facmin , min( sqrt(eps/r) , facmax ) ) * h;
    h = h_update;

    end

end
end
```

2.1 2.4

We test the implementations of Eulers Implicit Methods with both fixed and adaptive step sizes. The results of the considerations are shown in figure 9 and 10. We note that the methods generally converge slower. This is a qualitative observation as did not use the tic toc function systematically on these problems, but it makes sense that the implicit methods would be slower, especially for low tolerance or high fixed step size. The adaptive method is much more effective for the implicit case than the explicit, as there are a factor 6 fewer time steps.

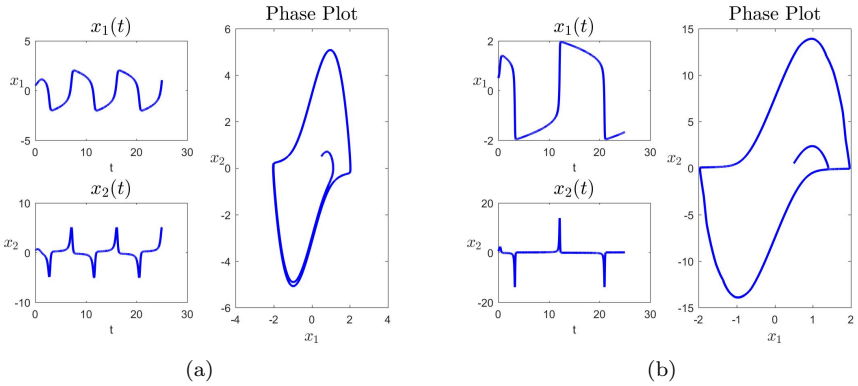


Figure 9: (a) Numerical Solution of the Van der Pol Problem, with μ=3, $N = 10^4$. Using the implicit Eulers method with fixed step size. (b) Numerical Solution of the Van der Pol Problem, with μ=10, $N = 10^4$. Using the implicit Eulers method with fixed step size.

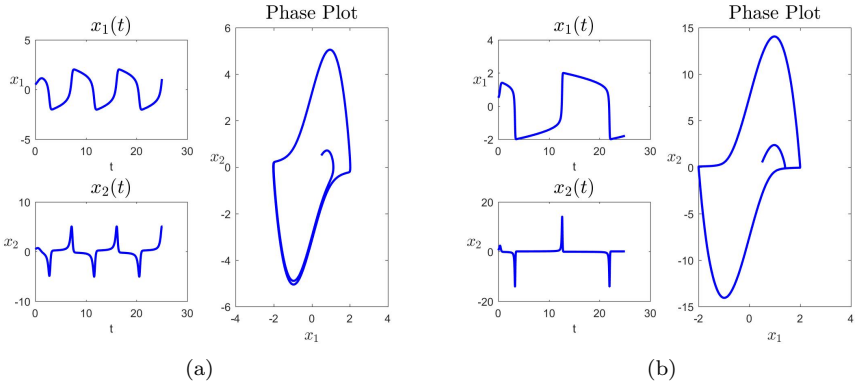


Figure 10: (a) Numerical Solution of the Van der Pol Problem, with μ=3, $h_0 = 10^{-2}$. Using implicit Eulers method with adaptive step size. Using 1263 time steps. $tol = 10^{-5}$(b) Numerical Solution of the Van der Pol Problem, with μ=10, $h_0 = 10^{-2}$. Using implicit Eulers method with adaptive step size. Using 1033 time steps, $tol = 10^{-5}$

2,5)

The methods are tested on the Predator-Prey system in figure

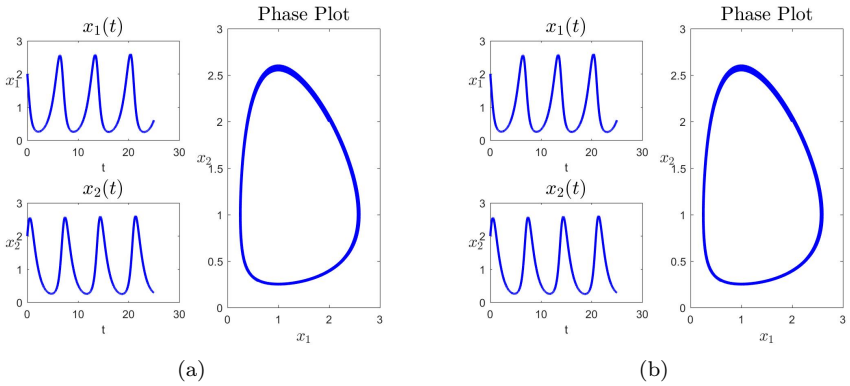


Figure 11: (a) Numerical Solution of the Prey Predator, with $N = 10^4$. Using Eulers method with fixed step size.(b) Numerical Solution of the Prey Predator, with $h_0 = 10^{-2}$. Using Eulers method with adaptive step size. 3885 steps

2.6)

For the matlab solutions, refer to section 1. The experiments with different step sizes and tolerances for the implicit methods, are showcased in figure 12 and 13.

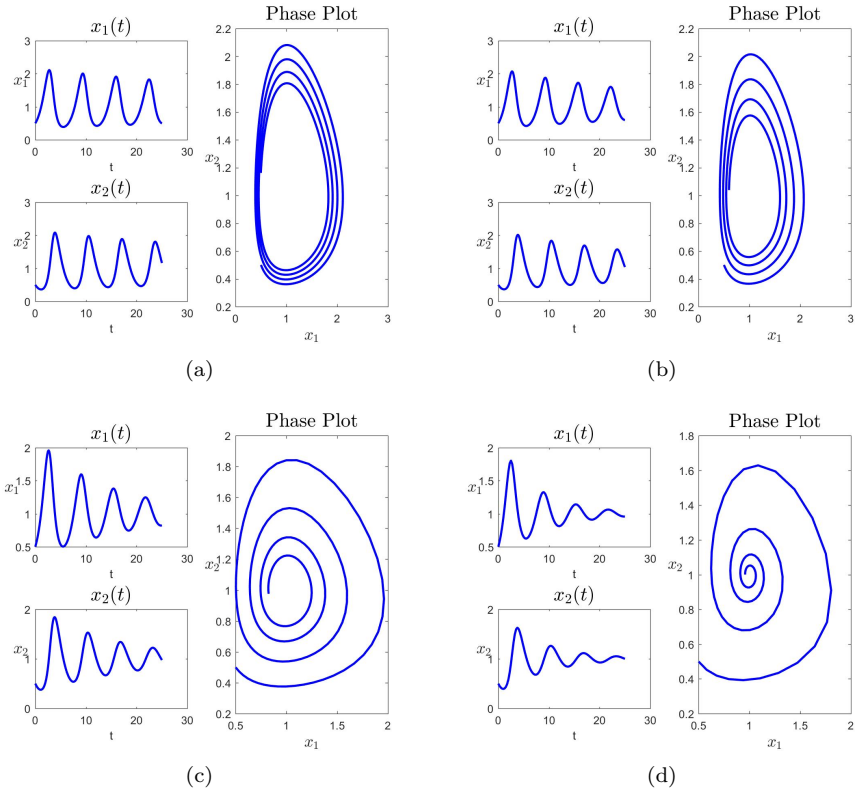


Figure 12: Numerical solution to the Predator-Prey problem for different step sizes, using the Implicit Euler Method with fixed step size. (a) 1000 steps. (b) 500 steps. (c) 200 steps. (d) 100 steps.

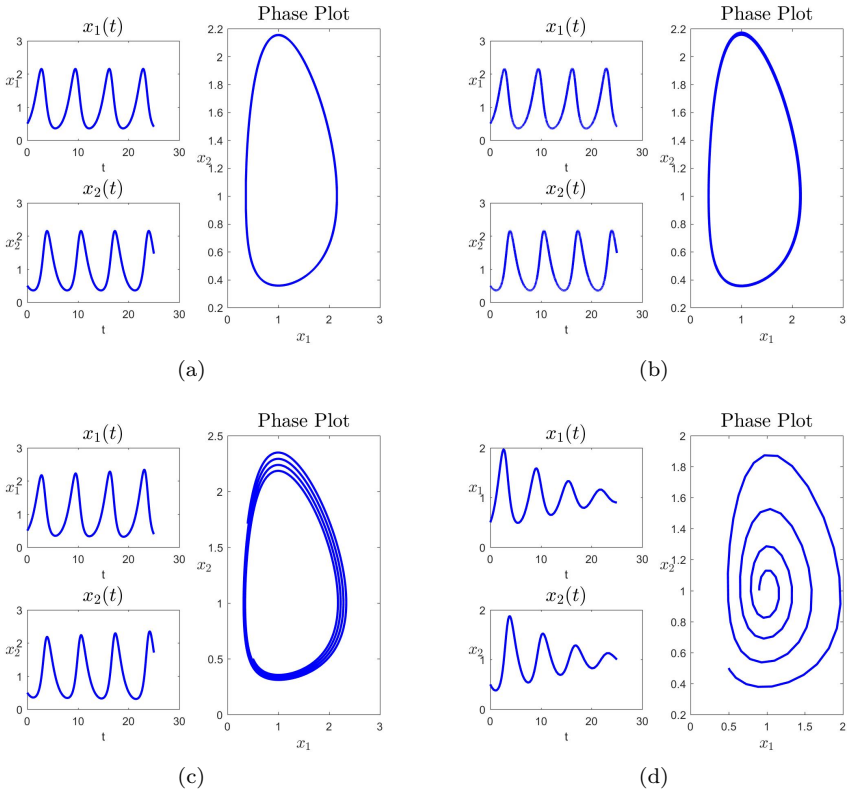
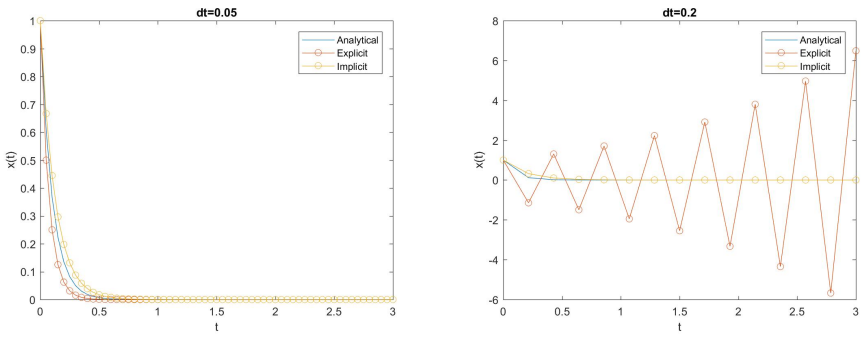


Figure 13: Numerical solution to the Predator-Prey problem for different step sizes, using the Implicit Euler Method with fixed step size. (a) 111439 steps. (b) 11182 steps. (c) 1156 steps. (d) 79 steps.

2.7)

The choice of whether to use the Explicit or Implicit method comes down to the equation you are trying to solve. The Implicit method adds complications, such as the use of a equation to solve in each step and the implicit and explicit methods have the same order. The extra computation steps introduced by the implicit method makes its use unnecessary on well behaved problems, but the utility shows itself when the step size used, is not small enough to ensure the stability of the explicit method. This is particularly evident with so called stiff problems, which exhibit very differing "time scales". That is, they vary both very slow and very fast in the domain in which they are defined. We can illustrate this with an example. For the test equation $x' = \lambda x$, we get this kind of behaviour for large λ.



(a) (b)

Figure 14: Solution to the test equation with $\lambda = -10$ using both implicit and explicit methods. (a) For a step size of 0.05, we see both methods converge towards the analytical solution. (b) For a step size of 0.2, we see that the implicit method still converges, while the explicit method becomes unstable.

The implementation of the implicit methods follows the explicit ones, the difference mainly being in the algorithm itself.

3 Test equation for ODEs

We consider the test equation.

$$\dot{x}(t) = \lambda x(t) \qquad x(0) = x_0 \tag{8}$$

In this case we are interested in the case where $\lambda = -1$ and $x_0 = 1$.

3,1)

The problem has a simple closed form solution, which we can find by separation of the variables.

$$\begin{aligned} \frac{dx(t)}{dt} &= \lambda x(t) \\ &\rightarrow \\ \frac{dx(t)}{x(t)} &= \lambda dt \\ &\rightarrow \\ \int_{x_0}^{x} \frac{dx(t)}{x(t)} &= \int_0^t \lambda dt \\ &\rightarrow \\ \ln x - \ln x_0 &= \lambda t \\ &\rightarrow \\ x(t) &= x_0 e^{\lambda t} \end{aligned}$$

So for our specific choice of constants in this section, the analytical solution is $x(t) = e^{-t}$.

3,2)

The Local Truncation Error (LTE) quantifies the error made by a numerical method in one step. It measures the difference between the exact solution and the numerical approximation, under the assumption that the previous step was exact. We use the notation in the BOOK, denoting the exact solution at a discrete time point t_k as $y(t_k)$ and the approximation at that same point as y_k. For Euler's explicit Method we use the forward difference operator. The error e_k is meant as the error as we step from t_k to t_{k+h}

$$e_k = \frac{x(t_k + h) - x(t_k)}{h} - x'(t_k) \tag{9}$$

Expanding the functions in the fraction using Taylor's theorem, we obtain.

$$\frac{x(t_k + h) - x(t_k)}{h} = x'(t_k) + \frac{1}{2}hx''(\xi) \qquad \xi \in [t_k, t_k + h] \tag{10}$$

$$\rightarrow \tag{11}$$

$$e_k = \frac{1}{2}hx''(\xi) \qquad \xi \in [t_k, t_k + h] \tag{12}$$

The LTE is thus $O(h)$ and as h goes to 0, the LTE does as well, making the method consistent. We say that the method is first order correct. In general, a method is p'th order correct, if the LTE from this definition is $O(h^p)$. Something to bear in mind is, that in some texts, the LTE is defined as the one-step error. This is the error associated with one-step of the iterative process and can thus for the explicit Euler Method, it is the difference between $x(t_k + h)$ and the iteration by which we wish to approximate it. It proportional to $p+1$, as we are multiplying by h. When measuring numerical error in our programs, we are always measuring the one step error and as such, this is the version we will denote as the LTE. Assuming x_k is exact, then the LTE is.

$$e_k = x_{k+1} - x(t_{k+1}) \tag{13}$$

The Global Truncation Error (GTE) is a measure of the total error at a time t_k, composed of the effect of the LTE accumulated in each step taken to reach the time t_k. In general for a numerical method, this quantity may be hard to estimate, due to the various sources of errors, including truncation errors and round-off errors inherent in the floating point representation of numbers in computational systems.

$$e_k = x_k(t_k) - x(t_k) \tag{14}$$

Where x_k is the numerical solution and x is the exact solution. The idea is that the preceeding step not assumed to be exact and so the error can accumulate.

3,3)

When we compute the local and global errors, we elect to use the values at the second time step for the local error and the last time step for the global error. That is because the value at the initial point is correct as it corresponds the the IC. The second step is then a good representation of 1 step error. The numerical values are a bit redundant, as they are included in the next section.

3,4)

We plot the local and the global error of the Euler methods with fixed time steps, versus the time step used. The result is shown in figure 15.

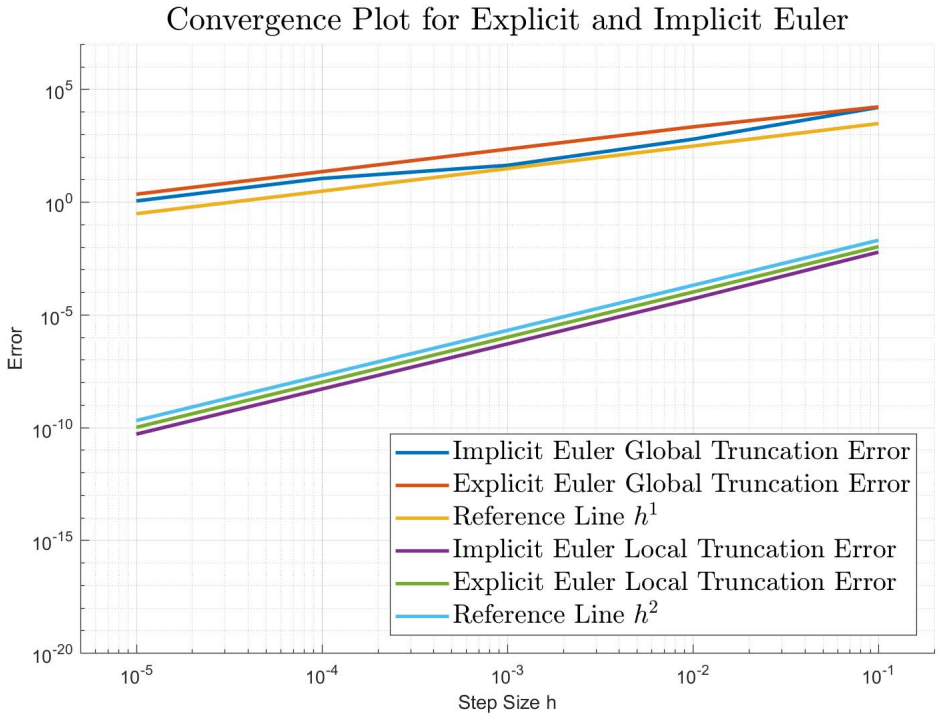


Figure 15: Convergence for the explicit and implicit Euler methods. It shows the LTE and GTE as functions of step size. The LTE is shown to be $O(h^2)$ and the GTE is shown to be $O(h)$.

The numerical calculations seem to show that both the implicit and explicit methods have an LTE that goes as $O(h^2)$ and a GTE that goes as $O(h)$. From the considerations in the derivations of the methods in section 1.1 and 1.2, we see that this is what is expected from asymptotic considerations. The error that we actually measure is the one-step error. This confirms that both errors behave as order 1 methods. The order of a method is the power of the truncation error as h goes to 0.

3,5)

See last section.

3,6)

Even though a numerical method is consistent, we might not be able to guarantee that the method converges toward the right solution. This property is called stability. Stability is the property that the error goes to zero as the truncation error goes to zero. There are different ways of defining stability. Suppose we are given a numerical method to solve equation 8. As we found earlier in the section, the exact solution to this equation is $x(t) = e^{\lambda t}$. The limit of this expression goes to 0 iff $\lambda \in \mathcal{C}$ has real part strictly smaller than 0. As our discrete time has $t_k = kh$, this can be rephrased The stability area of a numerical method, is the set of numbers $h\lambda \in \mathcal{C}$ so that the limiting value of the approximations x_k go to 0 as k goes to ∞. We can find the values of h and λ appropriate for a given numerical method from the iterative scheme. First for Eulers Explicit Method. By inserting the test equation into one step of Eulers method.

$$x_{k+1} = x_k + h\lambda x_k \tag{15}$$
$$= (1 + h\lambda)\, x_k \tag{16}$$

So to satisfy the condition, it must be the case that $|1 + h\lambda| < 1$. This region is portrayed in figure 22(a). For the implicit method, we can again insert the test equation.

$$x_{k+1} = x_k + h\lambda x_{k+1} \tag{17}$$
$$\rightarrow \tag{18}$$
$$(1 - h\lambda)\, x_{k+1} = x_k \tag{19}$$
$$\rightarrow \tag{20}$$
$$x_{k+1} = \left(\frac{1}{1 - h\lambda}\right) x_k \tag{21}$$

Thus the condition for the implicit method is that $\left|\frac{1}{1-h\lambda}\right| < 1$. This area is portrayed in figure 22(b). When a method is A-stable, any time step is allowed, as long as the eigenvalue of the problem has negative real part. The definition of A-stability is thus, that the entire left plane of the stability plane is contained in the region of stability. We see from the diagrams that this condition is fulfilled by the implicit method but not by the explicit.

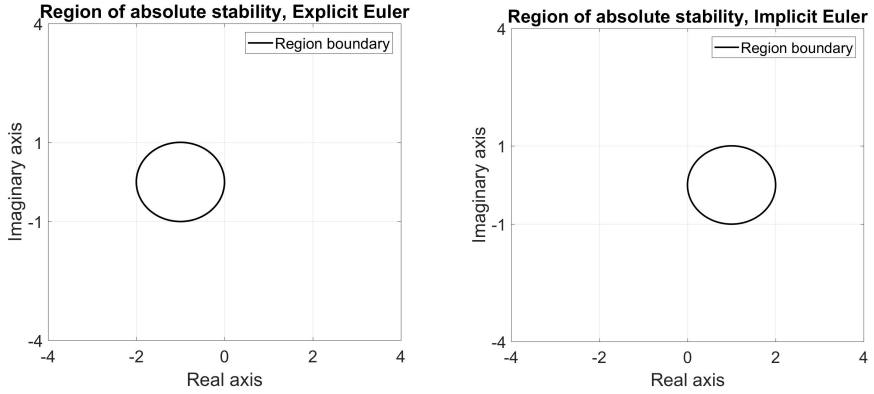


Figure 16: (a) Stability region for the explicit Euler method, inside the boundary (b)Stability region for the implicit Euler method, outside the boundary

4 Solvers for SDEs

We now consider Stochastic Differential Equations (SDE), of the following, general form. The equation is written is differential due to the brownian motion not being differentiable.

$$dx(t) = f(t, x(t), p_f)dt + g(t, x(t), p_g)d\omega(t) \qquad d\omega(t) \sim N_{iid}(0, Idt) \tag{22}$$

4,1)

A Multivariate Wiener Process is the mathematical formulation of Multivariate Brownian Motion. A Standard Wiener Process is a random variable $\omega(t)$, that has a continuous dependence on a $t \in [0, T]$ and satisfies the following conditions.

- $\omega(0) = 0$
- $0 < s < t < T : \omega(t) - \omega(s) \sim N(0, t - s)$
- $0 < s < t < u < v < T$: increments $[\omega(t) - \omega(s)]$ and $[\omega(v) - \omega(u)]$ are independent

We can call a small increment of this process $d\omega(t)$. This is called white noise. The integral of this is then the Brownian motion.

$$\omega(t) = \int_0^t d\omega(s) \tag{23}$$

We can make this process dicrete the same way as we would an integral in ordinary calculus. We replace it by a sum.

$$\omega(t) \approx \sum_{i=0}^{N} \Delta\omega_s \tag{24}$$

This way we can implement Brownian motion as an iterative process, by treating the discrete, white noise as being i.i.d. We write.

$$\omega_{k+1} = \omega_k + \Delta\omega_k \tag{25}$$

We implement the process for a multivariate vector x in matlab, by the use of the inbuilt rng. The only difference from the scalar version, presented in the lectures, is that we use a parameter nW to allow the resulting motion to be multivariate, and when creating our pseudorandom array, that the number is included as the length of a new dimension.

```
function [W,TW,dW] = StdWienerProcess(T,N,nW,Ns,seed)

%If the seed is specified, it is set.
if nargin == 4
    rng(seed);
end

% Wiener Process: Array of nW x N x Ns dimensions.
dt = T/N;
dW = sqrt(dt)*randn(nW,N,Ns);
W = [zeros(nW,1,Ns) cumsum(dW,2)];
TW = 0:dt:T;
```

4,2)

Now we consider the solution to the SDE in equation 22. As our code implements a discrete version of the Wiener process, we can by a simple extension solve by problem by application of the Euler-Maryama method. We first consider the Explicit-Explicit version.

```
function [X,stat] = SDE_ExplicitExplicitEuler(fun,gun,T,x0,W,varargin)
%Function for solving the stochastic differential equation using the
%Standard Wiener Process.

%Memory Allocation
X = nan(length(x0),length(T));
N = length(T) - 1;


%Stats
stat.nfun = 0;  %Function evaluations
stat.lerror = []; %local error
stat.gerror = []; %glocal error

%Time step and initial conditions
X(:,1) = x0;

%For Loop
for i = 1 : N
    dt = T(:,i+1) - T(:,i);
    dW = W(:,i+1) - W(:,i);

    f = feval(fun,T(i),X(:,i),varargin{:});
    g = feval(gun,T(i),X(:,i),varargin{:});

    X(:,i+1) = X(:,i) + f*dt+ g.*dW ;
    stat.nfun = stat.nfun + 1;
end


end
```

4,3)

For the implicit version, the diffusion term of the solution method is unaffected. The difference comes in that the f term makes the equation implicit and so we must use a variation of a nonlinear equation solver at each step. We use Newtons method.

```
function [X,stat] = SDE_ImplicitExplicitEuler(fun,gun,T,x0,W,varargin)
%Function for solving the stochastic differential equation using the
%Standard Wiener Process. The function uses an Implicit Euler on the
%deterministic part and an Explicit Euler on the stochastic part

%Memory Allocation
X = nan(length(x0),length(T));
N = length(T) - 1;

%Stats
stat.nfun = 0;  %Function evaluations
stat.lerror = []; %local error
stat.gerror = []; %glocal error

%Time step and initial conditions
X(:,1) = x0;
maxit = 100;
tol = 10^(-3);
```

```
%For Loop
for i = 1 : N
    dt = T(:,i+1) - T(:,i);
    dW = W(:,i+1) - W(:,i);

    g  = feval(gun,T(i),X(:,i),varargin{:});
    f = feval(fun,T(i),X(:,i),varargin{:});

    %%%%%% Implicit Solution %%%%%%

    %Implicit Initial Guess using Explicit Euler
    Psi = X(:,i) + g*dW;
    X_init = f*dt + Psi;

    %Solve next iterate using Newton's Methon
    X(:,i+1) = NewtonMethodSDE(fun,T(i),Psi,dt,X_init,tol,maxit,varargin{:});

    %%%%%% Implicit Solution %%%%%%
end
end
```

4,4)

Both the Explicit-Explicit and the Implicit-Explicit Methods are very reminiscent of the classical Euler's Methods used earlier in the project. The difference is that we now have a new, stochastic term added to the right hand side of the IVP. The same way that the time axis is discretized, we can also discretize the Wiener Process, using the time discretization as the variance for the Wiener Process. As we already have a discrete version of the process from section 1.1, we can state the method as an explicit-explicit, iterative formula. Using the form of the SDE and the integral derivation of the Explicit Eulers method. We use dt for h here.

$$x_{k+1} = x_k + f(t_k, x_k)dt + g(t_k, x_k)\Delta\omega_k \tag{26}$$

Instead of using the left hand evaluation of the integral of f in the derivation of the method, we could use the right hand one, leaving us with an implicit method. The stochastic term is unaffected, as this would complicate matters extensively. This implicit-explicit method can then be stated as follows.

$$x_{k+1} = x_k + f(t_{k+1}, x_{k+1})dt + g(t_k, x_k)\Delta\omega_k \tag{27}$$

In this hybrid method, the deterministic term is solved as in the implicit Euler method, while the stochastic term is advanced as in the explicit Euler method. To these methods, we have to supply two functions, which govern different aspects of the solution.

4,5)

We can realize SDE versions of the Van der Pol problem by adding a diffusion term. We can use both a state independent and dependent version of the diffusion term. Here the system is written on differential form.

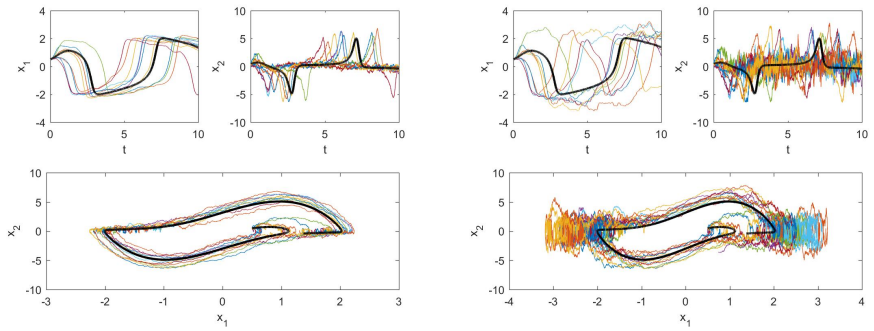
$$dx_1 = x_2(t)dt \tag{28}$$

$$dx_2 = \left(\mu(1 + x_1(t)^2)x_2(t) - x_1(t)\right)dt + \sigma d\omega(t) \tag{29}$$

$$dx_1 = x_2(t)dt \tag{30}$$

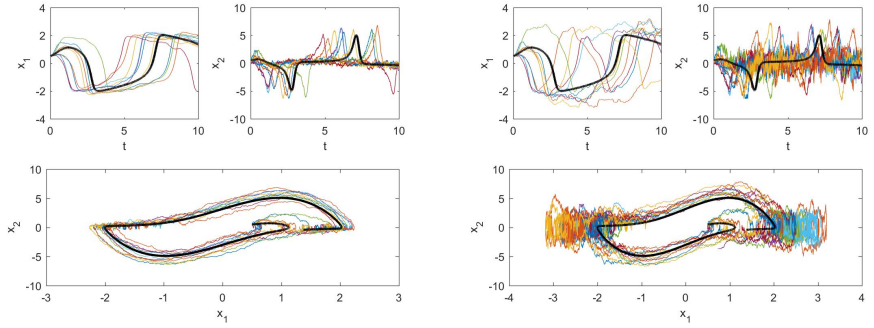
$$dx_2 = \left(\mu(1 + x_1(t)^2)x_2(t) - x_1(t)\right) dt + \sigma(1 + x_1(t)^2)d\omega(t) \tag{31}$$

The second version of the system essentially amplifies the diffusive term as time goes. We can solve the problem for a variety of parameters and step sizes. To keep the illustrations tractable, when they are plotted, they are mostly done so with only 10 instances of the solution. They are plotted against the solution to the non-stochastic version of the Van der Pol problem as well, which corresponds to setting $\sigma = 0$. This just reduces the solvers to Eulers method. In all of the test cases, the IC $x_0 = [0.50.5]$ is used. The result is shown in figures 17, 18, 21 and 22.



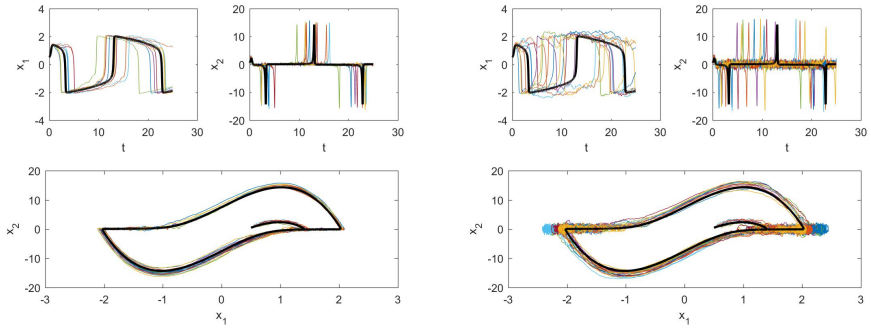
(a) (b)

Figure 17: Explicit Euler-Maruyama solutions to the Van der Pol problem with 10 instances, 10000 steps, $\mu = 3$. Non-diffusive solution in black. (a) Diffusion version 1 (b) Diffusion version 2



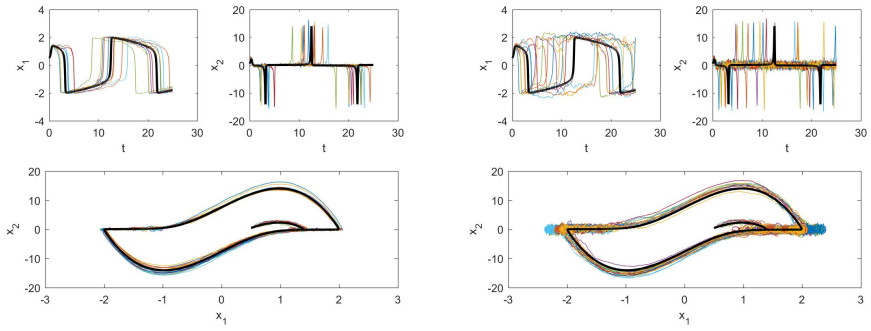
(a) (b)

Figure 18: Implicit Euler-Maruyama solutions to the Van der Pol problem with 10 instances, 10000 steps, $\mu = 3$. Non-diffusive solution in black. (a) Diffusion version 1 (b) Diffusion version 2



(a) (b)

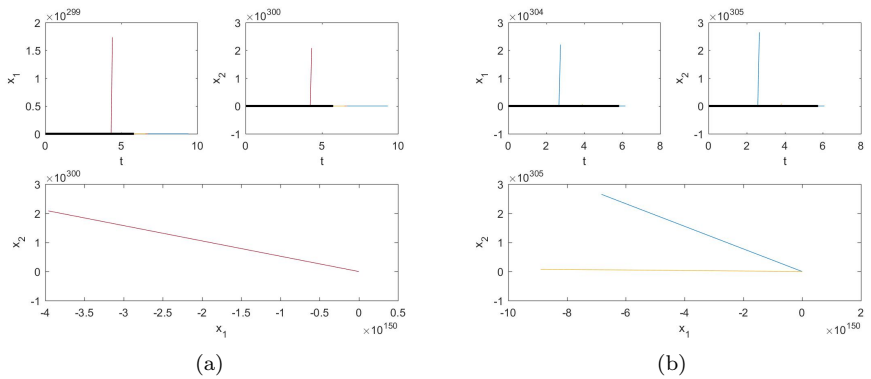
Figure 19: Explicit Euler-Maruyama solutions to the Van der Pol problem with 10 instances, 10000 steps, $\mu = 10$. Non-diffusive solution in black. (a) Diffusion version 1 (b) Diffusion version 2



(a) (b)

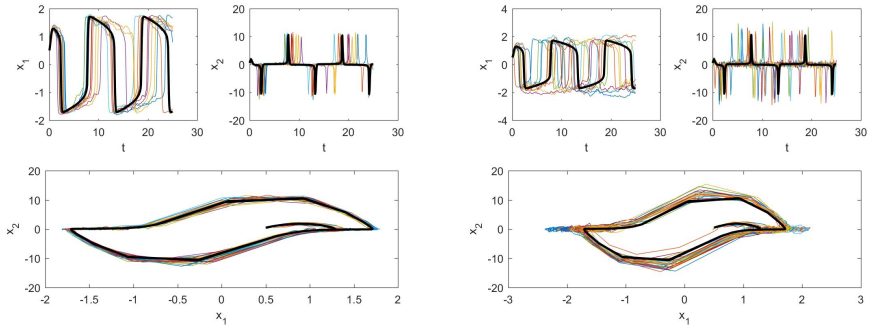
Figure 20: Implicit Euler-Maruyama solutions to the Van der Pol problem with 10 instances, 10000 steps, $\mu = 10$. Non-diffusive solution in black. (a) Diffusion version 1 (b) Diffusion version 2

As a final addendum, we can see whether the methods behave as expected for when stability is becoming a factor. As we see in figure **??** and **??**, the distinction here between the implicit and explicit methods seem to carry over, as we see the explicit-explicit version become very unstable for large step size and large μ.



(a) (b)

Figure 21: Explicit Euler-Maruyama solutions to the Van der Pol problem with 10 instances, 300 steps, $\mu = 10$. Non-diffusive solution in black. (a) Diffusion version 1 (b) Diffusion version 2



(a) (b)

Figure 22: Implicit Euler-Maruyama solutions to the Van der Pol problem with 10 instances, 300 steps, $\mu = 10$. Non-diffusive solution in black. (a) Diffusion version 1 (b) Diffusion version 2

5 Test equation for SDEs

For this section, we will look at the Geometric Brownian Motion, encapsulated in the following SDE.

$$dx(t) = \lambda x(t)dt + \sigma x(t)d\omega(t) \qquad d\omega(t) \sim N_{iid}(0, Idt) \tag{32}$$

5,1)

The analytic solution can be found by the use of Stochastic Calculus.

$$x(t) = x_0 \exp\left(\left(\lambda - \frac{1}{2}\sigma^2\right)t + \sigma\omega(t)\right) \tag{33}$$

This then is the stochastic solution we are referring to, when mentioning the analytical solution. The equation can be seen to be a solution to the SDE in equation 32, not by insertion directly, but by the use of Ito's formula. In figure 23 and 24 We show how such a solution can look, both analytically and numerically, using different seeds, to show the stochastic nature of the solutions. Choosing a different seed causes a much greater change in the solution, than choosing a numerical method over the analytical one.

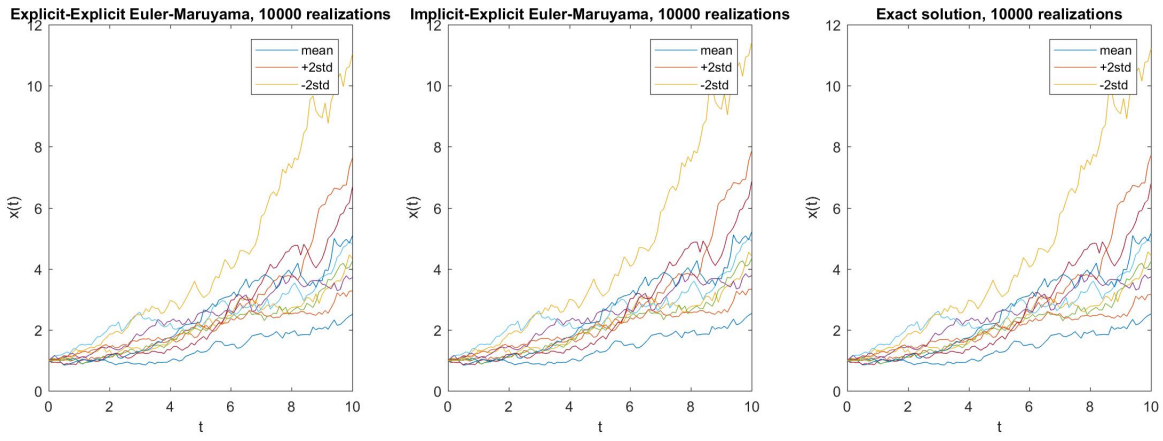


Figure 23: An example of 10 Solution instances for the geometric brownian motion. $\lambda = 0.15$, $\sigma = 0.15$

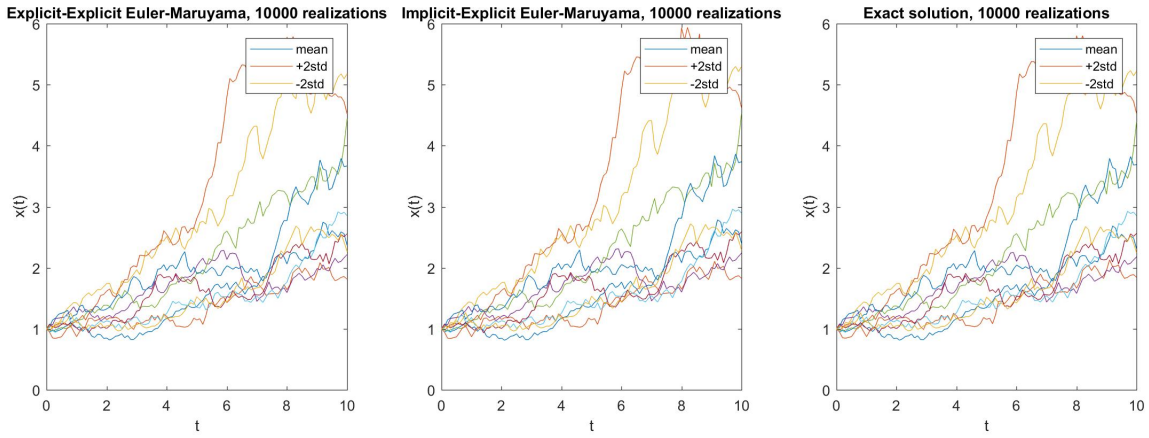


Figure 24: An example of 10 Solution instances for the geometric brownian motion using a different seed. (1000) $\lambda = 0.15$, $\sigma = 0.15$

5,2)

Instead of comparing all of the different stochastic solutions we find to the same number of analytical solutions, it could be prudent to take the mean and the standard deviations as measures to quantify the solution space at each time t_k. By plotting the mean and ± 2 standard deviations, we get a sense of the distribution of solutions for the different methods and can more readily compare to the numerical solutions. Especially as we are interested in the case at which we will be using 10000 solution instances. We first plot a illustrative example, using 20 instances.

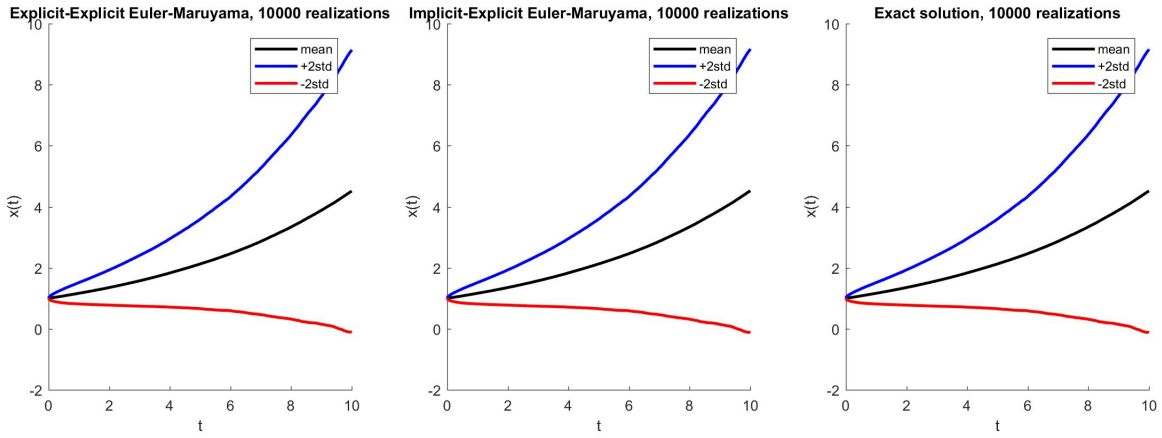


Figure 25: Mean and $\pm$ 2 std for the geometric brownian motion, with $\lambda = 0.15$, $\sigma = 0.15$. We clearly see a drift and a widening diffusion as time goes.

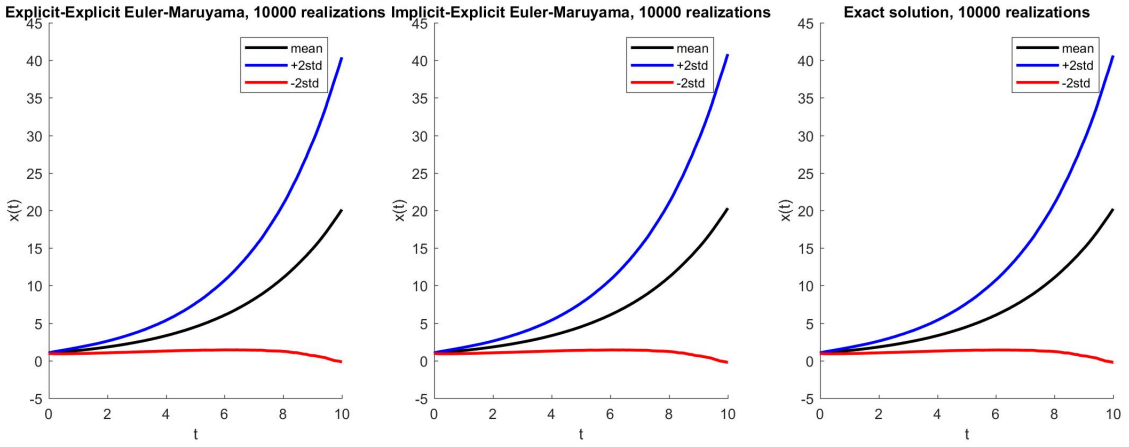


Figure 26: Mean and $\pm$ 2 std for the geometric brownian motion, with $\lambda = 0.30$, $\sigma = 0.15$. In this case we see stronger drift, but proportionally the diffusion seems to act as in the previous example.

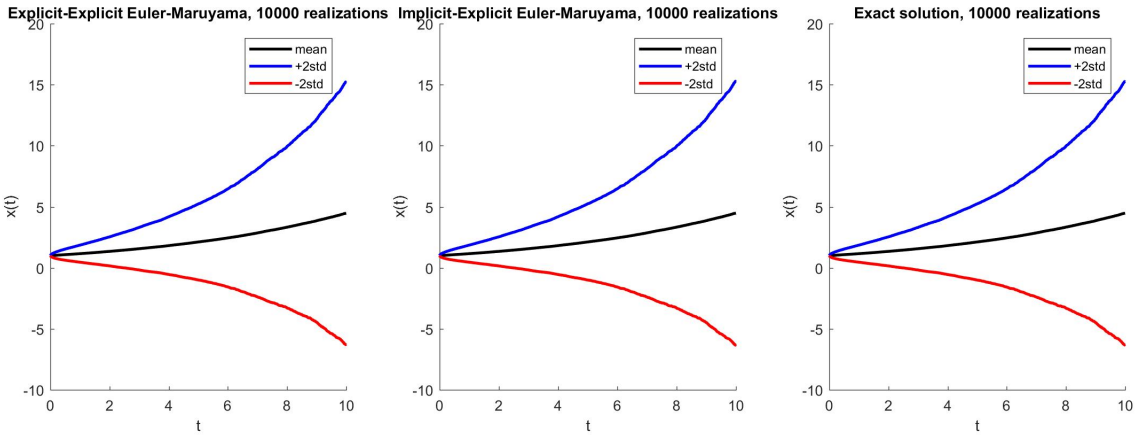


Figure 27: Mean and $\pm$ 2 std for the geometric brownian motion, with $\lambda = 0.15$, $\sigma = 0.30$. Here we see the mean unchanged from the first example, but a much larger diffusion as time goes.

5,3)

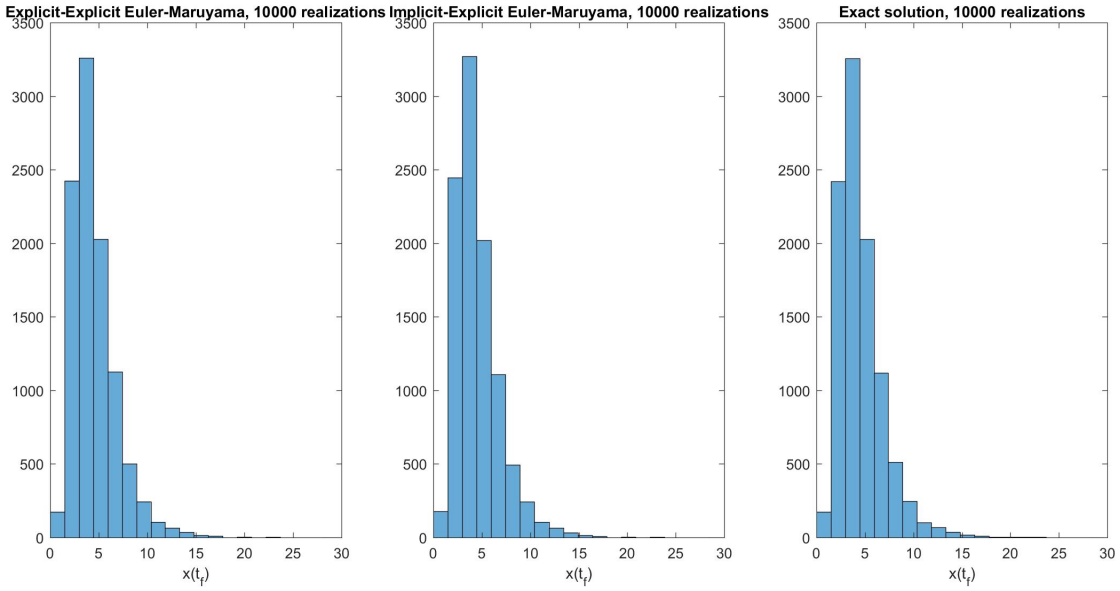


Figure 28: Histogram of the final distribution $x(t_f)$ of the solutions, with $\lambda = 0.15$, $\sigma = 0.15$

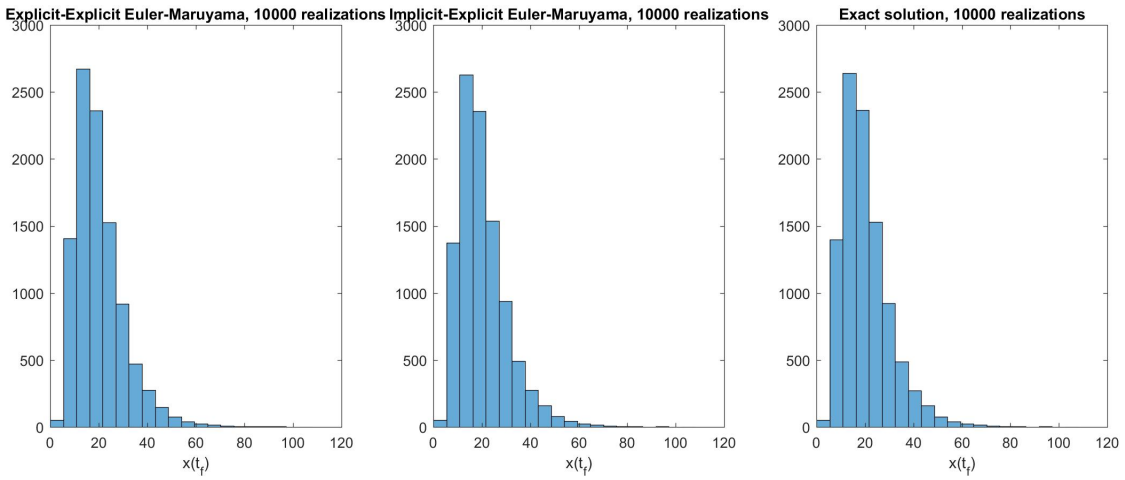


Figure 29: Histogram of the final distribution $x(t_f)$ of the solutions, with $\lambda = 0.30$, $\sigma = 0.15$

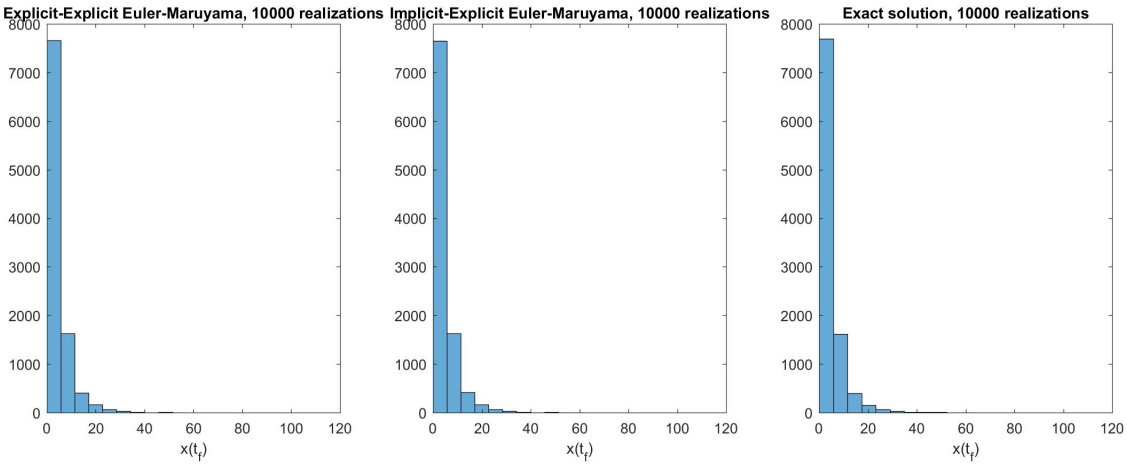


Figure 30: Histogram of the final distribution $x(t_f)$ of the solutions, with $\lambda = 0.15$, $\sigma = 0.30$

5,4)

The analytical versions are included in the plots where we are testing the numerical solutions.

1,5)

If we were comparing methods for which there was no stochastic influence, we would simple plot the step size versus the error on a logarithmic scale. We have to find another measure now, and for SDE's, we have looked at 2 nonequal ways to do this. So-called strong and weak error. For now we look at the strong error, which is defined by taking the mean of the sum of the difference in values between analytical and numerical.

$$e_k = mean\left(x_k - x(t_k)\right) \tag{34}$$

5,6)

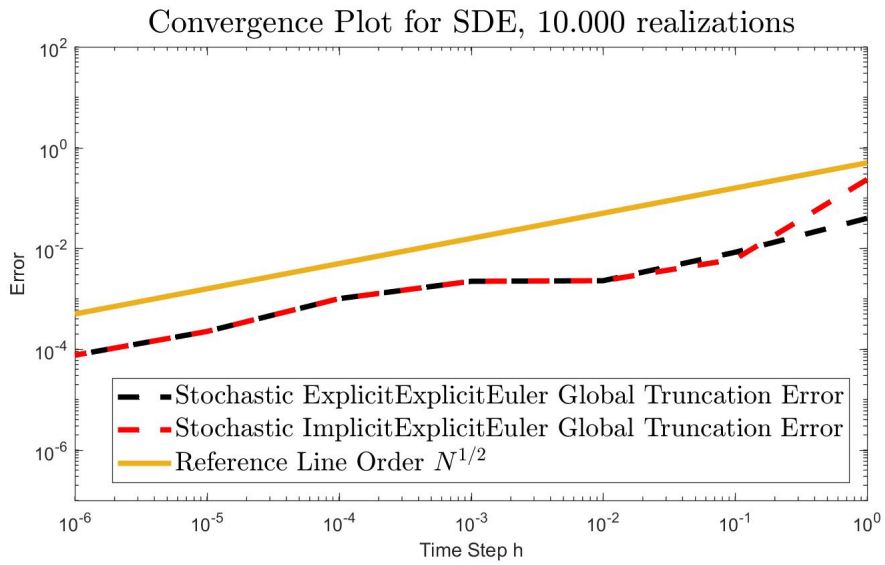


Figure 31: Convergence plot of the explicit-explicit and implicit-explicit euler-maruyama methods. The methods are shown to exhibit order $O(h^{1/2}$, consistent with expectations.

Mini Project 2

6 Classical Runge-Kutta Method: Fixed Time Step

The Runge-Kutta methods takes advantage of the fact that consecutive function evaluations within the interval $[t_n, t_{n+1}]$ can lead to a higher order method. It is in practice carried out by the intermediate evaluations X_i at various times given by $T_i = t_n + c_i h$ where $c_i \leq 1$. The amount of intermediate evaluations characterizes the method and is described as *stages*. Consider the intermediate steps in the explicit 4-staged *Classical Runge-Kutta* method (ERK4C) below. The function denoted by f here is the associated with the usual initial-value problem.

$$\begin{aligned}
&T_1 = t_n && X_1 = x_n \\
&T_2 = t_n + \frac{1}{2}h && X_2 = x_n + \frac{1}{2}hf(T_1, X_1) \\
&T_3 = t_n + \frac{1}{2}h && X_3 = x_n + \frac{1}{2}hf(T_2, X_2) \\
&T_4 = t_n + h = t_{n+1} && X_4 = x_n + hf(T_3, X_3) \\
&t_{n+1} = t_n + h && x_{n+1} = x_n + h\left[\frac{1}{6}f(T_1, X_1) + \frac{1}{3}f(T_2, X_2) + \frac{1}{3}f(T_3, X_3) + \frac{1}{6}f(T_4, X_4)\right]
\end{aligned} \tag{35}$$

The next iteration x_{n+1} in (35) is seen to be computed using the intermediate evaluations by steps of length $h/2$. The Runge-Kutta methods can be represented in a compact matrix notation. The matrix form associated with ERK4C is provided below in (36)

$$\begin{array}{c|cccc}
0 & 0 & 0 & 0 & 0 \\
1/2 & 1/2 & 0 & 0 & 0 \\
1/2 & 0 & 1/2 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 \\
\hline
X & 1/6 & 1/3 & 1/3 & 1/6
\end{array} \tag{36}$$

A MATLAB script of the implementation of the ERK4C method is found in Appendix 12.3. The implementation takes advantage of the matrix representation of the coefficient matrices to easily calculate the $n^{th}, n \in [2, ..., s]$ stage solution. This is in practice carried out in a for-loop where for each stage the intermediate evaluation $X(i) = X_i$ and function evaluation $F(i) = F(T_i, X_i)$ is computed and stored in a vector on the index dictated by the for-loop. New intermediate evaluations can then be computed by vector multiplication between F and the corresponding transposed column of A (to yields a scalar). The relevant matrices used for the ERK4C are shown below.

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1/2 & 1/2 & 0 & 0 \\ 0 & 1/2 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \qquad c = \begin{bmatrix} 0 \\ 1/2 \\ 1/2 \\ 1 \end{bmatrix} \qquad b = \begin{bmatrix} 1/6 \\ 1/3 \\ 1/3 \\ 1/6 \end{bmatrix} \tag{37}$$

The solver is applied on the usual test equation

$$\begin{aligned}
\dot{x}(t) &= \lambda x(t) \qquad t_0 \leq t \\
x(t) &= x(t_0)e^{(\lambda[t-t_0])}
\end{aligned}$$

for $\lambda = \{-25, -20, -10, -1, 1, 10, 20, 25\}$ in order to investigate the stability of the method for different eigenvalues (λ). The test is shown below in Figure (32) where the step size was $h = 0.1$. The method appears to perform well for $|\lambda| < 20$ but issues seem to arise outside this range as evident from the plots of $\lambda = \{-20, -25, 20, 25\}$. The issues were found to vanish for $h = 0.01$ however it is relevant to investigate the stability properties of the method in order to establish whether these issues continue to arise for any given $\lambda \in \mathbb{C}$. The stability region of the method

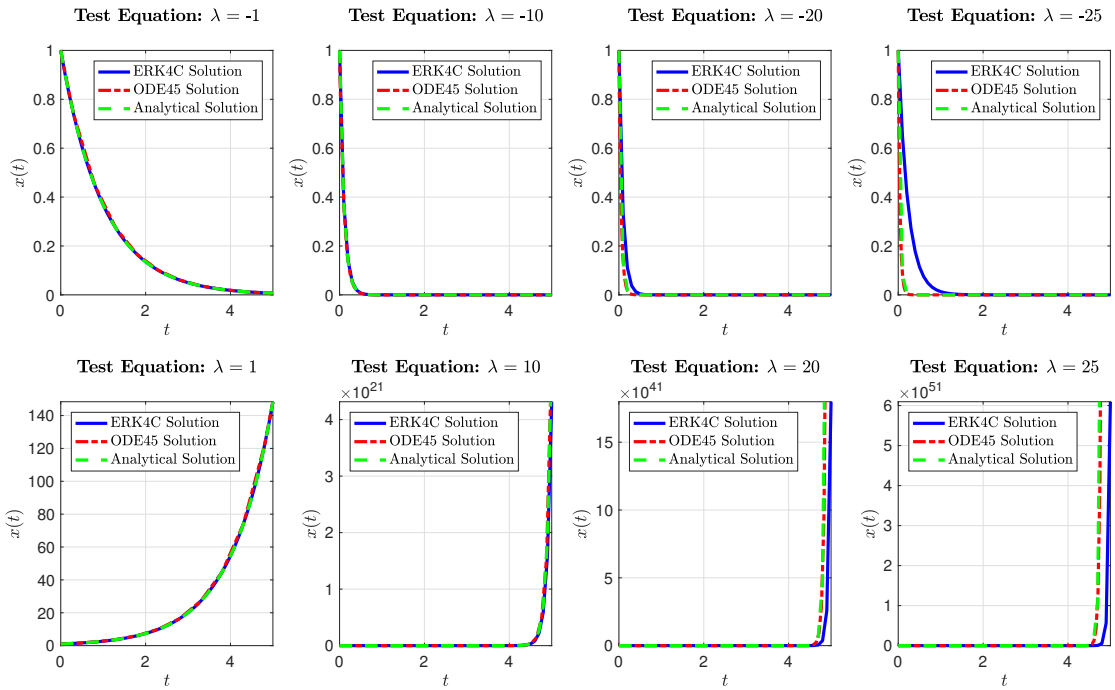


Figure 32: Numerical solutions to the test equation for various values of λ using ERK4C with step size $h = 0.1$. The plots reveal that for high values of λ the solution does not converge towards the analytical solution. This effect is amplified for $\lim_{\lambda \to \pm\infty}$ as evident from the stability region shown in Figure (33)

can be computed from the transfer function (38) which is the region for which $|R(z)| < 1$. The region is shown in Figure (33).

$$R(z) = 1 + zb^T(I - zA)^{-1}e \qquad z \in \mathbb{C} \tag{38}$$

It is evident from inspection that the ERK4C primarily exhibits stability within the rectangle whose corners are defined by the four complex numbers $z_1 = 0, z_2 = -2, z_3 = -2 + i, z_4 = i$ and as such the method is only stable in the test case $\lambda = -1$ of Figure (32). The instability effect is not clearly seen for the values of lambda outside the stability region for the test equation however so any attempts are omitted here.

Convergence of the method has been investigated in order to confirm appropriate implementation. The convergence properties of the ERK4C should be of order $\mathcal{O}(h^4)$ for the global truncation errors (GTE) while the local truncation error (i.e the one-step error here) should be of order $\mathcal{O}(h^5)$. These properties are confirmed in Figure (34).

The following includes a demonstration of the ERK4C on both the Van Der Pol and the Prey-Predator problem. The solution found by the algorithm is compared to the solutions from the integrated MATLAB ODE solvers (ODE45 and ODE15s). The solution to the former problem can be found in Figure (35) and Figure (36). These solutions were found using $h = 0.01$. It was necessary to use a small step-size to obtain convergence. Tests were carried out using both $h = 0.1$ and $h = 0.5$. In the first case the solution to the case $\mu = 10$ did not converge while in the second case neither converged.

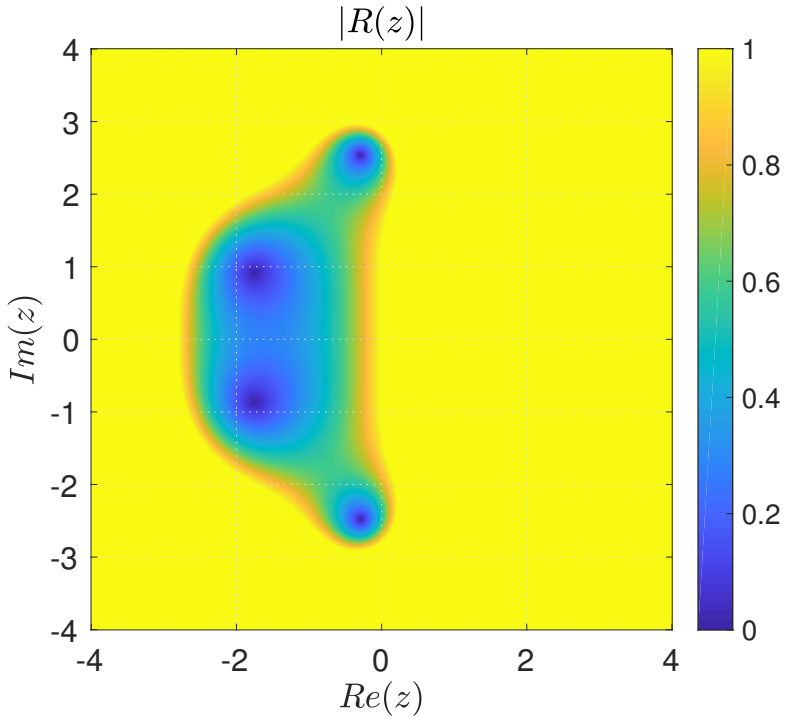


Figure 33: Absolute region of stability for the fourth order Classical Runge-Kutta method.

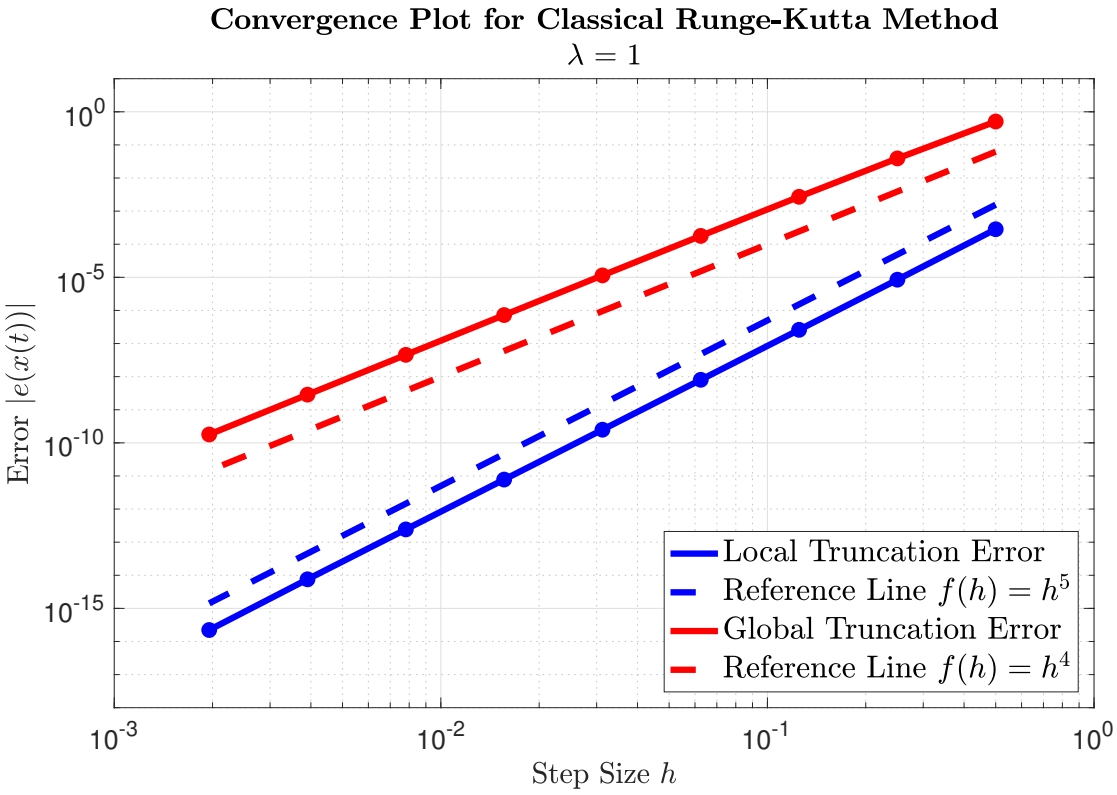


Figure 34: Convergence demonstration for ERF4C. As expected the leading term in the GTE is of fourth order, and the leading term in the LTE is of fifth order.

It is clear from inspection that the solution is in correspondence with the MATLAB algorithms, perhaps even at a higher accuracy as evident from the phase plot. It is clear that both ODE45 and ODE15s lacks smoothness in the solution where the gradient changes abruptly. While this could seem to indicate superiority of the ERK4C over ODE45 and ODE15s a glance at the plot legends will reveal that the amount of steps used by the ERK4C is between 10 and 20 times larger. The reason is that the integrated solvers employ an adaptive step size algorithm in order to obtain a given accuracy. The solution can be improved simply by requesting a higher accuracy and as illustrated in Figure (37) using a relative and absolute tolerance of $A_{tol} = R_{tol} = 10^{-6}$ is enough to achieve an equally smooth solution. Notice at the same time that the solution requires only that the steps taken roughly triples ($224 \rightarrow 656$).

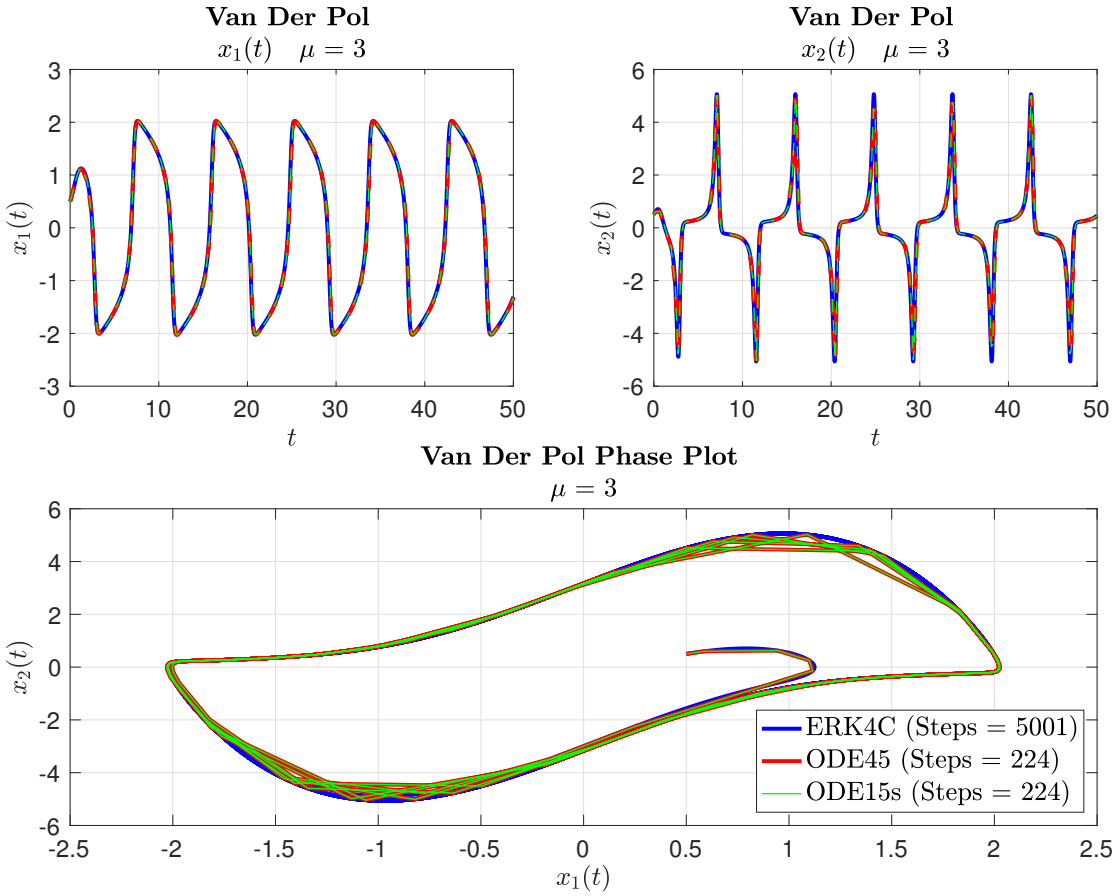


Figure 35: Solution to the VanDerPol for $\mu = 3$ problem using both the implemented ERK4C and MATLABs ODE45 and ODE15s solvers. The test shows that all solvers agree on the solution found by the ERK4C method. While the accuracy appears to be better than that of ODE45 and ODE15 especially for the phase plot, the amount of steps necessary is approximately an order of magnitude larger.

The solution to the Prey Predator problem is in equivalent fashion provided in Figure (38). The ERK4C solution is found using a step size of $h = 0.01$ and using the differential equation parameters $p_1 = p_2 = 2$. The points made for the Van Der Pol problem carries over to this problem: In order for the ODE45 and ODE15s solvers to achieve ERK4C accuracy relative and absolute tolerances of $A_{tol} = R_{tol} = 10^{-9}$ had to be used, however the task is accomplished in just 1/3 of the steps (1501 against 555).

In summary it has been demonstrated through the testing of the implemented ERK4C method that the implementation converges as expected and that it accurately solves both the Van Der Pol and the Prey-Predator problem. The method has however proven in comparison to MATLABs

ODE45 and ODE15s methods to be inefficient as the amount of steps taken (and thus function evaluations) are much larger. In the following section an adaptive step size ERK4C method will be implemented in an attempt to optimize this aspect of the solver.

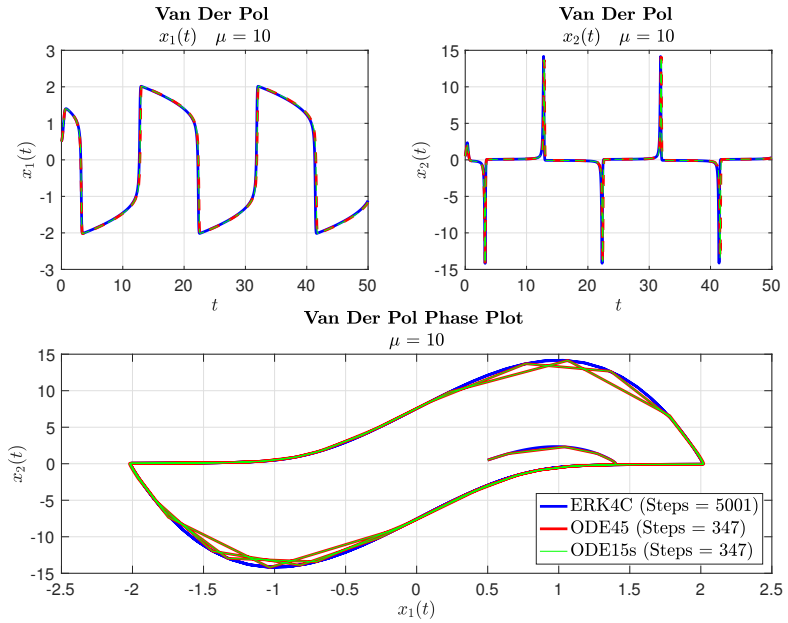


Figure 36: Solution to the VanDerPol for $\mu = 10$ problem using both the implemented ERK4C and MATLABs ODE45 and ODE15s solvers. The test shows that all solvers agree on the solution found by the ERK4C method. While the accuracy appears to be better than that of ODE45 and ODE15 especially for the phase plot, the amount of steps necessary is approximately an order of magnitude larger.

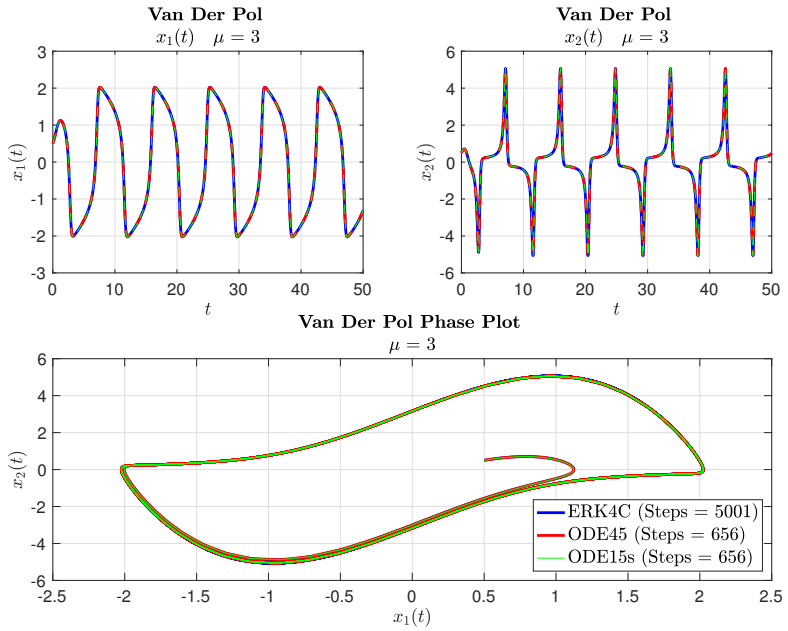


Figure 37: Solution to the VanDerPol for $\mu = 3$ problem using both the implemented ERK4C and MATLABs ODE45 and ODE15s solvers with absolute and relative tolerance of $A_{tol} = R_{tol} = 10^{-6}$. The solution from ODE45 and ODE15s achieves the same smoothness and accuracy from visual inspection as the ERK4C using a tenth of the steps.

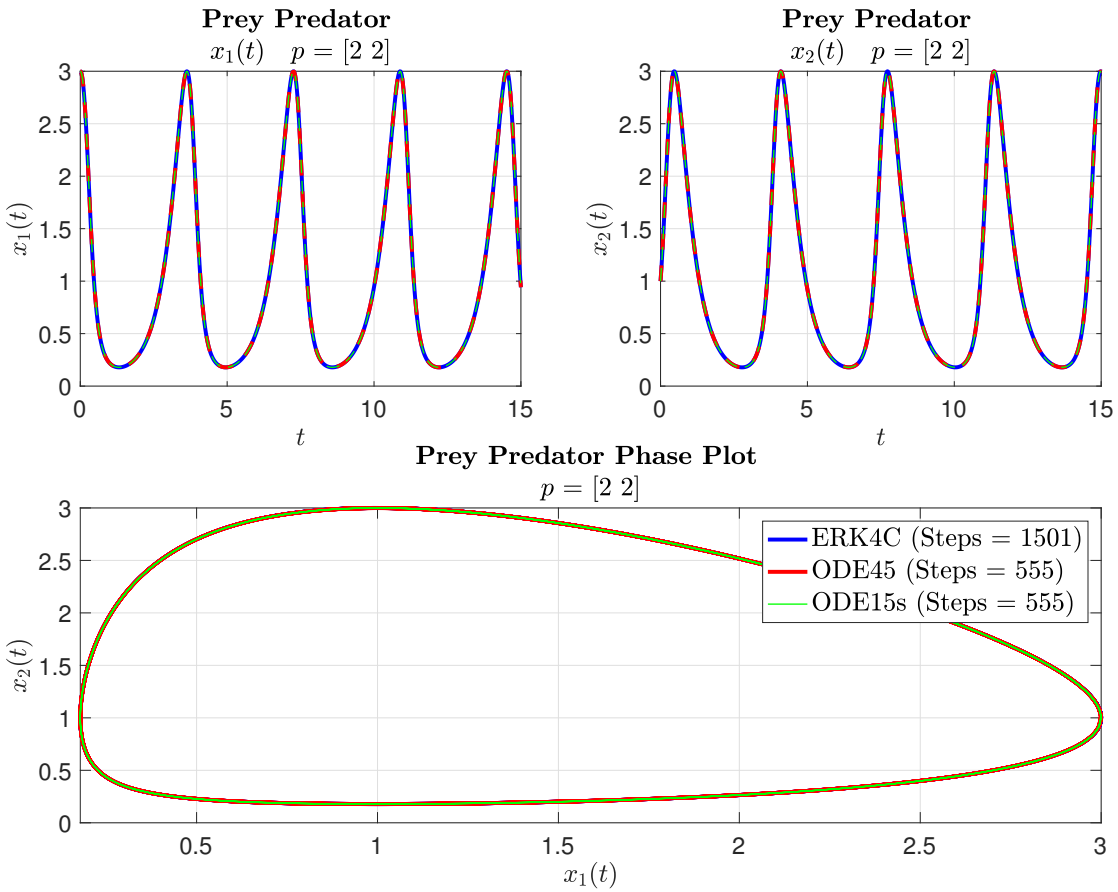


Figure 38: Solution to the Prey-Predator problem with $p_1 = p_2 = 2$ using both the implemented ERK4C and MATLABs ODE45 and ODE15s solvers. The absolute and relative tolerance is set to $A_{tol} = R_{tol} = 10^{-9}$. The MATLAB solvers achieve same solution smoothness and accuracy with 1/3 of the steps used by ERK4C.

7 Classical Runge-Kutta Method: Adaptive Time Step

As it was shown in the previous section in order to optimize algorithm efficiency an adaptive step size extension to the ERK4C solver has to be implemented. The complete MATLAB script for this implementation can be seen in Appendix (12.4). The implementation uses step-doubling and both next and half-step points are computed using the same matrix-format style as described for the fixed step ERK4C. The update of the step size is carried out using the asymptotic step-size controller suited for a fourth order method. This is given by

$$h_{k+1} = \left(\frac{\epsilon_{k+1}}{r_{k+1}}\right)^{1/(1+p)} h_k = \left(\frac{\epsilon_{k+1}}{r_{k+1}}\right)^{1/5} h_k \tag{39}$$

Though (39) is the exact step-size controller, in practice the implementation is carried out using a slightly altered controller such that the change in h is limited by the user introduced parameters f_{min} and f_{max}. This step-size controller is shown in (40). This feature accounts for a smoother step-size transition when the solution exhibits high non-linearity in areas where the gradient is rapidly changing.

$$h = \max\left\{f_{min}\,, \min\left\{f_{max}\,, \left(\frac{\epsilon}{r}\right)^{1/5}\right\}\right\} h \tag{40}$$

The values used here are $f_{min} = 0.1$ and $f_{max} = 5$ as recommended from the slides of Lecture 8 in the course.

The implementation is tested on the usual test equation in order to compare with differences without adaptive step size. This is shown in Figure (39) for $\lambda = \{1, 10, 15, 25\}$. The solution does not display divergence from the analytical near the end of the interval as it was shown to do without the adaptive step size in Figure (32).

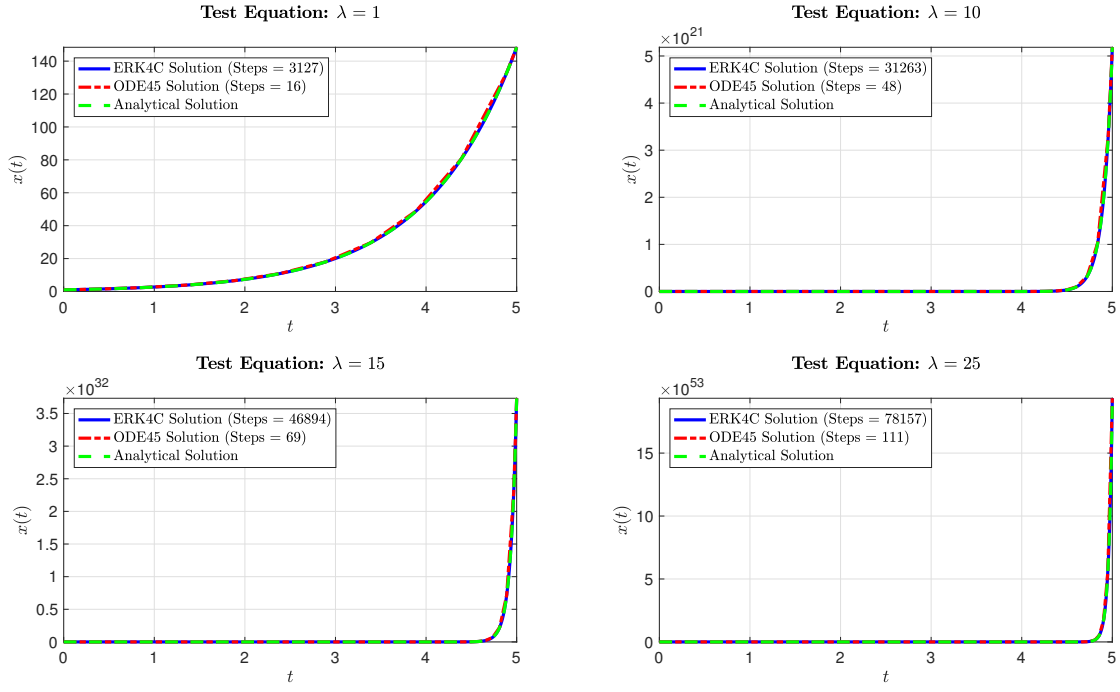


Figure 39: Numerical solutions to the test equation for various values of λ using ERK4C with an adaptive step size using an absolute and relative tolerance of $A_{tol} = R_{tol} = 10^{-3}$. Notice the large amount of steps taken compared to ODE45.

Notice however the huge difference between the steps taken by this new implementation against ODE45. The tolerances used here were $A_{tol} = R_{tol} = 10^{-3}$ (Note that the incomparable toler-

ance used by ODE45 was set to the default which happens to be $A_{tol,ODE45} = R_{tol,ODE45} \approx 10^{-3}$ depending on the problem). Close visual inspection reveals that ODE45 is indeed less accurate. In an attempt to compete with the ODE45 it was investigated for which tolerances roughly the same number of steps were taken. The findings are shown in Figure (40) and reveal that this is achieved for $A_{tol} = R_{tol} = 0.75$. One can notice the non-smoothness of the solution by inspecting the plot associated with $\lambda = 1$. Though not provided in the report a closer look around the solution shows that ODE45 and ERK4C approximates the analytical solution with roughly the same error.

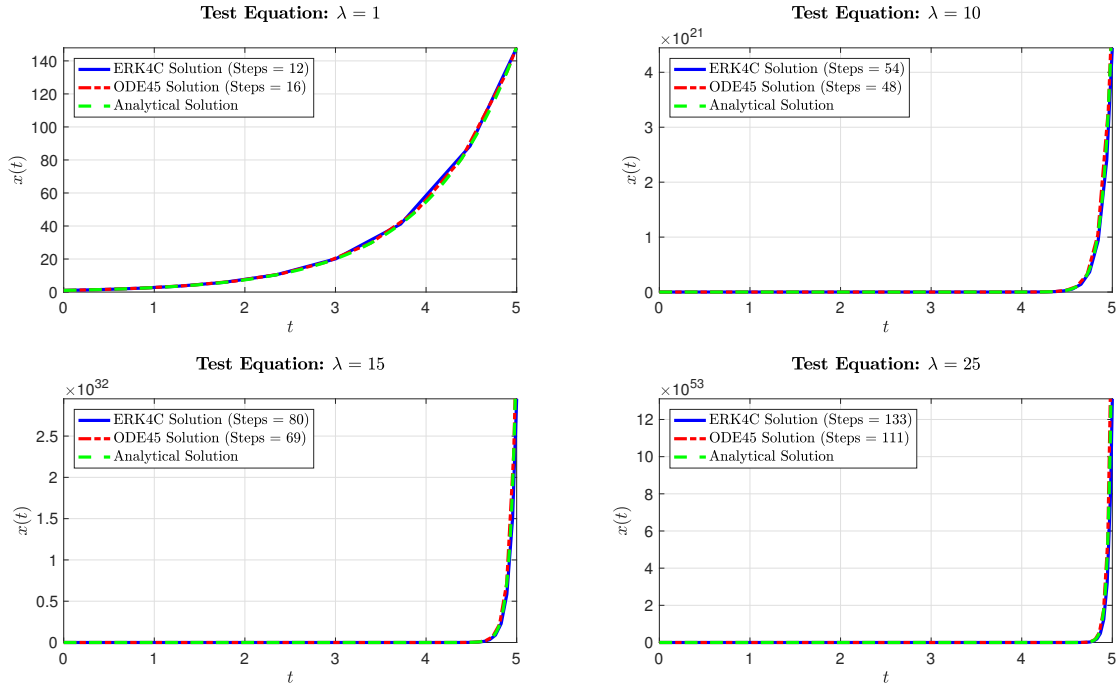


Figure 40: Numerical solutions to the test equation for various values of λ using ERK4C with an adaptive step size using an absolute and relative tolerance of $A_{tol} = R_{tol} = 0.75$. The amount of steps taken is similar to ODE45.

Additional information in the case of $\lambda = 25$ is provided in Figure (41). The initial step size of $h = 0.1$ is seen to rapidly decrease down to $h \approx 0.04$ where it remains. The step accepter described by R also quickly converges toward the desired value given by $\epsilon = 0.8$. While it is not immediately clear from the plot since the solution growths so rapidly, the error plot reveal additional information. It is evident that the errors are ever increasing which was not obvious from visual inspection of the solution plots. Whether or not this is an indication of the instability effects at this eigenvalue is not certain.

The implementation is likewise tested on both Van Der Pol and Prey-Predator as previously. The Van Der Pol plots are shown in Figure (42) and Figure (43) for the cases $\mu = \{3, 10\}$ respectively. The solutions were found using absolute and relative tolerances such as to match ODE45 and ODE15s tolerances which were set to $A_{tol,ODE45,ODE15s} = R_{tol,ODE45,ODE15s} = 10^{-6}$. It required different absolute and relative tolerances for the ER4KC solver depending on the value of μ. The used values were $A_{tol,\mu=3} = R_{tol,\mu=3} = 7 \cdot 10^{-2}$ and $A_{tol,\mu=10} = R_{tol,\mu=10} = 6 \cdot 10^{-2}$. Using adaptive step size it would seem that the ERK4C have been improved to such a degree that it performs equally to both ODE45 and ODE15s on the Van Der Pol problem. In the case of $\mu = 10$ however one can see that the ERK4C solution slightly translates towards the right as time progresses, relative to the solutions from ODE45 and ODE15s. Development in the step size, step-accepter and error is shown in Figure (44). The periodicity of the solution is replicated in these plots.

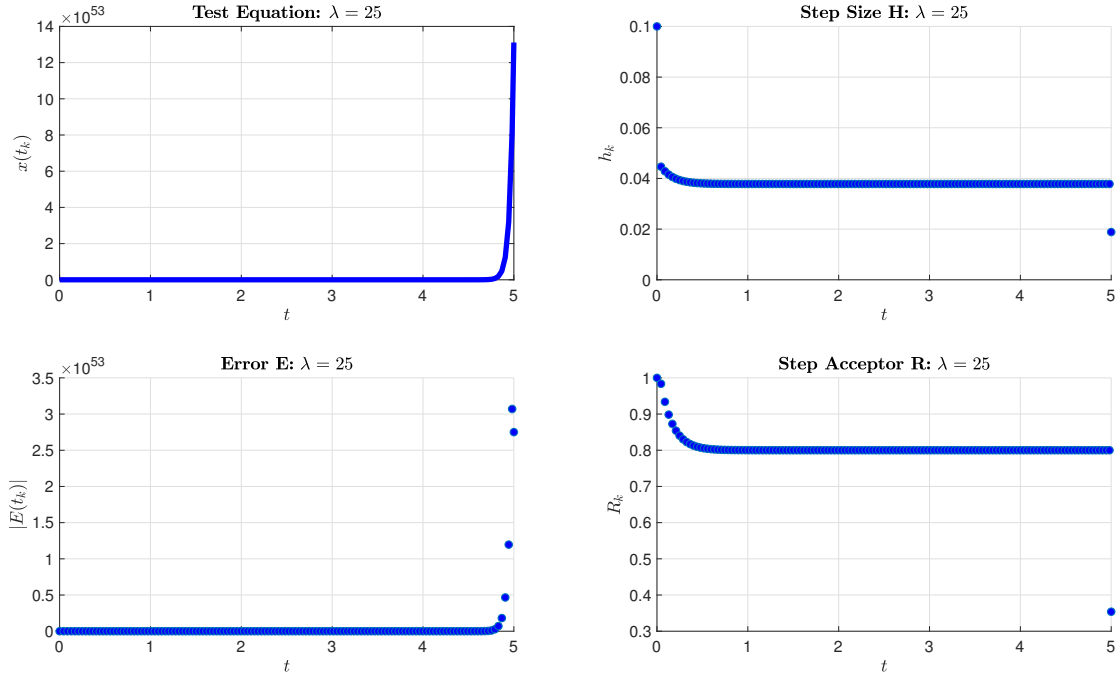


Figure 41: Numerical solution using ER4KC with adaptive step size to the test equation for $\lambda = 25$ with associated plots of the error, step size and step-accepter parameter.

The solution to the Prey-Predator problem can be seen in Figure (45) and the associated parameter plots in Figure (46). The conclusion is equivalent to that just provided for the Van Der Pol problem. Using tolerances for the adaptive ERK4C of $A_{tol} = R_{tol} = 3 \cdot 10^{-2}$ and $A_{tol,ODE45,ODE15s} = R_{tol,ODE45,ODE15s} = 10^{-9}$ the same number of steps taken is achieved at the same accuracy through visual inspection.

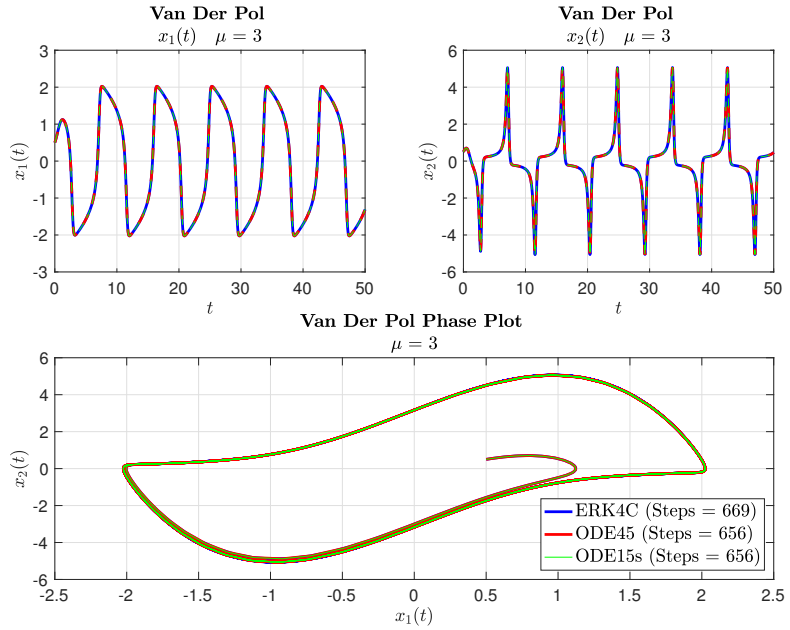


Figure 42: Solution to the VanDerPol problem for $\mu = 3$ using ERK4C with adaptive step size and MATLABs ODE45 and ODE15s solvers. Tolerances used for ERK4C were $7 \cdot 10^{-2}$ while ODE45 and ODE15s used 10^{-6}.

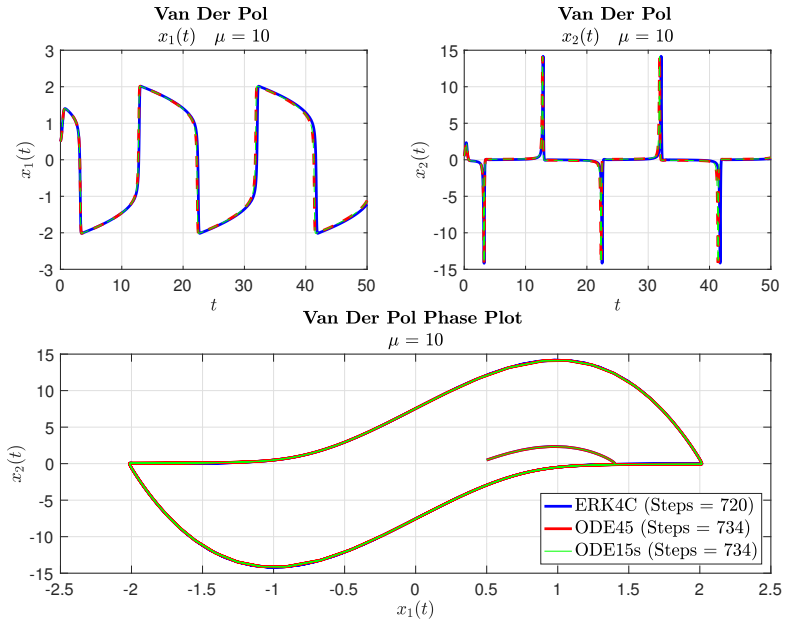


Figure 43: Solution to the VanDerPol problem for $\mu = 10$ using ERK4C with adaptive step size and MATLABs ODE45 and ODE15s solvers. Tolerances used for ERK4C were $6 \cdot 10^{-2}$ while ODE45 and ODE15s used 10^{-6}.

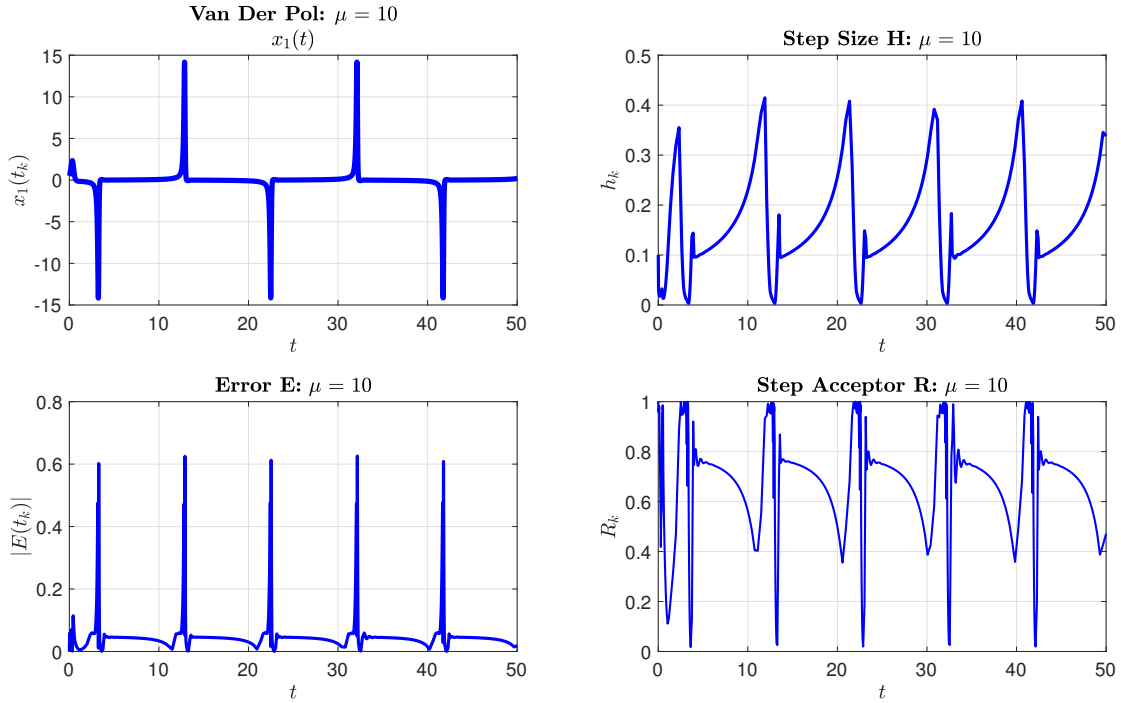


Figure 44: Numerical solution using ERK4C with adaptive step size to the Van Der Pol problem for $\mu = 10$ with associated plots of the error, step size and step-accepter parameter. The large fluctuations in all three parameters are caused by the very steep gradients in the solution.

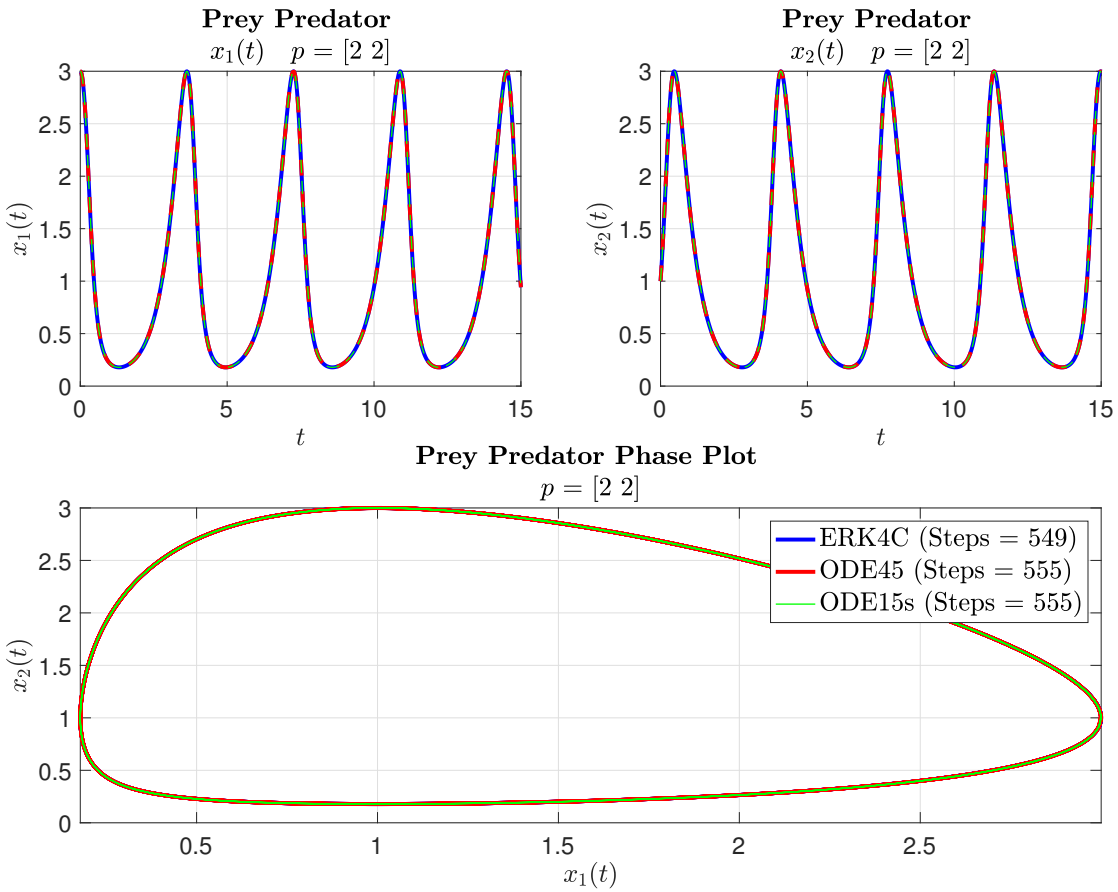


Figure 45: Solution to the Prey-Predator problem with $p_1 = p_2 = 2$ using ERK4C with adaptive step size and MATLABs ODE45 and ODE15s solvers. Tolerances used for ERK4C were $3 \cdot 10^{-2}$ while ODE45 and ODE15s used 10^{-9}.

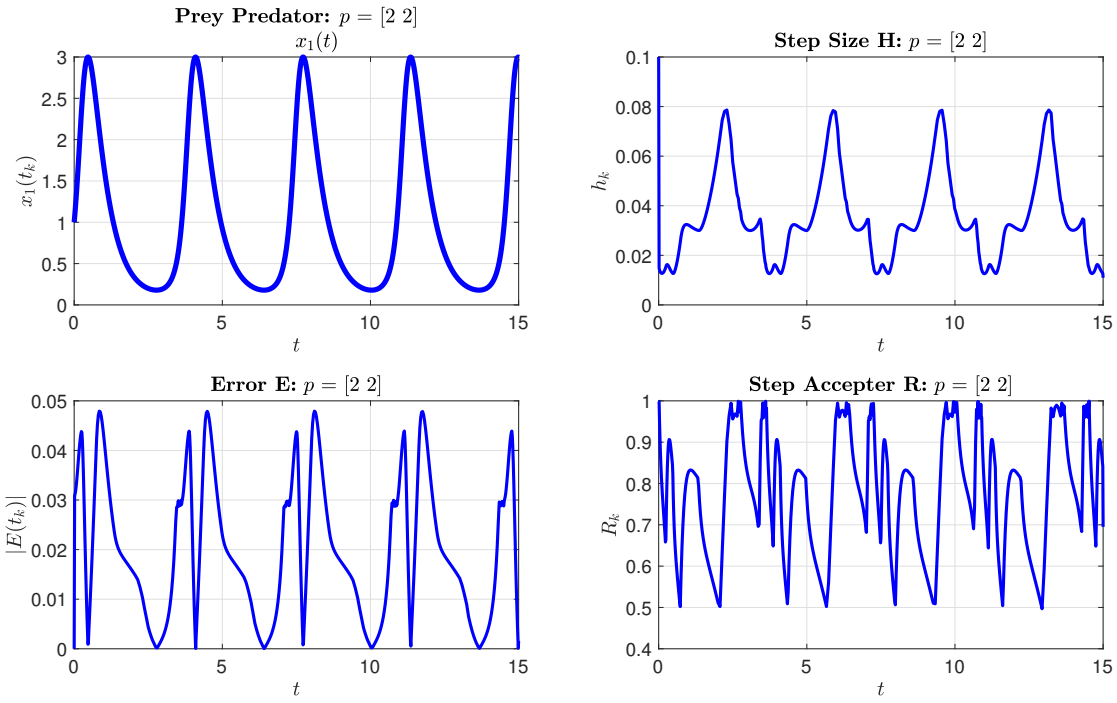


Figure 46: Numerical solution using ERK4C with adaptive step size to the Prey-Predator problem for $p_1 = p_2 = 2$ with associated plots of the error, step size and step-accepter parameter.

8 Dormand-Prince 5(4)

The DormandPrince5(4) is an explicit Runge-Kutta method which uses an adaptive step size. The method calculates both a fifth order and a fourth order solution as indicated by the numbers in its name. The difference between these solutions are regarded as the approximation to the error (of fourth order). The MATLAB solver referred to as ODE45 uses this method [MATLAB, 2018]. The large Butcher Tableau for the method is provided in (41) below.

0	0	0	0	0	0	0	0
1/5	1/5	0	0	0	0	0	0
3/10	3/40	9/40	0	0	0	0	0
4/5	44/45	−56/15	32/9	0	0	0	0
8/9	19372/6561	−25360/2187	64448/6561	−212/729	0	0	0
1	9017/3168	−355/33	46732/5247	49/176	−5103/18656	0	0
1	35/384	0	500/1113	125/192	−2187/6784	11/84	0
X	35/384	0	500/1113	125/192	−2187/6784	11/84	0
$\hat{X}$	5179/57600	0	7551/16695	393/640	−90297/339200	187/2100	1/40

(41)

The implementation process is slightly simpler than for the ERK4C since the embedded error estimation removes the need for the entire step-doubling process. The exact implementation can be seen in Appendix (12.5). Note that the asymptotic step-size controller in this case is altered as described by (39) such the exponent is 1/6 instead. The stability region for the method is shown in Figure (47). It bares large resemblance to that shown for the ERK4C method though this is not odd as the method is just one order higher.

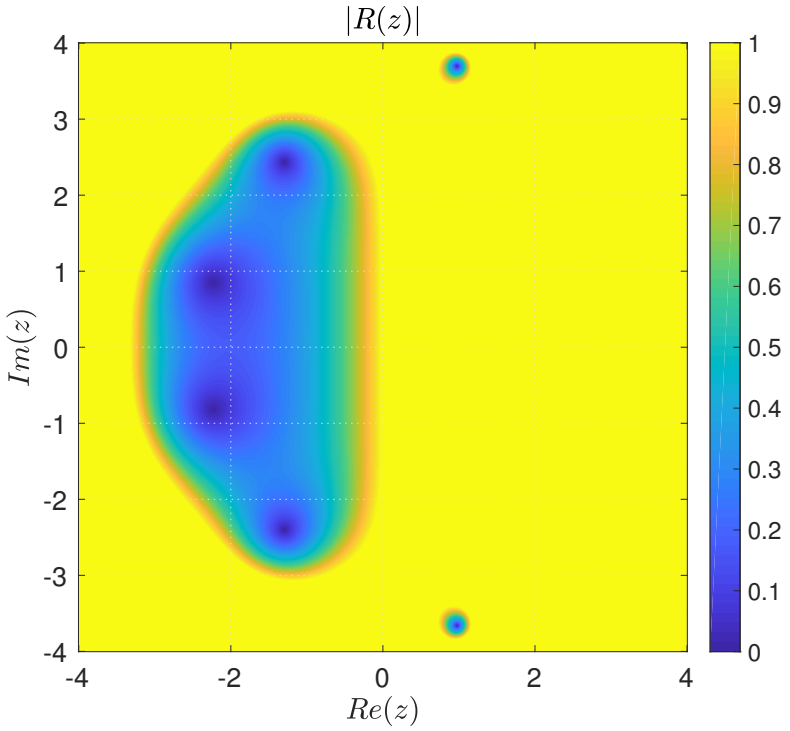


Figure 47: Stability region for the DormandPrince5(4) method. The region is close to identical to that of ERF4KC shown in Figure (33)

The increased order has rewarded two additional regions of absolute convergence. They lie symmetrically around the real-axis roughly at $z = \pm 3 \pm i$ but their radius is quite small. The method has been tested on the test equation against ODE45 and the results are shown in Figure (48). This time results are shown for just two values of $\lambda = \{1\,,\,25\}$ in order to avoid unnecessary repetition. As already mentioned this is exactly the algorithm used by ODE45 thus running against this algorithm is an excellent way to confirm proper implementation. The tolerances of ODE45 were chosen to 10^{-6}. In order for the two methods to use roughly the same steps the tolerances of DormandPrince had to be set to $A_{tol,DP} = R_{tol,DP} = 7 \cdot 10^{-5}$. In contrast to previously in the case of the ERK4C it indeed seems that the tolerances assume similar values now.

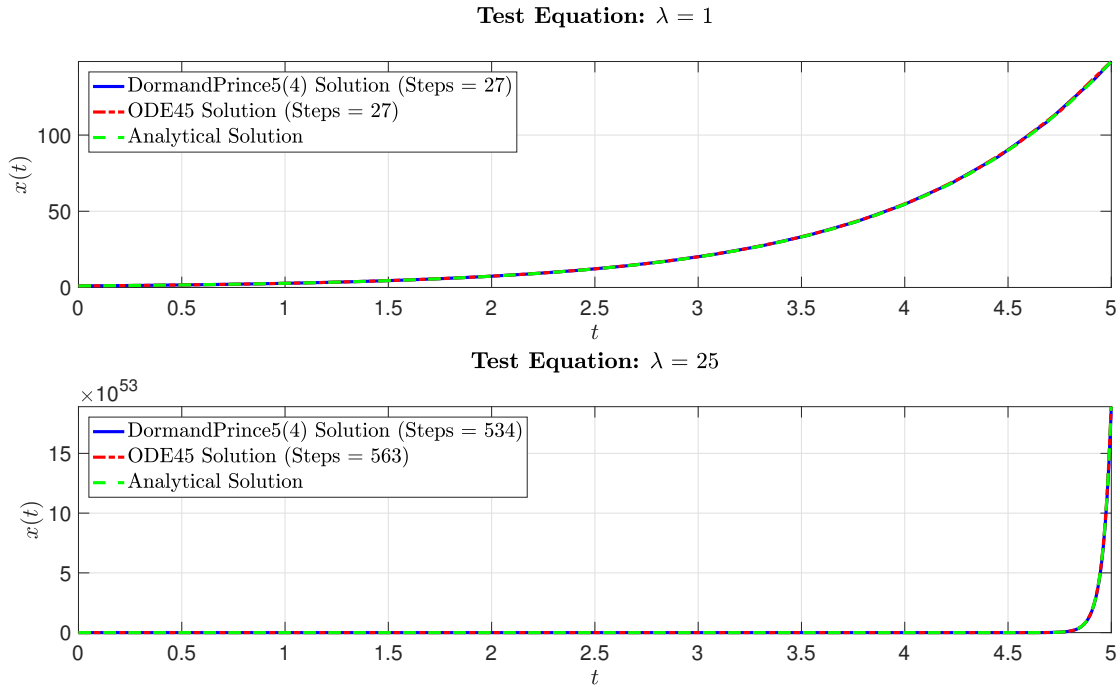


Figure 48: Numerical solution to the test equation for $\lambda = \{1\,,\,25\}$ using the DormandPrince algorithm. The solution is seen to match that from ODE45 and the analytical and the amount of steps taken is almost identical to ODE45.

The DormandPrince tested on the VanDerPol problem provided the results shown in Figure (49), Figure (50) and Figure (51). Tolerances used were 10^{-6} for ODE45 and ODE15s while the DormandPrince used μ-dependant values of $A_{tol,DP,\mu=3} = R_{tol,DP,\mu=3} = 3.5 \cdot 10^{-5}$ and $A_{tol,DP,\mu=10} = R_{tol,DP,\mu=10} = 2.5 \cdot 10^{-5}$. The solutions for the various methods are indistinguishable by visual inspection as expected (or hoped for at least) thus we conclude that implementation is at the very least correct to sufficient degree.

The tests on the Prey-Predator problem are likewise shown in Figure (52) and Figure (53). The conclusions remain identical to those just presented for the Van Der Pol. Tolerances used for ODE45 and ODE15s were 10^{-9} while DormandPrince was set to $A_{tol,DP} = R_{tol} = 1.4 \cdot 10^{-5}$.

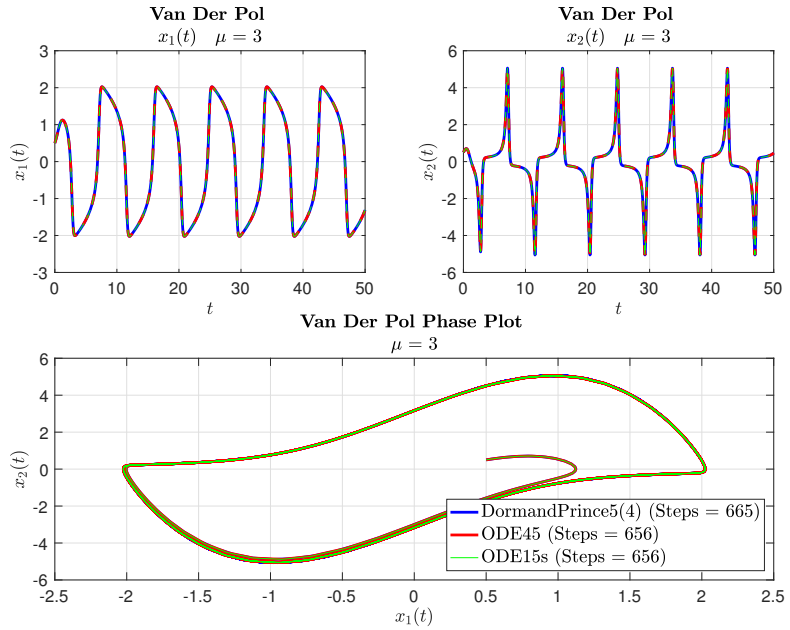


Figure 49: Solution to the VanDerPol problem for $\mu = 3$ using DormandPrince5(4) and MATLABs ODE45 and ODE15s solvers. Tolerances used for DormandPrince were $3.5 \cdot 10^{-5}$ while ODE45 and ODE15s used 10^{-6}

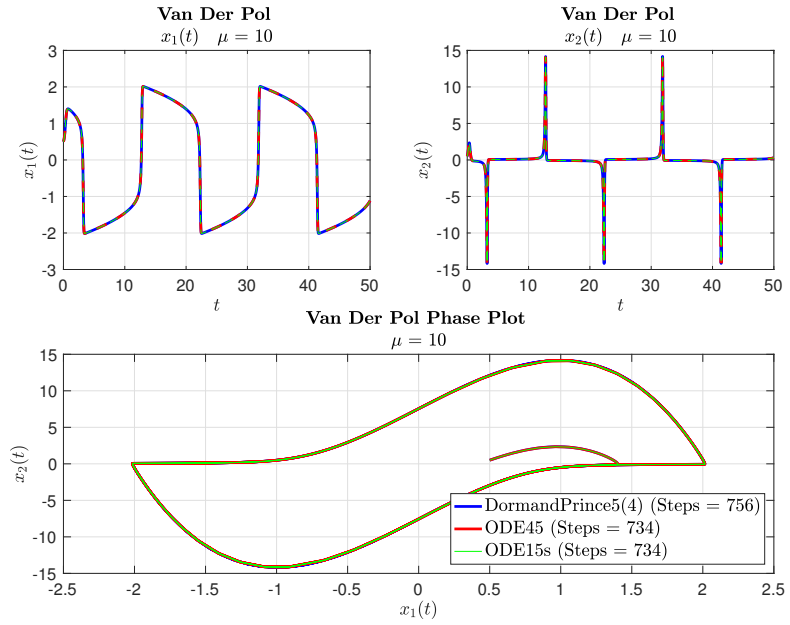


Figure 50: Solution to the VanDerPol problem for $\mu = 10$ using DormandPrince5(4) and MATLABs ODE45 and ODE15s solvers. Tolerances used for DormandPrince were $2.5 \cdot 10^{-5}$ while ODE45 and ODE15s used 10^{-6}

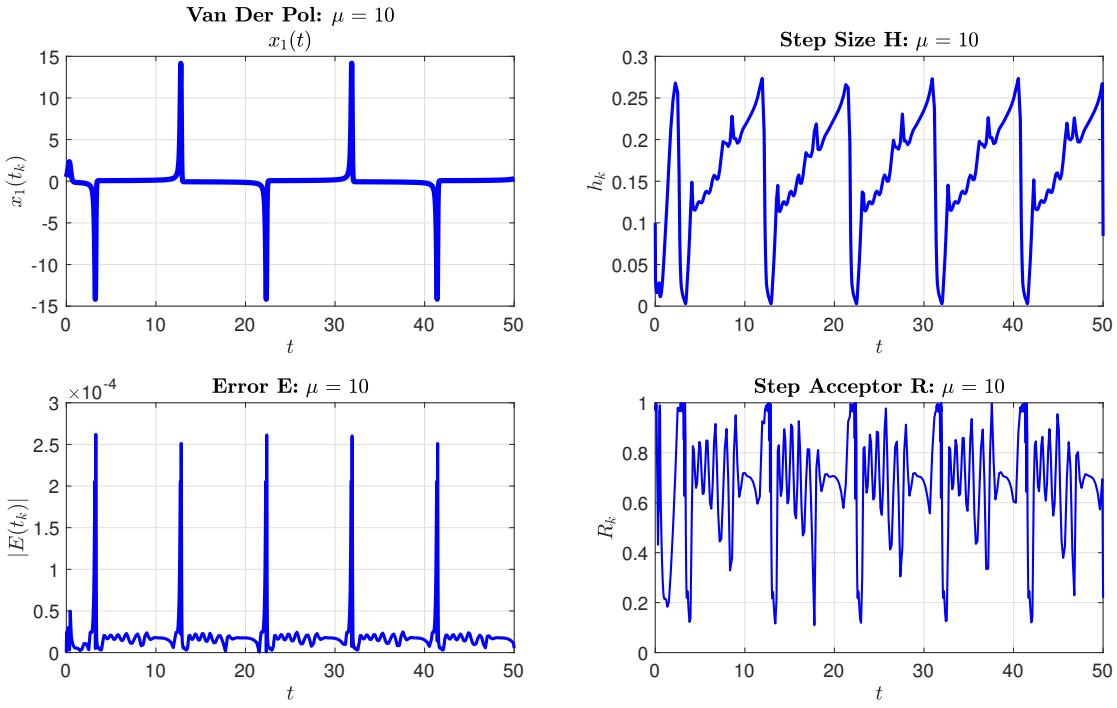


Figure 51: Numerical solution DormandPrince5(4) to the Van Der Pol problem for $\mu = 10$ with associated plots of the error, step size and step-accepter parameter.

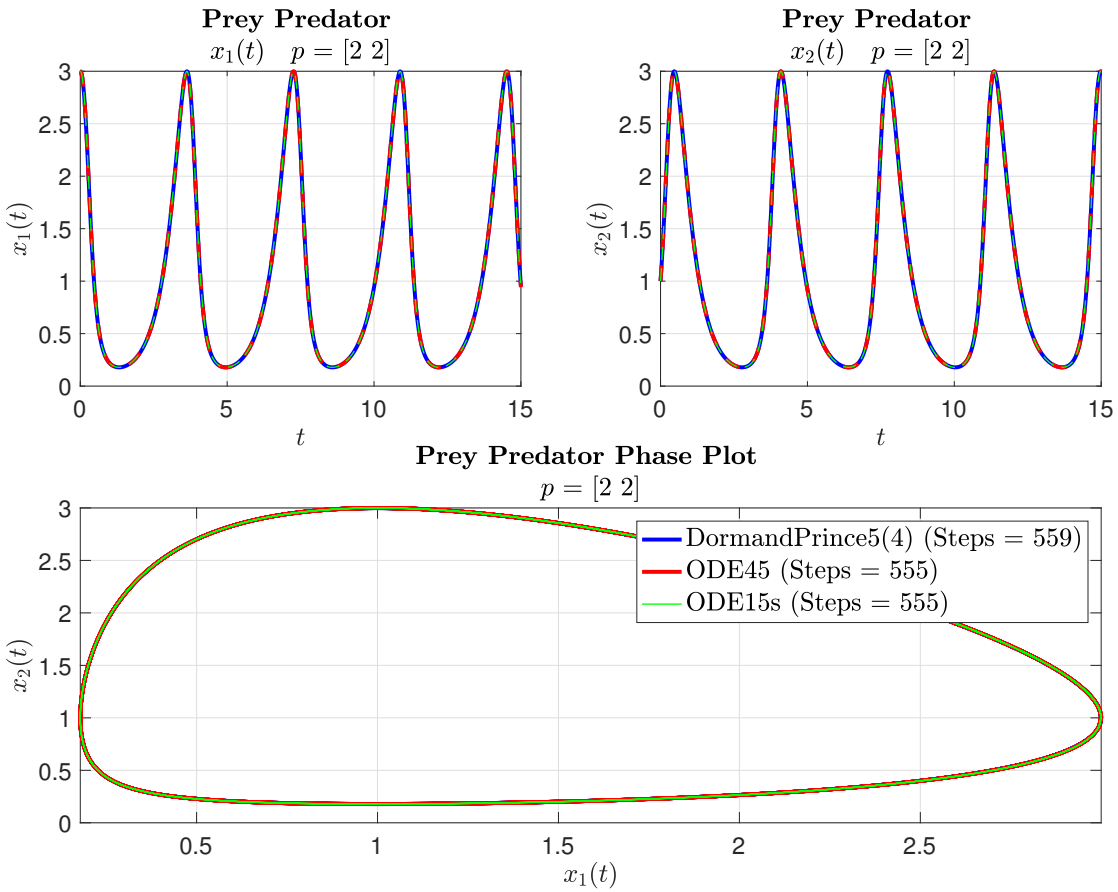


Figure 52: Solution to the Prey-Predator problem with $p_1 = p_2 = 2$ using DormandPrince, MATLABs ODE45 and ODE15s. Tolerances used for DormandPrince were $1.4 \cdot 10^{-5}$ while ODE45 and ODE15s used 10^{-9}.

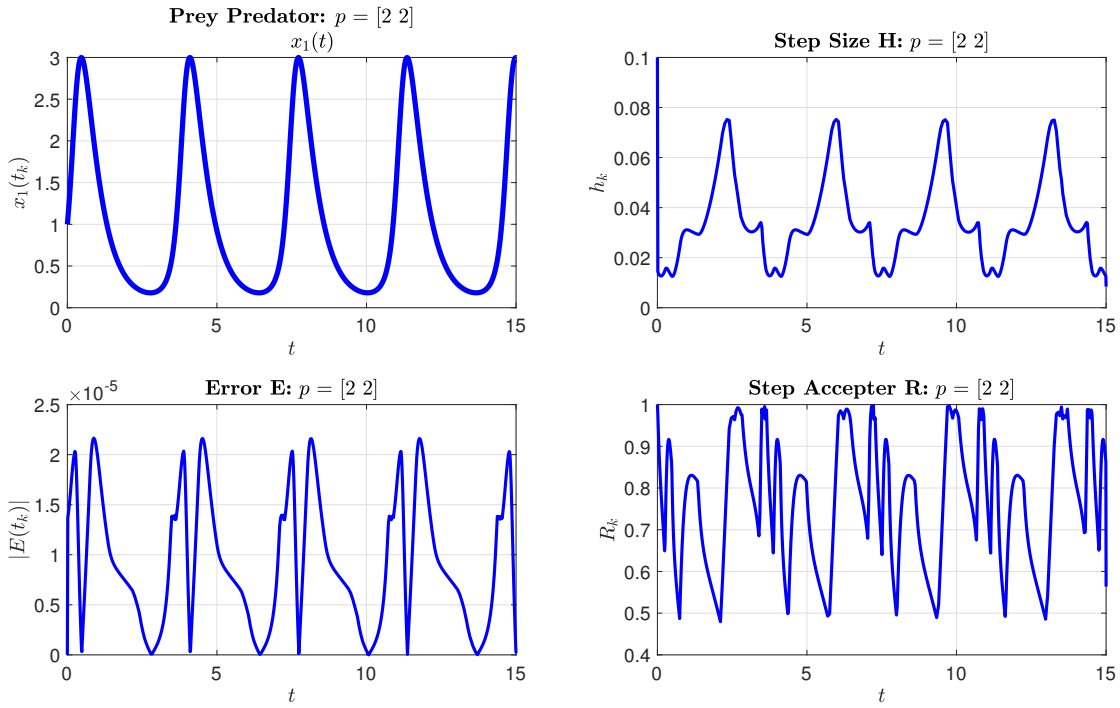


Figure 53: Numerical solution using DormandPrince to the Prey-Predator problem for $p_1 = p_2 = 2$ with associated plots of the error, step size and step-accepter parameter.

Mini Project 3

9 Design your own Explicit Runge-Kutta Method

9,1)

The design of a three-staged third order Runge-Kutta method with a second order embedded method can be achieved by satisfying the consisting and order conditions. Consider the general Butcher Tableau given in (42)

$$\begin{array}{c|ccc} c_1 & a_{11} & a_{12} & a_{13} \\ c_2 & a_{21} & a_{22} & a_{23} \\ c_3 & a_{31} & a_{32} & a_{33} \\ \hline X & b_1 & b_2 & b_3 \\ \hat{X} & \hat{b}_1 & \hat{b}_2 & \hat{b}_3 \end{array} \tag{42}$$

The method is explicit so immediately six entries in the A-matrix vanish. It is desired that the first stage value is given by $T_1 = t_n$ and $X_1 = x_n$ thus an additional constraint is that $c_1 = a_{11} = 0$. The system then reduces to

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ c_2 & a_{21} & 0 & 0 \\ c_3 & a_{31} & a_{32} & 0 \\ \hline X & b_1 & b_2 & b_3 \\ \hat{X} & \hat{b}_1 & \hat{b}_2 & \hat{b}_3 \end{array} \tag{43}$$

Unfolding the consistency equations $Ce = Ae$ yields the following two equations

$$c_2 = a_{21}$$
$$c_3 = a_{31} + a_{32}$$

The order conditions (see (64) for details) up to third order yields four additional equations

$$1 = b_1 + b_2 + b_3$$
$$\frac{1}{2} = b_2 c_2 + b_3 c_3$$
$$\frac{1}{3} = b_2 c_2^2 + b_3 c_3^2$$
$$\frac{1}{6} = b_3 a_{32} c_2$$

The system has two additional free parameters. The solution can be fixed by setting $c_3 = 1$. This makes sense as the final step then ends in the next temporal iterate $T_3 = t_n + h$. Furthermore fix $a_{21} = 1/2$. The solution coefficients are then uniquely defined. The embedded method is a repeated process of fulfilling the order conditions up the second order using the found coefficients of the C and A matrices. The 2. order equations are

$$1 = \hat{b}_1 + \hat{b}_2 + \hat{b}_3$$
$$\frac{1}{2} = \frac{1}{2}\hat{b}_2 - \hat{b}_3$$

The method must also *not* satisfy at least one third order condition. Fixation of $b_2 = 1/2$ yields one such solution. In conclusion the ERK3(2) method takes the form provided in (45) where the error coefficients are also stated. These are calculated in the following section.

9,2)

The error coefficients can readily be derived as the difference between the 3.order coefficients and the embedded coefficients.

$$e = \left[\frac{1}{6} - \frac{1}{4}, \frac{2}{3} - \frac{1}{2}, \frac{1}{6} - \frac{1}{4}\right] = \left[-\frac{1}{12}, \frac{1}{6}, -\frac{1}{12}\right] \tag{44}$$

9,3)

The Butcher Tableau takes the following form

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ 1/2 & 1/2 & 0 & 0 \\ 1 & -1 & 2 & 0 \\ \hline X & 1/6 & 2/3 & 1/6 \\ \hat{X} & 1/4 & 1/2 & 1/4 \\ \hline E & -1/12 & 1/6 & -1/12 \end{array} \tag{45}$$

9,4)

The implemented ERK3(2) solver is presented in Appendix (12.6). The details of the implementation are much like the Classical Runge-Kutta method with an additional error calculation added. The solution to the test equation can be seen in Figure (54). The solution is found using a step size of $h = 0.1$. A close inspection reveals that the method performs very well when compared to both ODE45 and ODE15s though the amount of steps taken are roughly twofold higher.

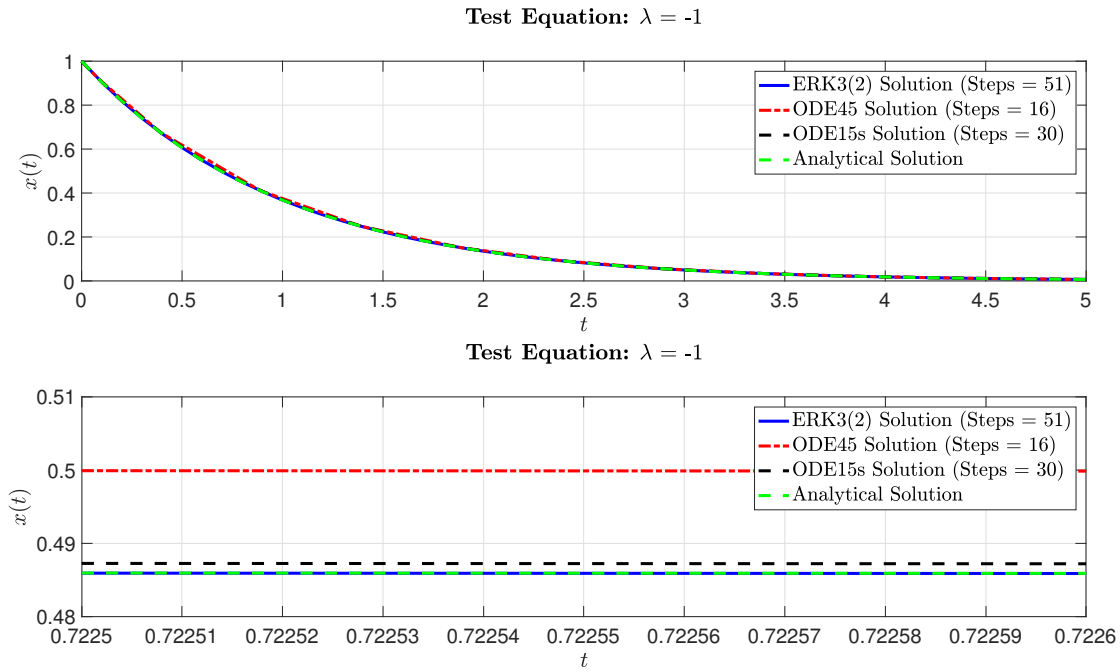


Figure 54: Numerical solution to the test equation using the derived ERK3(2) method.

9,5)

The method dictates that the local truncation errors should be order three. The convergence plot shown in Figure (55) demonstrates this property as expected.

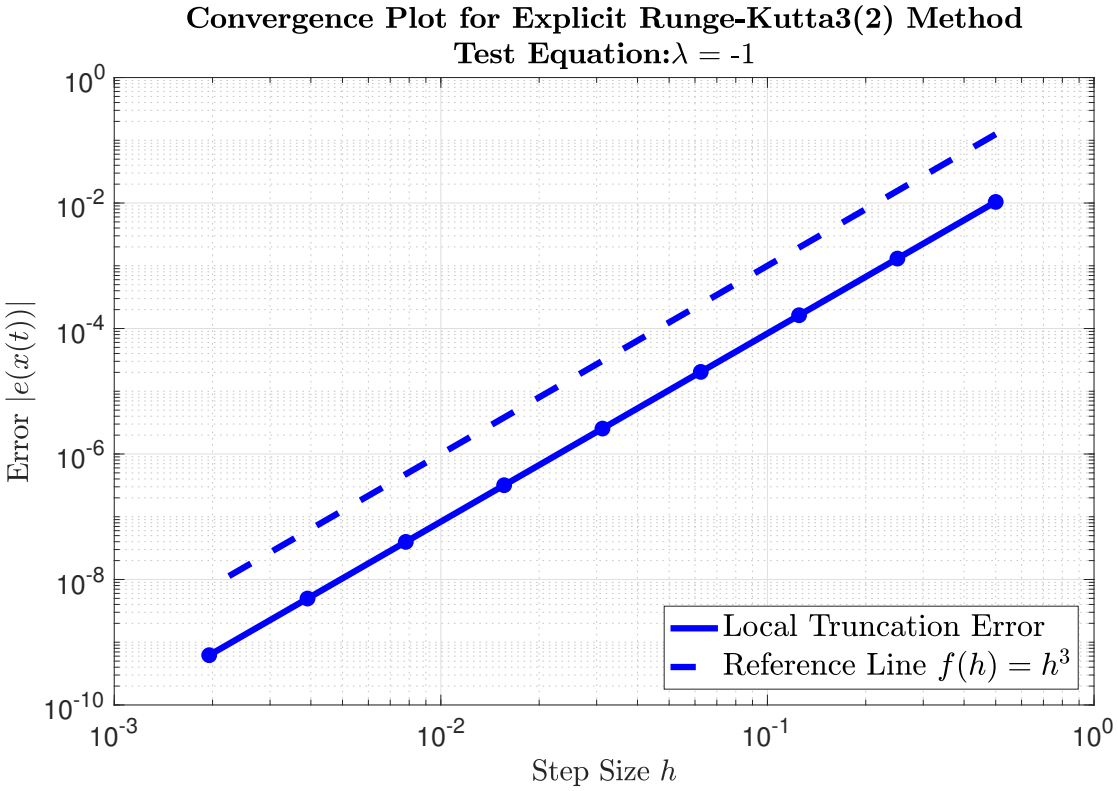


Figure 55: Convergence test performed on the test equation with $\lambda = -1$ for the ERK3(2) method. The plot demonstrates that the local truncation errors are of order 3 as expected.

9,6)

The transfer function can be computed from

$$
\begin{aligned}
R(\lambda h) &= 1 + zb^T \left(I - \lambda h A\right)^{-1} e \\
&= 1 + (\lambda h) \begin{bmatrix} 1/6 \\ 2/3 \\ 1/6 \end{bmatrix}^T \left(\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} - \lambda h \begin{bmatrix} 0 & 0 & 0 \\ 1/2 & 0 & 0 \\ -1 & 2 & 0 \end{bmatrix} \right)^{-1} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \\
&= 1 + \lambda h \begin{bmatrix} 1/6 \\ 2/3 \\ 1/6 \end{bmatrix}^T \begin{bmatrix} 1 \\ \frac{1}{2}\lambda h \\ (\lambda h)^2 + \lambda h + 1 \end{bmatrix} \\
&= \underline{\underline{1 + \lambda h + \frac{1}{2}(\lambda h)^2 + \frac{1}{6}(\lambda h)^3}}
\end{aligned}
$$

The transfer function is the three first terms in the Taylor-expansion of $\exp(\lambda h)$. The function is shown in Figure (56).

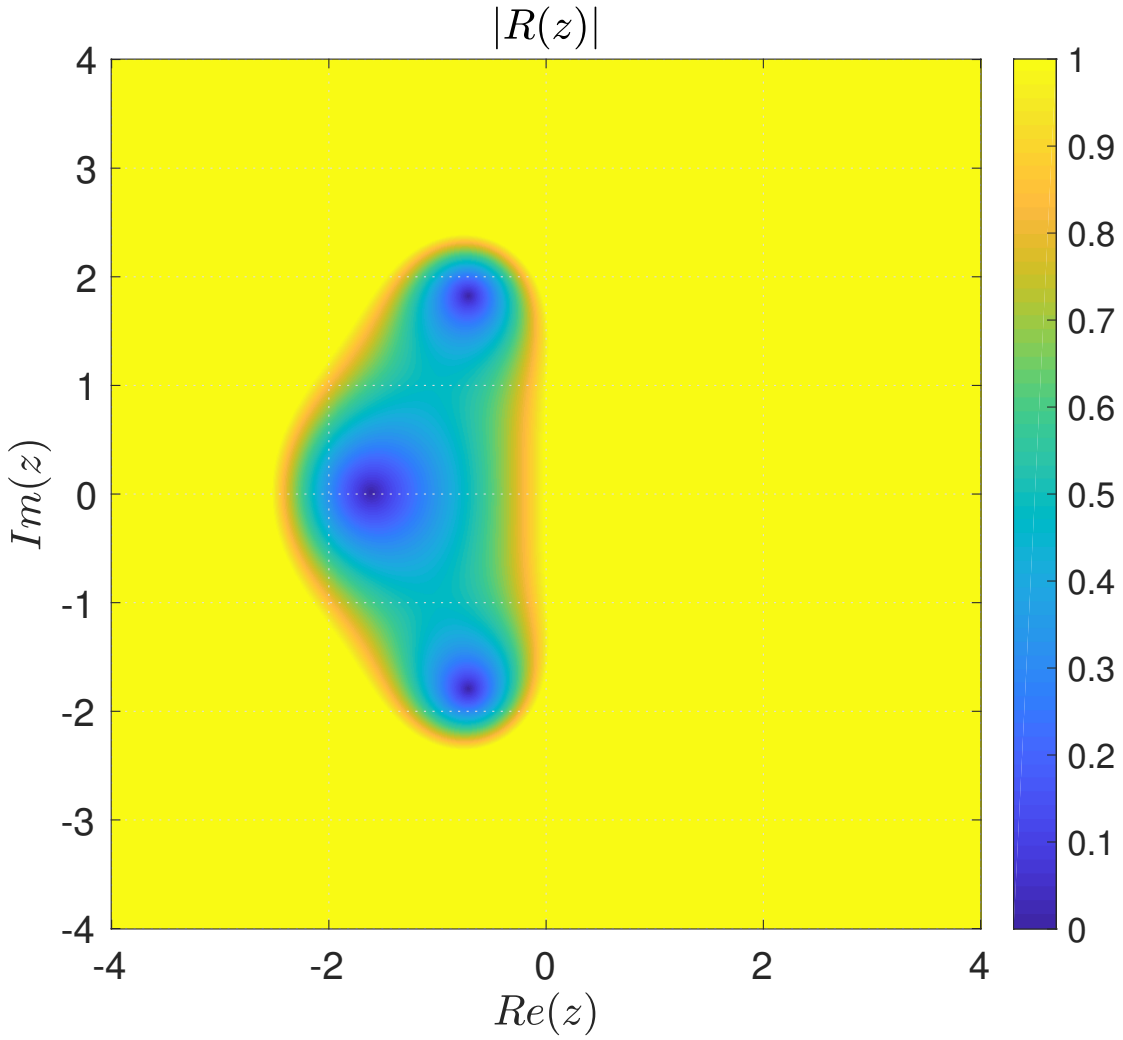


Figure 56: Region of absolute stability for the ERK3(2) method.

9,7)

The ERK3(2) was tested on the Van Der Pol problem against the ODE15s solver. The result is shown in Figure (57). These are found using $h = 0.01$ for the ERK3(2). Absolute and relative tolerance set for ODE15s were 10^{-6}. The two solutions are seen to match each other though the number of steps taken by the ERK3(2) is much larger than ODE15s.

An additional extra method with adaptive step size has been implemented. This can be found in Appendix (12.7). The method was also tested on the Van Der Pol problem using tolerances $A_{tol} = R_{tol} = 5 \cdot 10^{-4}$. The tolerance for ODE15s was set to 10^{-6}. The results and associated parameter statistics can be found in Figure (58) and Figure (59). In all cases the ERK3(2) method has proven to perform well against ODE15s and as such confirms correct implementation and usefulness of the method.

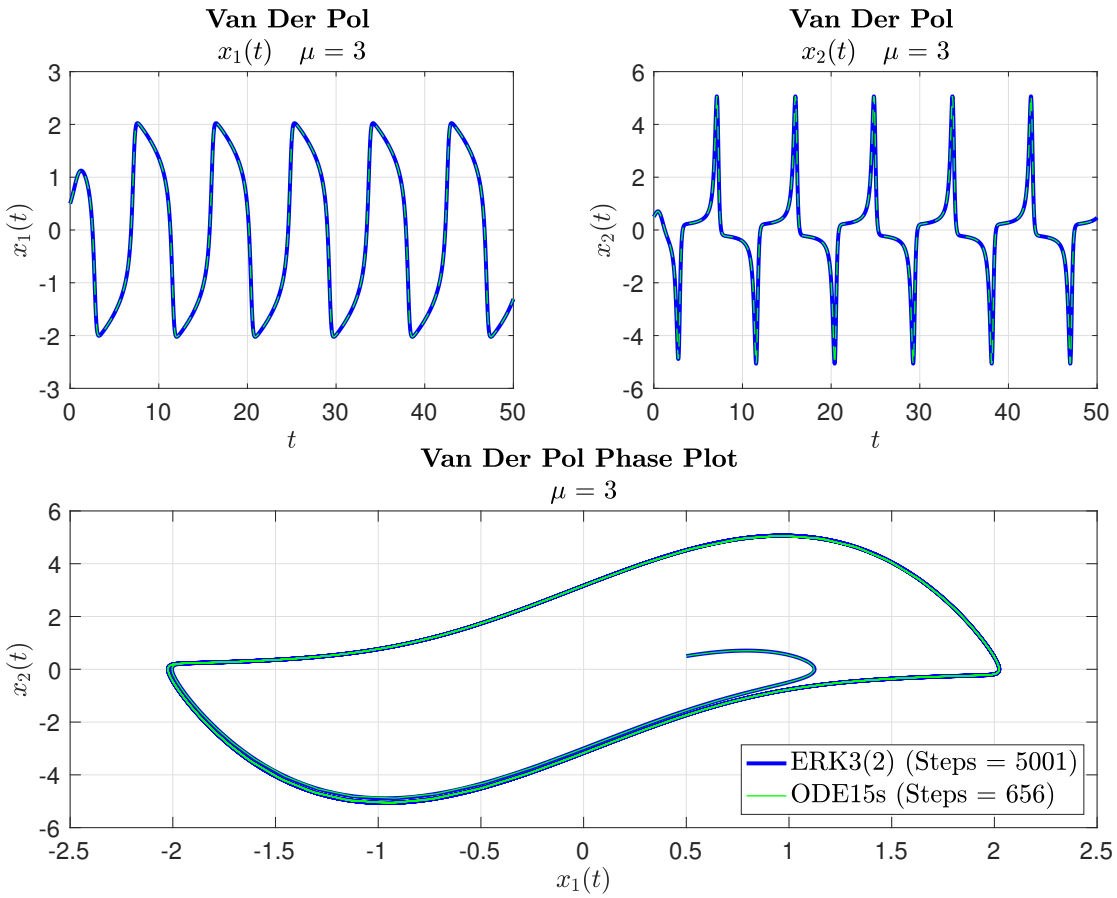


Figure 57: Numerical solution to the Van Der Pol problem for $\mu = 3$ using both ERK3(2) and ODE15s.

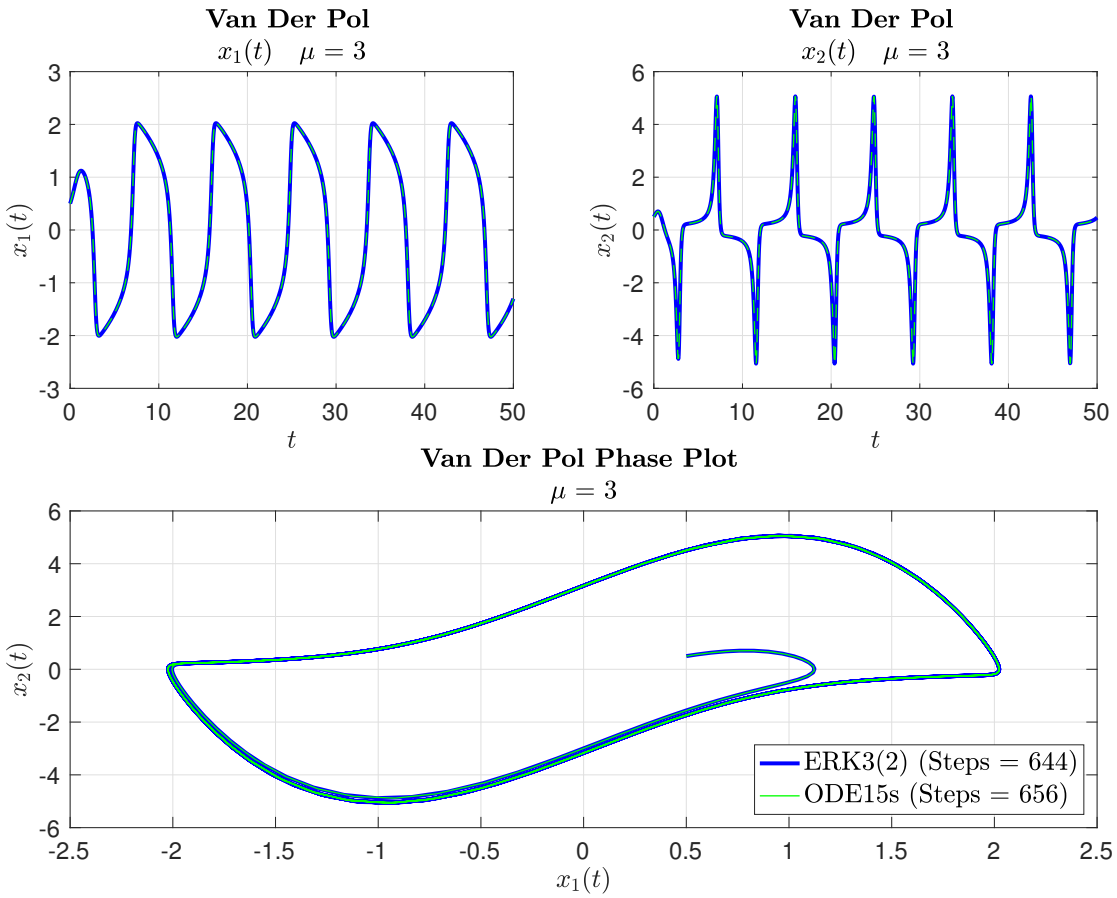


Figure 58: Numerical solution to the Van Der Pol problem for $\mu = 3$ using both ERK3(2) with adaptive step size and ODE15s.

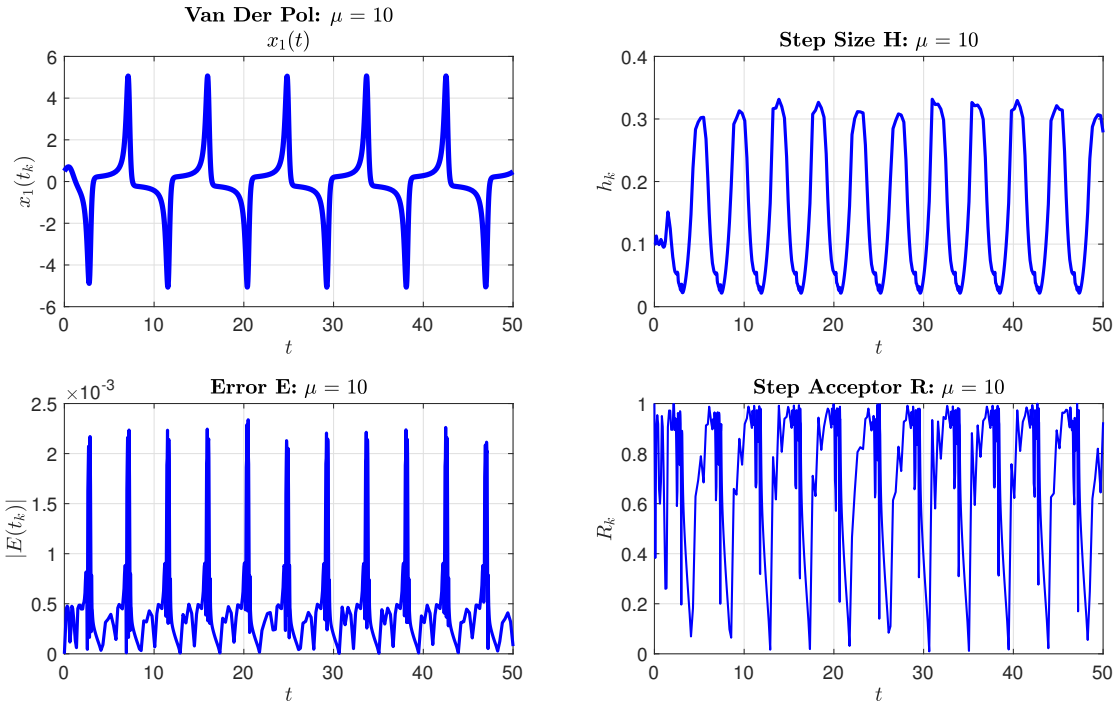


Figure 59: Numerical solution using ERK3(2) with adaptive step size to the Van Der Pol problem for $\mu = 3$ with associated plots of the error, step size and step-accepter parameter.

10 ESDIRK23

The Explicit Singly Diagonal Implicit Runge-Kutta (ESDIRK) methods is a series of explicit methods with implicit inner stages, which has some nice stability properties. Due to the shape of the ESDIRK methods the Butcher Tableau contains the same elements in the diagonal which enables the use of an approximation to the Jacobian to solve the inner implicit stages in a so-called inexact Newton solver. This is turn the reduces the number of Jacobian evaluations. More on this in the following sections regarding the implementation of the method.

10,1)

Firstly we see that the ESDIRK23 method has three stages ($s = 3$). All the ESDIRK-methods follows an explicit scheme, and have the feature that the diagonal elements of the Butcher Tableau are the same (referred to by γ) except for the first which has to be zero for the scheme to be explicit. Thus the Butcher Tableau we are looking at takes the shape seen in equation 46.

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ c_2 & a_{21} & \gamma & 0 \\ c_3 & a_{31} & a_{32} & \gamma \\ \hline x & b_1 & b_2 & b_3 \\ \hat{x} & \hat{b}_1 & \hat{b}_2 & \hat{b}_3 \\ \hline e & d_1 & d_2 & d_3 \end{array} \tag{46}$$

Considering the Butcher Tableau given in eq. 46 one can write the two following consistency equations:

$$\begin{aligned} c_2 &= a_{21} + \gamma \\ c_3 &= a_{31} + a_{32} + \gamma \end{aligned} \tag{47}$$

Furthermore we see that the ESDIRK23-method uses a second order method to progress, and the embedded scheme used to estimate the error is of third order. This leads to the following order equations:

$$\begin{aligned} 1 &= b_1 + b_2 + b_3 \\ \frac{1}{2} &= b_2 c_2 + b_3 c_3 \end{aligned} \tag{48}$$

The method should be A-stable and L-stable. This means that the second order explicit method needs to be A-stable and L-stable. For the second order scheme to be A-stable the last row in the Butcher A matrix must be equal to the Butcher b vector. Note that this makes the equations for the first order-condition and the second consistency-condition identical such that $c_3 = 1$.
For it to be L-stable the following stability equation must be satisfied:

$$0 = a_{21} b_2 - \gamma b_1 \tag{49}$$

We now only need one more equation to find the unknowns in the Butcher A matrix, b vector, and c vector. Here we use the equation for the second stage given on slide 65 in the power point (Explicit Singly Diagonal Implicit Runge-Kutta (ESDIRK) Integration Methods), which takes the form

$$\gamma c_2 = \frac{1}{2} c_2^2 \tag{50}$$

To determine the Butcher $\hat{b}$ vector we use the following order equation, coming from the requirement that the embedded method needs to be of third order.

$$\begin{aligned} 1 &= \hat{b}_1 + \hat{b}_2 + \hat{b}_3 \\ \frac{1}{2} &= \hat{b}_2 c_2 + \hat{b}_3 c_3 \\ \frac{1}{3} &= \hat{b}_2 c_2^2 + \hat{b}_3 c_3^2 \\ \frac{1}{6} &= 2\hat{b}_2 c_2 \gamma + \hat{b}_3 (c_2 b_2 + c_3 b_3) \end{aligned} \tag{51}$$

Since there is only three unknowns in the Butcher $\hat{b}$ vector only the first three order equations in eq. 51 will be needed and used.

The values of the elements in the Butcher tableau are then determined by solving the above equations (eq. 47, 48, 49, 50 and the Butcher Tableau ends up taking the form

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ 2\gamma & \gamma & \gamma & 0 \\ 1 & \frac{\sqrt{2}}{4} & \frac{\sqrt{2}}{4} & \gamma \\ \hline x & \frac{\sqrt{2}}{4} & \frac{\sqrt{2}}{4} & \gamma \\ \hat{x} & \frac{4-\sqrt{2}}{12} & \frac{4+3\sqrt{2}}{12} & \frac{2-\sqrt{2}}{6} \\ \hline e & \frac{\sqrt{2}-1}{3} & -\frac{1}{3} & \frac{2-\sqrt{2}}{3} \end{array} \qquad \text{where:} \quad \gamma = \frac{2-\sqrt{2}}{2} \tag{52}$$

Note that when solving the system it returns two possible values for $\gamma = \frac{2 \pm \sqrt{2}}{2}$, however only the one with the minus is a valid solution for the scheme when requiring that $0 < c_2 < 1$.

The solution shown in eq. 52 matches the one given in the lecture notes (Explicit Singly Diagonal Implicit Runge-Kutta (ESDIRK) Integration Methods).

Writing out the scheme for the ESDIRK23:

$$\begin{aligned} T_1 &= t_n & X_1 &= x_n \\ T_2 &= t_n + 2\gamma h & X_2 &= x_n + h\gamma f(T_1, X_1) + h\gamma f(T_2, X_2) \\ T_3 &= t_n + h & X_3 &= x_n + h\frac{6\gamma - 1}{12\gamma} f(T_1, X_1) + h\frac{6\gamma - 1}{12\gamma} f(T_2, X_2) + h\gamma f(T_3, X_3) \\ t_{n+1} &= T_3 & x_{n+1} &= X_3 \end{aligned} \tag{53}$$

where t_n and x_n is the time and function value at step n, respectively. Note that the value for the next step (x_{n+1}) is equal to the third inner stages, which is due to the last row in Butcher A-matrix being equal to the Butcher b vector. This saves us computing the value of the next step (x_{n+1}).

10,2)

The code for out implementation of the ESDIRK23 with adaptive step size (see section 12.8 for Matlab code) uses a while-loop to loop over the time interval, as for any other adaptive step size solver. We then use another while-loop to run each point in the interval until it has produced a following step with a satisfying low error.

The next value (x_{n+1}) is estimated using the scheme shown in eq. 53. As indicated in eq. 53 the inner stage 2 and 3 are implicit, since they depend on the function evaluated at the stage itself ($X_2 \propto f(X_2)$, $X_3 \propto f(X_3)$). To solve this implicit scheme we use a Newton solver to minimize the residual function (R(z)) of the stages seen in the following eq. 60.

$$\begin{aligned} R(X_2) &= X_2 - h\gamma f(T_1, X_1) - h\gamma f(T_2, X_2) - x_n \\ R(X_3) &= X_3 - h\frac{6\gamma - 1}{12\gamma} f(T_1, X_1) - h\frac{6\gamma - 1}{12\gamma} f(T_2, X_2) - h\gamma f(T_3, X_3) - x_n \end{aligned} \tag{54}$$

The following is the Matlab code for the inexact Newton solver:

```
function [x,Slow,divergent,k] = InexactNewtonsMethodODE(func, Jac, F, ii, tk,...
                                phi, dt, xinit, tol, minit, maxit, AT, varargin)
    % initialize
    k = 0;
    Slow = 0;
    divergent = 0;
    gamma = AT(2,2);
    x = xinit;
    % Compute the iteration matrix and LU factorize it
    J = feval(Jac,tk+dt,x,varargin{:}); % Evaluate the Jacobian
    M = eye(size(J))-dt*gamma.*J;
    [L,U] = lu(M,'vector');
    % Initial residual function
    F(:,ii) = feval(func,tk+dt,x,varargin{:});
    R = x - F(:,1:ii)*AT(1:ii,ii)*dt-phi;
    Convergent = 0;
    % Loop until convergent and below minimum and maximum number of
    % Newton iterations
    while (~Convergent ||  k <= minit) && k < maxit
        k=k+1; % Coun the number of Newton iterations
        ROld = R; % Save the previous residual function
        dx = U\(L\R); % Calculate the change to the residual
        x = x - dx; % Change the x value
        F(:,ii) = feval(func,tk+dt,x,varargin{:}); % Evaluate function at new x
        R = x - F(:,1:ii)*AT(1:ii,ii)*dt - phi; % Calculate the residual
        alpha = norm(R,'inf')/norm(ROld,'inf'); % Estimate convergence rate
        if norm(R,'inf') < tol*0.1 % Test for convergence
            Convergent = 1;
        elseif alpha > 0.1 % Test for slow convergence
            disp('Slow !!!!')
            Slow = 1;
            break
        end
    end
    if k >= maxit % Test for divergence
        disp('Divergent !!!!')
        divergent = 1;
    end
end
```

The initial guess for a solution to the roots given in eq. 60 is produced via a Backward Euler method. To save computational power an inexact Newton solver is employed. The M matrix is being approximated to

$$M = I - h\gamma J(t + h, x_{init}) \tag{55}$$

where $J(t + h, x_{init})$ is the Jacobian evaluated in the initial guess, and I is the identity matrix of the same size as the Jacobian.
This reduces number of Jacobian evaluations and is a trademark of the ESDIRK methods. This is also why they are constructed using the repeated factor γ in the diagonal. The inexact Newton solver will run until the residual function has been minimized to below its threshold. This threshold is set to $0.1tol$, where tol is the tolerance of the ESDIRK solver. The Inexact

Newton solver is stopped if either of three events occur: 1) It reaches its maximum number of iterations. 2) It is assumed to diverge or 3) if it converges too slow. This is measured by the "speed" of convergence $alpha > 0.1$, where:

$$\alpha = \frac{R_{n+1}}{R_n} \tag{56}$$

In either case of divergence or slow convergence the Newton solver is stopped and the step size is reduced by the minimum step size reduction given by facmin.

When the Newton solver has converged the value in the next step is set to $x_{n+1} = X_3$ (as discussed above), and the error is estimated by the Butcher d vector (given in eq: 52). The r-value is then estimated by:

$$r = \frac{|error|}{|x|reltol} \tag{57}$$

where reltol is the relative tolerance.
If $r \leq 1$ the step is accepted and the x-value saved, along with it other statistical information from the process. The time step is hereafter advanced by the step size $t = t + h$.

To control the step size the code uses a PI-controller, where if the step is step is accepted in changes as:

$$h_{n+1} = \left(\frac{h_n}{h_{n-1}} \frac{\epsilon}{r_n} \frac{r_{n-1}}{r_n} \right)^{1/(p+1)} \tag{58}$$

where p is the order of the method here $p = 2$.
if the step is rejected it changes as:

$$h_{n+1} = \left(\frac{\epsilon}{r_n} \right)^{1/(p+1)} \tag{59}$$

where change in step size for both cases is bounded by the minimum and maximum allowed rate of change (facmin, facmax).

10,3)

Testing our implementation of the ESDIRK23 on the Van der Pol problem result in the following figure 60 and 61 for the two different value of μ.

The number of step, shown in the legends on figure 60 and 61, clearly show how our implementation of the ESDIRK23 is not as effective for the Van der Pol problem as Matlabs ode15s solver. Our ESDIRK23 uses many more steps (close to 5 time as many) as the ode15s solver.

The ESDIRK23 uses many more steps that the explicit Runge-Kutta method constructed earlier. For this reason an explicit Runge-Kutta method is better than the ESDIRK23, since it do not require as much computational power. This is however only the case for "small" ($\mu < 9$) values of μ, since the explicit Runge-Kutta does not have the same stability properties, it is not A-stable nor L-stable!

10,4)

The stability region can (as done earlier) be plotted using the Butcher A matrix and b vector, by evaluating the function $R(z) = 1 + z\mathbf{b}' (\mathbf{I} - z\mathbf{A})^{-1} \mathbf{e}$ for z spanning over a region in the complex plane, as done and shown in figure 62.

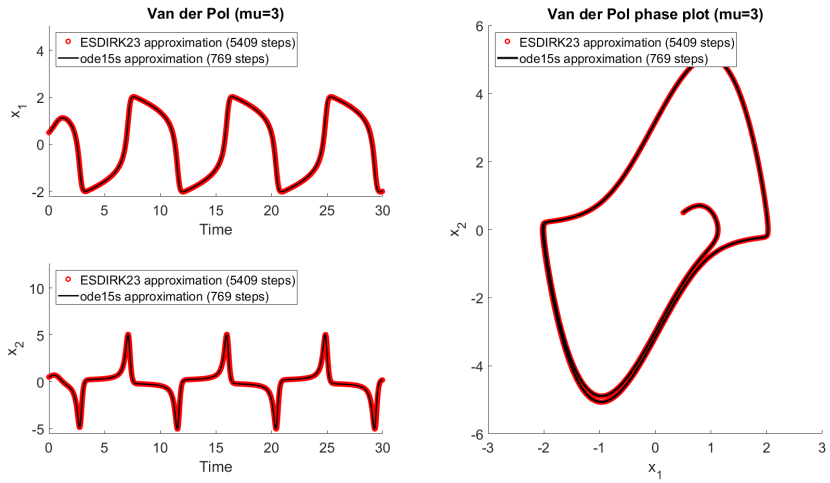


Figure 60: The solution from solving the Van der Pol problem with our implementation of the ESDIRK23 for $\mu = 3$. The initial value is $x = [0.5, 0.5]$, the initial step size is $h_0 = 10^{-3}$, the epsilon factor is $\epsilon = 0.8$ as suggested in the slides, and the absolute and relative tolerance is both 10^{-5}. We compare our solution to the solution produced by the stiff Matlab solver ode15s, which is run with the same absolute and relative tolerances.

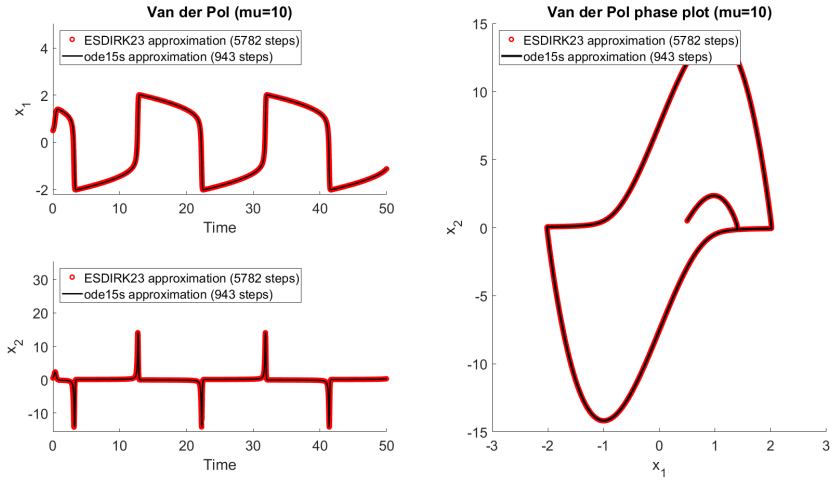


Figure 61: The solution from solving the Van der Pol problem with our implementation of the ESDIRK23 for $\mu = 10$. The initial value is $x = [0.5, 0.5]$, the initial step size is $h_0 = 10^{-3}$, the epsilon factor is $\epsilon = 0.8$ as suggested in the slides, and the absolute and relative tolerance is both 10^{-5}. We compare our solution to the solution produced by the stiff Matlab solver ode15s, which is run with the same absolute and relative tolerances.

The region of stability is the region where $|R(z)| < 1$, which in figure 62 is the region not colored yellow. Figure 62 shows that the instability region is a closed region in the complex plane.

The method is A-stable and L-stable by design (see eq. 49), and is illustrated in figure 62, where the entire left complex plane can be seen to be in the stability region, making it A-stable, and the values of $|R(z)|$ can be seen to go to zero as z goes to infinity, making it L-stability. This can be verified by the following to equations regarding the transfer function:

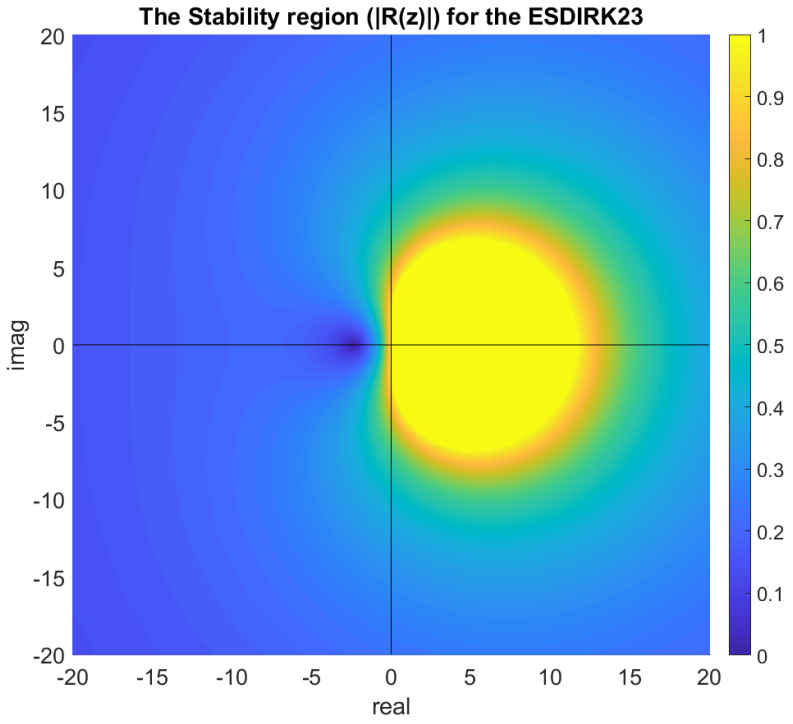


Figure 62: The stability region for the ESDIRK23, evaluated in the complex plane for values $z = [-20 - i20, 20 + i20]$. The region of stability is the region not colored yellow, where the value of $|R(z)| < 1$.

$$\begin{aligned} R(z) &= 1 + z \begin{bmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & \frac{2-\sqrt{2}}{2} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ -z\left(\frac{2-\sqrt{2}}{2}\right) & 1 - z\left(\frac{2-\sqrt{2}}{2}\right) & 0 \\ -z\left(\frac{\sqrt{2}}{2}\right) & -z\left(\frac{\sqrt{2}}{2}\right) & 1 - z\left(\frac{2-\sqrt{2}}{2}\right) \end{bmatrix}^{-1} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \\ &= 1 + \frac{4z\left(z\sqrt{2} - \frac{3}{2}z + 1\right)}{\left(z\sqrt{2} - 2z + 2\right)^2} \end{aligned} \tag{60}$$

$$\lim_{z \to \infty} |R(z)| = 0 \tag{61}$$

The fact that the ESDIRK23 is L-stable means that is can be applied to a large group of problems, where the method should return stable solutions. If the value of λ in the test equation is however positive, one need to make sure not to pick step sizes, which is so small that one enter the instability region! This is a problem for rapidly changing function with small values of λ.

11 Fully Implicit Runge-Kutta Method - Radau5

The Radau5 is a three stage method, which is fully implicit. A fully implicit method has a Butcher tableau with only non-zero diagonal elements. This makes it computational expensive to run such a scheme.

11,1)

The Butcher tableau for the three staged Radau5 is as follows:

$$\begin{array}{c|ccc} \frac{4-\sqrt{6}}{10} & \frac{88-7\sqrt{6}}{360} & \frac{296-169\sqrt{6}}{1800} & \frac{-2+3\sqrt{6}}{225} \\ \frac{4+\sqrt{6}}{10} & \frac{296+169\sqrt{6}}{1800} & \frac{88+7\sqrt{6}}{360} & \frac{-2-3\sqrt{6}}{225} \\ 1 & \frac{16-\sqrt{6}}{36} & \frac{16+\sqrt{6}}{36} & \frac{1}{9} \\ \hline x & \frac{16-\sqrt{6}}{36} & \frac{16+\sqrt{6}}{36} & \frac{1}{9} \end{array} \tag{62}$$

The equation for the scheme looks like:

$$\begin{aligned} T_1 &= t + h\left(\frac{4-\sqrt{6}}{10}\right) \\ \mathbf{X}_1 &= \mathbf{x} + \left(\frac{88-7\sqrt{6}}{360}\right) F\left(T_1, \mathbf{X}_1\right) + \left(\frac{296-169\sqrt{6}}{1800}\right) F\left(T_2, \mathbf{X}_2\right) + \left(\frac{-2+3\sqrt{6}}{225}\right) F\left(T_3, \mathbf{X}_3\right) \\ T_2 &= t + h\left(\frac{4+\sqrt{6}}{10}\right) \\ \mathbf{X}_2 &= \mathbf{x} + \left(\frac{296+169\sqrt{6}}{1800}\right) F\left(T_1, \mathbf{X}_1\right) + \left(\frac{88+7\sqrt{6}}{360}\right) F\left(T_2, \mathbf{X}_2\right) + \left(\frac{-2-3\sqrt{6}}{225}\right) F\left(T_3, \mathbf{X}_3\right) \\ T_3 &= t + h \\ \mathbf{X}_3 &= \mathbf{x} + \left(\frac{16-\sqrt{6}}{36}\right) F\left(T_1, \mathbf{X}_1\right) + \left(\frac{16+\sqrt{6}}{36}\right) F\left(T_2, \mathbf{X}_2\right) + \left(\frac{1}{9}\right) F\left(T_3, \mathbf{X}_3\right) \end{aligned} \tag{63}$$

It is clear from the scheme above that this is a implicit method.

11,2)

To verify that the Radau5 is of fifth order all the order equations has to match with the elements in the Butcher tableau in eq. 62. The equation up to fourth order is given in the slides, and written in vector and matrix form in the following:

$$\begin{aligned} 1 &= \mathbf{b}'\mathbf{e} && \text{(1st order)} \\ \tfrac{1}{2} &= \mathbf{b}'\mathbf{C}\mathbf{e} && \text{(2nd order)} \\ \tfrac{1}{3} &= \mathbf{b}'\mathbf{C}^2\mathbf{e} \\ \tfrac{1}{6} &= \mathbf{b}'\mathbf{A}\mathbf{C}\mathbf{e} && \text{(3rd order)} \\ \tfrac{1}{4} &= \mathbf{b}'\mathbf{C}^3\mathbf{e} \\ \tfrac{1}{8} &= \mathbf{b}'\mathbf{C}\mathbf{A}\mathbf{C}\mathbf{e} \\ \tfrac{1}{12} &= \mathbf{b}'\mathbf{A}\mathbf{C}^2\mathbf{e} \\ \tfrac{1}{24} &= \mathbf{b}'\mathbf{A}^2\mathbf{C}\mathbf{e} && \text{(4th order)} \end{aligned} \tag{64}$$

where $\mathbf{b}'$ is the Butcher b vector transposed, $\mathbf{A}$ is the Butcher a matrix, $\mathbf{e}$ is a vector of ones, and Butcher c vector written as a matrix ($\mathbf{C} = \mathbf{I}\mathbf{c}$).

Using Maple it was possible to verify that the list of equations 64 are true for the elements in the Radau5 Butcher tableau given in eq. 62 (Se Maple document attached together with the Matlab scripts). This mean it at least is a fourth order method.

To verify that it is a fifth order method the order five equations must be derived. To do so requires the order five rooted trees, which are shown in figure 63.

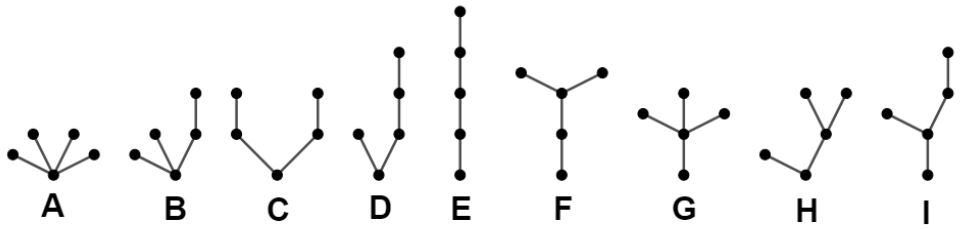


Figure 63: The nine fifth order rooted trees.

Using the rooted trees in figure 63 the necessary properties to consider when interested in constructing the order equations is given in the following tabular.

τ	A	B	C	D	E	F	G	H	I
$r(\tau)$	5	5	5	5	5	5	5	5	5
$\gamma(\tau)$	5	10	20	30	120	60	20	15	40
$\Phi(\tau)$	$\mathbf{b}'\mathbf{C}^4\mathbf{e}$	$\mathbf{b}'\mathbf{C}^2\mathbf{ACe}$	$\mathbf{b}'\mathbf{AC}^2\mathbf{Ae}$	$\mathbf{b}'\mathbf{CA}^2\mathbf{Ce}$	$\mathbf{b}'\mathbf{A}^3\mathbf{Ce}$	$\mathbf{b}'\mathbf{A}^2\mathbf{C}^2\mathbf{e}$	$\mathbf{b}'\mathbf{AC}^3\mathbf{e}$	$\mathbf{b}'\mathbf{CAC}^2\mathbf{e}$	$\mathbf{b}'\mathbf{ACACe}$

This turns into the following order equations for the fifth order rooted trees when using the expression $\Phi(\tau) = 1/\gamma(\tau)$:

$$\left.\begin{array}{r} \frac{1}{5} = \mathbf{b}'\mathbf{C}^4\mathbf{e} \\ \frac{1}{10} = \mathbf{b}'\mathbf{C}^2\mathbf{ACe} \\ \frac{1}{20} = \mathbf{b}'\mathbf{AC}^2\mathbf{Ae} \\ \frac{1}{30} = \mathbf{b}'\mathbf{CA}^2\mathbf{Ce} \\ \frac{1}{120} = \mathbf{b}'\mathbf{A}^3\mathbf{Ce} \\ \frac{1}{60} = \mathbf{b}'\mathbf{A}^2\mathbf{C}^2\mathbf{e} \\ \frac{1}{20} = \mathbf{b}'\mathbf{AC}^3\mathbf{e} \\ \frac{1}{15} = \mathbf{b}'\mathbf{CAC}^2\mathbf{e} \\ \frac{1}{40} = \mathbf{b}'\mathbf{ACACe} \end{array}\right\} \quad \text{(5th order)} \tag{65}$$

The fifth order equation has as well be verify using Maple (see same attached Maple document). The fact that the elements in the Butcher tableau in eq. 62 satisfies all the order equations in eq. 64 and 65 shows that the Radau5 is truly a fifth order method.

11,3)

As usual for a Runge-Kutta method can the transfer function $R(z)$ ($z = h\lambda$) be calculated from the equation:

$$R(z) = 1 + z\mathbf{b}'\left(\mathbf{I} - z\mathbf{A}\right)^{-1}\mathbf{e} \tag{66}$$

where $\mathbf{I}$ is the identity matrix, the $\mathbf{A}$ is the Butcher A matrix, $\mathbf{b}'$ is the transposed Butcher b vector, and $\mathbf{e}$ is a vectors of ones of the same dimensions as the Butcher b vector.

Writing out eq. 67 using the Butcher tableau for the Radau5 the equation becomes:

$$
\begin{aligned}
R(z) = 1 + z &\begin{bmatrix} \frac{16-\sqrt{6}}{36} & \frac{16+\sqrt{6}}{36} & \frac{1}{9} \end{bmatrix} \\
&\times \begin{bmatrix} 1 - z\left(\frac{88-7\sqrt{6}}{360}\right) & -z\left(\frac{296-169\sqrt{6}}{1800}\right) & -z\left(\frac{-2+3\sqrt{6}}{225}\right) \\ -z\left(\frac{296+169\sqrt{6}}{1800}\right) & 1 - z\left(\frac{88+7\sqrt{6}}{360}\right) & -z\left(\frac{-2-3\sqrt{6}}{225}\right) \\ -z\left(\frac{16-\sqrt{6}}{36}\right) & -z\left(\frac{16+\sqrt{6}}{36}\right) & 1 - z\left(\frac{1}{9}\right) \end{bmatrix}^{-1} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \\
&= 1 - \frac{z\left(z^2 - 6z + 60\right)}{z^3 - 9z^2 + 36z - 60}
\end{aligned}
\tag{67}
$$

where the $\times$ sign is to indicate that the line was broken at a multiplication sign.

11,4)

The stability region for the Radau5 method is plotted by evaluating eq. 67 in a region of the complex plane, see figure 64.

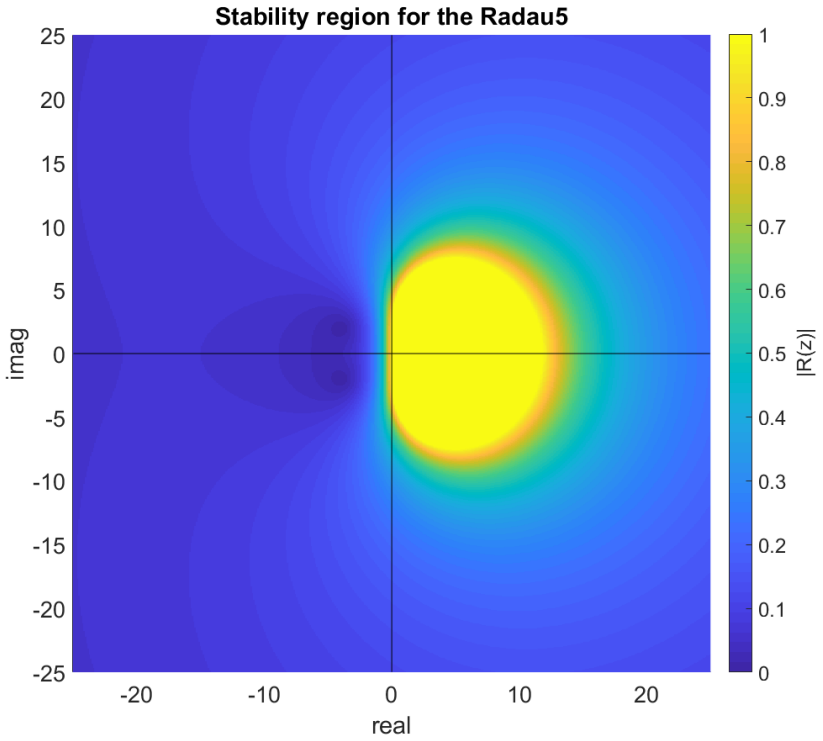


Figure 64: The stability region for the Radau5, evaluated in the complex plane for values $z = [-25-i25, 25+i25]$. The region of stability is the region not colored yellow, where the value of $|R(z)| \leq 1$.

The region of stability is the region where $|R(z)| < 1$, which in figure 64 is the region not colored yellow. Figure 64 shows that the instability region is a closed region in the complex plane similar to the ESDIRK23 method.

11,5)

Considering figure 64 for the stability region of the Radau5 method, it can be seen to be A-stable since the entire left complex plane is in the stability region, and it is most likely also L-stable since it qualitatively looks like $|R(z)|$ goes to zero as z goes to infinity.
To make sure it is L-stable we test the limits of the transfer function derived in eq. 67 as z goes to infinity:

$$\lim_{z\to\infty} |R(z)| = 1 - \frac{z^3}{z^3} = 0 \tag{68}$$

which shows that Radau5 truly is L-stable.

11,6)

The implementation of the Radau5 is very similar to that of the ESDIRK23 from the previous section, the matlab code can be seen in section 12.9.
To implement an adaptive step size a while-loop is used to loop though the time interval. To ensure the error is within the user defined tolerances another while-loop is used, to loop until the error is small enough. The error is being control by changing the step size.

The two major differences between the Radau5 and the ESDIRK23, is that Radau5 is fully implicit so rather than solving one stage at the time all three stages must be solved at once, and the other fact is that the Radua5 does not have a embedded error estimate!
To solve all three stages at once a new Newton solver must be developed. This new Newton solver must be able to the following residual function:

$$\begin{aligned} R(\mathbf{X}) = &\begin{bmatrix} \mathbf{X}_1 \\ \mathbf{X}_2 \\ \mathbf{X}_3 \end{bmatrix} \\ &- \begin{bmatrix} \left(\frac{88-7\sqrt{6}}{360}\right) F\left(T_1, \mathbf{X}_1\right) + \left(\frac{296-169\sqrt{6}}{1800}\right) F\left(T_2, \mathbf{X}_2\right) + \left(\frac{-2+3\sqrt{6}}{225}\right) F\left(T_3, \mathbf{X}_3\right) \\ \left(\frac{296+169\sqrt{6}}{1800}\right) F\left(T_1, \mathbf{X}_1\right) + \left(\frac{88+7\sqrt{6}}{360}\right) F\left(T_2, \mathbf{X}_2\right) + \left(\frac{-2-3\sqrt{6}}{225}\right) F\left(T_3, \mathbf{X}_3\right) \\ \left(\frac{16-\sqrt{6}}{36}\right) F\left(T_1, \mathbf{X}_1\right) + \left(\frac{16+\sqrt{6}}{36}\right) F\left(T_2, \mathbf{X}_2\right) + \left(\frac{1}{9}\right) F\left(T_3, \mathbf{X}_3\right) \end{bmatrix} - \mathbf{x}_n \end{aligned} \quad labeleq : 11 : 12 \tag{69}$$

where $\mathbf{X} = [\mathbf{X}_1, \mathbf{X}_2, \mathbf{X}_3]$ denotes the vector containing the x-value for all three stages, note $\mathbf{X}_i$ can them self be vectors, T_i is the three time steps, F denotes the function, and $\mathbf{x}_n$ is the previous x value (stating with the initial value).

To do this we decide to straighten out our vector, which for the Van der Pol problem would make the $\mathbf{X}$ vector look like $\mathbf{X}_s = [X_{1,1}, X_{1,2}, X_{2,1}, X_{2,2}, X_{3,1}, X_{3,2}]$ with dimensions 6×1. This means the matrices also has to be modified to fix this syntax. To keep the matrix vector product between the F-vector and the Butcher A matrix the same, the Butcher A matrix must have each of its elements multiplied by an identity matrix, changing it for the Van der Pol problem to:

$$A_{6\times 6} = \begin{bmatrix} a_{11} & 0 & a_{12} & 0 & a_{13} & 0 \\ 0 & a_{11} & 0 & a_{12} & 0 & a_{13} \\ a_{21} & 0 & a_{22} & 0 & a_{23} & 0 \\ 0 & a_{21} & 0 & a_{22} & 0 & a_{23} \\ a_{31} & 0 & a_{32} & 0 & a_{33} & 0 \\ 0 & a_{31} & 0 & a_{32} & 0 & a_{33} \end{bmatrix} \tag{70}$$

The matrix $\mathbf{M}$ used to estimate the change in the residual vector, must take the following form to match the straightened residual vector:

$$M_{6\times 6} = h_n \begin{bmatrix} 1-J_{11} & -J_{12} & 0 & 0 & 0 & 0 \\ -J_{13} & 1-J_{14} & 0 & 0 & 0 & 0 \\ 0 & 0 & 1-J_{21} & -J_{22} & 0 & 0 \\ 0 & 0 & -J_{23} & 1-J_{24} & 0 & 0 \\ 0 & 0 & 0 & 0 & 1-J_{31} & -J_{32} \\ 0 & 0 & 0 & 0 & -J_{33} & 1-J_{34} \end{bmatrix} \tag{71}$$

where the J's represents the jacobians, the first index specify at which stage it is evaluated, and the second index the element in the jacobian matrix (for the Van der Pol problem being 2×2 in each stage). h_n denotes the step size in the n'th iteration.

To reduces the computational power we use the approximation that $J_1 = J_2 = J_3$, which make it an inexact Newton solver.

Now having rewritten all the vectors and matrices, the Newton solver becomes straight forward. The following Matlab code shows how we in practise implemented it. Note how all the vectors and matrices is being constructed to fit the structure descried above.

The following is the matlab implementation of the inexact Newton solver for a implicit system.

```
function [x,Slow,divergent,k] = ImplicitInexactNewtonsMethodODE(func, Jac,...
                        T, dt, xinit, tol, minit, maxit, s, A, cc, varargin)
% initialize
    k = 0;
    Slow = 0;
    divergent = 0;
    % Shape x and c vectors
    x = []; c= [];
    for ii = 1:s
        x = [x;xinit];
        c = [c;cc(ii);cc(ii)];
    end
    phi = x;
    % Compute the iteration matrix and LU factorize it
    J = feval(Jac,T(s),x,varargin{:});
    M = zeros(size(J)*s);
    AI = zeros(size(J)*s);
    % Shape the M and A matrix
    for ii = 1:s
        M(2*ii-1:2*ii,2*ii-1:2*ii) = eye(size(J))-dt.*J;
        for jj = 1:s
            AI(2*ii-1:2*ii,2*jj-1:2*jj) = A(ii,jj).*eye(size(J));
        end
    end
    AIT = transpose(AI);
    % LU factorize
    [L,U,pp] = lu(M,'vector');
    for ii = 1:s % shape F vector
        F(1,2*ii-1:2*ii) = feval(func,T(ii),xinit,varargin{:});
    end
    x = x + F*c*dt; % Backward Eulor for first guess
    for ii = 1:s % shape F vector
        F(1,2*ii-1:2*ii) = feval(func,T(ii),x(2*ii-1:2*ii),varargin{:});
    end
    % Initialize the residual function
    R = x - transpose(F*AIT).*dt-phi;
```

```
    Convergent = 0;
    % Loop until converged or/and minit < k < maxit
    while (~Convergent ||  k <= minit) && k < maxit
        k=k+1;
        ROld = R; % Save preveous residual
        dx = U\(L\R); % Estimate change in x
        x = x - dx; % Change x
        for ii = 1:s % shape F vector
            F(1,2*ii-1:2*ii) = feval(func,T(ii),x(2*ii-1:2*ii),varargin{:});
        end
        R = x - transpose(F*AIT).*dt - phi; % update residual
        alpha = norm(R,'inf')/norm(ROld,'inf'); % Measure rate of convergence
        if norm(R,'inf') < tol*0.1 % Test id converged
            Convergent = 1;
        elseif alpha > 0.1 % Test if slow convergence
            Slow = 1;
            break
        end
    end
    if k >= maxit % Test if diverged
        disp('Divergent !!!!!')
        divergent = 1;
    end
end
```

If the rate of convergence in the Newton solver becomes to slow, measured as in eq. 56, the solver will stop and the step size will decrease before trying again. The same applies if the solver loops for more than the allowed number of iteration, where the code is assumed to be divergent.

To estimate the error related to taking a step with a given step size, we use step-doubling since Radau5 does not have an embedded error estimator. It is assumed that if the Newton solver converged for the full step, then it will also converge for the half step.

Only if the Newton solver converged with a satisfying rate, will the step be considered as a valid step, and the error will be estimated by:

$$error = \hat{x} - x \tag{72}$$

$$r = \frac{|error|}{|x|reltol} \tag{73}$$

$$\tag{74}$$

where x is the next solution value, $\hat{x}$ is the solution from the step doubling, and $reltol$ is the relative tolerance.
The step is accepted if $r \leq 1$, as for any other method with adaptive step size.

The step size is being controlled be a PI-controller, as the ESDIRK23. If the step is accepted the next step is estimated as in eq. 58, if rejected as in eq. 59. Both these changes to the step size in bounded by the minimum and maximum allowed change in step size facmin and facmax, respectively.

11,7)

Testing our implementation of the Radau5 on the Van der Pol with $\mu = 10$, result in the solution shown in figure 65 and statistics about the solution can be seen in figure 66.

The solution for the Radau5 shown in figure 65 can be seen to produces a solution similar to that of Matlabs ode15s solver, however with just over 400 times the number of points! The behavior of the step size in figure 66 is as expected, however the step size never go above $5 \cdot 10^{-3}$. The

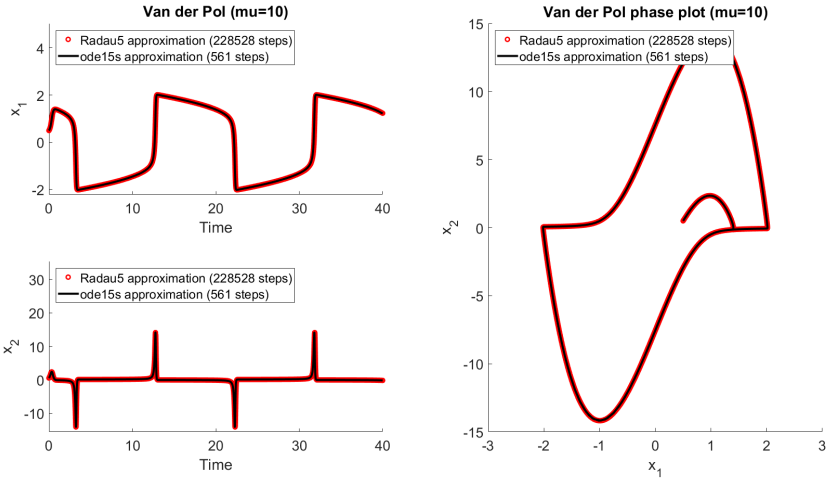


Figure 65: The solution from solving the Van der Pol problem with our implementation of the Radau5 for $\mu = 10$. The initial value is $x = [0.5, 0.5]$, the initial step size is $h_0 = 10^{-3}$, the epsilon factor is $\epsilon = 0.8$, and the absolute and relative tolerance is both 10^{-4}. We compare our solution to the solution produced by the stiff Matlab solver ode15s, which is run with the same absolute and relative tolerances.

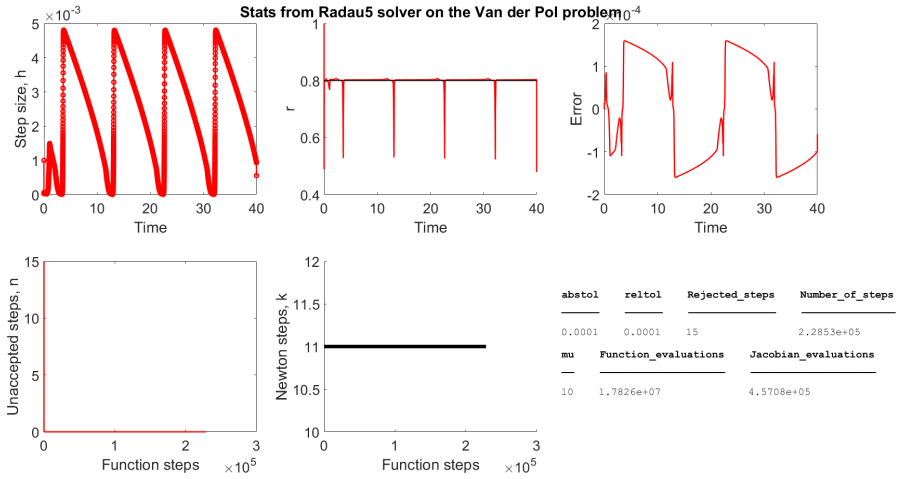


Figure 66: Plot of the statistics for solution to the Van der Pol problem in figure 65. The initial step size was $h_0 = 10^{-3}$, the epsilon factor is $\epsilon = 0.8$, the error is estimated by the embedded error estimator. Unaccepted steps only happen in the first time step, which can be explained in the graph for the step size where the initial step size can be seen to be to large. The minimum allowed number of Newton iterations is 11, and at no time step was it necessary to do more. It can be seen that the function was evaluated any more time than the jacobian.

number of Newton iterations is 11, which is the minimum number allowed.

It turns out that if the absolute and relative tolerance is lowered to below 10^{-4} (which was the tolerances used in figure 65 and 66), the solution no longer matches the solution produced by the ode15s. For too large tolerances the Van der Pol curve begins to oscillate to quickly, this can be seen in figure 67 with its solution statistics shown in figure 68. At lower tolerances the number of time steps is much smaller (as expected), but the method finds another solution curve the that the one found by ode15s. This trend continues as the tolerances is lowered further.

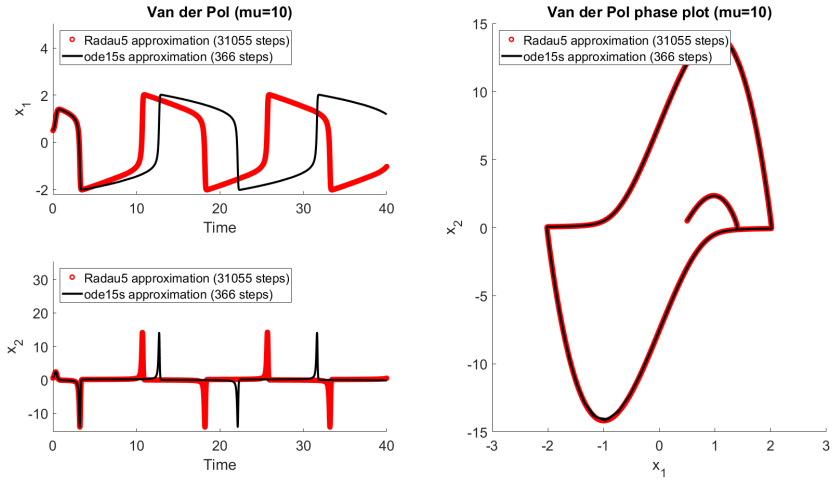


Figure 67: The solution from solving the Van der Pol problem with our implementation of the Radau5 for $\mu = 10$. The initial value is $x = [0.5, 0.5]$, the initial step size is $h_0 = 10^{-3}$, the epsilon factor is $\epsilon = 0.8$, and the absolute and relative tolerance is both 10^{-3}. We compare our solution to the solution produced by the stiff Matlab solver ode15s, which is run with the same absolute and relative tolerances.

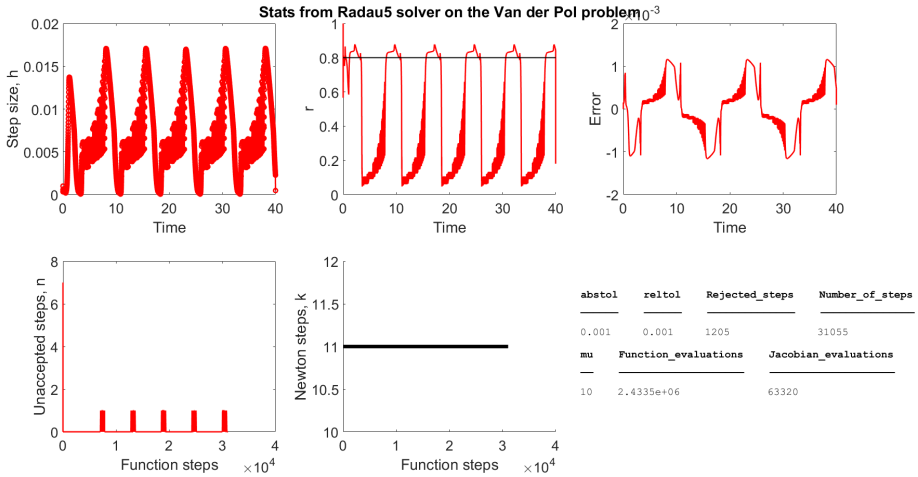


Figure 68: Plot of the statistics for solution to the Van der Pol problem in figure 67. The initial step size was $h_0 = 10^{-3}$, the epsilon factor is $\epsilon = 0.8$, the error is estimated by the embedded error estimator. The minimum allowed number of Newton iterations is 11, and at no time step was it necessary to do more. It can be seen that the function was evaluated any more time than the jacobian.

11,8)

a)

The Radau5 requires more than 400 times the number of steps to solve the Van der Pol (tested with $mu = 5$), however it do posses much better stability properties by being both A-stable and L-stable.
To insure that our explicit Runge-Kutta (RK32) produces useful result for $\mu = 10$, we need somewhat loss to 1000 time steps, and in that case it would be superior to the Radau5, simply because it requires less computational power.

Note however that for larger μ Radau5 may be a better choice, since it handles stiff prob-

lems better than our explicit Runge-Kutta. The Radau5 major defeat is the fact that it does not run well for large tolerances.

b)

Both the ESDIRK23 and the Radau5 requires a large number of time steps to be able to solve the Van der Pol problem well. ESDIRK23 works however best at low tolerances and is still faster to run than the Radau5, which is mainly because it isn't fully implicit (does not have coupled inner stages) and it having an embedded error estimator. Do to its semi implicit scheme does it also have the nice stability properties of being both A-stable and L-stable. This means it also works well for some stiff problems, and for the Van der Pol with $\mu = 10$ it is far superior to the Radau5.

Note again that as the problem increases in stiffness, by increases μ, the Radau5 becomes more and more favorable to the ESDIRK23, since a fully implicit method is more stable against stiffness.

12 Matlab code

Mini Project 1 Code

12.1 NewtonsMethodODE.m

```
function x = NewtonsMethodODE(tk, xk, dt, xinit, tol, maxit, varargin)
    k = 0;
    t = tk + dt;
    x = xinit;
    %[f,J] = feval(FunJac,t,x,varargin{:})
    [f,J] = VanderPolfunjac(t,x,varargin{:});
    R = x - f*dt - xk;
    I = eye(size(xk));
    while( (k < maxit) & (norm(R,'inf') > tol) )
        k = k+1;
        dRdx = I - J*dt;
        dx = dRdx\R;
        x = x - dx;
        %[f,J] = feval(FunJac,t,x,varargin{:});
        [f,J] = VanderPolfunjac(t,x,varargin{:});
        R = x - f*dt - xk;
    end
end
```

12.2 ScalarSampleMeanStdVar.m

```
function [xmean,s,xmeanp2s,xmeanm2s]=ScalarSampleMeanStdVar(x)

% Calculation of mean, std and pm 2*std at each instance, hence dimension 3
xmean = mean(x,3);
s = std(x,0,3);
xmeanp2s = xmean + 2*s;
xmeanm2s = xmean - 2*s;
```

Mini Project 2 Code

12.3 ClassicalRungeKuttaSolver.m

```
function [Tout,Xout] = ClassicalRungeKuttaSolver(fun,tspan,x0,h,varargin)

%Butcher Tableau Information for ERK4C
s  = 4;
AT = [0 1/2 0 0 ; 0 0 1/2 0 ; 0 0 0 1 ; 0 0 0 0];
b  = [1/6 ; 1/3 ; 1/3 ; 1/6];
c  = [0; 1/2 ; 1/2 ; 1];

%Step-size Matrix-Vector Products
hAT = h*AT;
hb  = h*b;
hc  = h*c;

% Initial Parameters, Step Size, etc.
x  = x0;
t  = tspan(1);
tf = tspan(end);
N = (tf-t)/h;
```

```
nx = length(x0);

%Allocate Memory for Variables
T  = zeros(1,s);
X  = zeros(nx,s);
F  = zeros(nx,s);
Tout = zeros(N+1,1);
Xout = zeros(N+1,nx);

%Initial Values to Output
Tout(1) = t;
Xout(1,:) = x';

%For-Loop for the Stages of each Iteration
for n=1:N

    % Stage 1
    T(1) = t;
    X(:,1) = x;
    F(:,1) = feval(fun,T(1),X(:,1),varargin{:});

    % Stage 2,3,4
    T(2:s) = t + hc(2:s);
    for i=2:s
        X(:,i) = x + F(:,1:i-1)*hAT(1:i-1,i);
        F(:,i) = feval(fun,T(i),X(:,i),varargin{:});
    end

    % Next Iterate
    t = t + h;
    x = x + F*hb;

    % Save Iterate to Output
    Tout(n+1) = t;
    Xout(n+1,:) = x';

end
end
```

12.4 AdaptiveRungeKuttaSolver.m

```
function [T,X,H,tau,R] = AdaptiveRungeKuttaSolver(fun, tspan, x0, h0, abstol, reltol, varargin)
%Change x0 to avoid errors

nx = length(x0);
x0 = reshape(x0,nx,1);

facmin = 0.1;
facmax = 5;
eps = 0.8;

%Butcher Tableau Information for ERK4C
s  = 4;
A = [0 0 0 0 ; 1/2 0 0 0 ; 0 1/2 0 0 ; 0 0 1 0];
b  = [1/6 ; 1/3 ; 1/3 ; 1/6];
c  = [0; 1/2 ; 1/2 ; 1];

% Initial Parameters and First Output
x  = x0;
h = h0;
t  = tspan(1);
tf = tspan(end);

X = x;
T = t;
```

```
H = h0;
tau = 0;
R = 1;

%Allocate Memory for Stage variables
Xs  = zeros(nx,s);
Fs  = zeros(nx,s);

Xr  = zeros(nx,s);
Fr  = zeros(nx,s);

%While-Loop for the Stages of each Iteration
while t < tf
    %Final Stage
    if tf < t + h
        h = tf - t;
    end

    Accept = false;
    while ~Accept

        %%%%% 1. Guess Next Iterate %%%%%

        %Step-size Matrix-Vector Products
        hAs  = h*A;
        hbs  = h*b;
        hcs  = h*c;

        Ts = t + hcs; %Time Vector
        % Stage 1
        Xs(:,1) = x;
        Fs(:,1) = feval(fun,Ts(1),Xs(:,1),varargin{:});
        % Stage 2,3,4
        for i=2:s
            Xs(:,i) = x + Fs(:,1:i-1)*hAs(i,1:i-1)';
            Fs(:,i) = feval(fun,Ts(i),Xs(:,i),varargin{:});
        end
        xs = x + Fs*hbs;

        %%%%%% Refined Guess Next Iterate %%%%%
        hr = h/2;
        %Refined Step-size Matrix-Vector Products
        hAr = hr*A;
        hbr  = hr*b;
        hcr  = hr*c;

        Tr = t + hcr; %Refined Time Vector
        % Stage 1
        Xr(:,1) = x;
        Fr(:,1) = feval(fun,Tr(1),Xr(:,1),varargin{:});
        % Stage 2,3,4
        for i=2:s
            Xr(:,i) = x + Fr(:,1:i-1)*hAr(i,1:i-1)';
            Fr(:,i) = feval(fun,Tr(i),Xr(:,i),varargin{:});
        end
        xr = x + Fr*hbr;

        %%% Error Computation %%%
        error = xr - xs;
        r = max(abs(error)./max(abstol,abs(xr).*reltol));

        Accept = (r <= 1);
        if Accept
                t = t+h;
                x = xs;

                R = [R,r];
```

```
                tau = [tau,error(end)];
                H = [H,h];
                T = [T,t];
                X = [X,x];
        end
        %Asympototic Step Size Controller with Limited Change
        h = max( [facmin , min( [facmax , (eps/r)^(1/5) ] ) ] ) * h;

    end
end
end
```

12.5 DormandPrinceSolver.m

```
function [Tout,Xout,H,tau,R] = DormandPrinceSolver(fun, tspan, x0, h0, abstol, reltol, varargin)
%Change x0 to avoid errors

nx = length(x0);
x0 = reshape(x0,nx,1);

facmin = 0.1;
facmax = 5;
eps = 0.8;

%Butcher Tableau Information for Dormand-Prince5(4)
s  = 7;
A = [[0,0,0,0,0,0,0];...
[1/5,0,0,0,0,0,0];...
[3/40,9/40,0,0,0,0,0];...
[44/45,-56/15,32/9,0,0,0,0];...
[19372/6561,-25360/2187,64448/6561,-212/729,0,0,0];...
[9017/3168,-355/33,46732/5247,49/176,-5103/18656,0,0];...
[35/384,0,500/1113,125/192,-2187/6784,11/84,0]];
b = [35/385;0;500/1113;125/192;-2187/6784;11/84;0];
bhat = [5179/57600;0;7571/16695;393/640;-92097/339200;187/2100;1/40];
c = [0;1/5;3/10;4/5;8/9;1;1];
d = b-bhat;

% Initial Parameters and First Output
x  = x0;
h = h0;
t  = tspan(1);
tf = tspan(end);

Xout = x;
Tout = t;
H = h0;
tau = 0;
R = 1;

%Allocate Memory for Stage variables
X  = zeros(nx,s);
F  = zeros(nx,s);

%While-Loop for the Stages of each Iteration
while t < tf
    %Final Stage
    if tf < t + h
        h = tf - t;
    end

    Accept = false;
    while ~Accept

        %%%%% 1. Guess Next Iterate %%%%%
```

```
        %Step-size Matrix-Vector Products
        hA  = h*A;
        hb  = h*b;
        hc  = h*c;
        hd  = h*d;
        T = t + hc; %Time Vector
        % Stage 1
        X(:,1) = x;
        F(:,1) = feval(fun,T(1),X(:,1),varargin{:});
        % Stage 2,3,4
        for i=2:s
            X(:,i) = x + F(:,1:i-1)*hA(i,1:i-1)';
            F(:,i) = feval(fun,T(i),X(:,i),varargin{:});
        end
        xs = x + F*hb;

        %%% Error Computation %%%
        error = F*hd;
        r = max(abs(error)./max(abstol,abs(x).*reltol));

        Accept = (r <= 1);
        if Accept
                t = t+h;
                x = xs;

                R = [R,r];
                tau = [tau,error(end)];
                H = [H,h];
                Tout = [Tout,t];
                Xout = [Xout,x];
        end
        %Asympototic Step Size Controller with Limited Change
        h = max( [facmin , min( [facmax , (eps/r)^(1/6) ] ) ] ) * h;

    end
end
end
```

Mini Project 3 Code

12.6 RungeKutta32Solver.m

```
function [Tout,Xout,E] = RungeKutta32Solver(fun,tspan,x0,h,varargin)
%Change x0 to avoid errors

x0 = reshape(x0,length(x0),1);

%Butcher Tableau Information for ERK4C
s = 3;
A = [0 0 0 ; 1/2 0 0 ; -1 2 0];
b = [1/6 ; 2/3 ; 1/6];
bhat = [1/4 ; 1/2 ; 1/4];
c = [ 0 ; 1/2 ; 1];
d = b-bhat;

%Step-size Matrix-Vector Products
hA = h*A;
hb  = h*b;
hc  = h*c;
hd = h*d;

% Initial Parameters, Step Size, etc.
```

```
x  = x0;
t  = tspan(1);
tf = tspan(end);
N = round((tf-t)/h);
nx = length(x0);

%Allocate Memory for Variables
T  = zeros(1,s);
X  = zeros(nx,s);
F  = zeros(nx,s);
Tout = zeros(N+1,1);
Xout = zeros(N+1,nx);
E = zeros(N,nx);

%Initial Values to Output
Tout(1) = t;
Xout(1,:) = x';
E = 0;

%For-Loop for the Stages of each Iteration
for n=1:N
    T = t + hc; %Time Vector

    % Stage 1
    X(:,1) = x;
    F(:,1) = feval(fun,T(1),X(:,1),varargin{:});

    % Stage 2,3,4
    for i=2:s
        X(:,i) = x + F(:,1:i-1)*hA(i,1:i-1)';
        F(:,i) = feval(fun,T(i),X(:,i),varargin{:});
    end

    % Next Iterate
    t = t + h;
    x = x + F*hb;
    e = F*hd;

    % Save Iterate to Output
    Tout(n+1) = t;
    Xout(n+1,:) = x';
    E(n+1) = e;

end
end
```

12.7 RungeKutta32AdaptiveSolver.m

```
function [Tout,Xout,H,tau,R] = RungeKutta32AdaptiveSolver(fun, tspan, x0, h0, abstol, reltol, varargin)
%Change x0 to avoid errors

nx = length(x0);
x0 = reshape(x0,nx,1);

facmin = 0.1;
facmax = 5;
eps = 0.8;

%Butcher Tableau Information for Dormand-Prince5(4)
s = 3;
A = [0 0 0 ; 1/2 0 0 ; -1 2 0];
b = [1/6 ; 2/3 ; 1/6];
bhat = [1/4 ; 1/2 ; 1/4];
c = [ 0 ; 1/2 ; 1];
d = b-bhat;
```

```
% Initial Parameters and First Output
x  = x0;
h = h0;
t  = tspan(1);
tf = tspan(end);

Xout = x;
Tout = t;
H = h0;
tau = 0;
R = 1;

%Allocate Memory for Stage variables
X  = zeros(nx,s);
F  = zeros(nx,s);

%While-Loop for the Stages of each Iteration
while t < tf
    %Final Stage
    if tf < t + h
        h = tf - t;
    end

    Accept = false;
    while ~Accept

        %%%%% 1. Guess Next Iterate %%%%%

        %Step-size Matrix-Vector Products
        hA  = h*A;
        hb  = h*b;
        hc  = h*c;
        hd  = h*d;
        T = t + hc; %Time Vector
        % Stage 1
        X(:,1) = x;
        F(:,1) = feval(fun,T(1),X(:,1),varargin{:});
        % Stage 2,3,4
        for i=2:s
            X(:,i) = x + F(:,1:i-1)*hA(i,1:i-1)';
            F(:,i) = feval(fun,T(i),X(:,i),varargin{:});
        end
        xs = x + F*hb;

        %%% Error Computation %%%
        error = F*hd;
        r = max(abs(error)./max(abstol,abs(x).*reltol));

        Accept = (r <= 1);
        if Accept
                t = t+h;
                x = xs;

                R = [R,r];
                tau = [tau,error(end)];
                H = [H,h];
                Tout = [Tout,t];
                Xout = [Xout,x];
        end
        %Asympototic Step Size Controller with Limited Change
        h = max( [facmin , min( [facmax , (eps/r)^(1/6) ] ) ] ) * h;

    end
end
end
```

12.8 ESDIRK23solver.m

```
function [tout,yout,stats] = ESDIRK23solver(func,Jac,ta,tb,h0,x0,...
                                    abstol,reltol,eps,facmin,facmax,varargin)
    % ESDIRK23 solver with adaptive step size
    s = 3; p = 2;
    % Define the Butcher tableau
    gamma = (2-sqrt(2))/2;
    A = [[0,0,0];...
         [gamma,gamma,0];...
         [sqrt(2)/4,sqrt(2)/4,gamma]];
    AT = transpose(A);
    b = [sqrt(2)/4;sqrt(2)/4;gamma];
    bhat = [(4-sqrt(2))/12;(4+3*sqrt(2))/12;(2-sqrt(2))/6];
    c = [0;2*gamma;1];
    d = b-bhat;
    % Initial values
    t = ta;
    h = h0;
    x = x0;
    % Initial value stats output
    stats.n = [];
    stats.k = [];
    stats.R = 1;
    stats.tau = x'.*0;
    stats.hx = h0;
    stats.fEval = [];
    stats.JEval = [];
    % To print progress
    tstep = (tb-ta)*0.1; tn = 1;
    % First value is set to initial value
    yout = x';
    tout = t;
    % use the ESDIRK23 method
    m = 0;
    while t < tb
        m=m+1; % Count number of steps
        stats.fEval(m,1) = 0;
        stats.JEval(m,1) = 0;
        if t+h>tb
            h = tb-t;
        end
        n = 0;
        Accept = false;
        while ~Accept % Run until error is below tolerance
            n = n+1; % Count number of rejected steps (-1)
            Slow = 0; divergent = 0;
            % Stage 1
            T(1) = t;
            X(:,1) = x;
            F(:,1) = feval(func,T(1),X(:,1),varargin{:});
            % Stage 2 and 3
            T(2:s) = t + h*c(2:s);
            for ii = 2:s
                F(:,ii) = F(:,ii-1);
                xinit = X(:,ii-1) + F(:,1:ii)*AT(1:ii,ii).*h; % Initial guess
                % Call inexact Newton solver
                [X(:,ii),Slow,divergent,k] = InexactNewtonsMethodODE(func, Jac, F,...
                    ii , t, x, h*c(ii), xinit, abstol, 5, 1000, AT, varargin{:});
                F(:,ii) = feval(func,T(ii),X(:,ii),varargin{:}); % Evaluated the function
                stats.k(m,ii-1) = k; % Save the number of Newton interations
                if Slow || divergent % Check for divergence or slow convergence
                    break
                end
            end
            stats.fEval(m,1) = stats.fEval(m,1) + 1 + 2*(1+1+k);
```

```
            stats.JEval(m,1) = stats.JEval(m,1) + 2;
            if ~Slow && ~divergent % If not slow or divergent
                x = X(:,3); % save the next value for the solution
                error = F*d*h; % estimate the error
                r = max(abs(error)./max(abstol,abs(x).*reltol)); % Calculate the r value
                Accept = (r <= 1); % Test we r<=1, within tolerance
            end
            if Accept % If within tolerance
                % To print progress
                if t >= tstep
                    if tn == 1
                        fprintf(strcat('\n',num2str(tn)))
                    elseif tn == 9
                        fprintf(strcat(num2str(tn),'\n'))
                    else
                        fprintf(num2str(tn))
                    end
                    tn = tn+1;
                    tstep = tstep + (tb-ta)*0.1;
                end
                % Progress to the next time step
                t = t+h;
                % Svae stat values
                stats.n(end+1,1) = n;
                stats.R = [stats.R;r];
                stats.tau = [stats.tau;error'];
                stats.hx = [stats.hx;h];
                % Save to the output
                tout = [tout;t];
                yout = [yout;x'];
                % Chanve the step size
                if n == 1
                    h = max([facmin,min([facmax,(eps/r).^(1/(p+1))])])*h;
                else
                    h = max([facmin,min([facmax,...
                        (h/stats.hx(end-1)).*(eps*stats.R(end-1)/r^2).^(1/(p+1))])])*h;
                end
            elseif Slow || divergent
                % Decrease step size of slow congerce or divergence in
                % Newton solver
                h = h*facmin;
            else
                % lower step size if error is to large
                h = max([facmin,min([facmax,(eps/r).^(1/(p+1))])])*h;
            end
        end
    end
end
```

12.9 Radau5solver.m

```
function [tout,yout,stats] = Radau5solver(func,Jac,ta,tb,h0,x0,...
                                abstol,reltol,eps,facmin,facmax,varargin)
    % Radau5 solver with adaptive step size
    s = 3; p = 5; % Number of stages and order of method
    A = [[(88-7*sqrt(6))/360,(296-169*sqrt(6))/1800,(-2+3*sqrt(6))/225];...
        [(296+169*sqrt(6))/1800,(88+7*sqrt(6))/360,(-2-3*sqrt(6))/225];...
        [(16-sqrt(6))/36,(16+sqrt(6))/36,1/9]]; % Butcher Tableau A matrix
    c = [(4-sqrt(6))/10;(4+sqrt(6))/10;1]; % Butcher Tableau c matrix
    % Initial values
    t = ta;
    h = h0;
    x = x0;
    % Initial value stats output
    stats.n = [];
```

```
stats.k = [];
stats.R = 1;
stats.tau = x'.*0;
stats.hx = h0;
stats.fEval = [];
stats.JEval = [];
% To print progress
tstep = (tb-ta)*0.1; tn = 1;
% First value is set to initial value
yout = x';
tout = t;
m=0;
while t < tb
    m=m+1; % Count number of steps
    stats.fEval(m,1) = 0;
    stats.JEval(m,1) = 0;
    if t+h>tb
        h = tb-t;
    end
    n = 0;
    Accept = false;
    while ~Accept
        n = n+1; % Number of rejected time steps
        % inner time steps
        T(1:s) = t + h*c(1:s);
        % Call the implicit inexact Newton solver
        [X,Slow,divergent,k] = ImplicitInexactNewtonsMethodODE(func,...
            Jac, T, h, x, min(abstol, reltol), 10, 1000, s, A, c, varargin{:});
        X = reshape(X,length(x),s); % Reshape output
        stats.k(m,1) = k; % Save number of Newton iterations
        x = X(:,3); % Set next solution value
        stats.fEval(m,1) = stats.fEval(m,1) + 6 + 3*k;
        stats.JEval(m,1) = stats.JEval(m,1) + 1;
        % If Newton method converged with a acceptable rate
        if (~divergent && ~Slow)
            % Use step doubling
            hm = 0.5*h; % Half step size
            Tm = t + hm.*c; % New inner time steps
            % Call the implicit inexact Newton solver
            [Xhat,~,~,kk] = ImplicitInexactNewtonsMethodODE(func,...
                Jac, Tm, hm, x, min(abstol, reltol), 10, 1000, s, A, c, varargin{:});
            stats.fEval(m,1) = stats.fEval(m,1) + 6 + 3*kk;
            stats.JEval(m,1) = stats.JEval(m,1) + 1;
            Xhat = reshape(Xhat,length(x),s);
            xhat = Xhat(:,3); % Save solution from step doubling
            error = xhat - x; % Estimate error
            % Test if within toleraces
            r = max(abs(error)./max(abstol,abs(x).*reltol));
            Accept = (r <= 1);
        end
        if Accept % If accepted step
            % Print progress
            if t >= tstep
                if tn == 1
                    fprintf(strcat('\n',num2str(tn)))
                elseif tn == 9
                    fprintf(strcat(num2str(tn),'\n'))
                else
                    fprintf(num2str(tn))
                end
                tn = tn+1;
                tstep = tstep + (tb-ta)*0.1;
            end
            t = t+h; % Set next time step
            % Save stats
            stats.n(end+1,1) = n;
            stats.R = [stats.R;r];
```

```
                stats.tau = [stats.tau;error'];
                stats.hx = [stats.hx;h];
                tout = [tout;t];
                yout = [yout;x'];
                % PI control change in step size
                if n == 1
                    h = max([facmin,min([facmax,(eps/r).^(1/(p+1))])])*h;
                else
                    h = max([facmin,min([facmax,...
                        (h/stats.hx(end-1)).*(eps*stats.R(end-1)/r^2).^(1/(p+1))])])*h;
                end
            elseif divergent || Slow % If diverged or slow convergence rate
                h = h*facmin; % Reduce step size
            else % If step did not get accepted
                h = max([facmin,min([facmax,(eps/r).^(1/(p+1))])])*h;
            end
        end
    end
end
```

References

[MATLAB, 2018] MATLAB (2018). Matlab ode45. "https://se.mathworks.com/help/matlab/ref/ode45.html".