Implementation of an Explicit Runge-Kutta Solver with Adaptive Time-Stepping Based on Error Control

Adrian Zafari

Bachelor's thesis
2023:K31

Lund University

Faculty of Science
Centre for Mathematical Sciences
Numerical Analysis

Centrum Scientiarum Mathematicarum

Abstract

This work begins by motivating why one would consider using error-based control for a numerical solver. An introduction to some of the preliminary theory for studying numerical methods is given, after which, the relevant theory pertaining to the Runge-Kutta methods and error-based controllers used in this work is covered as well. Furthermore, it is motivated why the solver is implemented the way it is in Python. The effectiveness of the solver is then tested across several different problems including the linear test equation, the double pendulum, a nonlinear ODE problem, and a stiff problem. The results are compared with a benchmark solver from SciPy and are found to be comparable with the solver that is implemented, suffering from relatively small errors. However, it is noted that ill-suited controller parameters caused disturbances in the results in some cases. In the case of the nonlinear ODE problem and stiff test problem, a further comparison is made with a similar implementation in Julia that uses the same error-based controller which presented noteworthy, if inconclusive, findings. A discussion of the considerations made and difficulties that were encountered with the solver in the Python and Julia implementations studied is presented as well.

Popular Abstract

The study of differential equations has been a topic of great interest to many mathematicians over the centuries, having brought with it great developments in engineering and the sciences. With the advent of the 20th century and the gradual automation of the laborious task of computation, mankind has strived to develop ever more efficient and effective means to ensure progress. Numerical methods for solving differential equations as a result saw a great rise in prominence during this time given that with the aid of computers, we could begin to automate the process of solving different equations, allowing us to even contend with problems that could not be solved analytically. In order to best be able to harness the capabilities of computers, particularly due to hardware limitations, innovations needed to be made.

One such innovation is the humble controller, an algorithm which harnesses the simple principle of wanting to reach and maintain a selected standard for performance at all possible times in a dynamical system. These controllers may be used in situations where automation is needed such as ensuring that the cabin temperature of an aeroplane is kept within a certain threshold to ensure the wellbeing of the passengers and crew in spite of changing climate conditions. Mathematicians in the early 20th century saw apt grounds for the implementation of controllers in order to automate the normally sensitive time-stepping process, which is responsible for the generation of the time grid on which a given differential equation is evaluated. Should this time grid not be suitable for the problem, one could simply get completely wrong solutions. Thus instead of using a fixed time step and wasting precious computational power and time spent waiting, one may use an adaptive controller-based time-stepping scheme which automatically conforms to the demands imposed on the solver by the problem.

This work studies precisely the implementation of such a solver, using the error in the evaluation of the current time step to predict the best possible step size for the next time step. It begins with an introduction to some of the underlying theory required to contend with numerical solvers, the problems they will solve, and a bit of a deeper look at what exactly controllers are, focusing on the study of a specific type of controller known as a PID controller. Having implemented the solver in Python, it is tested for the aforementioned problems to explore its efficacy and reliability. Included is a comparison of the implemented solver with results produced by SciPy and for two problem cases, the implementation is compared with another in Julia which shares many of the same characteristics. Finally, the work is concluded with a discussion on the considerations made with respect to the Python and Julia solvers as well as the difficulties that were experienced working with the two implementations.

Contents

1 Introduction 4

2 Runge-Kutta Methods and Time Stepping 5
2.1 Method of Lines . . . 5
2.2 Truncation Error and Orders of Methods . . . 6
2.3 Explicit Runge-Kutta methods and FSAL . . . 7
2.4 Control Theory and Error-based step size controller . . . 8

3 Testing Runge-Kutta Methods and Controller Parameters 10
3.1 The Linear Test Equation . . . 10
3.1.1 Stability of Numerical Methods . . . 12
3.1.2 Stiffness . . . 13
3.1.3 Strong Stability Preserving RK Methods and Low Storage Implementations . . . 13
3.2 The Double Pendulum . . . 14
3.3 Nonlinear ODE Test Problem: Krogh, Problem 12 [21] . . . 16
3.4 Stiff Test Problem: Hairer, Nørsett, Wanner II, eq. (2.27) [14] . . . 18

4 Implementation and Usage Of The Solver In Python and Julia 20
4.1 Zero Division Errors . . . 20
4.2 Abnormally Slow Convergence Using BS3 Given Certain Tolerances . . . 20
4.3 Runtime Calculations . . . 20
4.4 Shortcomings of Comparing the Python and Julia Implementations . . . 20

5 Summary and Conclusion 21

Acknowledgements 21

Appendix 22
A.1: The Python Implementation of the Solver . . . 22

References 29

1 Introduction

In the sciences and engineering, hyperbolic partial differential equations (PDEs) are frequently used to model phenomena such as those encountered in fluid dynamics, acoustics and electrodynamics. These systems often cannot be solved analytically and so are reliant upon solutions produced using numerical methods. Explicit Runge-Kutta (RK) methods are one of the most commonly used numerical methods for solving temporal discretisations of hyperbolic PDEs (Section 2.1) due to their efficiency and parallel scalability [24]. The efficiency of a method is determined by the method's ability to choose the largest possible step size while maintaining stability (see Section 3.1.1) and accuracy demands. Given that numerical stability is one of the greatest obstacles when working with hyperbolic PDEs, various approaches have been developed to contend with it.

In 1928 Richard Courant, Kurt Friedrichs, and Hans Lewy [9] presented a reliable stability and convergence condition, now eponymously referred to as the CFL condition. For explicit time-stepping schemes, the CFL condition provides an estimate on the upper bound of the step size or equivalently the CFL number [24]. If we choose a CFL number such that the CFL condition is fulfilled, then we expect the numerical scheme not to introduce oscillations that are not inherent in the problem. The optimal CFL number depends on the selected spatial and temporal discretisations and sometimes may even be affected by the problem; it is usually found through trial and error [24]. This work an approach using error-based step size control suggested in [24] based on the earlier work of Gustav Söderlind [28]. Error-based controllers are algorithms in dynamical systems. They use the evaluation of local truncation error (see Section 2.2) throughout the time evolution of the system in order to conform to stability and convergence demands while maximising the step size. In comparison to CFL-based control, error-based control has the advantage of not requiring a manually tuned CFL number and allowing for control of the temporal error in the form of a tolerance when needed.

The objective of this work is to implement an explicit RK solver with error-based adaptive step size control that is capable of contending with a variety of problems, including nonlinear and stiff ODE problems. The solver is implemented in Python 3 and is based on the work done in [26]. To be able to use an explicit solver to contend with such problems is a very powerful advantage (see Section 3.1.2) that is otherwise regarded as impossible under normal circumstances. It is precisely due to this novel approach using error-based step size control that we can make such an augmentation and therefore save on computational power as well as give greater flexibility to the numerical solver. Thus, in this thesis, we shall explore the implementation and usage of such an error-control based solver. Given that the study of controllers is nontrivial, we shall commence with the preliminary theory required for understanding numerical methods along with a brief introduction to error-based control and controllers. Afterwards, the effectiveness of the proposed error-based controller approach for explicit RK methods shall be gauged through several problem cases. A section dedicated to discussing some of the considerations made when using the solver is also included.

2 Runge-Kutta Methods and Time Stepping

Before going into the implementation of the numerical solver that is the focus of this work, a preamble will be given to establish the foundation upon which the numerical solver is constructed.

2.1 Method of Lines

To be able to contend with a PDE problem numerically, we often time require discretising the problem into two separate components, temporal and spatial. One of the most commonly used methods to achieve this is the Method of Lines (MOL).

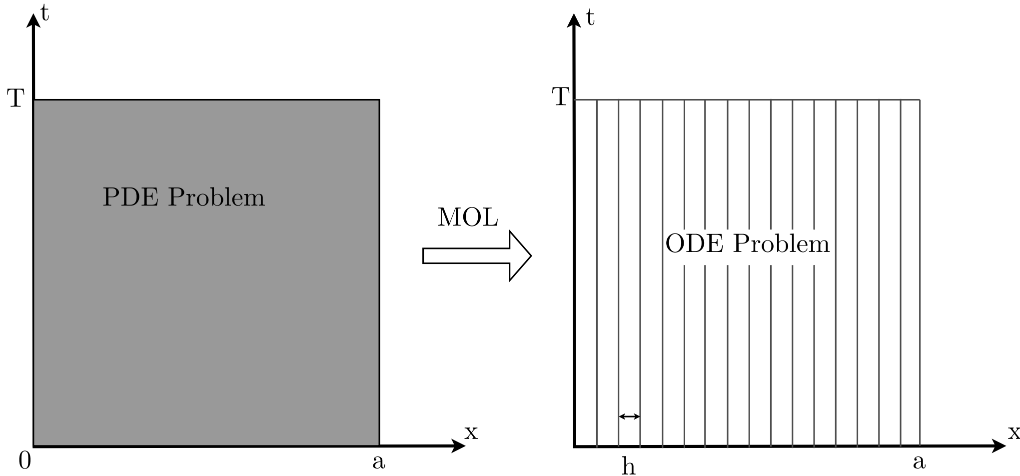


Figure 1: On the left we have a 2 dimensional problem with both variables x and t being continuous. MOL discretizes all but one dimension of the PDE problem, thus allowing for the problem to be solved along a discrete number of "lines" in the 2D case.

The method of lines discretises the spatial dimensions of the PDE problem, leaving the time variable continuous. The result is an ODE system which, given an initial value, can be solved using numerical methods. The resulting system, shown for an arbitrary function $u : [t_0, T] \to \mathbb{R}^m$ where m is the number of degrees of freedom, is thus given as,

$$\begin{cases} \frac{d}{dt}u(t) = f(t, u(t)),\ t \in [t_0, T], \\ u(t_0) = u_0. \end{cases} \tag{1}$$

Solving this ODE is the primary focus of many methods intended for PDEs, and as such, for solving this problem on the interval $[t_0, T]$, we call upon the RK method, which will be explored in Section 2.3.

2.2 Truncation Error and Orders of Methods

Before explaining numerical methods, it is important to establish the meaning of truncation error and the order of methods. Consider the following differential equation,

$$y' = f(t, y), \quad y(t_0) = y_0, \quad t \geq t_0. \tag{2}$$

We wish to find a numerical approximation y_n of the analytic solution $y(t_n)$ at discrete, equally spaced time steps t_n for $n = 1, 2, ..., N$ such that $h = t_n - t_{n-1}$ is constant. Now consider that we compute the sequence y_n using a one-step method, which is defined in the general form as follows below.

Definition 2.1 *A one step method for the computation of an approximation y_n of the solution of (2) on a grid $\{t_0, t_1, ..., t_N\}$, where t_0 refers to initial time and T the final time, has the form*

$$y_n = y_{n-1} + hK(t_{n-1}, y_{n-1}, h, f). \tag{3}$$

Herein, the function K is referred to as the increment function and can be seen as an estimate of the gradient of one step,

$$K \approx \frac{y(t_n) - y(t_{n-1})}{h}.$$

The local truncation error is the error in the solution at t_n that the increment function causes assuming we have knowledge of the analytic solution at t_{n-1}, that is to say, $y(t_{n-1})$ [27, pg. 317]. Formally, we define the local truncation error as follows.

Definition 2.2 *The local truncation error, τ_n, is the difference between the left and right-hand side of (3) and is of the form*

$$\tau_n = y(t_n) - y(t_{n-1}) - hK(t_{n-1}, y(t_{n-1}), h, f), \tag{4}$$

where K is the increment function.

If we now consider the aggregate of the local truncation error for all iterations up to t_n, assuming that we know of the analytic solution at the final time step, we have the global truncation error [27, pg. 317].

Definition 2.3 *The global truncation error, e_n, at a time step t_n with an increment function K is the sum of all local truncation errors of the prior time steps, given by*

$$\begin{aligned} e_n &:= y(t_n) - y_n \\ &= y(t_n) - (y_0 + hK(t_1, y_1, h, f) + ... + hK(t_{n-1}, y_{n-1}, h, f)). \end{aligned} \tag{5}$$

Using Definition 2.2, we may define the order of a method [18, pg. 323] as follows below.

Definition 2.4 *A numerical method has order p if for any sufficiently smooth solution the local truncation error is of order $\mathcal{O}(h^{p+1})$, meaning that there exists constants C and H such that*

$$|\tau_n| < Ch^{p+1}$$

for all $h < H$.

Order serves as a means to restrict the growth of the local error with respect to each successive approximation of the solution along the temporal grid by imposing an upper bound. Therefore, it follows that the higher the order of a method is, the smaller one would expect the rate of growth of the local error to be. Furthermore, using Definition 2.3, we may define an important concept known as convergence [18, pg. 5].

Definition 2.5 *A numerical method is convergent if for a step size h and global error e_n it holds that*

$$\lim_{h \to 0} \max |e_n| = 0.$$

2.3 Explicit Runge-Kutta methods and FSAL

In Section 2.1, we established the ODE system (1) that we wished to solve using the RK method. This section will explain how the explicit RK scheme is constructed in the regular and embedded cases. We begin by first positing the following definition.

Definition 2.6 *Given $c, b \in \mathbb{R}^s$ and $A \in \mathbb{R}^{s\times s}$, a strictly lower triangular matrix, an explicit Runge-Kutta method for (1) is of the form*

$$\begin{aligned} k_1 &= f(t_n, u_n) \\ k_2 &= f(t_n + c_2 h, u_n + (a_{21}k_1)h) \\ k_3 &= f(t_n + c_3 h, u_n + (a_{31}k_1 + a_{32}k_2)h) \\ &\vdots \\ k_s &= f(t_n + c_s h, u_n + (a_{s1}k_1 + a_{s2}k_2 + \cdots + a_{s,s-1}k_{s-1})h). \end{aligned} \tag{6}$$

Herein, k_i represents the stage derivatives of the Runge-Kutta scheme, $u_n \approx u(t_n)$ and $t_n = t_0 + nh$ for a constant time-stepping scheme. The stage derivatives are used to compute the solution at u_{n+1} as denoted by

$$u_{n+1} = u_n + h\sum_{i=1}^{s} b_i k_i, \tag{7}$$

where the sum represents the increment function of a simple Runge-Kutta method.

For a function $u(t)$, the stage derivatives serve as intermediary steps for gauging the gradient for a single step. k_1 uses the explicit Euler method at the beginning of the interval, giving a tangent at t_0; k_2 is the slope at a certain fraction of the step given by c_2 using u and k_1; k_3 is the slope at a factor c_3 of the step similar to before, now using u and k_2 and so on until k_s at the end of the interval. Figure 2 illustrates this for the classic RK4 (4 stage RK) method

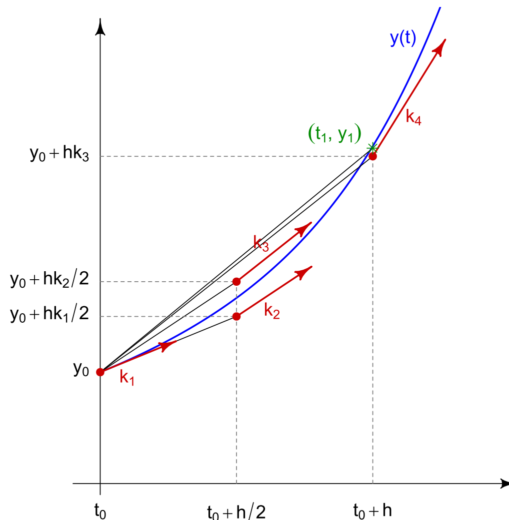


Figure 2: We note here that we are considering a function y as opposed to u however the same principle applies [16].

All RK schemes can be represented in a condensed form known as the Butcher Tableau [8], which provides all the necessary information for the implementation of the scheme.

$$\begin{array}{c|ccccc} 0 & & & & & \\ c_2 & a_{21} & & & & \\ c_3 & a_{31} & a_{32} & & & \\ \vdots & \vdots & & \ddots & & \\ c_s & a_{s1} & a_{s2} & \dots & a_{s,s-1} & \\ \hline & b_1 & b_2 & \dots & b_{s-1} & b_s \end{array}$$

The method that shall be used is a modification of the general explicit RK method, making two notable amendments. The first is the inclusion of an additional embedded step that can be seen in the Butcher Tableau below. The second is a property unique to some embedded RK schemes known as FSAL, First-Same-As-Last.

$$\begin{array}{c|c} c & A \\ \hline & b^T \\ & \widehat{b}^T \end{array}$$

We shall begin by explaining embedded RK schemes; herein, we have an embedded method adjacent to our main method [18, pg.113-118],

$$\begin{aligned} u_{n+1} &= u_n + h \sum_{i=1}^{s} b_i k_i, \\ \widehat{u}_{n+1} &= u_n + h \sum_{i=1}^{s} \widehat{b}_i k_i + \widehat{b}_{s+1} f(t_{n+1}, u_{n+1}). \end{aligned} \tag{8}$$

Here the k_i term is the same as in (6), but as can be seen we have an additional term that uses the value at u_{n+1}. For a main method of order q, the embedded method is usually selected to be one order lower, $\widehat{q} = q - 1$. The idea is that we wish to use $u - \widehat{u}$ to estimate the local truncation error, which is needed to determine the step size adaptively. If $\widehat{b}_{s+1} = 0$ then (8) is referred to as an ordinary RK pair, otherwise we call it an FSAL RK pair. The idea is to use the derivative of the new solution as an additional input for the error estimator. If the controller finds the step to be acceptable, then this calculation comes at no additional computational cost since the value $f(t_{n+1}, u_{n+1})$ must be computed at the next step regardless.

2.4 Control Theory and Error-based step size controller

As mentioned previously, a controller is an algorithm that operates within a dynamical system. These algorithms govern how system inputs (such as step size in the case of a numerical solver) are applied. The theory for controllers dates back to 1868 with James Clerk Maxwell pioneering the foundation of what would come to be known as the field of control theory [23]. Maxwell proposed the feedback controller, which takes in information from the system and produces an appropriate input variable with respect to that information. As an example, one can consider a computer's cooling fan increasing in RPM as being perpetrated by a controller on the motherboard that acts in response to a signal being sent from the thermal sensors. This signal is referred to as the process variable (PV). By feeding the PV through the controllers on the motherboard, the motherboard knows which fans need to run faster to cool the component that may be overheating. The desired temperature for the component to run at is referred to as the setpoint (SP) of the system. For the motherboard to know how hard it needs to make the fans work to achieve that cooling as optimally as possible, an error value, given by the difference between the SP and PV is required.

A particular type of controller that is popular in the industry [17] and will also be used here for our RK solver is the proportional-integral-derivative (PID) controller. A PID controller will calculate an error value $e(t)$ with respect to the given SP $= r(t)$ and PV $= y(t)$. The error is of the form,

$$e(t) = r(t) - y(t).$$

The controller applies a correction through three terms: the proportional control term, the integral term, and the derivative term [22]. The proportional control term is proportional to the current value of the error $e(t)$. If the error is large, that is to say, the difference between the PV and the desired SP is large, the controller output will be proportionately large by a gain factor K_p. We cannot rely on using proportional control alone because this will lead to a residual error between the SP and the preceding PV since the controller requires an error to generate an output response[22].

The integral term accounts for the past values of $e(t)$ and integrates them with respect to time to produce the integral term. Similar to the proportional term, it is scaled by a gain factor K_i. The integral term's purpose is to get rid of any residual error, the integral term will strive by adding a further control effect using the historic cumulative value of the error. When the error is eliminated, the integral term will no longer grow. Furthermore, the effect of the proportional control term will diminish as well; which thus prompts the integral term to grow once again.

Finally, we have the derivative term, which yields the best estimate for the future trend of $e(t)$ based on its current rate of change. It works to actively reduce the effect of the error generated by $SP - PV$ based on its rate of change and as such acts as a dampening term to prevent erratic fluctuations in the error. The greater the rate of change, the stronger the effect of the derivative term and similar to before, it is scaled by a gain factor K_d. A general introduction to PID controllers can be found in [22].

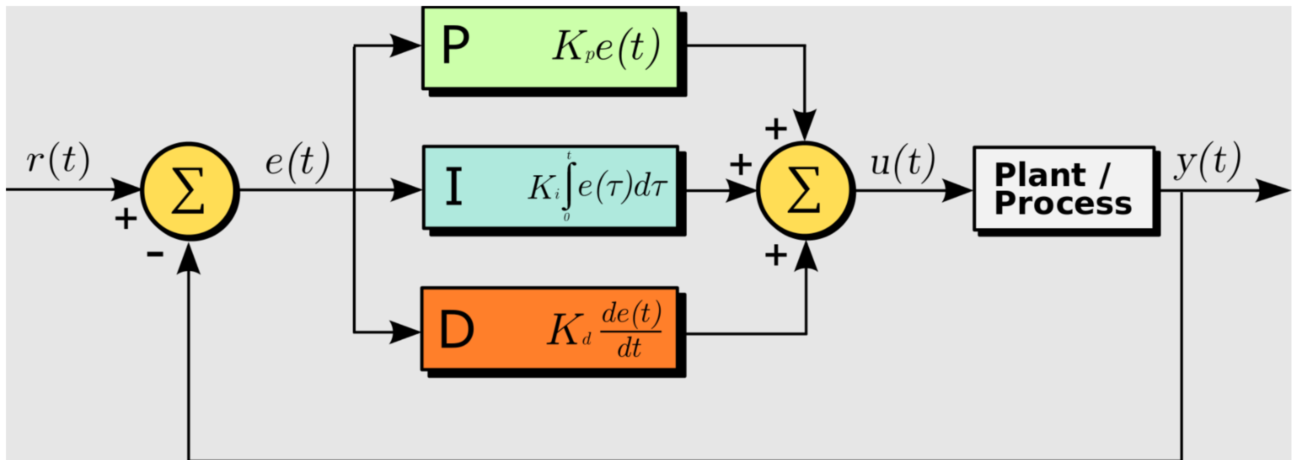


Figure 3: Block diagram of a PID controller in a feedback loop [30]. In mathematical terms, a control function is generalized as the sum of all the PID terms in the figure above, denoted as $u(t)$.

Figure 3 offers a visual guide as to how the PID controller works. However, despite the fail-safes in place thanks to the integral and derivative terms, numerical stability is not guaranteed by the PID controller and requires gain factors that are appropriate for a given problem (this is observed and explained further in Sections 3 and 4). We use PID controllers that select the new time step using (9) and the overall structure of the controller that we shall be using is similar to [24] and is of the form,

$$\Delta t_{n+1} = \varepsilon_{n+1}^{\beta_1/k} \varepsilon_n^{\beta_2/k} \varepsilon_{n-1}^{\beta_3/k} \Delta t_n, \tag{9}$$

$$\varepsilon_{n+1} = \frac{1}{w_{n+1}}, \quad w_{n+1} = \left(\frac{1}{m} \sum_{i=1}^{m} \left(\frac{u_i^{n+1} - \widehat{u}_i^{n+1}}{\text{atol} + \text{rtol} \max(|u_i^{n+1}|, |u_i^n|)} \right)^2 \right)^{1/2}. \tag{10}$$

It follows that for a main method of order q, embedded method of order $\widehat{q}$, $k = \min(q, \hat{q}) + 1$; usually, $\widehat{q} = q - 1$ and thus $k = q$. Thereafter, β_i is the gain factor for each control term (henceforth referred to as the controller parameters), m is the degrees of freedom in u and `atol` and `rtol` are the absolute and relative error tolerances. The absolute tolerance reflects the permitted absolute error in a solution for a given time step and the relative tolerance is the error permitted relative to the value of the solution. A notable difference between [24] and (10) is the usage of u_i^n as opposed to $\widehat{u}_i^{n+1}$ in the factor that multiplies `rtol`. The reason for this difference is because the Julia implementation that was accessed at the time [26], which served as a general guide for the Python implementation, used u_i^n as opposed to what is described in [24]. This discrepancy has little bearing on the results when tested in Python, however, given that in the overwhelming majority of evaluations, it was observed that u_i^{n+1} was greater than both u_i^n and $\widehat{u}_i^{n+1}$, as such it did not affect the results by a significant margin. Generally, it is standard that $\widehat{u}_i^{n+1}$ is used instead [2], however some implementations also utilise u_i^n in a similar way as what is shown in (10) [3].

All computations in Section 3 have been made with `rtol` = `atol`. The controller parameters, β_i are variant depending on the controller being used and the problem being considered. The equation for our relative error estimate, w_{n+1} is not the only possible choice, though it is the default choice in the context of powerful ODE software such as the `DifferentialEquations.jl` library in Julia [24]. The strength of such an equation is that we can give an error term that is independent of the spatial discretisation, which is ideal if one can use a program such as DUNE (The Distributed and Unified Numerics Environment) [5] to produce a spatial discretisation.

If the factor which multiplies out the previous step size, Δt_n in (9) is too small, the step Δt_{n+1} is rejected and a smaller step size is used instead, reduced by a factor of 4 and the solution at the ODE at the time step t_n is reevaluated. Notably, this is another point of difference between [24], however it was found to be a necessary change to ensure that the Python solver reliably converges for all problems and test cases. If the step taken is too large, then it can cause zero division errors, resulting in the solver being unable to converge. An accept safety of 0.81, the minimum magnitude of the step size scaling factor, has thus been set to avoid this. Should the safety condition not be met, the step is rejected and an alternative step size is used instead.

3 Testing Runge-Kutta Methods and Controller Parameters

In this section, we will examine the effectiveness of the controller implementation in Python. This will be done by testing the controller with different RK schemes for problems including the linear test equation, the double pendulum, a nonlinear ODE, and a stiff problem. Given that not all problems have simple analytic solutions, we use `SciPy.integrate.odeint()` which uses the LSODA ODE solver implemented in FORTRAN [4] to obtain a benchmark that we can use to compare our results with and gauge the error in our solution. This will be done in all cases barring the linear test equation due to the analytic solution being trivial.

Furthermore, the controller parameters, β are of great importance for a scheme to solve a given problem efficiently, if at all. For most problems, there exists a pair of optimal (PI) controller parameters for which the number of computational steps (usually the number of rejected steps) is minimized. This becomes significantly more apparent in the case of stiff problems where the choice of controller parameters greatly affects the efficiency and even accuracy of the result. These optimal parameters are obtained by testing the schemes for the best performance for a certain problem[24].

3.1 The Linear Test Equation

The linear test equation is the fundamental equation used to study numerical methods for solving differential equations. It is of the form,

$$\begin{cases} y'(t) = \lambda y(t),\ t \in [0, T], \\ y(0) = y_0 = 1, \end{cases} \tag{11}$$

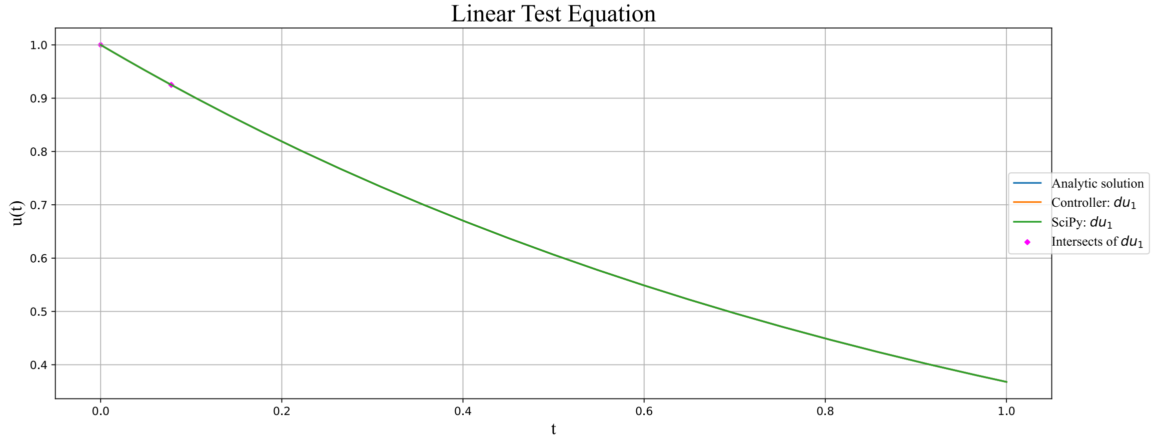
where T is the final time and 0 our initial time. For a system where $\lambda = -1$ and $T = 1$, the solution of the system is $y(t) = 1/e^t$, which is the case that we shall be considering here as well.

Table 1 gives an overview of how BS3(2)3$_F$ [6] (henceforth abbreviated to BS3), a 3rd order method of 4 stages (reduced to 3 evaluations per step due to it being an FSAL RK scheme) and BS5(4)7$_F$ [7] (abbreviated to BS5), a 5th order method of 8 stages (reduced to 7 due to it being an FSAL RK scheme) behave when trying to solve the linear test equation. Unique to this case, two different sets of controller parameters have been chosen to demonstrate the result of changing controller parameters.

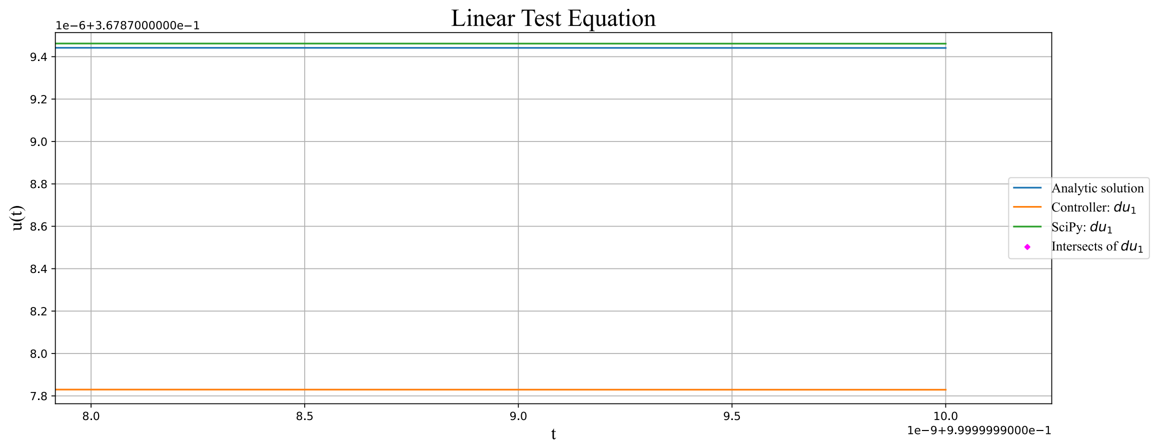
Table 1: Results of testing different schemes with different controller parameters β to solve the linear test equation at time $T = 1$. #A and #R show the number of accepted and rejected steps respectively, h is the initial step size and the Euclidean norm of the global error has been provided as well. The code was run 10,000 times and the minimum runtime was recorded (see Section 4.3).

Scheme	β	h	tolerance	#A	#R	#Error	Runtime (s)
BS3(2)3$_F$	$(0.60, -0.20, 0.00)$	10^{-3}	10^{-6}	30	3	1.61×10^{-6}	1.79×10^{-3}
	$(0.28, -0.23, 0.00)$	10^{-3}	10^{-6}	78	4	3.00×10^{-7}	4.61×10^{-3}
BS5(4)7$_F$	$(0.28, -0.23, 0.00)$	10^{-3}	10^{-6}	68	0	3.35×10^{-6}	1.07×10^{-2}
	$(0.60, -0.20, 0.00)$	10^{-3}	10^{-6}	20	0	1.30×10^{-5}	3.03×10^{-3}

It can be seen that for BS3, we have worse performance with respect to efficiency for solving the linear test equation with the second pair of controller parameters. Though one would be tempted to make the argument that we are achieving a lower error with the second pair of controller parameters, we can always decrease the tolerance by an order of magnitude for our solver, in which case the first set of parameters will always outperform the second in terms of minimum number of computations required with respect to the error. A mirrored case of this trend can be seen for BS5, wherein although we are solving the problem in more computational steps, our solution has a lower error. Some solvers lend themselves better to contending with some problems and to that end, it is the user's responsibility to try and find the solver that best suits their problems at hand. Whereas the step size h has been kept fixed for the solvers in Table 1, this will not always be the case. Depending on the problem, some solvers will require larger step sizes to function. This is generally a result of their respective stability regions, which are explained in Section 3.1.1.



(a) Solution curves for all dimensions of the problem. The points where the controller's solution curve lines up with the SciPy solution curve have also been shown.



(b) A close-up of the approximate area subject to the greatest difference between odeint and the controller-based approach, here shown to be $t = 1.0$.

Figure 4: BS3 and SciPy solving the linear test equation for the same conditions in Table 1 using the best-performing set of controller parameters. The x-axis and y-axis show time and the solution function $u(t)$ respectively.

As can be seen from Figure 4a, the controller-based solver can solve the linear test equation comparably to SciPy's with no regions exhibiting unstable or unexpected behaviour. If the solver is working well, then we would expect the method to be converging towards $1/e$ at $t = 1$ and the solution produced reflects this as well. By Figure 4b we see that we are under the solution curve that SciPy produces. Furthermore, SciPy's solver is over-evaluating its solution with respect to the analytic solution curve, however, it is still significantly closer than the controller's solution. Furthermore, it was found that the point of greatest difference between the SciPy solver and the controller was at $t = 1$, the final time, T for which we are solving the the linear test equation.

3.1.1 Stability of Numerical Methods

Having introduced the linear test equation, we may use it to explore the concept of numerical stability. Herein, we study whether the solution obtained by the numerical scheme converges to what we expect the analytic solution to be or if it amplifies the computational errors. To that end, we introduce the concept of a scheme's stability function in the following definition [18, pg. 59],

Remark 3.1 *For every Runge-Kutta scheme applied to the linear test equation $y' = \lambda y$, we have,*

$$y_{n+1} = R(h\lambda)y_n.$$

Here, $R(z)$ is the rational function denoted by,

$$R(z) = 1 + z\boldsymbol{b}^T(\boldsymbol{I} - z\boldsymbol{A})^{-1}\mathbf{1} \tag{12}$$

where $\boldsymbol{A}$ and $\boldsymbol{b}$ are both given by a scheme's Butcher Tableu, $\boldsymbol{I}$ is the identity matrix, and $\mathbf{1}$ is a vector of 1s. Equation (12) is called the scheme's stability function. If the method is explicit, then $R(z)$ is a polynomial of degree $s-1$, where s corresponds to the number of stages of the method.

Altogether, we are interested in the domain of this equation and how it maps $\mathbb{C}$. Using (12), we can get an idea of which values of our step size h our scheme will be numerically stable. To make the argument more formal, we introduce the following definition [18, pg. 60].

Definition 3.1 *A Runge-Kutta scheme's stability region, the region where the method is stable, is the set*

$$\mathcal{D} = \{z \in \mathbb{C} : |R(z)| \leq 1\}. \tag{13}$$

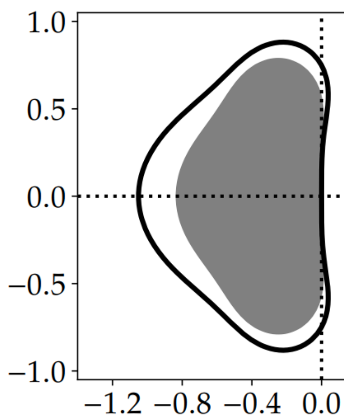
An important result that follows from using (12) and (13) together, is the concept of A-stability, wherein we have a condition that states that we are guaranteed exact solutions for which the origin is stable if $h\lambda$ lies in $\mathbb{C}^-$. This gives rise to the following theorem [18, pg. 61],

Theorem 3.1 *A Runge-Kutta method with stability function $R(z)$ is A-stable if and only if*

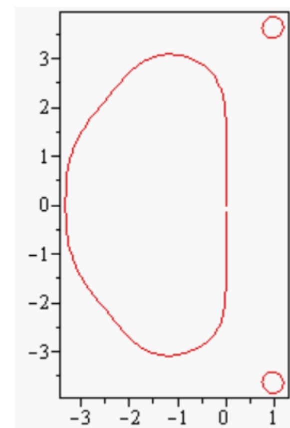
- *all poles of R have positive real parts, and*
- *$|R(i\omega)| \leq 1$ for all $\omega \in \mathbb{R}$.*

Proof. A proof can be found in [18, pg. 61-62].

Notably, the numerical methods that have been the focus of this paper all fail the condition set by Theorem 3.1 due to them being explicit. The stability function as obtained from (12) yields a polynomial $P(z)$ for all explicit Runge-Kutta methods and thus it follows that $P(z) \to \infty$ as $z \to \infty$. Figure 5 shows the stability regions of some of the methods used in this paper.



(a) 3rd order Bogacki-Shampine method [24]



(b) 5th order Dormand-Prince method [29]

Figure 5: The shaded grey part and area enclosed by the red line part represent the stable regions of the respective numerical schemes.

A-stability is a very useful quality for a numerical method, particularly when dealing with stiff (see Section 3.1.2) problems as it guarantees a stable solution for a greater variance of step sizes. However, this quality is not necessary to get good results for a stiff problem provided that the numerical method can meet the stability and accuracy demands of the problem; this case is explored in Section 3.4.

3.1.2 Stiffness

As was alluded to in the prior section, stiff problems are a prominent class of the differential equations that we encounter. The term "stiffness" however, lacks a formal definition and is rather described by the qualities the problem exhibits. A problem may be considered stiff if it has varying time scales, that is to say, some components of the solution decay comparatively faster than others. It is also often said that explicit methods are generally ineffective for solving stiff problems, due to implicit methods generally having A-stability. Stiff problems usually require an adaptive numerical method to efficiently solve the problem. This demand is due to the problem requiring more function evaluations than others, thus necessitating a form of step size control that is conscious of the stability demands imposed on the solver. This trade-off of accuracy for stability is explained by the *Second Dahlquist Barrier Theorem* [18, pg. 67] for linear multistep methods which states the following,

Theorem 3.3 *The highest order of an A-stable multistep method is* $p = 2$.

Proof. A proof can be found in [10].

As an additional note, for RK methods there are no maximum order restrictions for A-stability; A-stable RK methods can as a result be found at all orders, however, they can only be implicit [18, pg. 63]. Implicit methods come with the disadvantage of greater computational cost than explicit methods for a single step. Explicit methods require information about the system at present time t_n (such as an initial value at t_0) to solve for the solution of the system at time t_{n+1}. Implicit methods require information about the solution at both t_n and t_{n+1} to solve for the time step at t_{n+1}, hence why they are named the way they are. Such methods thus require more computations per step because they need to solve a non-linear system to solve for the t_{n+1} time step before the implicit method can use it to evaluate a solution at t_{n+1}. This generally gives a solution that is numerically stable for a greater variance of time steps than explicit methods at the cost requiring more computational time and memory per time step. Unless the step size restrictions imposed by the problem are sufficiently lax towards the point where the largest step size with an implicit scheme is considerably larger than what is possible with an explicit scheme, explicit methods are faster than implicit ones. The implementation of explicit methods is often significantly easier as well when compared to implicit methods, particularly of higher order. This is why it is useful to find methods for explicit Runge-Kutta schemes to solve stiff problems.

3.1.3 Strong Stability Preserving RK Methods and Low Storage Implementations

This section touches on two useful properties for RK methods that were used in [24], however, for reasons that will be explained, were not included in the Python implementation. The first of these, strong stability preserving (SSP) high order Runge-Kutta methods, were developed to be used in conjunction with the method of lines to solve hyperbolic PDEs and have found use for other applications as well [13]. These methods are motivated by the assumption that the explicit Euler time discretisation for the semi-discrete method of lines ODE is strongly stable under a certain norm, meaning that the inequality in (13) is a strict inequality. Thus, it is possible to find higher-order time discretisations that maintain this same stability property for the same norm while potentially under a different time stepping restriction. The SSP condition is useful to have for explicit schemes such as this work's error controller-based solver to be able to more comfortably compute problems that may have stiff regions where maintaining numerical stability is not simple, effectively by extending the stability region of the scheme [12]. However, they require more work to implement and be made compatible with the solver such as considerations of the CFL condition for a given spatial discretisation of a problem and as such, this family of RK methods have not been considered in this work.

A further amendment that is frequently considered in conjunction with an SSP implementation is the usage of Low-storage methods [19]. Usually, RK methods require simultaneous storage of all their stage derivatives for one point in time on the spatial grid, occupying m words, which we shall refer to henceforth as a register [24]. Low-storage RK methods seek to minimise the total number of registers; herein requiring three or four registers. The case where four registers are required is the minimum for a solver that is capable of supporting an embedded error estimator and rejecting steps. This is an optimisation of the code which minimises the amount of memory needed to be assigned for the solver to compute a solution across the temporal and spatial discretisation. Low-storage methods are very useful for problems that are solved on large spatial grids which demand many evaluations and thus more memory to solve it in a stable manner. This is why memory-minimising methods are considered frequently when it comes to using RK methods to solve temporal discretisations of method of lines approximations of hyperbolic PDEs. The implementation of these methods differs from the standard method for implementing RK schemes in the sense of particular considerations needing to be made with regard to the constraints placed upon the coefficients of the RK scheme such that the method can be turned into a low storage one [20]. Given that the scale of this work is small, such methods have not been considered in Section 3.

3.2 The Double Pendulum

The double pendulum is a chaotic system that, as implied, is highly sensitive to the initial conditions and exhibits very active dynamic behaviour. It is represented by 4 first order ODEs with two degrees of freedom, and is of the form,

$$\begin{cases} \dot{\theta_1} &= \dfrac{\partial H}{\partial p_{\theta_1}} = \dfrac{l_2 p_{\theta_1} - l_1 p_{\theta_2}\cos(\theta_1-\theta_2)}{l_1^2 l_2\left[m_1 + m_2\sin^2(\theta_1-\theta_2)\right]} \\ \dot{\theta_2} &= \frac{\partial H}{\partial p_{\theta_2}} = \frac{-m_2 l_2 p_{\theta_1}\cos(\theta_1-\theta_2)+(m_1+m_2)l_1 p_{\theta_2}}{m_2 l_1 l_2^2[m_1+m_2\sin^2(\theta_1-\theta_2)]} \\ \dot{p_{\theta_1}} &= -\frac{\partial H}{\partial \theta_1} = -(m_1+m_2)gl_1\sin\theta_1 - A + B\sin\left[2(\theta_1-\theta_2)\right] \\ \dot{p_{\theta_2}} &= -\frac{\partial H}{\partial \theta_2} = -m_2 g l_2 \sin\theta_2 + A - B\sin\left[2(\theta_1-\theta_2)\right], \end{cases} \tag{14}$$

where $p_{\theta_{1,2}}$ are the generalised momenta of the limbs of the pendulum. For the computations done in this section, it was considered that $l_1 = 1$, $l_2 = 1$, $m_1 = 1$, $m_2 = 1$, and $g = 9.81$. A and B are functions of $(\theta_1, \theta_2, p_{\theta_1}, p_{\theta_2})$ defined as,

$$A = \frac{p_{\theta_1} p_{\theta_2}\sin(\theta_1-\theta_2)}{l_1 l_2\left[m_1 + m_2\sin^2(\theta_1-\theta_2)\right]}$$

$$B = \frac{m_2 l_2^2 p_{\theta_1}^2 + (m_1+m_2) l_1^2 p_{\theta_2}^2 - 2m_2 l_1 l_2 p_{\theta_1} p_{\theta_2}\cos(\theta_1-\theta_2)}{2 l_1^2 l_2^2\left[m_1 + m_2\sin^2(\theta_1-\theta_2)\right]^2}.$$

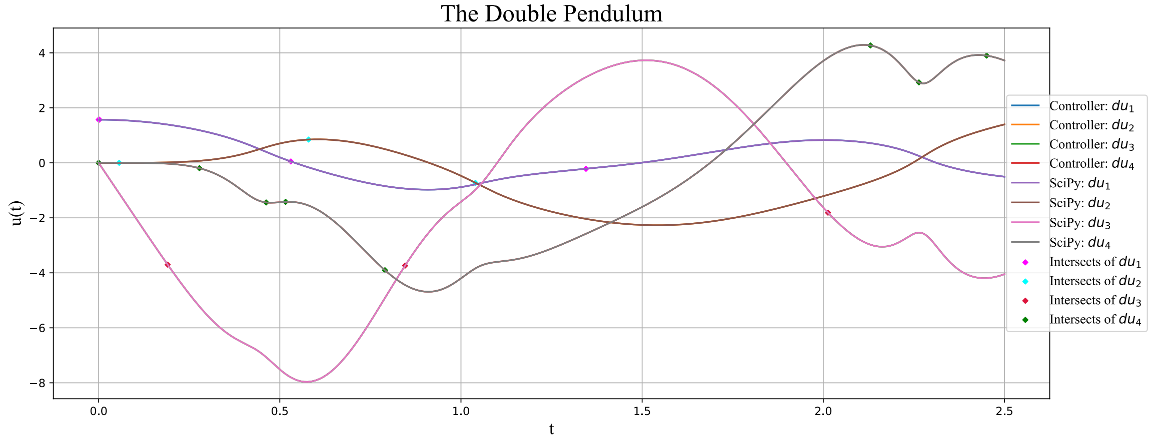
The utility of this problem is in its chaotic nature requiring adaptive methods to contend with it, testing the effectiveness of the controller. Not only are the systems computationally intensive, but they also lack analytic solutions and thus are usually solved with other adaptive RK methods.

Table 2: Results of testing different schemes to solve problem (14) at time $T = 2.5$. #A and #R show the number of accepted and rejected steps respectively, h is the initial step size and the Euclidean norm of the absolute error, comparing the result found to the LSODA solver in FORTRAN, has been provided. The code was run 1,000 times and the minimum runtime (see Section 4.3) was recorded.

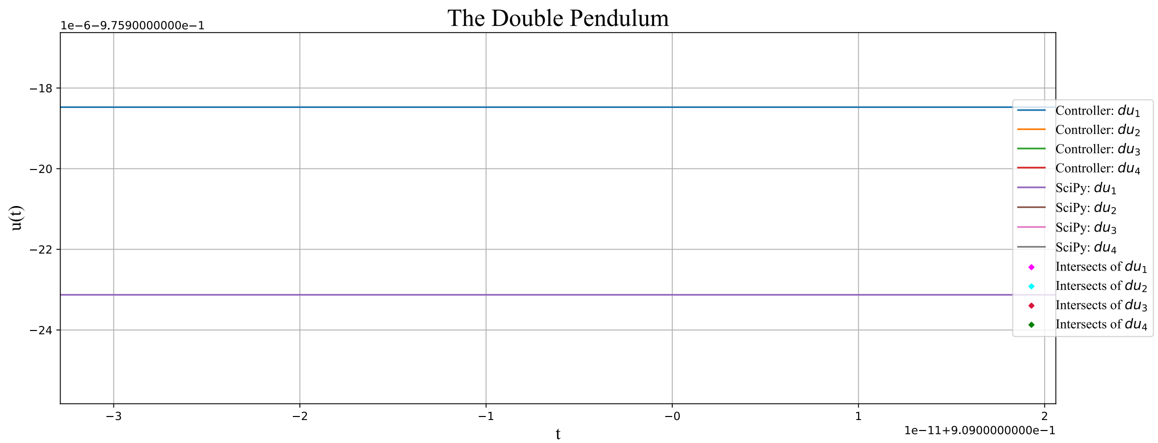
Scheme	β	h	tolerance	#A	#R	#Error	Runtime (s)
BS3(2)3$_F$	$(0.60, -0.20, 0.00)$	10^{-3}	10^{-6}	361	4	2.19×10^{-4}	3.21×10^{-2}
BS5(4)7$_F$	$(0.28, -0.23, 0.00)$	10^{-3}	10^{-6}	142	0	3.27×10^{-3}	3.01×10^{-2}
DP5(4)6$_F$	$(0.70, -0.40, 0.00)$	10^{-3}	10^{-6}	58	5	8.92×10^{-4}	1.07×10^{-2}
	$(0.70, -0.40, 0.00)$	10^{-2}	10^{-6}	55	3	6.60×10^{-4}	9.63×10^{-3}

From Table 2, we can see that BS3 edges out BS5 for this problem here due to having a smaller error at the cost of a slightly longer runtime. The total number of accepted steps is not always a reliable measure of efficiency given that two different methods could have varying orders of time complexity for a single step depending on their algorithm and implementation. The higher a scheme's order of convergence is, typically the more operations are required to be able to evaluate a single time step, however in considering the runtime, we consider the aggregate and as such get a more objective benchmark for efficiency (see Section 4). DP5(4)6$_F$ [11] (henceforth abbreviated to DP5) however outperforms both BS3 and BS5 in terms of efficiency. Despite this, BS3 still outperforms DP5 in terms of accuracy, which once again posits the compromise one makes between accuracy and efficiency. Given that both BS3 and DP5 have rejected steps, it is likely their controller parameters are not ideal for this problem and as such are causing step rejections. Notably, DP5 had better accuracy for larger step sizes. This suggests that the controller parameters for DP5 are ill-suited for this problem with these conditions.

In general, it is better to consider higher-order methods for certain problem types that are more chaotic. The easiest way of contending with problems like this is by utilising Julia's rich library of ODE solvers `OrdinaryDiffEq.jl` which has powerful (i.e. 9th order) schemes that are much better suited to contend with such problems. Unlike the linear test equation, both orders of RK schemes are now rejecting steps to maintain stability throughout the computation. A consistent trend here is that BS5 rejects fewer steps than BS3, which, despite performing arguably worse than BS3 in terms of accuracy, can be seen as having an easier time preserving stability and thus requiring fewer computational steps to solve the problem for the whole time interval, which is in line with expectations.



(a) Solution curves for all dimensions of the problem. The points where the controller's solution curves line up with the SciPy solution curves have also been shown.



(b) A close-up of an approximate area subject to the greatest difference between odeint and the controller-based approach, here shown to be $t = 0.909$.

Figure 6: BS3 and SciPy solving the double pendulum problem for the same conditions in Table 2. The x-axis and y-axis show time and the solution function $u(t)$ respectively.

The controller, similar to Figure 4, is once again able to contend with the problem and give comparable results to SciPy as can be observed in Figure 6a. There do not seem to be any particular areas in which errors cause obvious dissimilarities between the two solution curves. However, similar to before, we have time values where the local errors (taken here and henceforth as a measure of the difference between SciPy and the controller-based solver) are the largest with respect to one dimension of the ODE we are solving. Interestingly, this point is shared by du_1 and du_4 at around $t = 0.909$ and is therefore the time step that has been focused on in Figure 6b. It is not clear why this point is causing these two particular parts of the ODE the greatest error. The points of maximum local error for du_2 and du_3 are around $t = 1.54$ and $t = 0.576$ respectively.

3.3 Nonlinear ODE Test Problem: Krogh, Problem 12 [21]

Nonlinear differential equations are a type of differential equation that is not a linear equation in its unknown function or its derivatives. Generally, these differential equations exhibit chaotic properties and rarely have analytic solutions, even well-posed initial and boundary value problems are nontrivial in many cases. Fortunately, there is a test problem [21], which while still sensitive and problematic for maintaining numerical stability if the tolerances considered are not within certain thresholds, can be used for our purposes here. A modified version of the problem is explored in [15], which is the same version that we shall be exploring here. Furthermore, the problem shall also be solved using a similar implementation in Julia of the adaptive explicit RK solver [26] which utilises the same controller described in (10); the results of each implementation shall be compared. The problem is described as follows,

$$\begin{cases} u' = -Bu + U^T \left(\frac{z_1^2}{2} - \frac{z_2^2}{2}, z_1 z_2, z_3^2, z_4^2 \right)^T, \\ u(0) = (0, -2, -1, -1)^T, \end{cases} \tag{15}$$

with

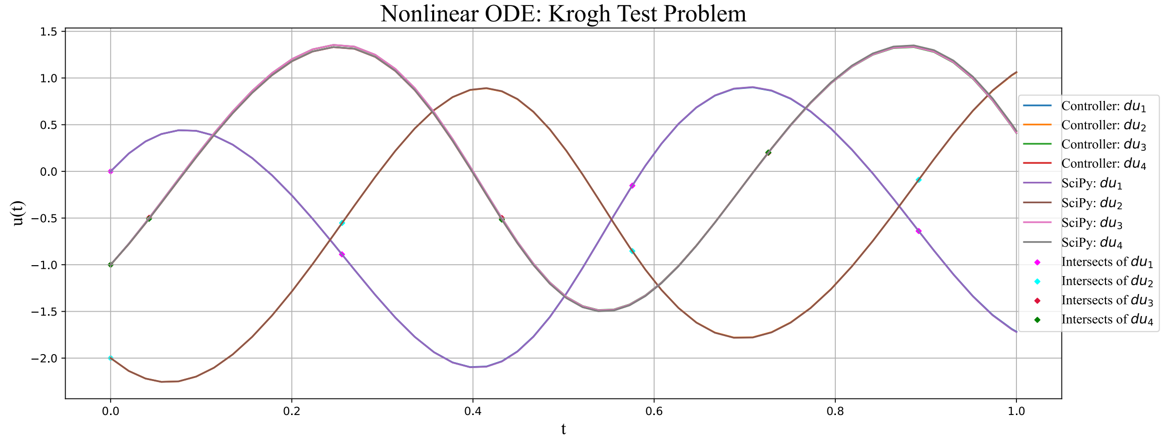
$$(z_1, z_2, z_3, z_3) = z = Uu, \quad B = U^T \begin{pmatrix} -10\cos(\phi) & -10\sin(\phi) & 0 & 0 \\ 10\sin(\phi) & -10\cos(\phi) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1/2 \end{pmatrix} U, \tag{16}$$

where $U \in \mathbb{R}^{4\times 4}$ is a unitary matrix containing $-1/2$ on the diagonal and $1/2$ in all other components. ϕ is a fixed parameter which can affect the number of rejected steps and the values of the dominant eigenvalues of the Jacobian. Particularly, as $t \to \infty$, the dominant eigenvalues become $-10|\cos(\phi)| \pm i10\sin(\phi)$. The system is solved for a time span of $[0, 1]$ and for a fixed value $\phi = \pi/2$, for which we expect no rejected steps [25].

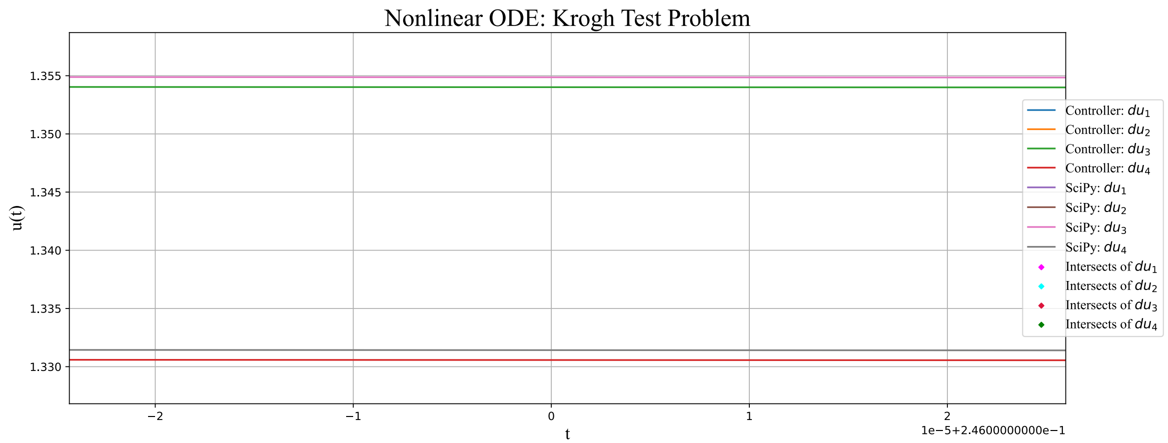
Table 3: Results of testing different schemes and implementations to solve the nonlinear ODE problem (15) at time $T = 1$. #A and #R show the number of accepted and rejected steps respectively, h is the initial step size and a measure of the Euclidean norm of the absolute error is given. All results are compared to what was found using SciPy, including the Julia implementation. The Python code was run 10,000 times (see Section 4.3) and the minimum runtime was recorded; runtime was not recorded for the Julia results due to similar testing conditions not being achievable at the time the data was gathered.

Scheme	β	h	tolerance	#A	#R	Error	Runtime (s)
BS3(2)3$_F$	$(0.60, -0.20, 0.00)$	10^{-2}	10^{-4}	52	0	6.69×10^{-3}	5.02×10^{-3}
BS5(4)7$_F$	$(0.28, -0.23, 0.00)$	10^{-2}	10^{-4}	37	0	2.50×10^{-3}	8.42×10^{-3}
DP5(4)6$_F$	$(0.70, -0.40, 0.00)$	10^{-2}	10^{-4}	25	2	1.47×10^{-3}	5.17×10^{-3}
(Julia) BS3(2)3$_F$	$(0.60, -0.20, 0.00)$	10^{-2}	10^{-4}	51	0	6.69×10^{-3}	
(Julia) BS5(4)7$_F$	$(0.28, -0.23, 0.00)$	10^{-2}	10^{-4}	27	0	3.45×10^{-6}	
(Julia) DP5(4)6$_F$	$(0.70, -0.40, 0.00)$	10^{-2}	10^{-4}	18	0	6.46×10^{-4}	

Table 3 shows the breadth of effectiveness for different numerical schemes and implementations. Even between schemes of equivalent orders of convergence, the results are not necessarily the same, and one method can still significantly outperform another method as can be seen with DP5 and BS5. While we note that this difference in error is subtle for BS5 and DP5 in the Python implementation, it is 2 orders of magnitude apart in the Julia implementation. When comparing the results produced by Julia and Python, for almost all schemes barring BS3, Julia performs significantly better, having smaller errors and requiring fewer steps for convergence. Given that the BS3 cases line up nearly identically with each other, it is unlikely that the cause of this discrepancy is an undiscovered flaw or bug in the Python implementation, despite the two rejected steps observed for Python's DP5 case being unexpected; rather it would seem that the Julia implementation is proficient at solving this particular problem, particularly with the BS5 scheme. No significant differences in the number of total steps was observed between the implementation in [26] and `OrdinaryDiffEq.jl` (also provided by the Julia implementation, but not included here) solvers. More test cases are required to arrive at a conclusive verdict regarding the efficacy of each implementation.



(a) Solution curves for all dimensions of the problem. The points where the controller's solution curves line up with the SciPy solution curves have also been shown.



(b) A close-up of an approximate area subject to the greatest difference between odeint and the controller-based approach, here shown to be around $t = 0.246$.

Figure 7: BS3 and SciPy solving the nonlinear Krogh problem for the same conditions in Table 3. The x-axis and y-axis show time and the solution function $u(t)$ respectively.

Upon first inspection of Figure 7a one may think that there are only 3 solution curves for a 4-dimensional problem, in reality, du_3 and du_4 follow similar, but not identical trends which cause them to almost overlap. Figure 7b shows that these two lines are separated by a greater magnitude than the relative separation between the solution curves of the solvers. The time step chosen here, $t = 0.246$ maximises the local error (taken as the difference between the solvers) for du_3. du_1, du_2 and du_4 experience maximal local errors at $t = 0.397$, $t = 1.0$, $t = 0.887$.

3.4 Stiff Test Problem: Hairer, Nørsett, Wanner II, eq. (2.27) [14]

Now that we have confirmed our method to be working well, we can test it for a stiff problem, the theory of which was covered previously in Section 3.1.2. The problem selected is the following,

$$\begin{cases} y_1' = -2000(\cos(x)\cdot y_1 + \sin(x)\cdot y_2 + 1), & y_1(0) = 1, \\ y_2' = -2000(-\sin(x)\cdot y_1 + \cos(x)\cdot y_2 + 1), & y_2(0) = 0 \end{cases}. \tag{17}$$

Just as before, we will be using the same BS3, BS5 and DP5 schemes and controller parameters. We will again be comparing the results to the implementation in Julia [26] that was used for the nonlinear ODE problem (Section 3.3). The final time for the problem has been set to $T = 1.57$ to ensure that the stiff region of the ODE is solved in its entirety. An advantage of this problem specifically is that the controller and its parameters have been optimised to be effective for it [24]. Thus, the results presented for BS3, DP5, and BS5 here may be considered to be reflective of optimal performance for the implementations in solving (17).

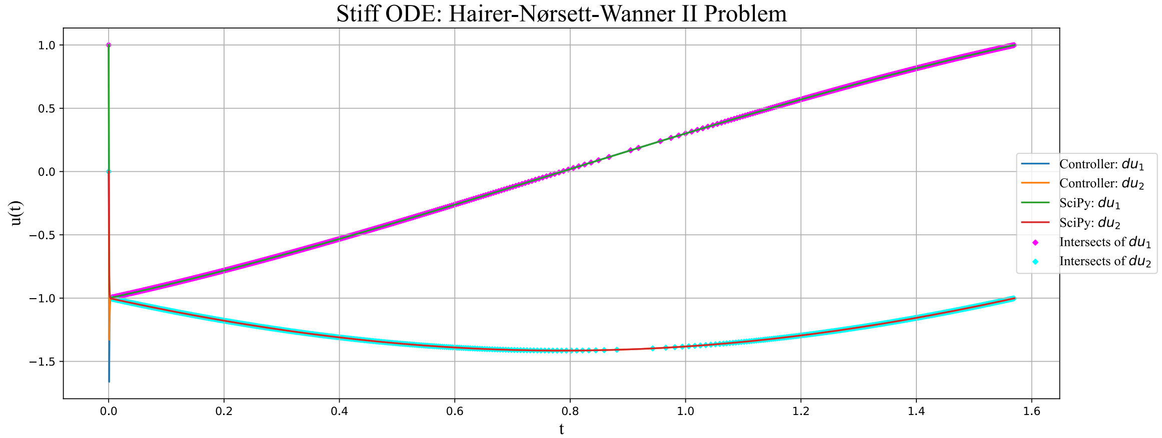
Table 4: Results of testing different schemes and implementations to solve the stiff test problem (17) at time $T = 1.57$. #A and #R show the number of accepted and rejected steps respectively, h is the initial step size and a measure of the Euclidean norm of the absolute error is given. All results are compared to what was found using SciPy, including the Julia implementation. The Python code was run 1,000 times (see Section 4.3) and the minimum runtime was recorded; runtime was not recorded for the Julia results due to similar testing conditions not being achievable at the time the data was gathered

Scheme	β	h	tolerance	#A	#R	Error	Runtime (s)
BS3(2)3$_F$	$(0.60, -0.20, 0.00)$	10^{-3}	10^{-4}	1329	4	1.22×10^{-3}	7.21×10^{-2}
BS5(4)7$_F$	$(0.28, -0.23, 0.00)$	10^{-3}	10^{-4}	820	0	3.20×10^{-2}	5.02×10^{-3}
DP5(4)6$_F$	$(0.70, -0.40, 0.00)$	10^{-3}	10^{-4}	991	2	9.82×10^{-3}	1.16×10^{-1}
(Julia) BS3(2)3$_F$	$(0.60, -0.20, 0.00)$	10^{-3}	10^{-4}	1328	5	1.19×10^{-3}	
(Julia) BS5(4)7$_F$	$(0.28, -0.23, 0.00)$	10^{-3}	10^{-4}	819	1	2.83×10^{-2}	
(Julia) DP5(4)6$_F$	$(0.70, -0.40, 0.00)$	10^{-3}	10^{-4}	985	49	9.81×10^{-3}	

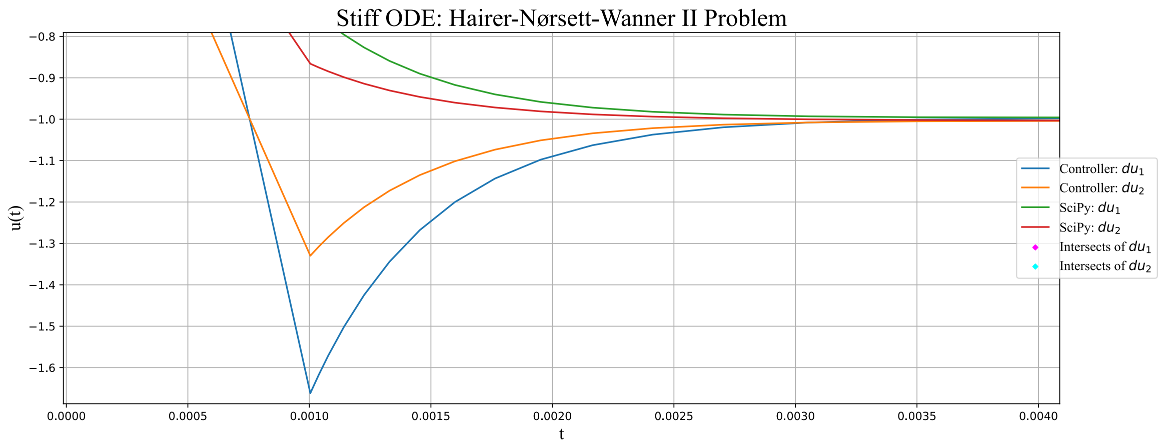
Table 4 suggests that for DP5, the Python solver is able to converge in fewer computational steps than the Julia implementation [26], as evidenced particularly by the number of rejected steps. Despite this, the Python solver has marginally lower accuracy. The results produced by the Python solver showed a stronger correlation to the solver in `OrdinaryDiffEq.jl` in terms of the total number of steps taken. The Python solver took only 4 more steps than the `OrdinaryDiffEq.jl` solver, rejecting the same number of steps.

A potential reason for why the results are not identical could be because Julia can use a variable precision for floating point arithmetic [1] for greater accuracy if necessary, as opposed to Python which uses the standard float which is 64 bits. This affects how the calculations are done and the degree of accuracy that can be achieved. Furthermore, the controller may be subject to certain implementation-related nuances, as is discussed further in Section 4. In comparison to what was observed in Table 3 for the nonlinear ODE problem, we can see that BS5 still produces a smaller error in the case of the Julia implementation, though by a significantly smaller margin and with the same number of total steps required for convergence. BS3 still serves as the case where both implementations provide the most similar results. An unexpected observation was to see Julia requiring more rejected and total steps for convergence in the DP5 case as opposed to Python, yet producing a similar error. Given the fact that the results for Python's DP5 is comparatively closer to that of `OrdinaryDiffEq.jl` for the stiff test problem, the observed differences between the implementations would not suggest one to be more efficacious than the other.

As mentioned in Section 3.1.2, A-stability is a commonly desired trait for a solver to be able to successfully evaluate stiff differential equations while maintaining numerical stability, a trait only found in implicit RK schemes. We have been successful in doing so with an explicit adaptive scheme using error-based step size control.



(a) Solution curves for all dimensions of the problem. The points where the controller's solution curves line up with the SciPy solution curves have also been shown.



(b) A close-up of the approximate area subject to the greatest difference between odeint and the controller-based approach.

Figure 8: BS3 and SciPy solving the Hairer-Nørsett-Wanner test problem for the same conditions in Table 4. The x-axis and y-axis show time and the solution function $u(t)$ respectively.

Beyond around $t = 0$, both SciPy and the controller behave well and are equally adept at contending with the demands on the solver that come with stiff problems. To this extent, Figure 8a is confirmation that the controller-based solver is indeed stable, even for an explicit RK method. However, the erratic behaviour of the first time step of the solver is an indication that something is causing errors. SciPy's solution curves start at the point given by the initial condition and stably converge towards the solutions for du_1 and du_2, whereas the controller's solution curves are seemingly in free fall up to a point. The exact position of this point is $t = 0.001$, as seen in figure 8b, the same as the initial step size used in our solver as seen in Table 4.

The controller requires an initial step size prior to gauging what step size should be used to try and maintain stability. As a result, it makes sense that using an explicit scheme with fixed step size, which as a consequence of Theorem 3.1 can never be A-stable, would be error prone in its first step. Fortunately, as soon as the controller comes into effect, the solver corrects itself and maintains stability throughout the remainder of the calculation. SciPy's controller not experiencing this issue is generally indicative that LSODA is using a method such as an implicit scheme to solve the issue of the first step being unstable. The instability of the first step cannot be amended by the controller and is an unfortunate, if not detrimental, shortcoming of adaptive explicit Runge-Kutta solvers.

4 Implementation and Usage Of The Solver In Python and Julia

The following section is meant to act as a description of the difficulties and constraints that were encountered when developing and working with the Python implementation as well as working with the Julia implementation given in [26]. Explanations of why certain things are implemented the way they are have also been provided to help better consolidate the rationale behind some choices in Section 3.

Note: It has been assumed absolute and relative tolerances are equal for this section.

4.1 Zero Division Errors

One of the most prominently experienced difficulties in getting the solver to work for all the cases examined in Section 3 is the matter of the controller avoiding zero division, which specifically may occur when computing the ε_{n+1} terms used in (9). While it is suggested that increasing step size will circumvent this flaw [26], this was notably found to not be the case with the version of the Julia implementation that was accessed at the time when the data was gathered. The solver was unable to avoid zero division errors, notably for the linear test equation (Section 3.1) and Double Pendulum cases (Section 3.2), where the Julia implementation would raise a zero division error for any step size value, irrespective of their magnitude. Since the data was gathered, it was later discovered that the Julia implementation accessed was an older version as compared to the one currently available in [26], which may suggest the presence of bugs being responsible for the discrepancies in the results that were found in Section 3.4 and perhaps also Section 3.3.

4.2 Abnormally Slow Convergence Using BS3 Given Certain Tolerances

Several different configurations of initial step size and tolerance were examined prior to settling on the ones presented in the tables of Section 3. The stiff test problem (Section 3.4) was observed to behave well in Python when solved with BS3 for almost all tolerances chosen from $[10^{-1}, 10^{-9}]$, with initial step size h chosen between $[10^{-1}, 10^{-9}]$ as well while keeping the number of rejected steps less than 100. There is notably a portion of the interval of tolerances that were examined which violates this trend for any step size, namely when `tol` is in the interval $[7.044 \times 10^{-8}, 2.998 \times 10^{-7}]$. In this case, the solver has anywhere from over 100 to over 1700 rejections. Notably, the solver still produces an error that is in line with expectations relative to the tolerance given to it. In the interest of seeing if this phenomenon is due to the selected scheme, BS5 was also tested with `tol` in the range $[7.044 \times 10^{-8}, 2.998 \times 10^{-7}]$ and was found to converge with less than 20 rejected steps for $h \in [10^{-1}, 10^{-9}]$. DP5 experiences zero division errors for any tolerance stricter than 10^{-6}, hence it could not be considered. A similar phenomenon was not observed for any other problem examined in Section 3, nor was this phenomenon observed in the Julia implementation for the same problem and configuration.

4.3 Runtime Calculations

The runtime was calculated in Python using the `timeit` module. This module can simulate a function call and its runtime a desired number of times and store the runtimes in an array. One particular choice that was made was to consider the minimum run time of a given scheme solving a problem as opposed to the average. The reason for this is that the minimum runtime for code is the case where there is the least noise caused by background processes running on the computer. The way the threads of the processor are used is not determined by the user for all processes and as such it is generally not reasonable to assume a thread dedicates all its processing power towards running the code. An average (arithmetic mean) of the runtimes would only succeed in giving a measure of how fast the code runs while under average noise conditions, which while one can argue to be a realistic performance metric, does not adequately test the efficiency of the code. A comparison of time complexity, while theoretically fruitful, is not possible given that the number of steps taken for an adaptive solver is dependent on the problem, its step size and tolerance parameters, and the implementation.

4.4 Shortcomings of Comparing the Python and Julia Implementations

Tables 3 and 4 give a measure of how well this Python implementation contends with a Julia implementation [26] for a variety of schemes for the nonlinear ODE and stiff test problems respectively. One must consider that the discrepancies, while certainly affected by rounding (as mentioned in Section 3.4), are also affected by subtle differences in the implementation that have been observed to have profound ramifications in the results each solver produces. Given that the Julia implementation failed to function for the linear test equation and double pendulum problems, a complete comparison between the implementations could not be made. Thus given the lack of test cases, the data produced by these implementations fail to suggest any conclusive result insofar as the comparison of the implementations is concerned.

5 Summary and Conclusion

Throughout this work, we have briefly touched upon the areas wherein numerical methods become necessary to contend with differential equations and motivated why one would consider using error-based control for a numerical solver. An introduction to the method of lines approximation for partial differential equations was given in Section 2.1 along with preliminary theory for numerical methods in Section 2.2. Section 2.3 served as an introduction to Runge-Kutta methods, embedded RK schemes and the FSAL property for increasing computational efficiency. Thereafter, in Section 2.4 control theory and PID controllers were touched on; followed by an explanation of the controller used for the adaptive explicit RK solver which is the primary focus of this work. In Section 3, the adaptive solver was tested across several different problems such as the linear test equation, the double pendulum, a nonlinear ODE problem, and a stiff problem. The results were compared with SciPy's `odeint` function, which uses the LSODA solver implemented in FORTRAN. In all cases, the results were comparable with the SciPy solver, suffering from relatively small errors, however, it was noted that ill-suited controller parameters caused disturbances in the data for some cases. After the study of the linear test equation in Section 3.1, an introduction was given to stability and the different conditions for preserving stability and efficiency were explained. In the case of the nonlinear ode problem and stiff test problem, covered in Sections 3.3 and 3.4 respectively, a further comparison was made with a similar implementation of the solver in Julia [26], and presented noteworthy, if inconclusive, findings. A discussion of the considerations made and difficulties that were encountered with the solver in the Python and Julia implementations studied was presented as well.

Overall, the solver was very successful in being able to contend with a variety of problems and FSAL RK schemes, while presenting comparable results to the Julia implementation. The algorithm itself is relatively efficient for a full-storage method, one that does not seek to minimise the number of registers. Because of that, it is doubtful if it is as efficacious as a low-storage method for solving memory-demanding problems such as those frequently encountered in computational fluid dynamics (CFD). Other works such as the 2021 paper by Ranocha, Dalein et.al [24] have explored this topic in greater depth, specifically exploring its application in CFD and proposing an approach to optimizing controller parameters for convection-dominated problems. Similar to what was found in this work, $BS3(2)_F$ of Bogacki and Shampine [6] performed quite well in comparison to the other RK schemes that were explored. It was mentioned in [24] that the best-performing strong-stability preserving method SSP3(2)4[3S*+] was rather sensitive to the tolerance chosen. However, this could alleviate the restriction of the step size in relation to tolerance which was mentioned in Section 4 and as such could be explored in the future.

A further extension of this work could be to implement it in the modular C++ library DUNE. DUNE is a very powerful tool that is used primarily for solving partial differential equations with grid-based methods as was discussed in Section 2.1. It has the means of converting a Python code into C++, compatible with the software; however, implementing this algorithm is not easy and requires the Python code to be made compatible with the conversion to ensure no bugs emerge. As a result, it could be worthwhile to directly implement the algorithm in C++ to be compatible with DUNE as well as to further improve the efficiency. This would allow for the analysis of a variety of interesting problems such as those found in meteorology as relating to atmospheric flow as well as other problems found in CFD. In its current form, the Python solver has been tested for a variety of different problems and has been able to produce respectable results in all instances. The code for the Python implementation has been attached as an appendix to this work (see Section A.2) and is capable of providing comparable results to Julia's `OrdinaryDiffEq.jl` library, as well as the implementation found in [26]. To that end, we have been successful in implementing an adaptive explicit RK solver that uses an error-based step size controller. It was found to be successful in contending with nonlinear and stiff ODEs as well as chaotic systems and presented reliable findings.

Acknowledgements

I would like to extend my gratitude to my supervisor, Robert Kloefkorn for his patience and assistance with this thesis and the opportunities he has provided me with throughout the time working together. I further extend these gratitudes to Hendrik Ranocha for taking the time to answer my questions in regards to his work which was pivotal in making the Python implementation successful. I am also thankful for much of the guidance that was provided to me during my studies by Claus Führer, whose sage wisdom and guidance were instrumental in re-evaluating my approach to mathematics over the years. Finally, I am eternally grateful to my friends and parents who inspire me to strive towards greater heights every day; may we all keep pushing each other to become better.

A Appendix

A.1 The Python Implementation of the Solver

```python
from numpy import  reciprocal, abs, dot, sin, cos, e, pi, shape, linspace, set_printoptions, sqrt, longdouble, array_split, where, sign, argwhere, diff
import numpy as np
import os
from scipy.integrate import odeint
import matplotlib.pyplot as plt
import matplotlib.font_manager as font_manager

import timeit

"""Test Problems"""
def linearTest(t, y): # Linear Test Equation
    gamma = -1.0
    z = gamma*y
    return z

def simplePendulum(t, theta): #Simple Pendulum
    g = 9.81
    L = 1

    alphaPr1 = theta[1]
    alphaPr2 = (-g/L)*sin(theta[0])
    return array([alphaPr1, alphaPr2])

def doublePendulum(t, u): #Double pendulum
    l1, l2, m1, m2, g = 1, 1, 1, 1, 9.81

    th1, th2, mom1, mom2 = u

    h1 = (mom1*mom2*sin(th1-th2))/(l1*l2*(m1+m2*(sin(th1-th2)**2)))
    h2 = (m2*(l2**2)*(mom1**2) +(m1+m2)*(l1**2)*(mom2**2)-2*m2*l1*l2*mom1*mom2*cos(th1-th2))/(2*(l1**2)*(l2**2)*(m1+m2*(sin(th1-th2)**2))**2)

    du1 = (l2*mom1 -l1*mom2*cos(th1-th2))/((l1**2)*l2*(m1 + m2*(sin(th1-th2)**2)))

    du2 = (-m2*l2*mom1*cos(th1-th2) + (m1+m2)*l1*mom2)/(m2*l1*(l2**2)*(m1 +m2*sin(th1 -th2)**2))

    du3 = -(m1 +m2)*g*l1*sin(th1) -h1 + h2*sin(2*(th1-th2))

    du4 = -m2*g*l2*sin(th2) +h1 -h2*sin(2*(th1-th2))
    return array([du1, du2, du3, du4])

def hairerNorsett(t, u): #Hairer, N rsett, Wanner II, eq. (2.27)
    du1 = -2000 * ( cos(t) * u[0] + sin(t)* u[1] +1)
    du2 = -2000 * ( -sin(t) * u[0] + cos(t)* u[1] +1)
    return array([du1,du2])

def krogh_nonlinear(t, u, phi = pi/2): #Nonlinear problem from https://doi.org/10.48550/arXiv.2307.12677 , section 4.1
    z1 = (-u[0] + u[1] + u[2] + u[3])/2
    z2 = ( u[0] - u[1] + u[2] + u[3])/2
    z3 = ( u[0] + u[1] - u[2] + u[3])/2
    z4 = ( u[0] + u[1] + u[2] - u[3])/2

    du1 = u[0]*(5*cos(phi) - 0.375) +  u[1]*(-5*cos(phi) - 0.375) + u[2]*(0.125 - 5*sin(phi)) +  u[3]*(-5*sin(phi) - 0.125) - 0.125*z1**2 + 0.5*z1*z2 + 0.125*z3**2 + 0.5*z4**2

    du2 = u[0]*(-5*cos(phi) - 0.375) +  u[1]*(5*cos(phi) - 0.375) + u[2]*(0.125 + 5*sin(phi)) +  u[3]*( 5*sin(phi) - 0.125) + 0.125*z1**2 - 0.5*z1*z2 + 0.125*z3**2 + 0.5*z4**2

    du3 = u[0]*(5*sin(phi) + 0.125) +  u[1]*(-5*sin(phi) + 0.125) + u[2]*(-0.375 + 5*cos(phi)) +  u[3]*(5*cos(phi) + 0.375) + 0.125*z1**2 + 0.5*z1*z2 - 0.125*z3**2 + 0.5*z4**2

    du4 = u[0]*(5*sin(phi) - 0.125) +  u[1]*(-5*sin(phi) - 0.125) + u[2]*(0.375 + 5*cos(phi)) +  u[3]*(5*cos(phi) - 0.375) + 0.125*z1**2 + 0.5*z1*z2 + 0.125*z3**2 - 0.5*z4**2

    return array([du1, du2, du3, du4])

def zeros(n):
    return np.zeros(n,dtype=float)

def ones(n):
    return np.ones(n, dtype= float)
```

```python

def array(L):
    return np.array(L, dtype= float)

def errorTimePoints(benchmark_data,experimental_data): #Filters through the benchmark and experiemntal data sets and finds the time points that are the most error prone
    differences = []
    for i in range(N):
        if N > 1:
            differences.append(abs(benchmark_data[0][i]-experimental_data[i]))
        else:
            differences.append(abs(benchmark_data[i]-experimental_data[i]))

    greatest_diff = []

    for i in range(N):
        greatest_diff.append(max(differences[i]))

    difference_indexes = []
    for i in range(N):
        difference_indexes.append(where(differences[i] == greatest_diff[i])[0][0])

    tErrVals= []
    for i in range(len(difference_indexes)):
        tErrVals.append(t_grid[difference_indexes[i]])

    print(f"Time values of greatest difference for each dimension: {tErrVals}")

def lin_test_soln(t):
    return 1/(e**t)

def interpolated_intercept(x, y1, y2):
    """Find the intercept of two curves, given by the same x data"""

    def intercept(point1, point2, point3, point4):
        """find the intersection between two lines
        the first line is defined by the line between point1 and point2
        the first line is defined by the line between point3 and point4
        each point is an (x,y) tuple.

        So, for example, you can find the intersection between
        intercept((0,0), (1,1), (0,1), (1,0)) = (0.5, 0.5)

        Returns: the intercept, in (x,y) format
        """

        def line(p1, p2):
            A = (p1[1] - p2[1])
            B = (p2[0] - p1[0])
            C = (p1[0]*p2[1] - p2[0]*p1[1])
            return A, B, -C

        def intersection(L1, L2):
            D  = L1[0] * L2[1] - L1[1] * L2[0]
            Dx = L1[2] * L2[1] - L1[1] * L2[2]
            Dy = L1[0] * L2[2] - L1[2] * L2[0]

            x = Dx / D
            y = Dy / D
            return x,y

        L1 = line([point1[0],point1[1]], [point2[0],point2[1]])
        L2 = line([point3[0],point3[1]], [point4[0],point4[1]])

        R = intersection(L1, L2)

        return R

    idx = np.argwhere(np.diff(np.sign(y1 - y2)) != 0)
    xc, yc = intercept((x[idx], y1[idx]),((x[idx+1], y1[idx+1])), ((x[idx], y2[idx])), ((x[idx+1], y2[idx+1])))
    return xc,yc

class FSALRungeKutta:
    def __init__(self, f, N, A, b, bembd, c, beta, accept_safety =0.81, *args, **kwargs):
        self.f = f
        self.N = N #degrees of freedom
        self.A = A
```

```
        self.b = b
        self.bembd = bembd
        self.c = c
        self.beta = beta
        self.accept_safety = accept_safety
        self.k = [zeros(self.N)]*len(self.c) #zeros((len(self.c),N))

    def computeError(self, y, y_embd, y_prev, abstol, reltol):
        err = 0.
        err_n = 0
        for i in range(self.N):
            tol = abstol + reltol * max(abs(y[i]), abs(y_prev[i]))
            if tol > 0:
                err += (abs(y[i] - y_embd[i]) / tol)**2
                err_n += 1
        return sqrt(err / err_n)

    def fsal_step(self,f,t,y,h):

        for i in range(1, len(self.A[0])):
            argument = zeros(self.N)
            for j in range(len(self.A[0])):
                argument +=  self.A[i][j] * self.k[j]
            self.k[i] = f(t + self.c[i] * h, y + h * argument)

        y3_temp = zeros(self.N)
        for i in range(len(self.b)):
            y3_temp += self.b[i]*self.k[i]
        y_third = y + y3_temp*h #Third Order, main

        y2_temp = zeros(self.N)
        for i in range(len(self.bembd)):
            y2_temp += self.bembd[i]*self.k[i]
        y_second = y + y2_temp*h #Second Order, Embedded

        self.k[0] = self.k[-1] #This is the assignment part of the FSAL algorithm the new first k[0] (which will be used in calculations) is assigned here, check line 64
        return y_third, y_second

    def RK_step(self,f,t,y,h): #Single step of a basic non embedded RK method

        for i in range(0, len(self.A[0])):
            argument = zeros(self.N)
            for j in range(len(self.A[0])):
                argument +=  self.A[i][j] * self.k[j]
            self.k[i] = f(t + self.c[i] * h, y + h * argument)

        y3_temp = zeros(self.N)
        for i in range(len(self.b)):
            y3_temp += self.b[i]*self.k[i]
        y_third = y + y3_temp*h #Third Order, main

        return y_third

    def ODE_problem(self, f, y0, t0, tf):
        return (f, y0, t0, tf)

    def RK_solve(self, f , y0, t0, tf, h):
        t = t0
        y = zeros(self.N)

        y[0] = self.RK_step(f,t, y0, h)

        t += h
        while t < tf:
            if t + h > tf:
               h = tf - t

            y = self.RK_step(f,t,y,h) #solution
            t += h

            if t == tf:
                break
        # print(t)
        return y

    def fsal_solve(self,f, y0, t0, tf, h_init, abstol, reltol, adaptive = True):
        t = t0
```

```
220         y = [zeros(self.N)]*3
221         y[0]=y0
222         h = h_init
223         err = zeros(3)
224         n_accept = 2
225         n_reject = 0
226         time_axis = [t0]
227 
228         solution_axis = [None]*N #The following 3 lines are only here for graphing purposes, they should be commented out otherwise along with line 272-273
229         for i in range(N):
230             solution_axis[i] = [y[0][i]]
231 
232         self.k[0] = f(t,y0) #The first stage derivative is only calculated once here. Afterwards the fsal_step manually assigns k[0] = k[-1]
233         y[2][:] = y0[:] # setting uprev with the initial condition
234         y[0], y[1] = self.fsal_step(f,t,y0,h_init) #y third order (solution) and (embedded) second order respectively
235 
236         t = t0 + h # First two steps are of the same size as the h_init
237         y_sol = y[0]
238         err[1] = err[0] = self.computeError(y[0], y[1], y[2], abstol, reltol)
239         time_factor = err[0]**(self.beta[0]/3) * err[1]**(self.beta[1]/3) * err[2]**(self.beta[2]/3)
240 
241         h = h_init
242         n = 1
243 
244         while t < tf:
245             if t + h > tf:
246                 h = tf - t  # Adjusts step size to reach the final time exactly
247 
248             n = n+1
249 
250             y[2][:] = y[0][:]
251             y[0], y[1] = self.fsal_step(f,t,y[0],h)
252 
253 
254             controller_error = self.computeError(y[0] ,y[1] , y[2], abstol, reltol)
255             err[0] = reciprocal(controller_error )
256 
257 
258             time_factor = err[0]**(self.beta[0]/(len(self.b)-1)) * err[1]**(self.beta[1]/(len(self.b)-1)) * err[2]**(self.beta[2]/(len(self.b)-1))
259 
260             if time_factor >= self.accept_safety: # accept safety = 0.81
261                 err[2] = err[1]
262                 err[1] = err[0]
263                 t += h
264                 y_sol = y[0]
265 
266                 if adaptive: #Condition to allow for adaptive and non adaptive implementation
267                     h *= time_factor
268 
269                 n_accept += 1
270                 time_axis.append(t)
271 
272                 for i in range(N): #Only here for grpahing, comment out if graphing is not desired.
273                     solution_axis[i].append(y[0][i])
274 
275                 # print(f"The process is {round(t/tf*100, 2)}% complete") #Loading bar, useful for showing if the loop is stuck or just slow; should be commented out for performance
276             else:
277                 if adaptive:
278                     h *= 0.25
279                     n_reject += 1
280                     y[0] = y_sol
281 
282             if t == tf:
283                 # os.system('cls' if os.name == 'nt' else 'clear') #Clears the loading bar, uncomment only if loading bar is uncommented.
284                 break
285 
286         return t, y_sol, n_accept, n_reject, time_axis, solution_axis
287 
288 
```

```
289 class BS32(FSALRungeKutta): #3rd order Bogacki-Shampine with 2nd order embedded method and 4 steps
290     def __init__(self,f,N,beta,*args,**kwargs):
291         A = [[ 0, 0, 0, 0],
292              [1/2, 0, 0, 0],
293              [ 0, 3/4, 0, 0],
294              [2/9, 1/3, 4/9, 0]]
295         b = [2/9, 1/3, 4/9, 0]
296         bembd = [7/24, 1/4, 1/3, 1/8]
297         c = [0, 1/2, 3/4, 1]
298         FSALRungeKutta.__init__(self, f, N, A, b, bembd, c, beta, accept_safety =0.81, *args, **kwargs) #Accept Safety highly dependant on problem and controller parameters
299
300 class RKDP(FSALRungeKutta): #5th order Dormand-Prince with 4th order embedded method
301     def __init__(self,f,N,beta,*args,**kwargs):
302         A = [[ 0, 0, 0, 0, 0, 0, 0],
303              [ 1/5, 0, 0, 0, 0, 0, 0],
304              [ 3/40, 9/40, 0, 0, 0, 0, 0],
305              [ 44/45, -56/15, 32/9, 0, 0, 0, 0],
306              [ 19372/6561, -25360/2187, 64448/6561, -212/729, 0, 0, 0],
307              [ 9017/3168, -355/33, 46732/5247, 49/176, -5103/18656, 0, 0],
308              [ 35/384, 0, 500/1113, 125/192, -2187/6784, 11/84, 0]]
309         b = [ 35/384, 0, 500/1113, 125/192, -2187/6784, 11/84, 0]
310         bembd = [ 5179/57600, 0, 7571/16695, 393/640, -92097/339200, 187/2100, 1/40]
311         c = [0, 1/5, 3/10, 4/5, 8/9, 1, 1]
312         FSALRungeKutta.__init__(self, f, N, A, b, bembd, c, beta, accept_safety =0.62, *args, **kwargs)
313
314 class BS54(FSALRungeKutta): #5th order Bogacki-Shampine method with 4th order embedded method,
315     def __init__(self,f,N,beta,*args,**kwargs):
316         A = [[ 0, 0, 0, 0, 0, 0, 0, 0],
317              [ 1/6, 0, 0, 0, 0, 0, 0, 0],
318              [ 2/27, 4/27, 0, 0, 0, 0, 0, 0],
319              [ 183/1372, -162/343, 1053/1372, 0, 0, 0, 0, 0],
320              [ 68/297, -4/11, 42/143, 1960/3861, 0, 0, 0, 0],
321              [ 597/22528, 81/352, 63099/585728, 58653/366080, 4617/20480, 0, 0, 0],
322              [ 174197/959244, -30942/79937, 8152137/19744439, 666106/1039181, -29421/29068, 482048/414219, 0, 0],
323              [ 587/8064, 0, 4440339/15491840, 24353/124800, 387/44800, 2152/5895, 7267/94080, 0]]
324         b = [ 587/8064, 0, 4440339/15491840, 24353/124800, 387/44800, 2152/5985, 7267/94080, 0]
325         bembd = [ 2479/34992, 0, 123/416, 612941/3411720, 43/1440, 2272/6561, 79937/1113912, 3293/556956]
326         c = [0, 1/6, 2/9, 3/7, 2/3, 3/4, 1, 1]
327         FSALRungeKutta.__init__(self, f, N, A, b, bembd, c, beta, accept_safety =0.81, *args, **kwargs)
328
329 class Calvo65(FSALRungeKutta): #6th order Calvo method with 5th order embedded method
330     def __init__(self,f,N,beta,*args,**kwargs):
331         A = [[ 0, 0, 0, 0, 0, 0, 0, 0, 0],
332              [ 0.133, 0, 0, 0, 0, 0, 0, 0, 0],
333              [ 0.050, 0.150, 0, 0, 0, 0, 0, 0, 0],
334              [ 0.075, 0, 0.225, 0, 0, 0, 0, 0, 0],
335              [ 0.441, -1.143, 0.695,  0.567, 0, 0, 0, 0, 0],
336              [ -1.904, 4.317, 1.068, -3.937, 1.216, 0, 0, 0, 0],
337              [ 4.831, -7.633, -7.401, 13.165, -2.682, 0.707, 0, 0, 0],
338              [ 6.048, -9.414, -9.717, 16.746, -3.511, 0.867, -0.018, 0, 0],
339              [ 0.061,  0, 0.285, 0.044, 0.305, 0.164, 0.516, -0.376, 0]]
340         b = [ 0.061,  0, 0.285, 0.044, 0.305, 0.164, 0.516, -0.376, 0]
341         bembd = [0.0298, 0, 0.512, -0.232, 0.414, 0.159, 0.189, -0.122, 0.050]
342         c = [0, 0.133, 0.200, 0.300, 0.560, 0.760, 0.987, 1. , 1.]
343         FSALRungeKutta.__init__(self, f, N, A, b, bembd, c, beta, accept_safety =0.71, *args, **kwargs)
344
345 class RK4(FSALRungeKutta): #4th prder Runge-Kutta Method, not FSAL
346     def __init__(self,f,N,beta,*args,**kwargs):
347         A = [[ 0, 0, 0, 0],
348              [1/2, 0, 0, 0],
349              [ 0, 1/2, 0, 0],
350              [ 0, 0,   1, 0]]
351         b = [1/6, 1/3, 1/3, 1/6]
352         bembd = [0, 0, 0 , 0]
353         c = [0, 1/2, 1/2, 1]
354         FSALRungeKutta.__init__(self, f, N, A, b, bembd, c, beta, accept_safety =0.71, *args, **kwargs)
355
356 class ExpEuler(FSALRungeKutta): #Explicit Euler method
```

```
357     def __init__(self,f,N,beta,*args,**kwargs):
358         A = [[0.]]
359         b = [1.]
360         bembd = [0]
361         c = [0.]
362         FSALRungeKutta.__init__(self, f, N, A, b, bembd, c, beta, accept_safety =0.71, *args,
    **kwargs)
363 
364 """Dimension & Scheme selection"""
365 #Dimension of Problem, N:
366 N = 2
367 
368 rkFSAL = BS32(None, N, [0.60,-0.20,0.00] )  #Works good
369 # rkFSAL = BS32(None, N, [0.28,-0.23,0.00] )
370 
371 # rkFSAL = RKDP(None, N, [0.70,-0.40,0.00] )  #Strange behaviour w.r.t. optimal contorller
    parameters not being optimal
372 # rkFSAL = RKDP(None, N, [0.6,-0.20,0.00] )
373 
374 # rkFSAL = BS54(None, N, [0.28,-0.23,0.00])
375 # rkFSAL = BS54(None, N, [0.6,-0.20,0.00])
376 
377 # rkFSAL = Calvo65(None, N, [0.60,-0.20,0.00])  #Not as good as BS54, even for Hairer N rsett
     problem, possible errors in Butcher Tableu
378 
379 # rkSTD = RK4(None, N, [0.60,-0.20,0.00] )
380 # rkSTD = ExpEuler(None, N, [0.60,-0.20,0.00] )
381 
382 """ Choose the problem: (function, initial condition, initial time, final time) """
383 problem = rkFSAL.ODE_problem(linearTest    , ones(N),               0, 1    )   #N = 1
384 
385 # problem = rkFSAL.ODE_problem(simplePendulum, array([pi/2,0]) ,     0, 2    )    #N = 2
386 
387 # problem = rkFSAL.ODE_problem(doublePendulum, array([pi/2,0,0,0]) , 0, 2.5    )    #N = 4
388 
389 problem = rkFSAL.ODE_problem(hairerNorsett , array([1,0]),         0, 1.57 )    #N = 2
390 
391 # problem = rkFSAL.ODE_problem(krogh_nonlinear , array([0,-2,-1,-1]),  0, 1)      #N = 4
392 
393 """Solver Parameters"""
394 h_init = 1e-3 # Minimum step size depends on the problem and method
395 abstol = reltol = 1e-4  #tolerances, set equal in Julia code
396 
397 
398 """ Solving the Problem """
399 
400 t, y, n_accept, n_reject, t_grid, SOL = rkFSAL.fsal_solve(*problem, h_init, abstol, reltol)
401 
402 
403 y_expected = odeint(problem[0], problem[1], t_grid , tfirst=True )
404 
405 y_lit = y_expected[-1]
406 
407 """ Printing the results """
408 set_printoptions(precision=9) #Variable mantissa for desired numbre of sig. figs. when printed
409 
410 
411 # y_lit = 1/(e) #literature value for linear test equation
412 
413 
414 print(f"y = {y} at t = {t}\nTotal step = {n_accept+n_reject}\nAccepted = {n_accept}\nRejected
    = {n_reject}\nError = {abs(y-y_lit)}\nEuclidean norm of error: {sqrt(dot(y-y_lit,y-y_lit )
    )}")
415 
416 # errorTimePoints(y_expected,SOL) #Shows time values where SciPy and the controller are the
    furthest apart
417 
418 # runTime = min(timeit.repeat(stmt=’rkFSAL.fsal_solve(*problem, h_init, abstol, reltol) ’,
    globals=globals(), number = 1, repeat=1000))
419 # print(f"Runtime = {runTime} ")
420 
421 """
422 Note on runtime:
423 
424 The fastest run case for code is the case where there is the least noise, the magnitude of
    which is randomly determined by the background processes running on the computer.
425 Min is used because of external noise from other software being the primary culprit for the
    noise and not python itself.
```

```
426 Mean and standard deviation only give you a measure of how background noise is affecting the thread's computation time on average, it does not tell you how fast the code itself is.
427 """
428
429 """Error for comparing with Julia"""
430 # julia_y = array([ -0.6385901210310782,0.2504868017901877]) #the 2 numbers that came out of sol_diffeq.u[end] in the Julia code.
431 # print(f"Additional error for the julia code: {sqrt(dot(julia_y -y_expected[-1],julia_y - y_expected[-1] ))}")
432
433
434 """Plotting of problem, modify w.r.t. the case"""
435 analytic_vals = zeros(len(t_grid))
436
437 for i in range(len(t_grid)): #This gives the y values for an analytic solution, given that we only haVe the linear test equation, it has not been extended beyond one dimension.
438     analytic_vals[i] = lin_test_soln(t_grid[i])
439
440
441 if N >= 1: #This is for the scipy solution. litvals will give you a matrix of column vectors with each solution for each dimension of the ODE
442     litvals = array_split(y_expected, N, axis = 1)
443 else:
444     litvals = y_expected[0]
445
446 plt.figure(figsize = (16,6))
447
448 # plt.plot(t_grid, analytic_vals, label = "Analytic solution") #Only for the linear test equation
449
450 for i in range(N):
451     plt.plot(t_grid, SOL[i], label = rf"Controller: $du_{i+1}$")
452
453 for i in range(N):
454     plt.plot(t_grid, litvals[i], label = rf"SciPy: $du_{i+1}$")
455
456 colours = ['magenta', 'cyan', 'crimson', 'green'] #Used for interpolation markers
457 for i in range(N):
458     xc, yc = interpolated_intercept(array(t_grid) , array(SOL[i]).flatten() , array(litvals[i]).flatten())
459     plt.scatter(xc, yc, marker= 'D', color = colours[i] , s=10, label=rf'Intersects of $du_{i+1}$')
460
461 font = font_manager.FontProperties(family='Times New Roman',style='normal', size=12)
462
463 plt.title('Stiff ODE: Hairer-N rsett-Wanner II Problem', fontname='Times New Roman', size = 22)
464 plt.grid()
465 plt.xlabel('t', fontname='Times New Roman', size = 16)
466 plt.ylabel('u(t)', fontname='Times New Roman', size = 16)
467 plt.legend(loc = 'center left', bbox_to_anchor =(0.95,0.5),prop = font)
468 plt.show()
```

References

[1] https://docs.julialang.org/en/v1/manual/integers-and-floating-point-numbers/#, Accessed Dec 2023.

[2] https://petsc4py.readthedocs.io/en/stable/manual/ts/, Accessed Mar 2024.

[3] https://sundials.readthedocs.io/en/latest/sunadaptcontroller/SUNAdaptController_links.html, Accessed Mar 2024.

[4] https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.odeint.html, Accessed Feb 2024.

[5] Peter Bastian, Markus Blatt, Andreas Dedner, Nils-Arne Dreier, Christian Engwer, René Fritze, Carsten Gräser, Christoph Grüninger, Dominic Kempf, Robert Klöfkorn, Mario Ohlberger, and Oliver Sander. The Dune framework: Basic concepts and recent developments. *Computers Mathematics with Applications*, 81:75–112, 2021. Development and Application of Open-source Software for Problems with Numerical PDEs.

[6] P. Bogacki and L.F. Shampine. A 3(2) Pair of Runge - Kutta formulas. *Applied Mathematics Letters*, 2(4):321–325, 1989.

[7] P. Bogacki and L.F. Shampine. An efficient Runge-Kutta (4,5) pair. *Computers Mathematics with Applications*, 32(6):15–28, 1996.

[8] John C. Butcher. *Differential equations, numerical methods and algebraic analysis*, page 24–24. Springer, 2021.

[9] R. Courant, K. Friedrichs, and H. Lewy. Über Die Partiellen Differenzengleichungen der mathematischen Physik. *Mathematische Annalen*, 100(1):32–74, 1928.

[10] Germund G. Dahlquist. A special stability problem for linear multistep methods. *BIT Numerical Mathematics*, 3(1):27–43, Mar 1963.

[11] J.R. Dormand and P.J. Prince. A family of embedded Runge-Kutta formulae. *Journal of Computational and Applied Mathematics*, 6(1):19–26, 1980.

[12] Dale E. Durran. *Strong-Stability Preserving Methods*, page 56–56. Springer Verlag, second edition, 2010.

[13] Sigal Gottlieb and Lee-Ad J. Gottlieb. Strong Stability Preserving Properties of Runge–Kutta Time Discretization Methods for Linear Constant Coefficient Operators. *Journal of Scientific Computing*, 18(1):83–109, Feb 2003.

[14] Ernst Hairer and Gerhard Wanner. *Solving ordinary differential equations II stiff and differential-algebraic problems.* Springer Berlin Heidelberg, 2010.

[15] D.J. Higham and G. Hall. Embedded Runge-Kutta formulae with stable equilibrium states. *Journal of Computational and Applied Mathematics*, 29(1):25–33, 1990.

[16] HilberTraum. File:Runge-Kutta slopes.svg. https://commons.wikimedia.org/wiki/File:Runge-Kutta_slopes.svg, Nov 2017, Accessed Oct 2023.

[17] William Hughes. Beyond PID: 6 advanced strategies to add value to modern process control. *Control Engineering*, Aug 2023.

[18] Arieh Iserles. *A First Course in the Numerical Analysis of Differential Equations.* Cambridge Texts in Applied Mathematics. Cambridge University Press, 2 edition, 2008.

[19] David I. Ketcheson. Highly Efficient Strong Stability-Preserving Runge–Kutta Methods with Low-Storage Implementations. *SIAM Journal on Scientific Computing*, 30(4):2113–2136, 2008.

[20] David I. Ketcheson. Runge-Kutta Methods with Minimum-Storage Implementations. *Journal of Computational Physics*, 229(5):1763–1773, 2010.

[21] Fred T. Krogh. On Testing a Subroutine for the Numerical Integration of Ordinary Differential Equations. *J. ACM*, 20(4):545–562, oct 1973.

[22] Araki M. and Heinz D. Unbehauen. *CONTROL SYSTEMS, ROBOTICS AND AUTOMATION - Volume II: System Analysis and Control: Classical Approaches-II*, volume 2, page 58–60. EOLSS Publications, 2009.

[23] James C. Maxwell. On Governors. *Proceedings of the Royal Society of London*, 16:270–283, 1868.

[24] Hendrik Ranocha, Lisandro Dalcin, Matteo Parsani, and David I. Ketcheson. Optimized Runge-Kutta Methods with Automatic Step Size Control for Compressible Computational Fluid Dynamics. *Communications on Applied Mathematics and Computation*, 4(4):1191–1228, nov 2021.

[25] Hendrik Ranocha and Jan Giesselmann. Stability of step size control based on a posteriori error estimates. `https://arxiv.org/abs/2307.12677`, 07 2023.

[26] Hendrik Ranocha and Jan Giesselmann. Reproducibility repository for "Stability of step size control based on a posteriori error estimates". `https://github.com/ranocha/2023_RK_error_estimate`, Accessed Mar 2023.

[27] Endre Suli and D. F. Mayers. *Initial Value Problems For ODEs*, page 317–317. Cambridge University Press, 2003.

[28] Gustaf Söderlind. Automatic Control and Adaptive Time-Stepping. *Numerical Algorithms*, 31(1/4):281–310, Dec 2002.

[29] Hakan Tiftikci. Stability regions of Runge-Kutta solvers. `https://www.mapleprimes.com/posts/100157-Stability-Regions-Of-RungeKutta-Solvers`, Dec 2010, Accessed Oct 2023.

[30] Arturo Urquizo. File:PID.svg. `https://commons.wikimedia.org/wiki/File:PID.svg`, Wikimedia Commons, Oct 2008, Accessed Oct 2023.

Bachelor’s Theses in Mathematical Sciences 2023:K31
ISSN 1654-6229

LUNFNA-4054-2023

Numerical Analysis
Centre for Mathematical Sciences
Lund University
Box 118, SE-221 00 Lund, Sweden

http://www.maths.lu.se/