Python Tools for the Investigation of Optimal Explicit Runge-Kutta Methods

By

Shivam Singla

September 09, 2021, Halifax, Nova Scotia

A Thesis Submitted to Saint Mary's University, Halifax, Nova Scotia in Partial Fulfilment of the Requirements for the Degree of Bachelor of Science, Honours in Computing Science and Major in Mathematics

Approved: Dr. Paul Muir

Supervisor

Approved: Dr. Wendy Finbow-Singh

Reader

Date: September 09, 2021

Abstract

Python Tools for the Investigation of Optimal Explicit Runge-Kutta Methods

By

Shivam Singla

The investigation of many real-world applications involves mathematical models that consist of systems of ordinary differential equations (ODEs). This thesis mainly focuses on the ODEs with the initial values known as initial value ordinary differential equations. These equations are typically solved using numerical methods to obtain approximate solutions. A popular class of numerical methods to solve an initial value ODE are the Explicit Runge-Kutta (ERK) methods. This thesis considers Python software for the investigation of ERK methods. ERK methods can be used to obtain approximate solutions at a discrete set of points across the domain of interest, with h being the distance between the points. An ERK method is said to be of order p if the error of the numerical solution is proportional to h^p. In this thesis, we consider Python software for the determination of optimal ERK methods of orders 1 to 4. The Python software also has the capability to solve a set of test problems using various ERK methods in order to allow for a comparison of the accuracy of the numerical solutions obtained from the ERK methods. The Python software can also be used to extend the discrete approximate solutions from the ERK methods to obtain continuous approximate solutions over the entire domain using Hermite interpolation. To assess the accuracy of the continuous solution approximation, the software can also be used to compute the defect of the continuous approximate solution, where defect is the amount by which the continuous approximate solution fails to satisfy the ODE.

Date: September 09, 2021

Contents

Chapter 1

Introduction

Initial Value Ordinary Differential Equations (ODEs) arise within mathematical models in a wide variety of applications such as the Predator-Prey problem, COVID-19 models, population growth and decay problems, survivability with AIDS problems, economics and finance problems, etc. Typically, these initial value ODEs are too complicated to be solved by hand. Approximate numerical solutions must be computed. There are a wide variety of numerical methods for solving initial value ODEs, but in this thesis, we are going to focus on one of the most popular classes of methods called '*Explicit Runge-Kutta methods'*.

In this thesis, we survey specific examples of Explicit Runge-Kutta methods that have been developed over the years. We also show the general forms of Explicit Runge-Kutta methods of orders $2, 3$ and 4. These general forms have free coefficients. One important part of this thesis involves determining optimal values for these free coefficients so that the methods are as accurate as possible. We have developed Python software for determining optimal values for the free coefficients based on minimization of the Principal Error Coefficient (to be defined in Chapter 2) of the Runge-Kutta method.

We also compare some standard Explicit Runge-Kutta methods with the Runge-Kutta methods that have optimal values for the free coefficients. The comparison involves applying these methods to solve various initial value ODEs. We use a Python tool we have developed for applying Explicit Runge-Kutta methods to selected test initial value ODEs. The tool computes the error at the end of the time domain for the problems that have a known exact solution along with the stepsize used and the order of convergence (to be defined in Chapter 2). For

the problems with an unknown exact solution, the Python tool provides the numerical solution approximation at the end of the time domain along with the stepsize used.

The Explicit Runge-Kutta methods give discrete solution approximations at certain points across the domain. Based on the use of Hermite interpolants, we can extend the discrete solution to produce a continuous solution approximation across the whole domain.

However, the quality of this continuous numerical solution must be assessed. Once we have computed the continuous solution approximation, we plug it into the initial value ODE to check how large the defect is. The defect is the amount by which the continuous approximate solution fails to satisfy the differential equation. The Python tool also computes the defect for the continuous approximate solution. Then, it provides a plot of the continuous approximate solution and a plot of the defect, which gives a measure of the accuracy of the continuous approximate solution.

In this thesis, we investigate an important question regarding choosing the free coefficients of a general ERK method to minimize the Principal Error Coefficient of the method. The actual error of a computed numerical solution depends on the linear combination of products of the components of the Principal Error Coefficient and higher derivatives of the right hand side of the ODE. On the other hand, the Principal Error Coefficient has components that depend only on the ERK method. This means that there is a difference between choosing the free coefficients of an ERK method to minimize the Principal Error Coefficient and minimizing the actual error of a numerical solution computed by the ERK method. Therefore, we expect that the optimal ERK methods obtained by minimizing only the Principal Error Coefficient may not deliver the smallest error, depending on the problem, especially for cases where the standard methods are nearly optimal. Based on numerical testing on a set of test initial value ODEs, we investigate how often the optimal ERK methods are actually able to deliver the smallest error.

As mentioned above, a standard use of an ERK method provides a set of discrete solution approximations at points across the problem domain. A standard ERK solver would employ an algorithm for estimating the error of the discrete solution approximations and adjusting the ERK methods so that the error for each discrete solution approximation is less than a user-provided tolerance. Another important aspect of this thesis is to introduce a simple method for extending the discrete solution to contain a continuous approximate solution for which the accuracy can be assessed by examining its corresponding defect. As the continuous numerical solution is the solution that is returned to the user by current ODE solvers, it is important to assess the accuracy of the continuous approximate solution rather than only the discrete approximate solution.

The thesis is organized as follows:

1. Chapter 2 Background: This chapter includes an explanation of the Initial Value ODEs, general and standard forms of ERK methods, optimal ERK methods along with order conditions and Principal Error Coefficients, continuous approximate solutions, Hermite interpolants and the computation of the defect.
2. Chapter 3 Software Implementation: This chapter includes the documentation and description of the Python software created for optimization of the ERK methods. It also includes the Python software for testing ERK methods and obtaining a continuous approximate solution and its defects.
3. Chapter 4 Results and Discussion: This chapter includes the results of the optimization software and the discussion on the optimal ERK methods and their comparison with standard methods. It also includes the experimental confirmation of the order of convergence of the methods along with the comparison between the standard ERK methods and optimal ERK methods when used on the test initial value ODEs. Finally,

this chapter includes the results and discussions on the continuous approximate solutions and their corresponding defects.

4. Chapter 5 Summary, Conclusions, and Future Work: This chapter includes the summary and conclusions of this thesis along with some suggestions for future work.
5. Appendix: The appendix includes the python scripts which build the optimization software as well as the software for testing the ERK methods and computing continuous approximate solution and their defects.

Chapter 2

Background

In this chapter, we first introduce Initial Value Ordinary Differential Equations (IVODEs) with examples as well as numerical methods for solving those IVODEs. We focus on Explicit Runge-Kutta (ERK) methods [Butc87]. We present the general form for ERK methods and provide some examples. Then, we discuss order conditions [Butc87] and Principal Error Coefficients [Butc87] for ERK methods and explain how these can be used to obtain optimal ERK methods of a given order. This chapter also includes an overview of continuous extensions [Butc87] of discrete solutions from ERK methods. Finally, we discuss assessment of the quality of the continuous solution approximation based on the computation of the defect [Enri89] of the continuous numerical solution.

2.1 Initial Value Ordinary Differential Equations

An IVODE, also known as Initial Value Problem (IVP), is an ordinary differential equation (ODE) together with an initial condition. The initial condition specifies the initial value of the solution to the ODE at a specific point in the domain. In this section, we show the general form for an IVODE and give some examples that we use later in the thesis as test problems.

2.1.1 General Form

The general form of an IVODE is

$$y: \mathbb{R} \to \mathbb{R}^m \qquad \text{and} \qquad y'(t) = f\big(t, y(t)\big),$$

in which $f: \mathbb{R} \times \mathbb{R}^m \to \mathbb{R}^m$. It also has an initial condition which specifies

$$y(t_0) = y_0$$

where t_0 is a given point and $y_0 \in \mathbb{R}^m$ is a given constant vector.

For $m > 1$, $y(t)$ is the vector $\big(y_1(t), y_2(t), \dots, y_m(t)\big)^T$ and the differential equation is replaced by a system of equations,

$$y_i'(t) = f_i\big(t, y_1(t), y_2(t), \dots, y_m(t)\big), \qquad i = 1, 2, \dots, m.$$

2.1.2 Example 1

First, we start with an example of an IVODE [SAP97],

$$y' = -2xy^2,$$

with initial value, $y(0) = 1$ and exact solution,

$$y(x) = \frac{1}{1 + x^2}.$$

The final t value for this IVODE that is used in this thesis is 1.

2.1.3 Example 2

This is another example of an IVODE [SAP97],

$$y' = \left(-\frac{1}{2}\right) y^3,$$

with initial value, $y(0) = 1$ and exact solution,

$$y(x) = \frac{1}{\sqrt{1 + x}}.$$

The final t value for this IVODE that is used in this thesis is 1.

2.1.4 Example 3

The next example of an IVODE is [SAP97],

$$y' = \left(\frac{1}{4}\right)\left(1 - \frac{y}{20}\right) y,$$

with initial value, $y(0) = 1$ and exact solution,

$$y(x) = \frac{20}{1 + 19e^{-\frac{x}{4}}}.$$

The final t value for this IVODE that is used in this thesis is 1.

2.1.5 Example 4

Another example of an IVODE is [SAP97],

$$y' = -\alpha y - e^{-\alpha x} \sin x,$$

where α is a constant between 0 and 1. We choose $\alpha = 0.1$ for our computations in this thesis. The initial value for this IVODE is $y(0) = 1$ and the exact solution is,

$$y(x) = e^{-\alpha x} \cos x.$$

The final t value for this IVODE that is used in this thesis is 1.

2.1.6 COVID-19 Model Example

This example for an IVODE is a standard COVID-19 model, an example of a SEIR epidemiological model, involving the following system of equations [Chri20],

$$y_1' = -\frac{\beta y_1 y_3}{N} + \mu N - \mu y_1,$$

$$y_2' = \frac{\beta y_1 y_3}{N} - (\alpha + \mu) y_2,$$

$$y_3' = \alpha y_2 - (\gamma + \mu) y_3,$$

$$y_4' = \gamma y_3 - \mu y_4,$$

where

$$y_1 = S\text{: Susceptible},$$

$$y_2 = E\text{: Exposed},$$

$$y_3 = I\text{: Infected},$$

$$y_4 = R\text{: Recovered},$$

and the constants are,

$$\alpha = \frac{1}{8}, \qquad \beta = 0.9, \qquad \gamma = 0.06, \qquad \mu = \frac{0.01}{365}, \qquad N = 37.741 \times 10^6.$$

The initial conditions for this IVODE are $y(0) = [N - y_2 - y_3;\ y_2;\ y_3;\ 0]$, where $y_2 = 103$ and $y_3 = 1$.

The final time that is used in this thesis is $t_f = 150$.

2.2 Numerical Methods for Solving IVODEs: Explicit Runge-Kutta Methods

To find the approximate solution of IVODEs, we use a popular family of methods known as the *'Explicit Runge-Kutta (ERK)* methods'. These methods are a generalization of the *'Euler method'*, also known as the *'Forward Euler method*'. A member of the *Runge-Kutta* family which is the most widely known method of this type, is the '*RK4 method'*, which is also known as the '*classical Runge-Kutta method*' or just '*the Runge-Kutta method'*. The order of convergence for the '*RK4 method*' is 4. The global error of an ERK method is the difference between the numerical solution it computes and the exact solution to an IVODE. The global error is proportional to some power of h, where h is the stepsize used by the ERK method. When the global error is proportional to h^p, we say that the global error is $O(h^p)$, and the ERK method is said to be of order p.

2.2.1 General Form

An ERK method can be used to obtain discrete numerical solution approximations, $y_n \approx y(t_n)$, at a set of points, $t_n = t_0 + nh$, where h is the stepsize used by the ERK method. (h is the distance between the t_n values.)

The general form for an s stage ERK method [Butc87] is,

$$y_{n+1} = y_n + h(b_1k_1 + b_2k_2 + \cdots + b_sk_s),$$

for $n = 0, 1, 2, \ldots$, where

$$k_1 = f(t_n, y_n),$$

$$k_2 = f\big(t_n + c_2h, y_n + h(a_{21}k_1)\big),$$

$$k_3 = f\big(t_n + c_3h, y_n + h(a_{31}k_1 + a_{32}k_2)\big),$$

…

$$k_s = f\Big(t_n + c_sh, y_n + h\big(a_{s1}k_1 + a_{s2}k_2 + \cdots + a_{s,s-1}k_{s-1}\big)\Big).$$

The vectors $k_1, k_2, \ldots, k_s$, are called the stages of the ERK method. To obtain a specific ERK method, one must specify the integer s (the number of stages), and the coefficients a_{ij} $(\text{for } 1 \leq j < i \leq s)$, b_i $(\text{for } i = 1, 2, \ldots, s)$ and c_i $(\text{for } i = 2, 3, \ldots, s)$. These coefficients are determined by requiring them to satisfy a set of equations known as Runge-Kutta order conditions. We discuss these conditions later in this chapter.

The matrix A with the elements a_{ij} is called the *Runge-Kutta matrix*, with b_i and c_i known as the *weights* and the *nodes* respectively.

The coefficients that define an ERK method are usually stored in a table, known as a *Butcher tableau* (named after John Butcher):

0					
c_2	a_{21}				
c_3	a_{31}	a_{32}			
$\vdots$	$\vdots$		$\ddots$		
c_s	a_{s1}	a_{s2}	…	$a_{s,s-1}$	
	b_1	b_2	…	b_{s-1}	b_s

where (usually)

$$\sum_{j=1}^{i-1} a_{ij} = c_i \text{ for } i = 2, 3, \dots, s.$$

2.2.2 The First Order ERK Method: The Forward Euler Method

The most basic explicit method for numerical integration of ordinary differential equations is the *'Forward Euler (FE) Method'*. It is the simplest example of a *Runge-Kutta* method.

For the general IVODE defined above, i.e.,

$$y'(t) = f\big(t, y(t)\big), \qquad y(t_0) = y_0,$$

and for a given solution approximation, y_n, at t_n, with a stepsize h, the FE method has the form

$$y_{n+1} = y_n + h\, f(t_n, y_n) \text{ for } n = 0, 1, 2, \dots .$$

This method is simple to use and works reasonably well. It is a first order method which means that if the stepsize h is reduced by a factor of 2, then the error will also be reduced by a factor of 2. The FE method has a global error that is $O(h)$, which means that the order for FE method is 1. The global error for y_n is $e_n = |y_n - y(t_n)|$.

2.2.3 Second Order ERK Methods

A second order ERK method is a method which has order 2 and provides twice the accuracy of the FE method. This means that if the stepsize h is reduced by a factor of 2, then the error will be reduced by a factor of 4. The global error for these methods is $O(h^2)$.

2.2.3.1 General Form

The general form for a two-stage, second order ERK method is

$$y_{n+1} = y_n + h\,(b_1 k_1 + b_2 k_2),$$

for $n = 0, 1, 2, \ldots$, where

$$k_1 = f(t_n, y_n), \qquad \text{and}$$

$$k_2 = f\big(t_n + c_2\, h,\ y_n + h(a_{21}\, k_1)\big).$$

This method is called a two-stage method because it involves two stage evaluations. The first stage is

$$k_1 = f(t_n, y_n)$$

and the second is

$$k_2 = f\big(t_n + c_2\, h,\ y_n + h(a_{21}\, k_1)\big).$$

The *Butcher tableau* for a two-stage, second order ERK method is:

0	0	0
c_2	a_{21}	0
	b_1	b_2

Here, the matrix A is

$$A = \begin{bmatrix} 0 & 0 \\ c_2 & 0 \end{bmatrix},$$

the weights are

$$b = \begin{bmatrix} 1 - \dfrac{1}{2c_2} & \dfrac{1}{2c_2} \end{bmatrix}$$

and the nodes are

$$c = [0 \quad c_2].$$

In order for this general two-stage ERK method to be second order, the coefficients must be chosen to satisfy order conditions. These conditions are discussed later in this chapter. When the conditions for second order are imposed on the coefficients of the general, two-stage,

second order ERK method, it turns out that all the coefficients can be expressed in terms of one free coefficient.

The *Butcher tableau* storing the coefficients for a two-stage, second order ERK method [Butc87] is:

0	0	0
c_2	c_2	0
	$1-\frac{1}{2c_2}$	$\frac{1}{2c_2}$

Substituting the coefficient values into the general form, we get

$$y_{n+1} = y_n + h\left(\left(1-\frac{1}{2c_2}\right)f(t_n, y_n) + \frac{1}{2c_2}f\big(t_n + c_2 h,\ y_n + c_2 h\, f(t_n, t_n)\big)\right).$$

There are several well-known two-stage, second order ERK methods, but we discuss only two of them. Those two methods are the *Explicit Midpoint method* [Butc87] and *Heun's method* [Butc87].

2.2.3.2 Explicit Midpoint Method

The *Explicit Midpoint method* is a two-stage, second order ERK method with the coefficient value $c_2 = \frac{1}{2}$.

After substituting the value $c_2 = \frac{1}{2}$ into the general form of two-stage, second order ERK method, we get

$$y_{n+1} = y_n + h\, f\left(t_n + \frac{h}{2},\ y_n + \frac{h}{2}\, f(t_n, y_n)\right)$$

for $n = 0, 1, 2, \ldots$. The *Butcher tableau* for the *Explicit Midpoint method* is:

0	0	0
$\frac{1}{2}$	$\frac{1}{2}$	0
	0	1

The name of the method itself suggests what's happening in the formula above. We can see that the first argument of the function f is evaluated at $t_n + \frac{h}{2}$, which is the midpoint between t_n and t_{n+1}. Note, the second argument of the function f looks like the FE method but with half the stepsize. In order to utilize this method, we first use the FE method with half the stepsize to compute a solution approximation and then use that approximation as the second argument for the function f to obtain the solution approximation at the end of the step. The two-stages involved in the computation of the *Explicit Midpoint method* are:

$$k_1 = f(t_n, y_n) \quad \text{and}$$

$$k_2 = f\left(t_n + \frac{h}{2},\ y_n + \frac{h}{2} f(t_n, y_n)\right) = f\left(t_n + \frac{h}{2},\ y_n + \frac{h}{2} k_1\right).$$

The *Explicit Midpoint method* provides more accurate results for a given choice of h, but it requires a bit more computation than the FE method.

2.2.3.3 Heun's Second Order Method

Heun's second order method is a two-stage, second order ERK method with the coefficient value $c_2 = 1$. After substituting the value $c_2 = 1$ into the general form of two-stage, second order ERK method, we get

$$y_{n+1} = y_n + h\left(\frac{1}{2} f(t_n, y_n) + \frac{1}{2} f\big(t_n + h,\ y_n + h\, f(t_n, y_n)\big)\right),$$

$$y_{n+1} = y_n + \frac{h}{2}\Big(f(t_n, y_n) + f\big(t_{n+1},\ y_n + h\, f(t_n, y_n)\big)\Big)$$

for $n = 0, 1, 2, \ldots$. The *Butcher tableau* for the *Heun's second order method* is:

0	0	0
1	1	0
	$\frac{1}{2}$	$\frac{1}{2}$

Note, in this method, the second argument of the second function f is from the FE method. We first obtain the solution approximation by using FE method and then use that approximation as the second argument in the second function f to find the solution approximation at the end of the step.

2.2.4 Third Order ERK Method

In this subsection, we discuss three-stage, third order ERK methods. This means that if the stepsize h is reduced by a factor of 2, then the error will be reduced by a factor of 8. The global error for these methods is $O(h^3)$.

2.2.4.1 General Form

The general form for a three-stage, third order ERK method is

$$y_{n+1} = y_n + h\,(b_1k_1 + b_2k_2 + b_3k_3),$$

for $n = 0, 1, 2, \ldots$, where

$$k_1 = f(t_n, y_n),$$

$$k_2 = f\big(t_n + c_2\,h,\; y_n + h(a_{21}\,k_1)\big) \quad \text{and}$$

$$k_3 = f\big(t_n + c_3h,\; y_n + h(a_{31}k_1 + a_{32}k_2)\big).$$

The general *Butcher tableau* for three-stage, third order ERK method is:

0	0	0	0
c_2	a_{21}	0	0
c_3	a_{31}	a_{32}	0
	b_1	b_2	b_3

where

$$c_2 = a_{21} \ and \ c_3 = a_{31} + a_{32}.$$

The coefficients must satisfy the order conditions for third order. When the order conditions are imposed, the coefficients of the method can all be expressed in terms of one or two free coefficients. It turns out that three-stage, third order methods have three different cases [Butc87].

Case 1. A 2-parameter family of three-stage, third order ERK methods with the conditions,

$c_2 \neq 0, \frac{2}{3}, c_3; c_3 \neq 0, c_2.$

In this case, there are two free variables c_2 and c_3. The *Butcher tableau* storing the coefficients for a three-stage, third order method for this case is:

0	0	0	0
c_2	c_2	0	0
c_3	$\frac{c_3(c_3 - 3c_2 + 3c_2^2)}{c_2(3c_2 - 2)}$	$\frac{c_3(c_2 - c_3)}{c_2(3c_2 - 2)}$	0
	$\frac{2 - 3(c_2 + c_3) + 6c_2c_3}{6c_2c_3}$	$\frac{c_3 - \frac{2}{3}}{2c_2(c_3 - c_2)}$	$\frac{\frac{2}{3} - c_2}{2c_3(c_3 - c_2)}$

Here, the matrix A is

$$A = \begin{bmatrix} 0 & 0 & 0 \\ c_2 & 0 & 0 \\ \frac{c_3(c_3 - 3c_2 + 3c_2^2)}{c_2(3c_2 - 2)} & \frac{c_3(c_2 - c_3)}{c_2(3c_2 - 2)} & 0 \end{bmatrix},$$

the weights are

$$b = \left[\frac{2 - 3(c_2 + c_3) + 6c_2c_3}{6c_2c_3} \quad \frac{c_3 - \frac{2}{3}}{2c_2(c_3 - c_2)} \quad \frac{\frac{2}{3} - c_2}{2c_3(c_3 - c_2)}\right]$$

and the nodes are

$$c = [0 \quad c_2 \quad c_3].$$

Substituting the coefficient values in the general form, we get

$$y_{n+1} = y_n + h\left(\left(\frac{2 - 3(c_2 + c_3) + 6c_2c_3}{6c_2c_3}\right)k_1 + \left(\frac{c_3 - \frac{2}{3}}{2c_2(c_3 - c_2)}\right)k_2 + \left(\frac{\frac{2}{3} - c_2}{2c_3(c_3 - c_2)}\right)k_3\right)$$

for $n = 0, 1, 2, \ldots$, where

$$k_1 = f(t_n, y_n),$$

$$k_2 = f\big(t_n + c_2\, h,\ y_n + h(c_2k_1)\big),$$

$$k_3 = f\left(t_n + c_3h,\ y_n + h\left(\left(\frac{c_3(c_3 - 3c_2 + 3c_2^2)}{c_2(3c_2 - 2)}\right)k_1 + \left(\frac{c_3(c_2 - c_3)}{c_2(3c_2 - 2)}\right)k_2\right)\right)$$

Case 2. A 1-parameter family of three-stage, third order ERK methods with the condition, $b_3 \neq 0$.

In this case, there is only one free variable, b_3. The *Butcher Tableau* storing the coefficients for a three-stage, third order method for this case is:

0	0	0	0
$\frac{2}{3}$	$\frac{2}{3}$	0	0
0	$-\frac{1}{4b_3}$	$\frac{1}{4b_3}$	0
	$\frac{1}{4} - b_3$	$\frac{3}{4}$	b_3

Here, the matrix A is

$$A = \begin{bmatrix} 0 & 0 & 0 \\ \frac{2}{3} & 0 & 0 \\ -\frac{1}{4b_3} & \frac{1}{4b_3} & 0 \end{bmatrix},$$

the weights are

$$b = \begin{bmatrix} \frac{1}{4} - b_3 & \frac{3}{4} & b_3 \end{bmatrix}$$

and the nodes are

$$c = \begin{bmatrix} 0 & \frac{2}{3} & 0 \end{bmatrix}.$$

Substituting the coefficient values in the general form, we get

$$y_{n+1} = y_n + h\left(\left(\frac{1}{4} - b_3\right)k_1 + \frac{3}{4}k_2 + b_3 k_3\right)$$

for $n = 0, 1, 2, \ldots$, where

$$k_1 = f(t_n, y_n),$$

$$k_2 = f\left(t_n + \frac{2h}{3},\ y_n + h\left(\frac{2k_1}{3}\right)\right),$$

$$k_3 = f\left(t_n, y_n + \frac{h}{4b_3}(k_2 - k_1)\right).$$

Case 3. A 1-parameter family of three-stage, third order ERK methods with the condition, $b_3 \neq 0$.

In this case, there is only one free variable, b_3. The *Butcher Tableau* storing the coefficients for a three-stage, third order method for this case is:

0	0	0	0
$\frac{2}{3}$	$\frac{2}{3}$	0	0
$\frac{2}{3}$	$\frac{8b_3 - 3}{12b_3}$	$\frac{1}{4b_3}$	0
	$\frac{1}{4}$	$\frac{3}{4} - b_3$	b_3

Here, the matrix A is

$$A = \begin{bmatrix} 0 & 0 & 0 \\ \frac{2}{3} & 0 & 0 \\ \frac{8b_3 - 3}{12b_3} & \frac{1}{4b_3} & 0 \end{bmatrix},$$

the weights are

$$b = \begin{bmatrix} \frac{1}{4} & \frac{3}{4} - b_3 & b_3 \end{bmatrix}$$

and the nodes are

$$c = \begin{bmatrix} 0 & \frac{2}{3} & \frac{2}{3} \end{bmatrix}.$$

Substituting the coefficient values in the general form, we get

$$y_{n+1} = y_n + h\left(\frac{k_1}{4} + \left(\frac{3}{4} - b_3\right)k_2 + b_3 k_3\right)$$

for $n = 0, 1, 2, \ldots$, where

$$k_1 = f(t_n, y_n),$$

$$k_2 = f\left(t_n + \frac{2h}{3},\ y_n + h\left(\frac{2k_1}{3}\right)\right),$$

$$k_3 = f\left(t_n + \frac{2h}{3},\ y_n + \frac{h}{4b_3}\left(\left(\frac{8b_3 - 3}{3}\right)k_1 + k_2\right)\right).$$

There are several well-known three-stage, third order ERK methods, but we discuss only two of them. The two methods are *Heun's third order method* [Butc87] and *Ralston's third order method* [Butc87].

2.2.4.2 Heun's Third Order Method

Heun's Third Order method is a three-stage, third order ERK method, Case 1, with the coefficient values $c_2 = \frac{1}{3}$ and $c_3 = \frac{2}{3}$. After substituting the values $c_2 = \frac{1}{3}$ and $c_3 = \frac{2}{3}$ into the general form of three-stage, third order ERK method, Case 1, we get

$$y_{n+1} = y_n + \frac{h}{4}(k_1 + 3k_3)$$

where

$$k_1 = f(t_n, y_n),$$

$$k_2 = f\left(t_n + \frac{h}{3},\ y_n + h\frac{k_1}{3}\right),$$

$$k_3 = f\left(t_n + \frac{2h}{3},\ y_n + h\frac{2k_2}{3}\right).$$

for $n = 0, 1, 2, \ldots$. The *Butcher Tableau* for the *Heun's third order method* is:

0	0	0	0
$\frac{1}{3}$	$\frac{1}{3}$	0	0
$\frac{2}{3}$	0	$\frac{2}{3}$	0
	$\frac{1}{4}$	0	$\frac{3}{4}$

2.2.4.3 Ralston's Third Order Method

Ralston's Third Order method is a three-stage, third order ERK method, Case 1, with the coefficient values $c_2 = \frac{1}{2}$ and $c_3 = \frac{3}{4}$. After substituting the values $c_2 = \frac{1}{2}$ and $c_3 = \frac{3}{4}$ into the general form of three-stage, third order ERK method, Case 1, we get

$$y_{n+1} = y_n + \frac{h}{9}(2k_1 + 3k_2 + 4k_3)$$

where

$$k_1 = f(t_n, y_n),$$

$$k_2 = f\left(t_n + \frac{h}{2},\ y_n + h\frac{k_1}{2}\right),$$

$$k_3 = f\left(t_n + \frac{3h}{4},\ y_n + h\frac{3k_2}{4}\right)$$

for $n = 0, 1, 2, \ldots$. The *Butcher Tableau* for the *Ralston's third order method* is:

0	0	0	0
$\frac{1}{2}$	$\frac{1}{2}$	0	0
$\frac{3}{4}$	0	$\frac{3}{4}$	0
	$\frac{2}{9}$	$\frac{1}{3}$	$\frac{4}{9}$

2.2.5 Fourth Order ERK Method

For a fourth order ERK method, if the stepsize h is reduced by a factor of 2, then the error will be reduced by a factor of 16. The global error for these methods is $O(h^4)$.

2.2.5.1 General Form

The general form for a four-stage, fourth order ERK method is

$$y_{n+1} = y_n + h\,(b_1k_1 + b_2k_2 + b_3k_3 + b_4k_4),$$

for $n = 0, 1, 2, \ldots$, where

$$k_1 = f(t_n, y_n),$$

$$k_2 = f\big(t_n + c_2\,h,\; y_n + h(a_{21}\,k_1)\big),$$

$$k_3 = f\big(t_n + c_3\,h,\; y_n + h(a_{31}k_1 + a_{32}k_2)\big) \quad \text{and}$$

$$k_4 = f\big(t_n + c_4\,h,\; y_n + h(a_{41}k_1 + a_{42}k_2 + a_{43}k_3)\big).$$

This method is called a four-stage method because it involves the four stage evaluations given above. The general *Butcher tableau* for a four-stage, fourth order ERK method is:

0	0	0	0	0
c_2	a_{21}	0	0	0
c_3	a_{31}	a_{32}	0	0
c_4	a_{41}	a_{42}	a_{43}	0
	b_1	b_2	b_3	b_4

The coefficients must satisfy the order conditions for fourth order. When these order conditions are imposed, the coefficients of the method can all be expressed in terms of one or two free coefficients. It turns out that, the four-stage, fourth order methods have five different cases [Butc87].

Case 1. A 2-parameter family of four-stage, fourth order ERK methods with the conditions $0, c_2, c_3, 1$ all distinct; $c_2 \neq \frac{1}{2}$ and $3 - 4(c_2 + c_3) + 6c_2c_3 \neq 0$.

In this case, there are two free variables, c_2 and c_3. The *Butcher Tableau* storing the coefficients for a four-stage, fourth order ERK method for this case is:

0	0	0	0	0
c_2	c_2	0	0	0
c_3	$\frac{c_3(3c_2-c_3-4c_2^2)}{2c_2(1-2c_2)}$	$\frac{c_3(c_3-c_2)}{2c_2(1-2c_2)}$	0	0
1	a_{41}	a_{42}	a_{43}	0
	$\frac{1-2(c_2+c_3)+6c_2c_3}{12c_2c_3}$	$\frac{2c_3-1}{12c_2(c_3-c_2)(1-c_2)}$	$\frac{1-2c_2}{12c_3(c_3-c_2)(1-c_3)}$	$\frac{3-4(c_2+c_3)+6c_2c_3}{12(1-c_2)(1-c_3)}$

where

$$a_{41}=\frac{c_3^2(12c_2^2-12c_2+4)-c_3(12c_2^2-15c_2+5)+(4c_2^2-6c_2+2)}{2c_2c_3(3-4(c_2+c_3)+6c_2c_3)},$$

$$a_{42}=\frac{(-4c_3^2+5c_3+c_2-2)(1-c_2)}{2c_2(c_3-c_2)(3-4(c_2+c_3)+6c_2c_3)},$$

$$a_{43}=\frac{(1-2c_2)(1-c_3)(1-c_2)}{c_3(c_3-c_2)(3-4(c_2+c_3)+6c_2c_3)}.$$

Here, the matrix A is

$$A=\begin{bmatrix} 0 & 0 & 0 & 0 \\ c_2 & 0 & 0 & 0 \\ \frac{c_3(3c_2-c_3-4c_2^2)}{2c_2(1-2c_2)} & \frac{c_3(c_3-c_2)}{2c_2(1-2c_2)} & 0 & 0 \\ a_{41} & a_{42} & a_{43} & 0 \end{bmatrix},$$

where a_{41}, a_{42} and a_{43} are as given above. The weights are

$$b=\begin{bmatrix}\frac{1-2(c_2+c_3)+6c_2c_3}{12c_2c_3} & \frac{2c_3-1}{12c_2(c_3-c_2)(1-c_2)} & \frac{1-2c_2}{12c_3(c_3-c_2)(1-c_3)} & \frac{3-4(c_2+c_3)+6c_2c_3}{12(1-c_2)(1-c_3)}\end{bmatrix}$$

and the nodes are

$$c=[0 \quad c_2 \quad c_3 \quad 1].$$

Case 2. A 1-parameter family of four-stage, fourth order ERK methods with the conditions $c_2 = c_3 = \frac{1}{2}, b_3 \neq 0$.

In this case, there is only one free variable, b_3. The *Butcher Tableau* storing the coefficients for a four-stage, fourth order ERK method for this case is:

0	0	0	0	0
$\frac{1}{2}$	$\frac{1}{2}$	0	0	0
$\frac{1}{2}$	$\frac{3b_3 - 1}{6b_3}$	$\frac{1}{6b_3}$	0	0
1	0	$1 - 3b_3$	$3b_3$	0
	$\frac{1}{6}$	$\frac{2}{3} - b_3$	b_3	$\frac{1}{6}$

Here, the matrix A is

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 & 0 \\ \frac{3b_3 - 1}{6b_3} & \frac{1}{6b_3} & 0 & 0 \\ 0 & 1 - 3b_3 & 3b_3 & 0 \end{bmatrix},$$

the weights are

$$b = \begin{bmatrix} \frac{1}{6} & \frac{2}{3} - b_3 & b_3 & \frac{1}{6} \end{bmatrix}$$

and the nodes are

$$c = \begin{bmatrix} 0 & \frac{1}{2} & \frac{1}{2} & 1 \end{bmatrix}.$$

Case 3. A 1-parameter family of four-stage, fourth order ERK methods with the conditions $c_2 = \frac{1}{2}, c_3 = 0, b_3 \neq 0$.

In this case, there is only one free variable, b_3. The *Butcher Tableau* storing the coefficients for a four-stage, fourth order ERK method for this case is:

0	0	0	0	0
$\frac{1}{2}$	$\frac{1}{2}$	0	0	0
0	$-\frac{1}{12b_3}$	$\frac{1}{12b_3}$	0	0
1	$-\frac{1}{2}-6b_3$	$\frac{3}{2}$	$6b_3$	0
	$\frac{1}{6}-b_3$	$\frac{2}{3}$	b_3	$\frac{1}{6}$

Here, the matrix A is

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 & 0 \\ -\frac{1}{12b_3} & \frac{1}{12b_3} & 0 & 0 \\ -\frac{1}{2}-6b_3 & \frac{3}{2} & 6b_3 & 0 \end{bmatrix},$$

the weights are

$$b = \begin{bmatrix} \frac{1}{6}-b_3 & \frac{2}{3} & b_3 & \frac{1}{6} \end{bmatrix}$$

and the nodes are

$$c = \begin{bmatrix} 0 & \frac{1}{2} & 0 & 1 \end{bmatrix}.$$

Case 4. A 1-parameter family of four-stage, fourth order ERK methods with the conditions $c_2 = 1, c_3 = \frac{1}{2}, b_4 \neq 0$.

In this case, there is only one free variable, b_4. The *Butcher Tableau* storing the coefficients for a four-stage, fourth order ERK method for this case is:

0	0	0	0	0
1	1	0	0	0
$\frac{1}{2}$	$\frac{3}{8}$	$\frac{1}{8}$	0	0
1	$1-\frac{1}{4b_4}$	$-\frac{1}{12b_4}$	$\frac{1}{3b_4}$	0
	$\frac{1}{6}$	$\frac{1}{6}-b_4$	$\frac{2}{3}$	b_4

Here, the matrix A is

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ \frac{3}{8} & \frac{1}{8} & 0 & 0 \\ 1-\frac{1}{4b_4} & -\frac{1}{12b_4} & \frac{1}{3b_4} & 0 \end{bmatrix},$$

the weights are

$$b = \begin{bmatrix} \frac{1}{6} & \frac{1}{6}-b_4 & \frac{2}{3} & b_4 \end{bmatrix}$$

and the nodes are

$$c = \begin{bmatrix} 0 & 1 & \frac{1}{2} & 1 \end{bmatrix}.$$

Case 5. A 1-parameter family of four-stage, fourth order ERK methods with the conditions $c_2 \neq 0, c_3 = \frac{1}{2}, b_2 = 0.$

In this case, there is only one free variable, b_4. The *Butcher Tableau* storing the coefficients for a four-stage, fourth order ERK method for this case is:

0	0	0	0	0
c_2	c_2	0	0	0
$\frac{1}{2}$	$\frac{4c_2 - 1}{8c_2}$	$\frac{1}{8c_2}$	0	0
1	$\frac{1 - 2c_2}{2c_2}$	$-\frac{1}{2c_2}$	2	0
	$\frac{1}{6}$	0	$\frac{2}{3}$	$\frac{1}{6}$

Here, the matrix A is

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 \\ c_2 & 0 & 0 & 0 \\ \frac{4c_2 - 1}{8c_2} & \frac{1}{8c_2} & 0 & 0 \\ \frac{1 - 2c_2}{2c_2} & -\frac{1}{2c_2} & 2 & 0 \end{bmatrix},$$

the weights are

$$b = \begin{bmatrix} \frac{1}{6} & 0 & \frac{2}{3} & \frac{1}{6} \end{bmatrix}$$

and the nodes are

$$c = \begin{bmatrix} 0 & c_2 & \frac{1}{2} & 1 \end{bmatrix}.$$

There are several well-known four-stage, fourth order ERK methods, but we discuss only two of them. Those two methods are the *Classical Runge-Kutta method* and the *3/8 Rule method*.

2.2.5.2 The Classical Runge-Kutta Method

The most widely known and used type of Runge-Kutta method is known as the *'classical Runge-Kutta method'*, *'RK4 method'* or *'the Runge-Kutta method'* [Butc87]. It is a four-stage, fourth order ERK method, Case 2, with the coefficient value, $b_3 = \frac{1}{3}$. After substituting the value $b_3 = \frac{1}{3}$ into the general form of four-stage, fourth order ERK method, Case 2, we get

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4),$$

where

$$k_1 = f(t_n, y_n),$$

$$k_2 = f\left(t_n + \frac{h}{2},\ y_n + h\frac{k_1}{2}\right),$$

$$k_3 = f\left(t_n + \frac{h}{2},\ y_n + h\frac{k_2}{2}\right),$$

$$k_4 = f(t_n + h,\ y_n + hk_3)$$

for $n = 0, 1, 2, \ldots$. The *Butcher Tableau* for the *classical Runge-Kutta method* is:

0	0	0	0	0
$\frac{1}{2}$	$\frac{1}{2}$	0	0	0
$\frac{1}{2}$	0	$\frac{1}{2}$	0	0
1	0	0	1	0
	$\frac{1}{6}$	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{6}$

2.2.5.3 The 3/8 Rule Method

The *3/8 Rule method* [Butc87] is a four-stage fourth order ERK method from Case 1 with coefficient values, $c_2 = \frac{1}{3}$ and $c_3 = \frac{2}{3}$. After substituting the values $c_2 = \frac{1}{3}$ and $c_3 = \frac{2}{3}$ into the general form of four-stage, fourth order ERK method, Case 1, we get

$$y_{n+1} = y_n + h\left(\frac{1}{8}k_1 + \frac{3}{8}k_2 + \frac{3}{8}k_3 + \frac{1}{8}k_4\right),$$

where

$$k_1 = f(t_n, y_n),$$

$$k_2 = f\left(t_n + \frac{h}{3},\ y_n + h\frac{k_1}{3}\right),$$

$$k_3 = f\left(t_n + \frac{2h}{3},\ y_n + h\left(-\frac{k_1}{3} + k_2\right)\right),$$

$$k_4 = f\big(t_n + h,\ y_n + h(k_1 - k_2 + k_3)\big).$$

for $n = 0, 1, 2, \ldots$. The *Butcher Tableau* for the *3/8 Rule method* is:

0	0	0	0	0
$\frac{1}{3}$	$\frac{1}{3}$	0	0	0
$\frac{2}{3}$	$-\frac{1}{3}$	1	0	0
1	1	-1	1	0
	$\frac{1}{8}$	$\frac{3}{8}$	$\frac{3}{8}$	$\frac{1}{8}$

2.3 Optimal ERK methods

From the discussion on ERK methods in the previous section, we now have general forms for ERK methods for second, third and fourth order. In this section, we are going to discuss the

order conditions and Principal Error Coefficients for ERK methods. Then we discuss how to determine optimal values for the free coefficients that appear in the general forms.

2.3.1 Order Conditions

We can obtain Runge-Kutta order conditions by comparing the Taylor series expansion of the exact solution with the Taylor series expansion of the solution given by the Runge-Kutta method [HNW87]. In order to give an example of how the order conditions can be obtained, we consider the case of the general three-stage, third order ERK method. Let's consider the IVODE,

$$y' = f(y)$$

and assume f is sufficiently smooth. Let

$$y = y(t), \qquad y' = y'(t), \qquad y'' = y''(t), \qquad y''' = y'''(t),$$

$$f \equiv f(x,y), \qquad f_x \equiv \frac{\partial f(x,y)}{\partial x}, \qquad f_{xx} \equiv \frac{\partial^2 f(x,y)}{\partial x^2},$$

$$f_{xy} \equiv f_{yx} \equiv \frac{\partial^2 f(x,y)}{\partial x \partial y}, \qquad f_{yy} \equiv \frac{\partial^2 f(x,y)}{\partial^2 y}.$$

Using a Taylor series expansion of the exact solution, we have

$$y(x_{n+1}) = y(x_n) + h\, y'(x_n) + \frac{h^2}{2} y''(x_n) + \frac{h^3}{6} y'''(x_n) + O(h^4).$$

Now, the derivative of $y(x_n)$ is

$$y'(x_n) = f\big(x_n, y(x_n)\big) \equiv f.$$

The second derivative of $y(x_n)$ is

$$y''(x_n) = \frac{\partial}{\partial x} f\big(x_n, y(x_n)\big) = \frac{\partial f}{\partial x} + \frac{\partial f}{\partial y}\,\frac{\partial y}{\partial x} = f_x + f_y\, f.$$

Finally, the third derivative of $y(x_n)$ is

$$y'''(x_n) = \frac{\partial^2 f}{\partial x^2}\big(x_n, y(x_n)\big) = \frac{\partial}{\partial x}\Big(f_x\big(x_n, y(x_n)\big) + f_y\big(x_n, y(x_n)\big) f\big(x_n, y(x_n)\big)\Big)$$

$$= \left[\frac{\partial}{\partial x}\Big(f_x\big(x_n, y(x_n)\big)\Big)\right] + \left[\frac{\partial}{\partial x}\Big(f_y\big(x_n, y(x_n)\big) f\big(x_n, y(x_n)\big)\Big)\right]$$

$$= \left[\frac{\partial}{\partial x} f_x + \frac{\partial}{\partial y} f_x \frac{\partial y}{\partial x}\right] + \left[f\big(x_n, y(x_n)\big) \frac{\partial}{\partial x} f_y\big(x_n, y(x_n)\big)\right] + \left[f_y\big(x_n, y(x_n)\big) \frac{\partial}{\partial x} f\big(x_n, y(x_n)\big)\right]$$

$$= \left[f_{xx} + f_{yx}\, f\right] + \left[f\left(\frac{\partial f_y}{\partial x} + \frac{\partial f_y}{\partial y}.\frac{\partial y}{\partial x}\right)\right] + \left[f_y\left(\frac{\partial f}{\partial x} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial x}\right)\right]$$

$$= \left[f_{xx} + f_{yx}\, f\right] + \left[f\big(f_{xy} + f_{yy}.f\big)\right] + \left[f_y\big(f_x + f_y\, f\big)\right]$$

$$= f_{xx} + 2f f_{xy} + f^2 f_{yy} + f_y\big(f_x + f f_y\big).$$

Let

$$F \equiv f_x + f f_y \text{ and } G \equiv f_{xx} + 2f f_{xy} + f^2 f_{yy};$$

then

$$y(x_{n+1}) = y(x_n) + hf + \frac{h^2}{2}F + \frac{h^3}{6}\big(F f_y + G\big) + O(h^4) \quad \ldots(1).$$

Next, let's consider the three-stage, ERK method that we discussed in Section 2.2.4.1. This method has the form

$$y(x_{n+1}) = y(x_n) + h(b_1 k_1 + b_2 k_2 + b_3 k_3)$$

where

$$k_1 = f\big(x_n, y(x_n)\big),$$

$$k_2 = f(x_n + c_2 h, y(x_n) + h c_2 k_1),$$

$$k_3 = f\Big(x_n + c_3 h, y(x_n) + h\big((c_3 - a_{32})k_1 + a_{32} k_2\big)\Big).$$

The *Butcher tableau* for this method is given in Section 2.2.4.1. We need to express each of the stages in terms of $f(x_n, y_n)$ and higher derivatives.

$$k_1 = f\big(x_n, y(x_n)\big) = f,$$

$$k_2 = f(x_n + c_2 h, y(x_n) + c_2 h\, f).$$

First, we expand k_2 using a Taylor series in the first variable:

$$k_2 = [f(x_n, y(x_n) + c_2 h\, f)] + [(c_2 h)\, f_x(x_n, y(x_n) + c_2 h\, f)] +$$

$$\left[\frac{(c_2 h)^2}{2} f_{xx}(x_n, y(x_n) + c_2 h\, f)\right] + O(h^3).$$

Next, we perform expansions using a Taylor series, for each of the terms in the [ ] brackets above, in the second variable:

$$k_2 = \left[f\big(x_n, y(x_n)\big) + (c_2 h\, f) f_y\big(x_n, y(x_n)\big) + \frac{(c_2 h\, f)^2}{2} f_{yy}\big(x_n, y(x_n)\big) + O(h^3)\right]$$

$$+ \Big[(c_2 h)\big[f_x\big(x_n, y(x_n)\big) + (c_2 h\, f) f_{xy}\big(x_n, y(x_n)\big) + O(h^2)\big]\Big]$$

$$+ \left[\frac{(c_2 h)^2}{2}\big[f_{xx}\big(x_n, y(x_n)\big) + O(h)\big]\right],$$

$$k_2 = f + (c_2 h) f f_y + \frac{(c_2 h)^2}{2} f^2 f_{yy} + (c_2 h) f_x + (c_2 h)^2 f f_{xy} + \frac{(c_2 h)^2}{2} f_{xx} + O(h^3),$$

$$k_2 = f + (c_2 h)\big(f_x + f f_y\big) + \frac{(c_2 h)^2}{2}\big(f_{xx} + 2 f f_{xy} + f^2 f_{yy}\big) + O(h^3),$$

$$k_2 = f + c_2 h\, F + \frac{(c_2 h)^2}{2} G + O(h^3).$$

The expansion for k_3 is far more tedious (see [HNW87]) but eventually gives,

$$k_3 = f + h\big(c_3 f_x + \big((c_3 - a_{32}) f + a_{32} k_2\big) f_y\big)$$

$$+ \frac{h^2}{2}\Big(c_3^2 f_{xx} + 2c_3\big((c_3 - a_{32}) f + a_{32} + k_2\big) f_{xy} + \big((c_3 - a_{32}) f + a_{32} k_2\big)^2 f_{yy}\Big) + O(h^3).$$

Substituting for k_2 as obtained earlier and solving for k_3 using F and G gives

$$k_3 = f + hc_3\,F + h^2\left(a_{32}c_2\,F\,f_y + \frac{c_3^2}{2}G\right) + O(h^3).$$

Substituting the expressions for the stages into

$$y(x_{n+1}) = y(x_n) + h(b_1k_1 + b_2k_2 + b_3k_3)$$

gives,

$$y(x_{n+1}) = y(x_n) + h(b_1 + b_2 + b_3)f + h^2(b_2c_2 + b_3c_3)F$$
$$+ h^3\left[(b_3a_{32}c_2)Ff_y + \frac{1}{2}(b_2c_2^2 + b_3c_3^2)G\right] + O(h^4) \quad \ldots(2).$$

Next, we need to compare the above with the expansion of the exact solution (1), which was

$$y(x_{n+1}) = y(x_n) + hf + \frac{h^2}{2}F + \frac{h^3}{6}\left(Ff_y + G\right) + O(h^4).$$

If we compare like terms to match the numerical and exact solutions, we must have

$$b_1 + b_2 + b_3 = 1 \quad \text{(First Order)},$$

$$b_2c_2 + b_3c_3 = \frac{1}{2} \quad \text{(Second Order)},$$

$$b_3a_{32}c_2 = \frac{1}{6} \quad \text{(Third Order)},$$

$$\frac{1}{2}(b_2c_2^2 + b_3c_3^2) = \frac{1}{6} \quad \text{(Third Order)}.$$

These are called *Runge-Kutta order conditions* for third order. These order conditions can also be written in the following form:

$$b^Te = 1, \qquad b^Tc = \frac{1}{2}, \qquad b^Tc^2 = \frac{1}{3} \qquad \text{and} \qquad b^TAc = \frac{1}{6}.$$

where e is a vector of ones and $c^2 = [c_1^2 \quad c_2^2 \quad c_3^2]^T$. The computations required to find the order conditions for fourth and fifth order methods are far more tedious. But after all the

computations have been done, we have the following order conditions for fourth order ERK methods [Butc87]:

$$b^T c^3 = \frac{1}{4},$$

$$b^T cAc = \frac{1}{8},$$

$$b^T Ac^2 = \frac{1}{12},$$

$$b^T A^2 c = \frac{1}{24},$$

and the following order conditions for fifth order ERK methods [Butc87]:

$$b^T c^4 = \frac{1}{5},$$

$$(bc^2)^T Ac = \frac{1}{10},$$

$$(bc)^T Ac^2 = \frac{1}{15},$$

$$(bc)^T A^2 c = \frac{1}{30},$$

$$b^T (Ac)^2 = \frac{1}{20},$$

$$b^T Ac^3 = \frac{1}{20},$$

$$b^T Ac(Ac) = \frac{1}{40},$$

$$b^T A^2 c^2 = \frac{1}{60},$$

$$b^T A^3 c = \frac{1}{120}.$$

Some of the order conditions above need to be interpreted in a certain manner. For example, take vector c for instance. The square of vector c, that is, c^2, is interpreted as component-wise product instead of the dot product. In component-wise product, the respective components of the two vectors are multiplied together resulting into a vector of the same size. For 2 vectors of size n,

$$a = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}, \qquad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix},$$

the component-wise product of the vectors a and b is

$$ab = \begin{bmatrix} a_1 b_1 \\ a_2 b_2 \\ \vdots \\ a_n b_n \end{bmatrix}.$$

Similarly, if the vector is multiplied by itself n times using the component-wise product, then the resulting vector would be

$$a^n = \begin{bmatrix} a_1^n \\ a_2^n \\ \vdots \\ a_n^n \end{bmatrix}.$$

Later in this thesis, we show how these order conditions are used within the process to optimize general ERK methods.

2.3.2 Principal Error Coefficients

The collection of order conditions that are one order higher than the order of an ERK method gives the Principal Error Coefficient of that method; it is denoted by E_p where p is the order of that ERK method. An essential point is that the Principal Error Coefficient gives a method-dependent but problem independent measure of the leading order term in the error of the solution that is obtained from the ERK method. Over a sufficiently large class of problems, a

method with a smaller Principal Error Coefficient is expected to be generally more accurate, i.e., have a smaller error, than a method with a larger Principal Error Coefficient. However, the Principal Error Coefficient is not exactly the same as the error of the solution obtained from the ERK method because, in the actual error, the unsatisfied order conditions that make up the components of the Principal Error Coefficient are multiplied by problem dependent derivatives of $f\big(t, y(t)\big)$. See equation (2) above. We use the results from the previous section to obtain the Principal Error Coefficients of the ERK methods. Let's start with the second order ERK methods. The Principal Error Coefficient for a second order ERK method is the vector [Butc87]

$$E_2 = \begin{bmatrix} \frac{1}{2}\left(b^T c^2 - \frac{1}{3}\right) \\ b^T A c - \frac{1}{6} \end{bmatrix}.$$

These are the weighted, unsatisfied order conditions for third order. Note that $\left(b^T c^2 - \frac{1}{3}\right)$ is weighted by $\frac{1}{2}$ because that is how it appears in the Taylor series given in the previous section. We consider the square of the 2-norm of the Principal Error Coefficient E_2; this is

$$\big||E_2|\big|_2^2 = \left(\frac{1}{2}\left(b^T c^2 - \frac{1}{3}\right)\right)^2 + \left(b^T A c - \frac{1}{6}\right)^2.$$

Similarly, the Principal Error Coefficient for a third order ERK method is the vector of weighted, unsatisfied order conditions for fourth order [Butc87]; it has the form

$$E_3 = \begin{bmatrix} \left(\frac{1}{6}\right)\left(b^T c^3 - \frac{1}{4}\right) \\ b^T c A c - \frac{1}{8} \\ \left(\frac{1}{2}\right)\left(b^T A c^2 - \frac{1}{12}\right) \\ b^T A^2 c - \frac{1}{24} \end{bmatrix}.$$

The square of the 2-norm of the Principal Error Coefficient E_3 is

$$||E_3||_2^2 = \left(\left(\frac{1}{6}\right)\left(b^T c^3 - \frac{1}{4}\right)\right)^2 + \left(b^T cAc - \frac{1}{8}\right)^2 + \left(\left(\frac{1}{2}\right)\left(b^T Ac^2 - \frac{1}{12}\right)\right)^2 + \left(b^T A^2 c - \frac{1}{24}\right)^2.$$

Lastly, the Principal Error Coefficient for a fourth order ERK method is the vector of weighted, unsatisfied order conditions for fifth order [Butc87]; this has the form

$$E_4 = \begin{bmatrix} \left(\frac{1}{24}\right)\left(b^T c^4 - \frac{1}{5}\right) \\ \left(\frac{1}{2}\right)\left((bc^2)^T Ac - \frac{1}{10}\right) \\ \left(\frac{1}{2}\right)\left((bc)^T Ac^2 - \frac{1}{15}\right) \\ (bc)^T A^2 c - \frac{1}{30} \\ \left(\frac{1}{2}\right)\left(b^T (Ac)^2 - \frac{1}{20}\right) \\ \left(\frac{1}{6}\right)\left(b^T Ac^3 - \frac{1}{20}\right) \\ b^T Ac(Ac) - \frac{1}{40} \\ \left(\frac{1}{2}\right)\left(b^T A^2 c^2 - \frac{1}{60}\right) \\ b^T A^3 c - \frac{1}{120} \end{bmatrix}.$$

The square of the 2-norm of the Principal Error Coefficient E_4 is

$$\begin{aligned} ||E_4||_2^2 = &\left(\left(\frac{1}{24}\right)\left(b^T c^4 - \frac{1}{5}\right)\right)^2 + \left(\left(\frac{1}{2}\right)\left((bc^2)^T Ac - \frac{1}{10}\right)\right)^2 + \left(\left(\frac{1}{2}\right)\left((bc)^T Ac^2 - \frac{1}{15}\right)\right)^2 \\ &+ \left((bc)^T A^2 c - \frac{1}{30}\right)^2 + \left(\left(\frac{1}{2}\right)\left(b^T (Ac)^2 - \frac{1}{20}\right)\right)^2 + \left(\left(\frac{1}{6}\right)\left(b^T Ac^3 - \frac{1}{20}\right)\right)^2 \\ &+ \left(b^T Ac(Ac) - \frac{1}{40}\right)^2 + \left(\left(\frac{1}{2}\right)\left(b^T A^2 c^2 - \frac{1}{60}\right)\right)^2 + \left(b^T A^3 c - \frac{1}{120}\right)^2. \end{aligned}$$

An optimal ERK method is obtained by choosing the free coefficients to minimize the 2-norm of the Principal Error Coefficient of the method. We consider the determination of optimal ERK methods in Chapter 4.

2.4 Continuous extensions of discrete solutions from ERK methods

An ERK method provides discrete numerical solutions, y_i, at a set of points, t_i, on the domain $[t_0, t_f]$. However, high quality software for solving initial value ODEs requires that a continuous numerical solution approximation be available to the user of the software. If we have a discrete numerical solution at certain points of the domain, we can extend that solution in order to obtain a continuous numerical solution approximation over the entire domain. This provides continuity in the numerical solution which then allows the user to plot that solution on a graph or find a solution value at any point on the domain. It is essential that the order of accuracy of the continuous solution approximation be at least as high as that of the discrete solution.

2.4.1 Hermite Interpolation

For orders 1 to 4, a continuous approximate solution of the appropriate order can be obtained using Hermite interpolation, which involves finding polynomial functions with specified function and derivative values. Since we want interpolants of fourth order, we need to use Hermite cubics [SAP97]. Choosing an interpolant that is of fourth order means that the interpolation error will be at least as small as the error of the discrete solution obtained from the ERK method.

Let's assume that t is in the subinterval $[t_i, t_{i+1}]$. Let $\theta = [t - t_i]/h_i$ where $h_i = t_{i+1} - t_i$. Here, θ is the relative distance of t from the point t_i in the subinterval. For example, θ will be $\frac{1}{2}$ if t is at the midpoint of the subinterval. The Hermite form of the continuous solution approximation, $u_i(t)$, on $[t_i, t_{i+1}]$ is

$$u_i(t) = y_i\, h_{00}(\theta) + h_i\, f_i\, h_{10}(\theta) + y_{i+1}\, h_{01}(\theta) + h_i\, f_{i+1}\, h_{11}(\theta),$$

where the Hermite basis polynomials, $h_{00}(\theta), h_{10}(\theta), h_{01}(\theta)$, and $h_{11}(\theta)$, are

$$h_{00}(\theta) = (1 + 2\theta)(1 - \theta)^2, \qquad h_{10}(\theta) = \theta(1 - \theta)^2,$$

$$h_{01}(\theta) = \theta^2(3 - 2\theta), \qquad h_{11}(\theta) = \theta^2(\theta - 1),$$

where

$$f_i = f(t_i, y_i), \qquad f_{i+1} = f(t_{i+1}, y_{i+1}),$$

$$y_i \approx y(t_i) \qquad \text{and} \qquad y_{i+1} \approx y(t_{i+1}).$$

Using Hermite interpolation allows us to get a continuous numerical solution approximation at any point on the domain. Hermite interpolants can be used for the ERK methods of order from 1 to 4. This is because, as mentioned above, the interpolation error for the Hermite interpolant is $O(h^4)$, which is the same as or smaller than the error of the solution from the ERK method.

2.5 The Defect of the Continuous Approximate Solution

Numerical methods are used to find the approximate solution to equations which don't have exact solutions or for which finding an exact solution is too complicated or time consuming because of the complexity of the problem. The approximate numerical solution will have an error. Thus, it is essential for the numerical method to also deliver an estimate of the error in the numerical solution that is returned.

One way of assessing the accuracy of a continuous approximate solution is to consider the defect of the approximate solution. In the previous section, we discussed Hermite interpolation and how it extends the discrete numerical solution approximation to give a continuous numerical solution approximation. But how well does that continuous numerical solution approximation satisfy the ODE? We answer that question by computing the defect of the continuous numerical solution approximation. As mentioned earlier, the defect is the amount by which the continuous approximate solution fails to satisfy the ODE. The defect,

$$\delta(t) = u'(t) - f\big(t, u(t)\big),$$

of a continuous numerical solution, $u(t)$, is a continuous function of t, and the question is how to estimate the maximum value of $|\delta(t)|$ on each step $[t_i, t_{i+1}]$. We can clearly see above that the value of the defect is problem dependent and therefore, the location of the maximum defect can vary from step to step and problem to problem.

Since, we already have the continuous numerical solution approximation, it is straightforward to evaluate $\delta(t)$ at a given point t. But generally, it is not straightforward to determine the location within $[t_i, t_{i+1}]$ where $\delta(t)$ is maximum. However, we would like to have an estimate of the maximum value of $\delta(t)$ on each step to assess the quality of $u(t)$. The standard approach to obtain this estimate is to sample the defect at several points within each step and use the maximum of these samples as an estimate of the maximum defect.

In Chapter 4, we compute the defect for a continuous approximate solution for several IVODEs and show the form that it has on a given step. We do not consider the task of sampling the defect to obtain an estimate of the maximum defect on each step. As explained above, this process is straightforward.

Chapter 3

Software Implementation

In this chapter, we describe the software implementation for the determination of optimal ERK methods, the representation of ERK methods and the testing of ERK methods on an ODE test set. This software also implements Hermite interpolants to provide a continuous solution approximation, allowing this solution to be plotted, and it implements a defect sampling algorithm so that plots of the defect can be obtained. We provide a detailed description of the structure of the software and its capabilities. The software is created in the Python language and can be found in the Appendix.

3.1 Optimization Software

The purpose of this software is to find the optimal values for the free coefficients by minimizing the Principal Error coefficient of the ERK method. This software uses the python library 'SciPy', and in particular the '*minimize()*' function from within that library, in order to find the optimal values for the free coefficients of a given ERK method. In this software, we use the Principal Error Coefficients E_n for $n = 2, 3, 4$, which we discussed in Section 2.3. The script for this software is *'Optimization.py'*, given in the Appendix. We first discuss the functions that initiate this software.

- ***displayMenu()***: This function prints the available choices to initiate the optimization software for a specific Principal Error Coefficient. Then, it asks for input from the user and saves that input in the integer variable *'choice'* which is then returned by the function. The available choices are:

1: Optimize $E2$

2: Optimize $E3$

3: Optimize $E4$

This function is initiated to provide a choice to the *initializeOptimizer()* function.

- ***initializeOptimizer(choice)***: This function takes the integer variable *'choice'* as the parameter and uses it to initiate the optimizer for a specific Principal Error Coefficient. Depending upon the value given for the parameter *'choice'*, the function *'optimize()'* is called with one of the following parameters:

  1: $E2$

  2: $E3$

  3: $E4$

  For E_4, there are five cases. So, before initiating the function *optimize*(E_4), the *chooseE4Case()* function is called to get the *'case'* value for the optimization of E_4. (For third order, we saw earlier that there are three cases, but we show in Chapter 4, that for two of these cases, we cannot choose the free coefficient to minimize the Principal Error Coefficient.) When this function has completed, the user will be provided with the following results:

  1: Minimized E_n value, where $n = 2, 3, 4$

  2: Optimal values for the free coefficients

  3: An indicator of whether the optimization software terminated successfully.

- ***chooseE4Case()***: This function prints all the cases for E_4 and returns the integer value for the case provided by the user. The cases for E_4 that are available in this function are the cases for four-stage, fourth order ERK methods; (Section 2.2.5).

3.1.1 Use of the Optimization Software

In order to use this software, the user is first required to install the python library 'SciPy' using pip. After that, the user should run the *'Optimization.py'* python script. Then, the user should enter their choice for the order of the method to be optimized and for E_4, choose the case as well. The script will provide the results of the optimization. An example is given below:

```
==== START: Research-Thesis\Optimization.py ====
1. Optimize E2
2. Optimize E3
3. Optimize E4
Enter your choice: 1
Message: 'Optimization terminated successfully.'
E2: 0.02777777777777779
Free Coefficients: [0.66666665]
```

3.2 Software for Testing Explicit Runge-Kutta Methods

The purpose of this software is to investigate the performance of the ERK methods identified in this thesis. In this software, the ERK methods of order 1 to 4 are implemented for all cases defined in Section 2.2 of this thesis. This software employs several python scripts which can be found in the Appendix. The scripts are:

- *main.py*
- *config.py*
- *EulersMethod.py*
- *Function.py*
- *HermiteInterpolation.py*
- *Methods.py*
- *ivode.py*
- *FileIO.py*

The script that is used to initiate this software is *'main.py'*. Let's first discuss the functions in this script which initiate the software.

- ***displayMenu()***: This function prints the available choices to initiate this software, which are

  1: *Specific IVODE on a specific method*:

  This option will initiate the function which allows the user to choose an ERK method to solve a specified IVODE and provide the results as output.

  2: *Specific IVODE on All Methods and Export results to a file*:

  This option will initiate the function which allows the user to use all the ERK methods to solve a specified IVODE and provide the results in a text file.

  Then, it asks for input from the user and saves that input in the integer variable *'choice'* which is then returned by the function. This function is initiated to provide a choice to the *chooseMenuOption()* function.

- ***chooseMenuOption(choice)***: This function takes the integer variable *'choice'* as a parameter and uses it to initiate the testing of one or more ERK methods. Depending upon the value given for the parameter *'choice'*, one of the following functions is called:

  1: *specificIVODESpecificMethod()*

  2: *specificIVODEAllMethods()*

- ***specificIVODESpecificMethod()***: This function initiates the specified ERK method to solve a specific IVODE chosen by the user. First, it asks the user to select the IVODE with its initial and final values. Then, it asks the user to select the specific ERK method which it uses to solve that IVODE. To find the numerical solution approximations,

stepsizes from 2^{-1} to 2^{-6} are used by the function. If the chosen IVODE has an exact solution, the function will provide the following information:

1: Error at t_f

   The absolute difference between the approximate and exact solution of an IVODE at the final time (t_f).

2: Stepsize

   The stepsize used to obtain the error for that IVODE.

3: Ratio of the errors

   The ratio of the error from the previous stepsize and the error from the current stepsize.

4: Order of Convergence

   The order of convergence for the ERK method used (See Section 2.2). (The order of convergence is easily determined from the ratios of the errors.)

5: Graph for Hermite interpolant

   A graph plotting the continuous numerical solution approximation obtained by using Hermite interpolation. The file containing the graph will be 'Hermite Interpolation.html' found in the 'Plots' folder.

6: Graph for Defect

   A graph plotting the defect in the continuous numerical solution approximation obtained by Hermite interpolation. The file containing the graph will be 'Defect.html' found in the 'Plots' folder.

If the IVODE does not have an exact solution, then the information provided will be:

1: Numerical solution approximation at t_f

The approximate numerical solution of an IVODE at the final time (t_f).

2: Stepsize

The stepsize used to obtain the approximate numerical solution for that IVODE.

3: Graph for Hermite interpolant

A graph plotting the continuous numerical solution approximation obtained by using Hermite interpolation. The file containing the graph will be 'Hermite Interpolation.html' found in the 'Plots' folder.

4: Graph for Defect

A graph plotting the defect in the continuous numerical solution approximation obtained by Hermite interpolation. The file containing the graph will be 'Defect.html' found in the 'Plots' folder.

- ***specificIVODEAllMethods()***: This function initiates all the ERK methods to solve a specific IVODE chosen by the user. First, it asks the user to select the IVODE with its initial and final values. Next, it uses all the ERK methods to solve that IVODE. Since this test produces a large amount of output, the results are saved in a text file. It also uses stepsizes from 2^{-1} to 2^{-6} to find the numerical solution approximations. If the chosen IVODE has an exact solution, it will provide the following information:

1: Error at t_f

The absolute difference between the approximate and exact solution of an IVODE at the final time (t_f).

2: Stepsize

The stepsize used to obtain the error for that IVODE.

3: Ratio of the errors

The ratio of the error from the previous stepsize and the error from the current stepsize.

4: Order of Convergence

The order of convergence for the ERK method used (See Section 2.2).

5: Relative to Minimum Error

The ratio of each error and the smallest error.

If the IVODE does not have an exact solution, then the information provided will be:

1: Numerical solution approximation at t_f

The approximate numerical solution of an IVODE at the final time (t_f).

2: Stepsize

The stepsize used to obtain the approximate numerical solution for that IVODE.

3.2.1 Use of the ERK Testing Software

This software uses the python library named 'bokeh' to plot the graph for Hermite interpolants and defects. In order to use this software, the user is first required to install the python library 'bokeh' using pip. After that, run the *'main.py'* python script. The user should enter their choice to initiate the software for a single ERK method or for all ERK methods. Then, the user should choose the IVODE to be solved. For the choice of single ERK method, the user should choose the specific ERK method to be used to solve the IVODE. The above-mentioned results will be provided along with a graph for the Hermite interpolant and a graph for defects. On

the other hand, if the initial choice was to use all the ERK methods, then the results of the computations will be provided in a text file. An example is given below:

```
======== START: Research-Thesis\main.py ========
1. Specific IVODE on Specific Method
2. Specific IVODE on All Methods and Export results to a
file
Enter your choice: 1

Simple: f1 t tfinal y0
Predator Prey: f2 t tfinal x y alpha beta gamma delta
Simple System: f3 t tfinal x y
Test F4: f4 t tfinal y0
Test F5: f5 t tfinal y0
Test F6: f6 t tfinal y0
Test F7: f7 t tfinal y0 alpha
Sample COVID-19 Model: f8 t tfinal

Enter the formula with values respectively (Use spaces
between the values like shown above):
f1 0 1 1

1. Forward Euler Method
2. Explicit Midpoint Method
3. Heun's Second Order Method
4. Second Order RK Method
5. Heun's Third Order Method
6. Ralston's Third Order Method
7. Third Order RK Method
8. RK4 Method
9. FourthOrderRKMethod

Enter the method with values respectively (Use spaces
between the values like shown above):
1

ee[0]: 0.11787944117144233      Steps: 0.5
ee[0]: 0.051473191171442334     Steps: 0.25     eeOld/ee: 2.2901133286805546    Order: 1
ee[0]: 0.024270525365625684     Steps: 0.125    eeOld/ee: 2.120810752796631     Order: 1
ee[0]: 0.01180531071964952      Steps: 0.0625   eeOld/ee: 2.0558989036373485    Order: 1
ee[0]: 0.005824151915125697     Steps: 0.03125  eeOld/ee: 2.0269578973361546    Order: 1
ee[0]: 0.002892916927534961     Steps: 0.015625 eeOld/ee: 2.0132454754200033    Order: 1
```

3.3 How to add a new IVODE

There are several python scripts that are used to create the software as mentioned in Section 3.2. The script related to the IVODEs is '*ivode.py*'. There are a few IVODEs that are already implemented in this script which can be used as test examples. To add a new IVODE, the user should create a new function with an appropriate name in the same script. The user should create three sections in this function using if-else statements with comparison operators on i. In this software, the IVODEs are implemented in the form of lists, so each section returns the results in the form of a list. The results returned from each section, with respective to the value of i, are the following:

1: For $i = 0$, the approximate numerical solution of the IVODE is returned.

2: For $i = 1$, the exact value for the IVODE is returned (if exists).

3: For $i = 2$, the error associated with the IVODE is returned (if exists).

After the user creates the IVODE function, the user should then edit the *'Function.py'* python script. The user should add the function call for that IVODE with a new *formulaNumber* in the *formula()* function. The user should use this *formulaNumber* to display it in the *displayFormulas()* function and set the initial values in *setFormulaValues()* function. An example is provided below.

Consider the example IVODE,

$$y'(t) = -y(t) \quad \text{with} \quad y(0) = 1.$$

The exact solution for this IVODE is

$$y(t) = e^{-t}.$$

The script for the IVODE above looks like the following:

```
def simple(i, t, y):
    # IVODE
    if (i == 0):
        return [-y[0]]
    # Exact solution for the IVODE
    elif (i == 1):
        return [math.exp(-t)]
    # Error associated with the solution for the IVODE
    else:
        return [y[0] - math.exp(-t)]
```

Chapter 4

Results and Discussion

In this chapter, we present results we have obtained using the software discussed in the previous chapter. We determine optimal ERK methods and compare them with standard ERK methods. Then, we present experimental confirmation of the order of convergence of the optimal methods using a test set of ODEs. After that, we apply the standard and optimal ERK methods to test sets to examine the accuracy of the solution approximations computed by the methods. The last section of this chapter considers augmenting the discrete numerical solutions computed using the ERK methods with continuous approximate solutions obtained by using Hermite interpolation. This section also considers the computation of the defect of a continuous solution approximation.

4.1 Optimal ERK methods and Comparison with Standard Methods

As we have discussed in Chapter 3, we have employed optimization software which allows us to minimize the Principal Error Coefficients for ERK methods to obtain optimal values for the free coefficients of the methods.

4.1.1 Second Order ERK Method Optimization

To optimize the general, two-stage, second order ERK method, we need to minimize the Principal Error Coefficient, E_2. We recall from Chapter 2 that for the general two-stage, second order ERK method, we have the coefficients in the form of *Butcher tableau* as follows:

$$\begin{array}{c|cc} 0 & 0 & 0 \\ c_2 & c_2 & 0 \\ \hline & 1-\frac{1}{2c_2} & \frac{1}{2c_2} \end{array}$$

We can see that c_2 is the only independent variable and a_{21}, b_1 and b_2 are all dependent upon c_2. From Chapter 2, the Principal Error Coefficient is

$$E_2 = \begin{bmatrix} \frac{1}{2}\left(b^T c^2 - \frac{1}{3}\right) \\ b^T Ac - \frac{1}{6} \end{bmatrix}.$$

Using the expressions for the coefficients of the general two-stage, second order ERK method, we have $b = \begin{bmatrix} 1 - \frac{1}{2c_2} & \frac{1}{2c_2} \end{bmatrix}$, $c = [0 \quad c_2]$ and the matrix $A = \begin{bmatrix} 0 & 0 \\ c_2 & 0 \end{bmatrix}$. To find the optimal c_2 value, we need to choose c_2 to minimize the square of the 2-norm of E_2:

$$\left||E_2|\right|_2^2 = \left(\frac{1}{2}\left(b^T c^2 - \frac{1}{3}\right)\right)^2 + \left(b^T Ac - \frac{1}{6}\right)^2.$$

After substituting the values for b, A and c, we get an expression which depends upon c_2:

$$\left||E_2|\right|_2^2 = \left(\frac{1}{2}\left(\left(\frac{1}{2c_2}\right){c_2}^2 - \frac{1}{3}\right)\right)^2 + \left(-\frac{1}{6}\right)^2, \text{or}$$

$$\left||E_2|\right|_2^2 = \left(\frac{1}{2}\left(\left(\frac{1}{2}\right)c_2 - \frac{1}{3}\right)\right)^2 + \frac{1}{36}.$$

The above expression can be minimized by choosing $c_2 = \frac{2}{3}$. This is the same value provided by the optimization software. For $c_2 = \frac{2}{3}$, we get

$$\left||E_2|\right|_2^2 = \frac{1}{36} = 0.027777776,$$

and the value for $\left||E_2|\right|$ is

$$\left||E_2|\right| = \frac{1}{6} = 0.166666.$$

As mentioned earlier in this thesis, there are several well-known second order ERK methods, such as the *Explicit-Midpoint method* and *Heun's second order method*. We compare the optimal method with these methods by finding their $\left||E_2|\right|$ values.

The *Explicit-Midpoint method* has $c_2 = \frac{1}{2}$. Putting this value of c_2 into the general expression for $\left||E_2|\right|_2^2$, we get

$$\left||E_2|\right|_2^2 = \left(\frac{1}{2}\left(\frac{1}{4} - \frac{1}{3}\right)\right)^2 + \frac{1}{36},$$

$$\left||E_2|\right|_2^2 = \frac{1}{576} + \frac{1}{36} = \frac{17}{576},$$

$$\left||E_2|\right|_2^2 = 0.029513888.$$

So, the value of $\left||E_2|\right|$ for the *Explicit-Midpoint method* is

$$\left||E_2|\right| = 0.17179606.$$

Here, we can clearly see that the value of $\left||E_2|\right|$ for the *Explicit-Midpoint method* is slightly larger than the value of $\left||E_2|\right|$ for the optimal method.

Now, let's have a look at *Heun's second order method*, for which $c_2 = 1$. Putting this value of c_2 into the general expression for $\left||E_2|\right|_2^2$, we get

$$\left||E_2|\right|_2^2 = \left(\frac{1}{2}\left(\frac{1}{2} - \frac{1}{3}\right)\right)^2 + \frac{1}{36},$$

$$\left||E_2|\right|_2^2 = \frac{1}{144} + \frac{1}{36} = \frac{5}{144},$$

$$\left||E_2|\right|_2^2 = 0.034722222.$$

So, the value of $\left||E_2|\right|$ for *Heun's second order method* is

$$\left||E_2|\right| = 0.18633899.$$

Here, we can clearly see that the value of $||E_2||$ for *Heun's second order method* is larger than the value of $||E_2||$ for the optimal method.

Here are the results from all the methods:

Methods	Variables	Principal Error Coefficient $||E_2||$
Optimal Method	$c_2 = \frac{2}{3}$	0.166666
The Explicit Midpoint Method	$c_2 = \frac{1}{2}$	0.17179606
Heun's second order Method	$c_2 = 1$	0.18633899

From the above table, we can see that the smallest value for Principal Error Coefficient, $||E_2||$, is $||E_2|| = 0.166666$ for $c_2 = \frac{2}{3}$. So, we conclude that $c_2 = \frac{2}{3}$ gives the optimal value for two-stage, second order ERK method. However, we can see that all three methods have $||E_2||$ values that are approximately the same which means that the two standard methods are close to being optimal.

4.1.2 Third Order ERK Method Optimization

Unlike the two-stage, second order ERK method case, we know that the general three-stage, third order ERK method has three special cases.

From Chapter 2, the Principal Error Coefficient is

$$E_3 = \begin{bmatrix} \left(\frac{1}{6}\right)\left(b^T c^3 - \frac{1}{4}\right) \\ b^T cAc - \frac{1}{8} \\ \left(\frac{1}{2}\right)\left(b^T Ac^2 - \frac{1}{12}\right) \\ b^T A^2 c - \frac{1}{24} \end{bmatrix}.$$

Then, the square of the 2-norm of E_3 is

$$||E_3||_2^2 = \left(\left(\frac{1}{6}\right)\left(b^T c^3 - \frac{1}{4}\right)\right)^2 + \left(b^T cAc - \frac{1}{8}\right)^2 + \left(\left(\frac{1}{2}\right)\left(b^T Ac^2 - \frac{1}{12}\right)\right)^2$$

$$+ \left(b^T A^2 c - \frac{1}{24}\right)^2.$$

We now consider the three cases for a three stage, third order ERK method that we presented in Chapter 2.

Case 1. A 2-parameter family of three-stage third order ERK methods has the following tableau with the conditions $c_2 \neq 0, \frac{2}{3}, c_3; c_3 \neq 0, c_2$.

0	0	0	0
c_2	c_2	0	0
c_3	$\frac{c_3(c_3 - 3c_2 + 3c_2^2)}{c_2(3c_2 - 2)}$	$\frac{c_3(c_2 - c_3)}{c_2(3c_2 - 2)}$	0
	$\frac{2 - 3(c_2 + c_3) + 6c_2c_3}{6c_2c_3}$	$\frac{c_3 - \frac{2}{3}}{2c_2(c_3 - c_2)}$	$\frac{\frac{2}{3} - c_2}{2c_3(c_3 - c_2)}$

As we can see, c_2 and c_3 are the two independent variables and the rest of the coefficients are dependent upon them.

For this case,

$$b = \left[\frac{2 - 3(c_2 + c_3) + 6c_2c_3}{6c_2c_3} \quad \frac{c_3 - \frac{2}{3}}{2c_2(c_3 - c_2)} \quad \frac{\frac{2}{3} - c_2}{2c_3(c_3 - c_2)}\right],$$

$$c = [0 \quad c_2 \quad c_3]$$

and the matrix

$$A = \begin{bmatrix} 0 & 0 & 0 \\ c_2 & 0 & 0 \\ \frac{c_3(c_3 - 3c_2 + 3c_2^2)}{c_2(3c_2 - 2)} & \frac{c_3(c_2 - c_3)}{c_2(3c_2 - 2)} & 0 \end{bmatrix}.$$

To find the optimal c_2 and c_3 values, we need to choose c_2 and c_3 to minimize the Principal Error Coefficient by minimizing the square of the 2-norm of E_3. After

substituting in E_3 with the above values for b, A and c, we get an expression which depends upon c_2 and c_3.

$$||E_3||_2^2 = \left(\left(\frac{1}{12}\right)\left(\frac{2}{3}(c_3 + c_2) - c_2c_3 - \frac{1}{2}\right)\right)^2 + \left(\frac{c_3}{6} - \frac{1}{8}\right)^2 + \left(\left(\frac{1}{12}\right)\left(c_2 - \frac{1}{2}\right)\right)^2 + \frac{1}{576}$$

Using the optimization software discussed in Chapter 3 to find the optimal values of c_2 and c_3 by minimizing the above expression, we get

$$c_2 = 0.49650476,$$

$$c_3 = 0.75174749,$$

yielding

$$\left|\left|E_3\right|\right|_2^2 = 0.0017479988.$$

So, the value for $\left|\left|E_3\right|\right|$ is

$$\left|\left|E_3\right|\right| = 0.041809076.$$

As mentioned earlier in this thesis, there are several well-known third order ERK methods. We consider two classic third order Runge-Kutta methods, which are examples of this case, *Heun's third order method* and *Ralston's third order method*. We compare our results with these methods by observing what $\left|\left|E_3\right|\right|$ values they have.

We start with *Heun's third order method* which has $c_2 = \frac{1}{3}$ and $c_3 = \frac{2}{3}$. Now, by putting these values of c_2 and c_3 in $\left|\left|E_3\right|\right|_2^2$, we get

$$\left|\left|E_3\right|\right|_2^2 = 0.0021433470.$$

So, the value of $\left|\left|E_3\right|\right|$ for *Heun's third order method* is

$$\left|\left|E_3\right|\right| = 0.046296296.$$

Here, we can clearly see that the value of $\left|\left|E_3\right|\right|$ by using *Heun's third order method* is slightly larger than the value of $\left|\left|E_3\right|\right|$ for the optimal method.

Next, we can consider *Ralston's third order method* which has $c_2 = \frac{1}{2}$ and $c_3 = \frac{3}{4}$. These values are reported in [Butc87] and agree quite closely with the values of the optimal method mentioned above. Now, by putting these values of c_2 and c_3 in $\left|\left|E_3\right|\right|_2^2$, we get

$$\left|\left|E_3\right|\right|_2^2 = 0.0017481674.$$

So, the value of $\left|\left|E_3\right|\right|$ for *Ralston's third order method* is

$$\left|\left|E_3\right|\right| = 0.041811092.$$

The value of $\left|\left|E_3\right|\right|$ by using *Ralston's third order method* is slightly larger than the value of $\left|\left|E_3\right|\right|$ for the optimal method, but *Ralston's third order method* is almost optimal.

Case 2. A 1-parameter family of three-stage third order ERK methods has the following tableau with the condition $b_3 \neq 0$.

0	0	0	0
$\frac{2}{3}$	$\frac{2}{3}$	0	0
0	$-\frac{1}{4b_3}$	$\frac{1}{4b_3}$	0
	$\frac{1}{4} - b_3$	$\frac{3}{4}$	b_3

As we can see, b_3 is the independent variable and the rest of the coefficients are dependent upon it.

For this case,

$$b = \left[\frac{1}{4} - b_3 \quad \frac{3}{4} \quad b_3\right],$$

$$c = \begin{bmatrix} 0 & \frac{2}{3} & 0 \end{bmatrix}$$

and the matrix,

$$A = \begin{bmatrix} 0 & 0 & 0 \\ \frac{2}{3} & 0 & 0 \\ -\frac{1}{4b_3} & \frac{1}{4b_3} & 0 \end{bmatrix}.$$

After substituting in E_3 with the above values for b, A and c, instead of getting an expression which depends upon b_3, we get a constant which is independent of the b_3 value.

$$\left||E_3|\right|_2^2 = \left(-\frac{1}{216}\right)^2 + \left(-\frac{1}{8}\right)^2 + \left(\frac{1}{72}\right)^2 + \left(-\frac{1}{24}\right)^2$$

$$\left||E_3|\right|_2^2 = 0.017575445$$

So, the value for $\left||E_3|\right|$ is

$$\left||E_3|\right| = 0.13257242.$$

As $\left||E_3|\right|$ does not depend upon b_3, the choice of b_3 is arbitrary. So, the value used in the software is $b_3 = \frac{1}{8}$.

Case 3. A 1-parameter family of three-stage third order ERK methods has the following tableau with the condition $b_3 \neq 0$.

0	0	0	0
$\frac{2}{3}$	$\frac{2}{3}$	0	0
$\frac{2}{3}$	$\frac{8b_3 - 3}{12b_3}$	$\frac{1}{4b_3}$	0
	$\frac{1}{4}$	$\frac{3}{4} - b_3$	b_3

As we can see, b_3 is the independent variable and the rest of the coefficients are dependent upon it.

For this case,

$$b = \begin{bmatrix} \frac{1}{4} & \frac{3}{4} - b_3 & b_3 \end{bmatrix},$$

$$c = \begin{bmatrix} 0 & \frac{2}{3} & \frac{2}{3} \end{bmatrix}$$

and the matrix,

$$A = \begin{bmatrix} 0 & 0 & 0 \\ \frac{2}{3} & 0 & 0 \\ \frac{8b_3 - 3}{12b_3} & \frac{1}{4b_3} & 0 \end{bmatrix}.$$

After substituting in E_3 with the above values for b, A and c, instead of getting an expression which depends upon b_3, we get a constant which is independent of the b_3 value.

$$\left|\left|E_3\right|\right|_2^2 = \left(-\frac{1}{216}\right)^2 + \left(-\frac{1}{72}\right)^2 + \left(\frac{1}{72}\right)^2 + \left(-\frac{1}{24}\right)^2$$

$$\left|\left|E_3\right|\right|_2^2 = 0.0021433470$$

So, the value for $\left|\left|E_3\right|\right|$ is

$$\left|\left|E_3\right|\right| = 0.046296296.$$

As $\left|\left|E_3\right|\right|$ does not depend upon b_3, the choice of b_3 is arbitrary. So, the value used in the software is $b_3 = \frac{3}{8}$.

Here are the results from all the methods:

Methods	Variables		Principal Error Coefficient $\left\|\left\|E_3\right\|\right\|$
Case 1 Optimal	$c_2 = 0.49650476$	$c_3 = 0.75174749$	0.041809076
Heun's third order Method	$c_2 = \frac{1}{3}$	$c_3 = \frac{2}{3}$	0.046296296
Ralston's third order Method	$c_2 = \frac{1}{2}$	$c_3 = \frac{3}{4}$	0.041811092
Case 2	N/A		0.13257242
Case 3	N/A		0.046296296

From the above table, we can see that the smallest value for Principal Error Coefficient, $\left\|\left\|E_3\right\|\right\|$, is $\left\|\left\|E_3\right\|\right\| = 0.041809076$ for the values $c_2 = 0.49650476$ and $c_3 = 0.75174749$. So, we conclude that Case 1 with values $c_2 = 0.49650476$ and $c_3 = 0.75174749$ is the optimal case for three-stage, third order ERK methods. However, we can see that four of the five methods have $\left\|\left\|E_3\right\|\right\|$ values that are almost the same. This means that the standard methods are close to optimal. It is only the Case 2 method which has a substantially larger $\left\|\left\|E_3\right\|\right\|$ value.

4.1.3 Fourth Order ERK Method Optimization

From Section 2.2, we know that the general four-stage, fourth order ERK method has five special cases.

From Chapter 2, the Principal Error Coefficient is

$$E_4 = \begin{bmatrix} \left(\frac{1}{24}\right)\left(b^T c^4 - \frac{1}{5}\right) \\ \left(\frac{1}{2}\right)\left((bc^2)^T Ac - \frac{1}{10}\right) \\ \left(\frac{1}{2}\right)\left((bc)^T Ac^2 - \frac{1}{15}\right) \\ (bc)^T A^2 c - \frac{1}{30} \\ \left(\frac{1}{2}\right)\left(b^T (Ac)^2 - \frac{1}{20}\right) \\ \left(\frac{1}{6}\right)\left(b^T Ac^3 - \frac{1}{20}\right) \\ b^T Ac(Ac) - \frac{1}{40} \\ \left(\frac{1}{2}\right)\left(b^T A^2 c^2 - \frac{1}{60}\right) \\ b^T A^3 c - \frac{1}{120} \end{bmatrix}.$$

The square of the 2-norm of E_4 is

$$\begin{aligned} ||E_4||_2^2 = &\left(\left(\frac{1}{24}\right)\left(b^T c^4 - \frac{1}{5}\right)\right)^2 + \left(\left(\frac{1}{2}\right)\left((bc^2)^T Ac - \frac{1}{10}\right)\right)^2 + \left(\left(\frac{1}{2}\right)\left((bc)^T Ac^2 - \frac{1}{15}\right)\right)^2 \\ &+ \left((bc)^T A^2 c - \frac{1}{30}\right)^2 + \left(\left(\frac{1}{2}\right)\left(b^T (Ac)^2 - \frac{1}{20}\right)\right)^2 + \left(\left(\frac{1}{6}\right)\left(b^T Ac^3 - \frac{1}{20}\right)\right)^2 \\ &+ \left(b^T Ac(Ac) - \frac{1}{40}\right)^2 + \left(\left(\frac{1}{2}\right)\left(b^T A^2 c^2 - \frac{1}{60}\right)\right)^2 + \left(b^T A^3 c - \frac{1}{120}\right)^2. \end{aligned}$$

Case 1. A 2-parameter family of four-stage, fourth order ERK methods has the following tableau with the conditions $0, c_2, c_3, 1$ all distinct; $c_2 \neq \frac{1}{2}$ and $3 - 4(c_2 + c_3) + 6c_2 c_3 \neq 0$.

0	0	0	0	0
c_2	c_2	0	0	0
c_3	$\frac{c_3(3c_2 - c_3 - 4c_2^2)}{2c_2(1-2c_2)}$	$\frac{c_3(c_3 - c_2)}{2c_2(1-2c_2)}$	0	0
1	a_{41}	a_{42}	a_{43}	0
	$\frac{1 - 2(c_2 + c_3) + 6c_2 c_3}{12 c_2 c_3}$	$\frac{2c_3 - 1}{12c_2(c_3 - c_2)(1 - c_2)}$	$\frac{1 - 2c_2}{12c_3(c_3 - c_2)(1 - c_3)}$	$\frac{3 - 4(c_2 + c_3) + 6c_2 c_3}{12(1 - c_2)(1 - c_3)}$

where

$$a_{41} = \frac{c_3^2(12c_2^2 - 12c_2 + 4) - c_3(12c_2^2 - 15c_2 + 5) + (4c_2^2 - 6c_2 + 2)}{2c_2c_3(3 - 4(c_2 + c_3) + 6c_2c_3)},$$

$$a_{42} = \frac{(-4c_3^2 + 5c_3 + c_2 - 2)(1 - c_2)}{2c_2(c_3 - c_2)(3 - 4(c_2 + c_3) + 6c_2c_3)},$$

$$a_{43} = \frac{(1 - 2c_2)(1 - c_3)(1 - c_2)}{c_3(c_3 - c_2)(3 - 4(c_2 + c_3) + 6c_2c_3)}.$$

As we can see, c_2 and c_3 are the two independent variables and the rest of the coefficients are dependent upon them.

For the expressions in E_4 for this case,

$$b = \left[\frac{1 - 2(c_2 + c_3) + 6c_2c_3}{12c_2c_3} \quad \frac{2c_3 - 1}{12c_2(c_3 - c_2)(1 - c_2)} \quad \frac{1 - 2c_2}{12c_3(c_3 - c_2)(1 - c_3)} \quad \frac{3 - 4(c_2 + c_3) + 6c_2c_3}{12(1 - c_2)(1 - c_3)}\right],$$

$$c = [0 \quad c_2 \quad c_3 \quad 1],$$

and the matrix

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 \\ c_2 & 0 & 0 & 0 \\ \frac{c_3(3c_2 - c_3 - 4c_2^2)}{2c_2(1 - 2c_2)} & \frac{c_3(c_3 - c_2)}{2c_2(1 - 2c_2)} & 0 & 0 \\ a_{41} & a_{42} & a_{43} & 0 \end{bmatrix},$$

where a_{41}, a_{42} and a_{43} are mentioned above.

After substituting for b, A and c, we get an expression which depends upon c_2 and c_3, but it is too complicated to show here. Using the optimization software to find the optimal values of c_2 and c_3 by minimizing the square of the 2-norm of E_4, we get

$$c_2 = 0.35774159,$$

$$c_3 = 0.59148821,$$

yielding

$$\left||E_4|\right|_2^2 = 0.00014345932.$$

So, the value for $\left||E_4|\right|$ is

$$\left||E_4|\right| = 0.011977450.$$

The above c_2 and c_3 values agree reasonably well with the optimal coefficient values reported in [Butc87] where

$$c_2 = 0.371615,$$

$$c_3 = \frac{3}{5} = 0.6,$$

yielding

$$\left||E_4|\right|_2^2 = 0.00014452215.$$

So, the value for $\left||E_4|\right|$ is

$$\left||E_4|\right| = 0.012021736.$$

In [Rals62], the optimal values are reported as

$$c_2 = 0.4,$$

$$c_3 = 0.455737,$$

yielding

$$\left||E_4|\right|_2^2 = 0.00018779888.$$

So, the value for $\left||E_4|\right|$ is

$$\left||E_4|\right| = 0.013703973.$$

As mentioned in Chapter 2, there is a well-known fourth-order ERK method known as the *3/8-Rule method*; we compare our optimal method with this method by finding its $\left||E_4|\right|$ value.

The *3/8-Rule method* is derived from Case 1 with $c_2 = \frac{1}{3}$ and $c_3 = \frac{2}{3}$. By putting these values foe c_2 and c_3 in $\left||E_4|\right|_2^2$, we get

$$\left||E_4|\right|_2^2 = 0.00016051287.$$

So, the value of $\left||E_4|\right|$ for *3/8-Rule method* is

$$\left||E_4|\right| = 0.012669367.$$

Here, we can clearly see that the value of $\left||E_4|\right|$ by using the *3/8-Rule method* is slightly larger than the value of $\left||E_4|\right|$ for the optimal method.

Case 2. A 1-parameter family of four-stage fourth order ERK methods has the following tableau with the conditions $c_2 = c_3 = \frac{1}{2}, b_3 \neq 0$.

0	0	0	0	0
$\frac{1}{2}$	$\frac{1}{2}$	0	0	0
$\frac{1}{2}$	$\frac{3b_3 - 1}{6b_3}$	$\frac{1}{6b_3}$	0	0
1	0	$1 - 3b_3$	$3b_3$	0
	$\frac{1}{6}$	$\frac{2}{3} - b_3$	b_3	$\frac{1}{6}$

As we can see, b_3 is the only independent variable and the rest of the coefficients are dependent upon it.

For this case,

$$b = \left[\frac{1}{6} \quad \frac{2}{3} - b_3 \quad b_3 \quad \frac{1}{6}\right],$$

$$c = \begin{bmatrix} 0 & \frac{1}{2} & \frac{1}{2} & 1 \end{bmatrix},$$

and the matrix

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 & 0 \\ \frac{3b_3 - 1}{6b_3} & \frac{1}{6b_3} & 0 & 0 \\ 0 & 1 - 3b_3 & 3b_3 & 0 \end{bmatrix}.$$

After substituting for b, A and c, we get an expression which depends upon b_3, but it is too complicated to show here. Using the optimization software to find the optimal value of b_3 to minimize the square of the 2-norm of E_4, we get

$$b_3 = 0.83316441,$$

yielding

$$\left|\left|E_4\right|\right|_2^2 = 0.00017132040.$$

So, the value for $\left|\left|E_4\right|\right|$ is

$$\left|\left|E_4\right|\right| = 0.013088942.$$

The corresponding value reported in [Rals62] for this case is

$$b_3 = \frac{5}{3} = 1.666666,$$

yielding

$$\left|\left|E_4\right|\right|_2^2 = 0.00017566068.$$

So, the value for $\left|\left|E_4\right|\right|$ is

$$\left|\left|E_4\right|\right| = 0.013253704.$$

As mentioned in Chapter 2, there is a well-known fourth-order ERK method known as *the classical Runge-Kutta method* which is widely used and is the most well-known of all Runge-Kutta methods. We find its $\left|\left|E_4\right|\right|$ value.

The classical Runge-Kutta method is obtained from Case 2 when $b_3 = \frac{1}{3}$. By putting this value of b_3 into $\left|\left|E_4\right|\right|_2^2$, we get

$$\left|\left|E_4\right|\right|_2^2 = 0.0002103829.$$

So, the value of $\left|\left|E_4\right|\right|$ for *the classical Runge-Kutta method* is

$$\left|\left|E_4\right|\right| = 0.014504582.$$

Here, we can clearly see that the value of $\left|\left|E_4\right|\right|$ by using *the classical Runge-Kutta method* is slightly larger than the value of $\left|\left|E_4\right|\right|$ for the optimal method.

Case 3. A 1-parameter family of four-stage fourth order ERK methods has the following tableau with the conditions $c_2 = \frac{1}{2}, c_3 = 0, b_3 \neq 0$.

0	0	0	0	0
$\frac{1}{2}$	$\frac{1}{2}$	0	0	0
0	$-\frac{1}{12b_3}$	$\frac{1}{12b_3}$	0	0
1	$-\frac{1}{2} - 6b_3$	$\frac{3}{2}$	$6b_3$	0
	$\frac{1}{6} - b_3$	$\frac{2}{3}$	b_3	$\frac{1}{6}$

As we can see, b_3 is the only independent variable and the rest of the coefficients are dependent upon it.

For this case,

$$b = \left[\frac{1}{6} - b_3 \quad \frac{2}{3} \quad b_3 \quad \frac{1}{6}\right],$$

$$c = \begin{bmatrix} 0 & \frac{1}{2} & 0 & 1 \end{bmatrix},$$

and the matrix

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 & 0 \\ -\frac{1}{12b_3} & \frac{1}{12b_3} & 0 & 0 \\ -\frac{1}{2} - 6b_3 & \frac{3}{2} & 6b_3 & 0 \end{bmatrix}.$$

After substituting for b, A and c, we get an expression which depends upon b_3, but it is too complicated to show here. Using the optimization software to find the optimal value of b_3 by minimizing the square of the 2-norm of E_4, we get

$$b_3 = -0.03968255,$$

yielding

$$\left|\left|E_4\right|\right|_2^2 = 0.00093086902.$$

So, the value for $\left|\left|E_4\right|\right|$ is

$$\left|\left|E_4\right|\right| = 0.030510146.$$

The corresponding value from [Rals62] for this case is

$$b_3 = -\frac{5}{78} = -0.064103,$$

yielding

$$\left|\left|E_4\right|\right|_2^2 = 0.0010003149.$$

So, the value for $\left|\left|E_4\right|\right|$ is

$$\left|\left|E_4\right|\right| = 0.031627756.$$

Case 4. A 1-parameter family of four-stage fourth order ERK methods has the following tableau with the conditions $c_2 = 1, c_3 = \frac{1}{2}, b_4 \neq 0$.

0	0	0	0	0
1	1	0	0	0
$\frac{1}{2}$	$\frac{3}{8}$	$\frac{1}{8}$	0	0
1	$1-\frac{1}{4b_4}$	$-\frac{1}{12b_4}$	$\frac{1}{3b_4}$	0
	$\frac{1}{6}$	$\frac{1}{6}-b_4$	$\frac{2}{3}$	b_4

As we can see, b_4 is the only independent variable and the rest of the coefficients are dependent upon it.

For this case,

$$b = \left[\frac{1}{6} \quad \frac{1}{6} - b_4 \quad \frac{2}{3} \quad b_4\right],$$

$$c = \left[0 \quad 1 \quad \frac{1}{2} \quad 1\right],$$

and the matrix

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ \frac{3}{8} & \frac{1}{8} & 0 & 0 \\ 1-\frac{1}{4b_4} & -\frac{1}{12b_4} & \frac{1}{3b_4} & 0 \end{bmatrix}.$$

After substituting for b, A and c, we get an expression which depends upon b_4, but it is too complicated to show here. Using the optimization software to find the optimal value of b_4 by minimizing the square of the 2-norm of E_4, we get

$$b_4 = 0.17543856,$$

yielding

$$\left|\left|E_4\right|\right|_2^2 = 0.00047513985.$$

So, the value for $\left|\left|E_4\right|\right|$ is

$$\left|\left|E_4\right|\right| = 0.021797702.$$

The corresponding value reported in [Rals62] for this case is

$$b_4 = \frac{10}{51} = 0.196078,$$

yielding

$$\left|\left|E_4\right|\right|_2^2 = 0.00047947996.$$

So, the value for $\left|\left|E_4\right|\right|$ is

$$\left|\left|E_4\right|\right| = 0.021897031.$$

Case 5. A 1-parameter family of four-stage fourth order ERK methods has the following tableau with the conditions $c_2 \neq 0, c_3 = \frac{1}{2}, b_2 = 0$.

0	0	0	0	0
c_2	c_2	0	0	0
$\frac{1}{2}$	$\frac{4c_2 - 1}{8c_2}$	$\frac{1}{8c_2}$	0	0
1	$\frac{1 - 2c_2}{2c_2}$	$-\frac{1}{2c_2}$	2	0
	$\frac{1}{6}$	0	$\frac{2}{3}$	$\frac{1}{6}$

As we can see, c_2 is the only independent variable and the rest of the coefficients are dependent upon it.

For this case,

$$b = \left[\frac{1}{6} \quad 0 \quad \frac{2}{3} \quad \frac{1}{6}\right],$$

$$c = \left[0 \quad c_2 \quad \frac{1}{2} \quad 1\right],$$

and the matrix

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 \\ c_2 & 0 & 0 & 0 \\ \dfrac{4c_2 - 1}{8c_2} & \dfrac{1}{8c_2} & 0 & 0 \\ \dfrac{1 - 2c_2}{2c_2} & -\dfrac{1}{2c_2} & 2 & 0 \end{bmatrix}.$$

After substituting for b, A and c, we get an expression which depends upon c_2, but it is too complicated to show here. Using the optimization software to find the optimal value of c_2 by minimizing the square of the 2-norm of E_4, we get

$$c_2 = 0.39999999,$$

yielding

$$\left||E_4|\right|_2^2 = 0.00016372492.$$

So, the value for $\left||E_4|\right|$ is

$$\left||E_4|\right| = 0.012795504.$$

Here are the results from all the cases:

Methods	Variables		Principal Error Coefficient $\left\|\|E_4\|\right\|$
Case 1 Optimal	$c_2 = 0.35774159$	$c_3 = 0.59148821$	0.011977450
3/8-Rule	$c_2 = \frac{1}{3}$	$c_3 = \frac{2}{3}$	0.012669367
Case 2 Optimal	$b_3 = 0.83316441$		0.013088942
The classical Runge-Kutta Method	$b_3 = \frac{1}{3}$		0.014504582
Case 3 Optimal	$b_3 = -0.03968255$		0.030510146
Case 4 Optimal	$b_4 = 0.17543856$		0.021797702
Case 5 Optimal	$c_2 = 0.39999999$		0.012795504

From the above table, we can clearly see that the smallest value for Principal Error Coefficient, $||E_4||$, is $||E_4|| = 0.011977450$. So, we conclude that Case 1 with values $c_2 = 0.35774159$ and $c_3 = 0.59148821$ is the optimal case for four-stage, fourth order ERK methods. However, we can see that Case 1, Case 2, Case 5, and the two standard methods have similar $||E_4||$ values so all of these methods are close to optimal. It is only the Case 3 and Case 4 methods that have substantially larger $||E_4||$ values.

4.2 Experimental Verification of Order of Convergence

In this section, we experimentally verify the order of convergence of some of the methods based on some numerical experiments performed on a test set of ODEs. For each method, we provide a table that has error ratios generated by the software for the test set, for several stepsizes.

Let us first define the term error ratio. For each method and test problem, we use the software to step from t_0 to t_f and we compute the exact error at t_f. The ratio between the error from the previous stepsize and the error from the current stepsize is known as the error ratio. Here, we are decreasing the stepsize by a factor of 2 to determine its effect on the error ratios. For a two-stage, second order ERK method, if the stepsize is decreased by a factor of 2, then the error should reduce by a factor of 4, approximately. This is because a second order method has an error that is $O(h^2)$. So, the error ratio for second order methods should be 4. For a three-stage, third order ERK method, if the stepsize is decreased by a factor of 2, then the error should reduce by a factor of 8, approximately, which would make the error ratio equal to 8 since the error for a third order method is $O(h^3)$. Similarly, for a four-stage, fourth order ERK method, if the stepsize is decreased by a factor of 2, then the error should reduce by a factor of 16, approximately, which would make the error ratio equal to 16 since the error for the fourth order method is $O(h^4)$.

We use the following IVODEs as the test sets.

Name	IVODE	Initial Condition	Exact Solution
IVODE 1	$y' = -2xy^2$	$y(0) = 1$	$y(x) = \frac{1}{1+x^2}$
IVODE 2	$y' = \left(-\frac{1}{2}\right)y^3$	$y(0) = 1$	$y(x) = \frac{1}{\sqrt{1+\mathrm{x}}}$
IVODE 3	$y' = \left(\frac{1}{4}\right)\left(1 - \frac{y}{20}\right)y$	$y(0) = 1$	$y(x) = \frac{20}{1+19e^{-\frac{x}{4}}}$
IVODE 4	$y' = -\alpha y - e^{-\alpha x}\sin x$	$y(0) = 1$	$y(x) = e^{-\alpha x}\cos x$, where $\alpha = 0.1$

The tables for all the cases of the optimal, two-stage, second order, three-stage, third order, and four-stage, fourth order ERK methods for the above IVODEs for stepsizes $\frac{1}{4}$, $\frac{1}{8}$, $\frac{1}{16}$, $\frac{1}{32}$ and $\frac{1}{64}$ are as follows:

Error Ratios for optimal, two-stage, second order ERK Method

IVODE / Stepsize	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{64}$
IVODE 1	31.28	7.06	2.12	3.37	3.74
IVODE 2	4.76	4.42	4.21	4.10	4.05
IVODE 3	3.83	3.92	3.96	3.98	3.99
IVODE 4	4.50	4.26	4.13	4.07	4.03

Error Ratios for optimal, three-stage, third order ERK Method, Case 1

IVODE / Stepsize	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{64}$
IVODE 1	54.21	13.99	4.09	6.67	7.44
IVODE 2	9.81	8.93	8.47	8.23	8.12
IVODE 3	7.66	7.83	7.91	7.96	7.98
IVODE 4	8.06	8.02	8.01	8.00	8.00

Error Ratios for optimal, three-stage, third order ERK Method, Case 2

IVODE / Stepsize	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{64}$
IVODE 1	10.38	9.08	8.65	8.32	8.16
IVODE 2	13.34	10.45	9.15	8.55	8.27
IVODE 3	7.95	7.99	8.00	8.00	8.00
IVODE 4	8.05	8.01	8.00	8.00	8.00

Error Ratios for optimal, three-stage, third order ERK Method, Case 3

IVODE / Stepsize	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{64}$
IVODE 1	6.54	8.02	8.144	8.10	8.05
IVODE 2	10.30	9.15	8.56	8.28	8.14
IVODE 3	7.66	7.83	7.91	7.96	7.98
IVODE 4	8.05	8.01	8.00	8.00	8.00

Error Ratios for optimal, four-stage, fourth order ERK Method, Case 1

IVODE / Stepsize	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{64}$
IVODE 1	33.36	24.76	20.15	18.01	16.98
IVODE 2	12.78	11.17	14.70	15.55	15.82
IVODE 3	15.27	15.63	15.81	15.90	15.93
IVODE 4	16.08	16.02	16.01	16.00	16.00

Error Ratios for optimal, four-stage, fourth order ERK Method, Case 2

IVODE / Stepsize	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{64}$
IVODE 1	4.64	12.17	14.95	15.63	15.85
IVODE 2	17.11	17.13	16.68	16.36	16.18
IVODE 3	15.28	15.63	15.81	15.91	15.94
IVODE 4	16.13	16.04	16.01	16.01	16.00

Error Ratios for optimal, four-stage, fourth order ERK Method, Case 3

IVODE / Stepsize	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{64}$
IVODE 1	14.15	19.75	18.35	17.26	16.64
IVODE 2	9.07	17.81	18.08	17.31	16.72
IVODE 3	15.40	15.70	15.85	15.92	15.92
IVODE 4	16.13	16.04	16.01	16.01	16.00

Error Ratios for optimal, four-stage, fourth order ERK Method, Case 4

IVODE / Stepsize	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{64}$
IVODE 1	10.04	15.23	15.91	16.01	16.02
IVODE 2	8.71	14.15	15.40	15.77	15.90
IVODE 3	15.19	15.59	15.79	15.90	15.94
IVODE 4	16.05	15.99	15.99	15.99	16.00

Error Ratios for optimal, four-stage, fourth order ERK Method, Case 5

IVODE / Stepsize	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{64}$
IVODE 1	1.044	15.23	15.91	16.01	16.02
IVODE 2	8.71	14.15	15.40	15.77	15.90
IVODE 3	15.19	15.59	15.79	15.90	15.94
IVODE 4	16.05	15.99	15.99	15.99	16.00

Looking at the tables above, we can see that when the stepsize is decreased by a factor of 2 at each step, the error ratios for each method are approximately as follows:

Two-stage second order ERK methods: 4

Three-stage third order ERK methods: 8

Four-stage fourth order ERK methods: 16

For bigger stepsizes, the error ratios might not be close to the expected values. But the smaller the stepsize gets, the more the error ratios approach the expected results. So, for the smallest stepsize results in the table, you can see that the ratios are approximately the same as we expect from the theory. This confirms that the ERK methods are correct since each method provides the order that is expected.

4.3 Comparison of standard and optimal ERK methods: Accuracy and Efficiency

4.3.1 Accuracy

Here we consider some standard methods along with the optimal 2nd, 3rd and 4th order ERK methods and compare them on several test problems. For each IVODE, we provide a table which has all the methods grouped according to their order. We check the accuracy of all the methods for the stepsize $\frac{1}{64}$. For the error results for each order, we take the smallest error

and then divide all the other errors from all the methods by the smallest error. The results are presented in the 'Rel. to Min.' column of the tables. The method with '1.0' as the value of 'Rel. to Min.' is the most accurate method of that order for that IVODE.

We first consider the IVODE,

$$y' = -2xy^2$$

with initial value, $y(0) = 1$ and exact solution,

$$y(x) = \frac{1}{1+x^2}.$$

Using the software discussed in Chapter 3, we apply all the methods to this IVODE to find the approximate solutions and corresponding errors. The results are shown in the table below:

Methods	**Errors**	**Rel. to Min.**
Explicit Midpoint Method (Second Order)	7.19×10^{-06}	2.4
Heun's Second Order Method	2.34×10^{-05}	7.7
Optimal Second Order ERK Method	3.05×10^{-06}	1.0
Heun's Third Order Method	4.06×10^{-08}	2.1
Ralston's Third Order Method	2.11×10^{-08}	1.1
Optimal Third Order ERK Method Case 1	1.90×10^{-08}	1.0
Optimal Third Order ERK Method Case 2	3.78×10^{-07}	19.8
Optimal Third Order ERK Method Case 3	1.29×10^{-07}	6.8
The Classical Runge-Kutta Method	4.07×10^{-10}	1.4
3/8 Rule Method (Fourth Order)	4.38×10^{-10}	1.6
Optimal Fourth Order ERK Method Case 1	2.81×10^{-10}	1.0
Optimal Fourth Order ERK Method Case 2	5.34×10^{-10}	1.9
Optimal Fourth Order ERK Method Case 3	4.31×10^{-10}	1.5
Optimal Fourth Order ERK Method Case 4	3.21×10^{-09}	11.4
Optimal Fourth Order ERK Method Case 5	3.21×10^{-09}	11.4

Table 4.3.1: Errors and Rel. to Min. values for all the ERK methods applied to IVODE $y' = -2xy^2$, $y(0) = 1$.

We next consider the IVODE,

$$y' = \left(-\frac{1}{2}\right) y^3$$

with initial value, $y(0) = 1$ and exact solution,

$$y(x) = \frac{1}{\sqrt{1+x}}.$$

We apply all the methods to this IVODE to find the approximate solutions and corresponding errors. The results are shown in the table below:

Methods	**Errors**	**Rel. to Min.**
Explicit Midpoint Method (Second Order)	9.58×10^{-06}	1.8
Heun's Second Order Method	5.43×10^{-06}	1.0
Optimal Second Order ERK Method	8.20×10^{-06}	1.5
Heun's Third Order Method	5.31×10^{-08}	1.4
Ralston's Third Order Method	3.67×10^{-08}	1.0
Optimal Third Order ERK Method Case 1	3.67×10^{-08}	1.0
Optimal Third Order ERK Method Case 2	1.03×10^{-07}	2.8
Optimal Third Order ERK Method Case 3	3.71×10^{-08}	1.0
The Classical Runge-Kutta Method	1.13×10^{-11}	2.3
3/8 Rule Method	4.94×10^{-12}	1.0
Optimal Fourth Order ERK Method Case 1	3.88×10^{-11}	7.8
Optimal Fourth Order ERK Method Case 2	7.77×10^{-11}	15.7
Optimal Fourth Order ERK Method Case 3	2.52×10^{-11}	5.1
Optimal Fourth Order ERK Method Case 4	2.98×10^{-11}	6.0
Optimal Fourth Order ERK Method Case 5	2.98×10^{-11}	6.0

Table 4.3.2: Errors and Rel. to Min. values for all the ERK methods applied to IVODE $y' = \left(-\frac{1}{2}\right) y^3$, $y(0) = 1$.

We next consider the IVODE,

$$y' = \left(\frac{1}{4}\right)\left(1 - \frac{y}{20}\right)y$$

with initial value, $y(0) = 1$ and exact solution,

$$y(x) = \frac{20}{1 + 19e^{-\frac{x}{4}}}.$$

We apply all the methods to this IVODE to find the approximate solutions and corresponding errors. The results are shown in the table below:

Methods	**Errors**	**Rel. to Min.**
Explicit Midpoint Method (Second Order)	5.72×10^{-07}	1.0
Heun's Second Order Method	6.32×10^{-07}	1.1
Optimal Second Order ERK Method	5.92×10^{-07}	1.0
Heun's Third Order Method	4.67×10^{-10}	1.4
Ralston's Third Order Method	5.13×10^{-10}	1.6
Optimal Third Order ERK Method Case 1	5.13×10^{-10}	1.6
Optimal Third Order ERK Method Case 2	3.29×10^{-10}	1.0
Optimal Third Order ERK Method Case 3	5.13×10^{-10}	1.6
The Classical Runge-Kutta Method	4.30×10^{-13}	1.1
3/8 Rule Method	4.27×10^{-13}	1.1
Optimal Fourth Order ERK Method Case 1	4.04×10^{-13}	1.0
Optimal Fourth Order ERK Method Case 2	3.93×10^{-13}	1.0
Optimal Fourth Order ERK Method Case 3	4.30×10^{-13}	1.1
Optimal Fourth Order ERK Method Case 4	4.67×10^{-13}	1.2
Optimal Fourth Order ERK Method Case 5	4.67×10^{-13}	1.2

Table 4.3.3: Errors and Rel. to Min. values for all the ERK methods applied on IVODE $y' = \left(\frac{1}{4}\right)\left(1 - \frac{y}{20}\right)y$, $y(0) = 1$.

The next IVODE we consider is

$$y' = -\alpha y - e^{-\alpha x} \sin x$$

where α is a constant between 0 and 1. We choose $\alpha = 0.1$ for our computations. The initial value for this IVODE is $y(0) = 1$ and exact solution is,

$$y(x) = e^{-\alpha x} \cos x.$$

We apply all the methods to this IVODE to find the approximate solutions and corresponding errors. The results are shown in the table below:

Methods	**Errors**	**Rel. to Min.**
Explicit Midpoint Method (Second Order)	8.48×10^{-06}	3.1
Heun's Second Order Method	8.74×10^{-06}	3.2
Optimal Second Order ERK Method	2.75×10^{-06}	1.0
Heun's Third Order Method	6.80×10^{-09}	1.0
Ralston's Third Order Method	7.07×10^{-09}	1.0
Optimal Third Order ERK Method Case 1	6.89×10^{-09}	1.0
Optimal Third Order ERK Method Case 2	1.27×10^{-08}	1.9
Optimal Third Order ERK Method Case 3	1.27×10^{-08}	1.9
The Classical Runge-Kutta Method	8.88×10^{-12}	1.8
3/8 Rule Method	4.91×10^{-12}	1.0
Optimal Fourth Order ERK Method Case 1	7.47×10^{-12}	1.5
Optimal Fourth Order ERK Method Case 2	8.88×10^{-12}	1.8
Optimal Fourth Order ERK Method Case 3	8.88×10^{-12}	1.8
Optimal Fourth Order ERK Method Case 4	5.37×10^{-12}	1.1
Optimal Fourth Order ERK Method Case 5	5.37×10^{-12}	1.1

Table 4.3.4: Errors and Rel. to Min. values for all the ERK methods applied on IVODE $y' = -\alpha y - e^{-\alpha x} \sin x$ where $\alpha = 0.1$, $y(0) = 1$.

For second order methods, referring to Tables 4.3.1 and 4.3.4, we see that the optimal second order ERK method has 'Rel. to Min.' value to be '1.0', which means it gives the most accuracy in those tables. However, in the case of Tables 4.3.2 and 4.3.3, *Heun's Second Order method* and the *Explicit Midpoint method* provide the most accurate results, respectively. For third order methods, Tables 4.3.1 and 4.3.2 show that the optimal third order ERK method, case 1, provides the most accurate results. However, in case of Tables 4.3.3 and 4.3.4, the optimal third order ERK method, case 2, and *Heun's Third Order method* provide the most accuracy, respectively. For fourth order methods, Table 4.3.1 shows that the optimal fourth order ERK method, case 1, provides the most accuracy, whereas Table 4.3.3 shows that the optimal fourth order ERK method, case 2, provides the most accuracy. However, Tables 4.3.2 and 4.3.4 show that the *3/8 Rule* provides the most accurate results.

Recall from Chapter 2 that the components of the Principal Error Coefficient are multiplied by problem dependent factors in the actual error. This is the reason why we get different results for each IVODEs. For second order, the optimal second order ERK method has an Average Rel. to Min. (ARM) of 1.125, while the other two second order methods have ARMs of 2.075 and 3.25. For third order, the optimal third order ERK method, Case 1, has an ARM of 1.15, *Ralston's third order method* has an ARM of 1.175, and *Heun's third order method* and cases 2 and 3 have ARMs of 1.475, 6.375, and 2.825, respectively. For fourth order, the *3/8 Rule method* has an ARM of 1.175, the *classical Runge-Kutta method* has an ARM of 1.65, and cases 1 through 5 have ARMs of 2.825, 5.1, 2.375, 4.925, and 4.925, respectively. So, for second and third orders, the optimal method has the best ARM, while for fourth order, the *3/8 Rule method* has the best ARM. We must note that substantially more testing on a much larger test set should be performed before any specific conclusions can be made.

4.3.2 Efficiency

In this subsection, we demonstrate that better accuracy implies better efficiency. We begin with second order ERK methods, which include the optimal second order ERK method, the *Explicit Midpoint method* and *Heun's method*, but instead of determining which one gives the smallest error, we set a specific error as the goal, and determine which method is able to compute a solution with that error most quickly i.e., in the fewest number of steps. We determine this by changing the stepsizes and running the software so that all methods achieve approximately the same error.

For this analysis, we start with the IVODE,

$$y' = -\alpha y - e^{-\alpha x} \sin x,$$

where $\alpha = 0.1$. The initial value for this IVODE is $y(0) = 1$. As one can see from Section 4.3.1, the smallest stepsize that we have used is $\frac{1}{64} = 0.015625$ and for this stepsize, *Heun's method* has an error of 8.74×10^{-6} for this IVODE. We now find out how large a stepsize the other methods use while still obtaining approximately the same accuracy.

We find that for the stepsize of 0.02765, the optimal second order ERK method achieves the same accuracy as *Heun's method* using the stepsize of 0.015625, whereas the *Explicit Midpoint method* achieves that accuracy using the stepsize of 0.01578. Since $t_f = 1$, this means that the optimal second order ERK method computes a solution of the desired accuracy using ≈ 36 steps, while the other two methods require about 63 steps indicating that the optimal method is about 40% more efficient.

Next, we consider the three-stage, third order ERK methods applied to the IVODE,

$$y' = -2xy^2,$$

with initial value, $y(0) = 1$. We use all the third order ERK methods identified in Section 4.3.1. In that section, we saw that for this IVODE, the optimal third order ERK methods, Case 2, has the largest error of 3.78×10^{-7} for the stepsize of $\frac{1}{64} = 0.015625$.

We find that at the stepsize of 0.04209, the optimal third order ERK method, Case 1, gives the same accuracy as the optimal third order ERK method, Case 2 using the stepsize of 0.015625. *Ralston's Third Order method* obtains the desired accuracy with a stepsize of 0.037035, *Heun's Third Order method* obtains the desired accuracy with a stepsize of 0.03445 and the optimal third order ERK method, Case 3, obtains the desired accuracy with a stepsize of 0.0223.

Since $t_f = 1$, this means that the optimal third order ERK method, Case 1, computes a solution of the desired accuracy using ≈ 24 steps, while *Ralston's Third Order method* requires ≈ 27 steps, *Heun's Third Order method* requires ≈ 29 steps, the optimal third order ERK method, Case 3, requires ≈ 45 steps and the optimal third order ERK method, Case 2, requires ≈ 64 steps to obtain the desired accuracy. This indicates that the optimal third order ERK method, Case 1, is about 62% more efficient, *Ralston's Third Order method* is about 58% more efficient, *Heun's method* is about 55% more efficient and the optimal third order ERK method, Case 3, is about 30% more efficient than the optimal third order ERK method, Case 2.

Finally, we use the four-stage, fourth order ERK methods to compute a numerical solution for the IVODE,

$$y' = -2xy^2,$$

with initial value, $y(0) = 1$. We use all the fourth order ERK methods identified in Section 4.3.1. In that section, we saw that for this IVODE, the optimal fourth order ERK methods, Case 4 and Case 5, have the largest error of 3.21×10^{-9} for a stepsize of $\frac{1}{64} = 0.015625$.

We find that for the stepsize of 0.02885, the optimal fourth order ERK method, Case 1, gives the same accuracy as the optimal fourth order ERK methods, Case 4 and Case 5. The *classical*

Runge-Kutta method obtains that accuracy with a stepsize of 0.02601, the optimal fourth order ERK method, Case 3, obtains the desired accuracy with a stepsize of 0.02105, the *3/8 Rule method* obtains the desired accuracy with a stepsize of 0.025925, and the optimal fourth order ERK method, Case 2, obtains the desired accuracy with a stepsize of 0.02425.

Since $t_f = 1$, this means that the optimal fourth order ERK method, Case 1, computes a solution of the desired accuracy using ≈ 35 steps, while the *classical Runge-Kutta method* requires ≈ 38 steps, the optimal fourth order ERK method, Case 3, requires ≈ 48 steps, *3/8 Rule method* requires ≈ 39 steps, the optimal fourth order ERK method, Case 2, requires ≈ 41 steps and the optimal fourth order ERK method, Case 4 and Case 5, both require ≈ 64 steps to obtain the desired accuracy. This indicates that the optimal fourth order ERK method, Case 1, is about 55% more efficient, the *classical Runge-Kutta method* and the *3/8 Rule method* are about 40% more efficient, the optimal fourth order ERK method, Case 3, is about 25% more efficient and the optimal fourth order ERK method, Case 2, is about 36% more efficient than the optimal fourth order ERK method, Case 4 and Case 5.

4.4 Continuous Approximate Solutions and Corresponding Defects

As we have discussed in Chapter 2, the approximate solutions provided by the ERK methods are not continuous. In order to make the approximate solutions continuous, we use Hermite interpolation. In this section, we investigate continuous approximate solutions as well as defects for these continuous approximate solutions.

4.4.1 Continuous Approximate Solutions

In this subsection, we provide some plots of the continuous approximate solutions for some of the test IVODEs computed using the optimal ERK methods. For IVODEs with the exact solutions, we compare the continuous approximate solutions with the exact solutions.

We begin with second order ERK methods. We use the optimal second order ERK method to compute a discrete approximate solution to the IVODE,

$$y' = -\alpha y - e^{-\alpha x} \sin x,$$

where $\alpha = 0.1$, with initial value, $y(0) = 1$, and then compute a continuous approximate solution using Hermite interpolation. The graph, plotting the continuous approximate solution and exact solution, is shown in Figure 4.4.1.

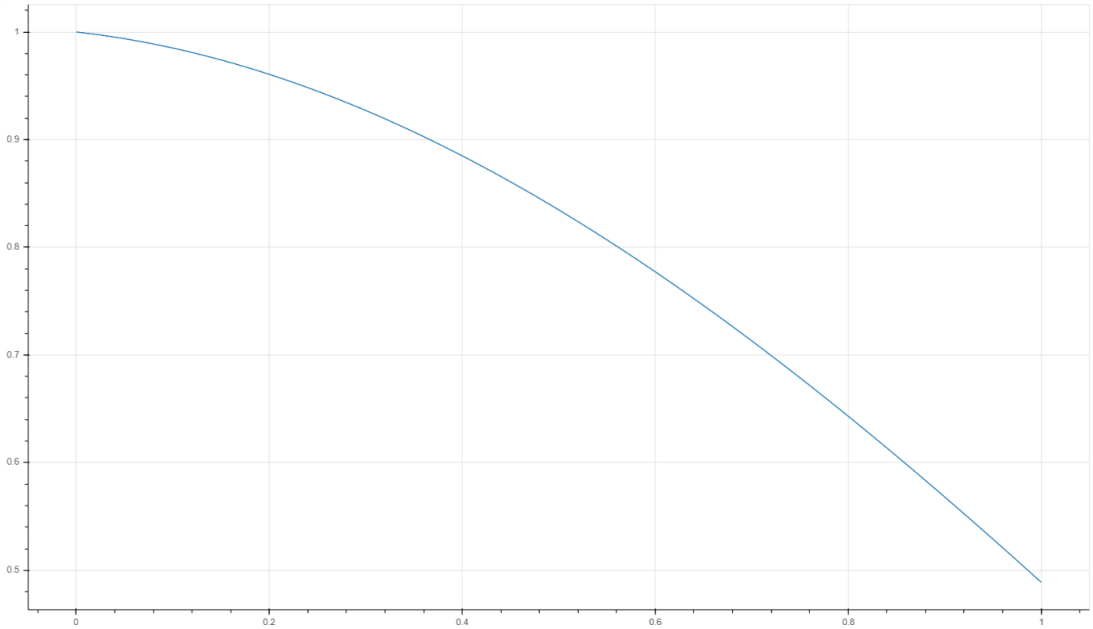

Figure 4.4.1: Exact solution and continuous approximate solution using the optimal second order ERK method and Hermite interpolation for the IVODE $y' = -\alpha y - e^{-\alpha x} \sin x$, $y(0) = 1$.

In the graph above, one can clearly see that the continuous approximate solution and exact solution agree quite well.

We next consider third order ERK methods. We use the optimal third order ERK method, Case 1, to compute a discrete approximate solution to the IVODE,

$$y' = \left(-\frac{1}{2}\right) y^3,$$

with initial value, $y(0) = 1$, and then compute a continuous approximate solution using Hermite interpolation. The graph, plotting the continuous approximate solution and exact solution, is shown in Figure 4.4.2.

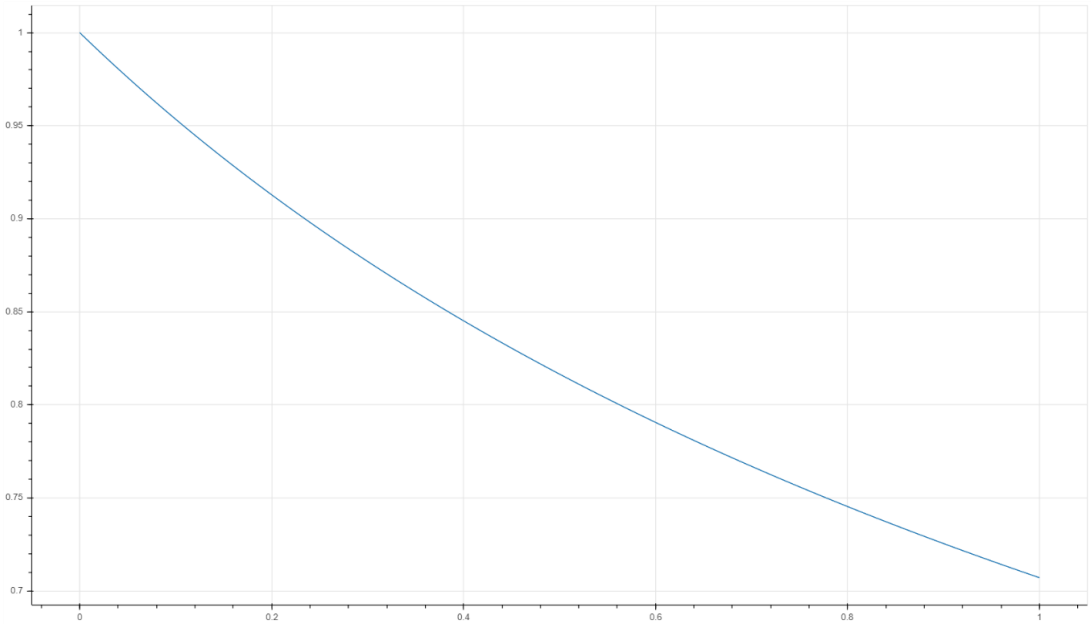

Figure 4.4.2: Exact solution and continuous approximate solution using the optimal third order ERK method, case 1, and Hermite interpolation for the IVODE $y' = \left(-\frac{1}{2}\right) y^3,\ y(0) = 1$.

In the graph above, one can clearly see that the continuous approximate solution and exact solution agree quite well.

We next consider fourth order ERK methods. We use the optimal fourth order ERK method, case 1, to compute a discrete approximate solution to the IVODE,

$$y' = -2xy^2,$$

with initial value, $y(0) = 1$, and then a compute continuous approximate solution using Hermite interpolation. The graph, plotting the continuous approximate solution and exact solution, is shown in Figure 4.4.3.

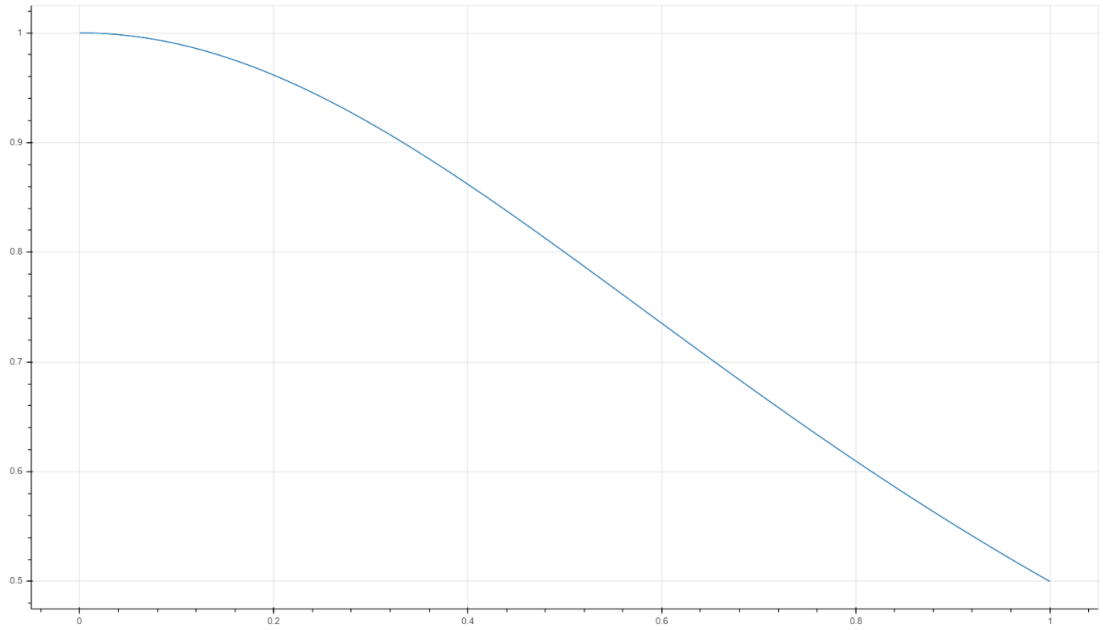

Figure 4.4.3: Exact solution and continuous approximate solution using the optimal fourth order ERK method, case 1, and Hermite interpolation for the IVODE $y' = -2xy^2,\ y(0) = 1$.

In the graph above, one can clearly see that the continuous approximate solution and exact solution agree quite well.

Finally, we consider the COVID-19 model introduced in Section 2.1, and we use the optimal fourth order ERK method, Case 1, to obtain an approximate solution for t from 0 to 150. We use Hermite interpolation to obtain a corresponding continuous approximate solution. This IVODE does not have an exact solution. We therefore only show the continuous approximate solution in Figure 4.4.4.

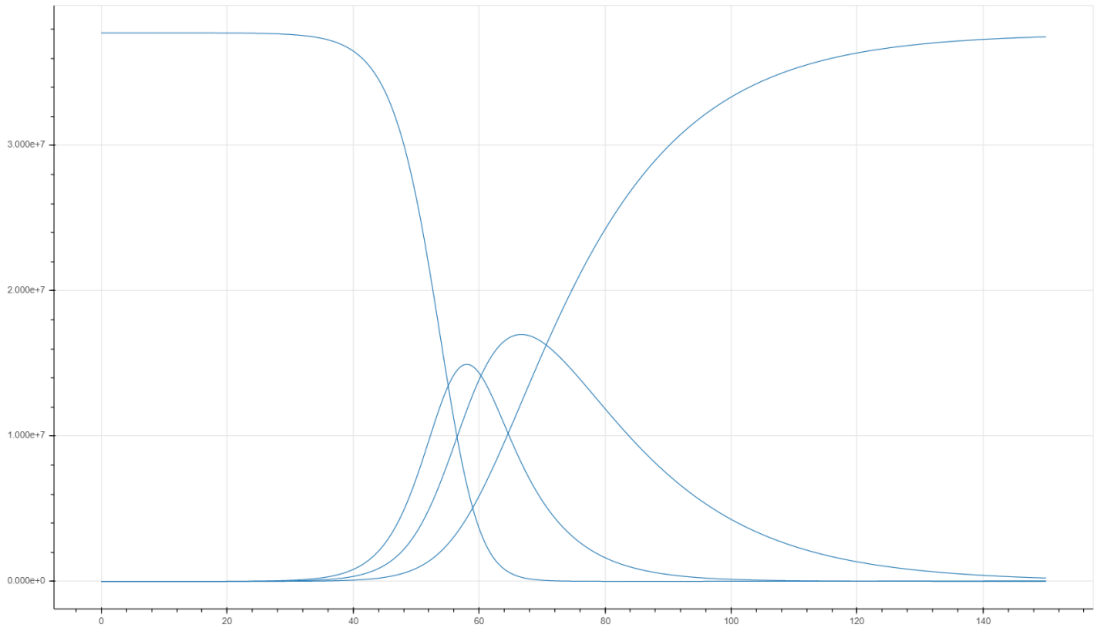

Figure 4.4.4: Continuous approximate solution using the optimal fourth order ERK method, Case 1, and Hermite interpolation to solve the COVID-19 model.

4.4.2 Defect of Continuous Approximate Solutions

In this subsection, we plot defects for the continuous approximate solutions of the IVODEs computed using optimal ERK methods of each order, together with Hermite interpolants. We consider the same IVODEs that we considered in Section 4.4.1. We plot the defect on one step. Recall that for a continuous approximate solution, $u(t)$, the defect is,

$$\delta(t) = u'(t) - f\big(t, u(t)\big).$$

We begin with second order ERK methods. We use the optimal second order ERK method to obtain a discrete approximate solution to the IVODE,

$$y' = -\alpha y - e^{-\alpha x} \sin x,$$

where $\alpha = 0.1$, with initial value, $y(0) = 1$ at a stepsize of 0.015625, and then we use Hermite interpolation to find a continuous approximate solution as we did in Section 4.4.1. We plot the defect based on this continuous approximate solution. For this test, we choose, arbitrarily, step number 23. The graph plotting the defect in steps 23 is shown in Figure 4.4.5. We see that the maximum defect is quite small $\approx 5 \times 10^{-6}$ in magnitude.

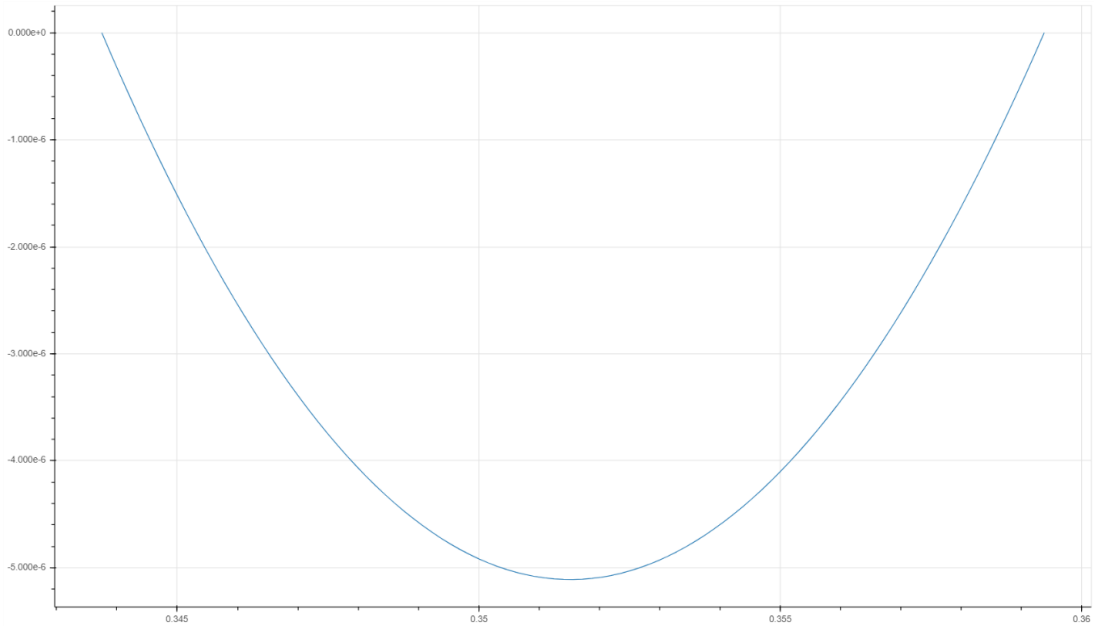

Figure 4.4.5: Defect, Step 23, of the continuous approximate solution of the IVODE $y' = -\alpha y - e^{-\alpha x}\sin x$, $y(0) = 1$ computed by using the optimal second order ERK method, and Hermite interpolation.

Next, we consider third order ERK methods. We use the optimal third order ERK method, Case 1, to obtain a discrete approximate solution to the IVODE,

$$y' = \left(-\frac{1}{2}\right)y^3,$$

with initial value, $y(0) = 1$ at a stepsize of 0.01, and then we use Hermite interpolation to find a continuous approximate solution as we did in Section 4.4.1. We plot the defect based on this continuous approximate solution. For this test, we choose, arbitrarily, step number 11. The graph plotting the defect in step 11 is shown in Figure 4.4.6. We see that the maximum defect is quite small; it has a magnitude of $\approx 2.5 \times 10^{-7}$.

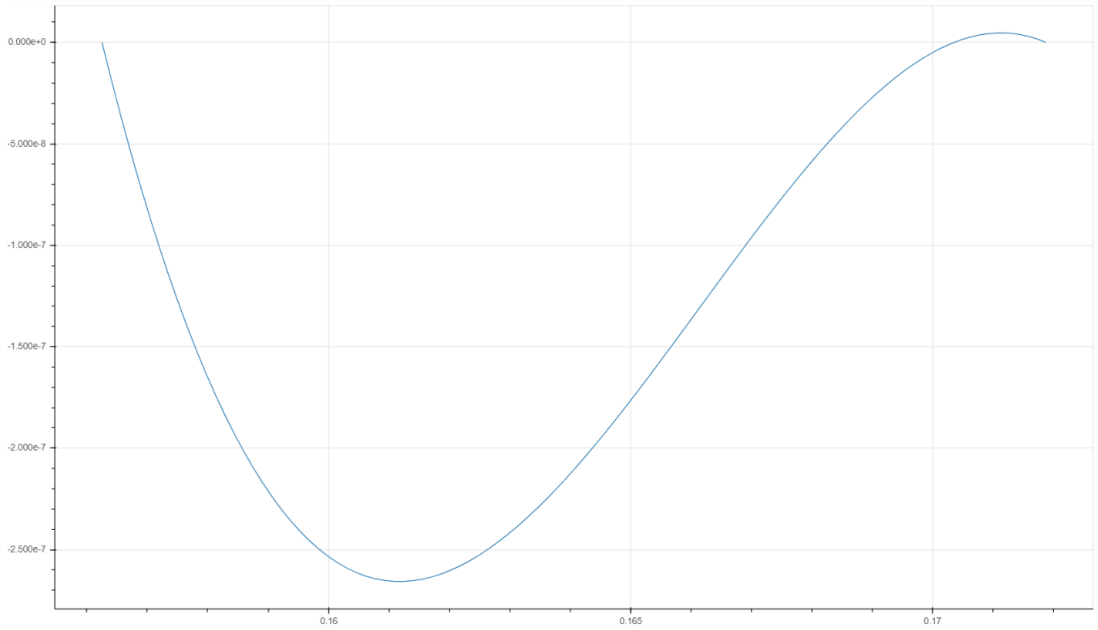

Figure 4.4.6: Defect, Step 11, of the continuous approximate solution of the IVODE $y' = \left(-\frac{1}{2}\right)y^3,\ y(0) = 1$ computed by using the optimal third order ERK method, case 1, and Hermite interpolation.

Finally, we consider fourth order ERK methods. We use the optimal fourth order ERK method, Case 1, to obtain a discrete approximate solution to the IVODE,

$$y' = -2xy^2,$$

with initial value, $y(0) = 1$, and then we again use Hermite interpolation to find a continuous approximate solution as we did in Section 4.4.1. We plot the defect based on this continuous approximate solution. For this test, we choose, arbitrarily, step number 8. The graph plotting the defect in step 8 is shown in Figure 4.4.7. We can see that the maximum defect is quite small; it has a magnitude of $\approx 6 \times 10^{-7}$.

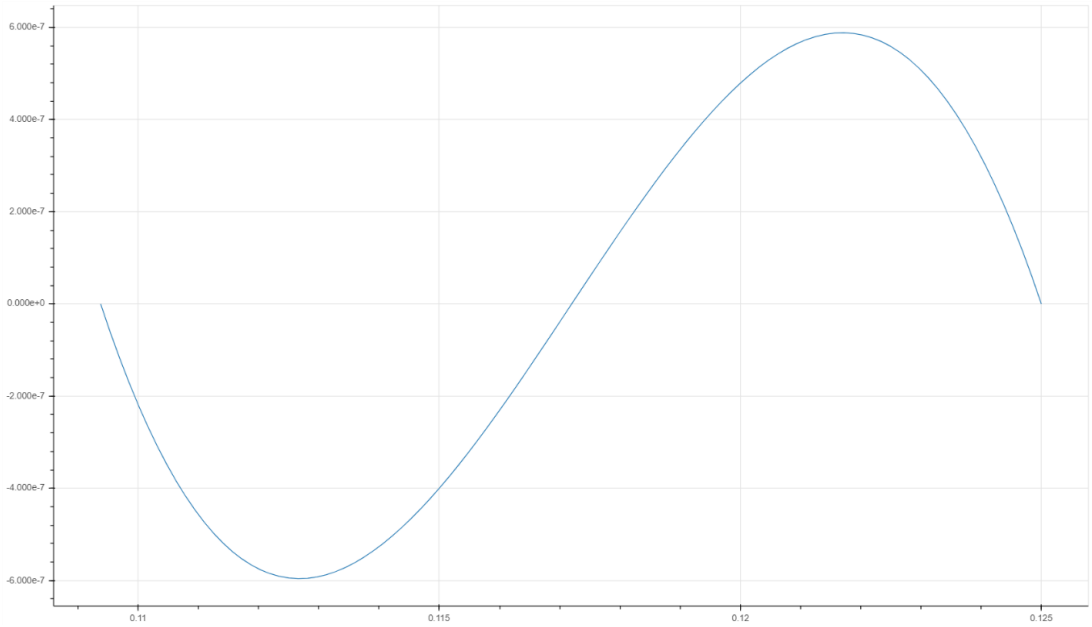

Figure 4.4.7: Defect, Step 8, of the continuous approximate solution of the IVODE $y' = -2xy^2$, $y(0) = 1$ computed by using the optimal fourth order ERK method, case 1, and Hermite interpolation.

We observe that the defects in all cases are quite small which indicates that the approximate continuous solution almost satisfies the given ODE.

Chapter 5

Summary, Conclusions, and Future Work

In this thesis, we have presented the general form for Initial Value ODEs along with some examples. We have presented general forms for Explicit Runge-Kutta methods of orders 2 to 4 and found the optimal values for the free coefficients based on minimizing the Principal Error Coefficient of the method using optimization software. We observed that the Principal Error Coefficients of several of the standard ERK methods are very close to those of the optimal ERK methods. We have also provided a comparison between the standard and optimal ERK methods by testing them on various IVODEs. We note that the problem itself has an effect on the error of the approximate numerical solution as the unsatisfied order conditions that make up the components of the Principal Error Coefficient are multiplied by problem dependent factors. We confirmed the order of convergence of the optimal methods that we obtained based on discrete approximate solutions computed by the ERK methods. We used Hermite interpolants to augment the discrete approximate solutions to obtain continuous approximate numerical solutions across the whole domain. This allowed us to plot the continuous approximate solution and to find the defect associated with it. We observed that the continuous approximate numerical solutions were quite accurate as their corresponding defects were quite small.

Regarding future work, further testing of the ERK methods using a larger test set would be helpful in order to determine how frequently the optimal methods actually lead to more accurate results. The next step would be to develop a new software that is able to perform adaptive step control based on some estimate of the maximum defect on each step. This would allow the python tool to provide more efficient results by adjusting the stepsizes. The

idea here is to adjust the stepsize in order to keep the maximum defect below the user-provided tolerance. Another idea would be to extend the research to higher order methods. That will also include using an interpolant of higher order than Hermite interpolation. Along with that, it would be helpful to add a graphical user interface aspect to the Python software. It would allow the user to use the software and enter new IVODEs more easily.

Bibliography

[Butc87] J.C. Butcher, The Numerical Analysis of Ordinary Differential Equations: Runge-Kutta and General Linear Methods, Wiley-Interscience, 1987.

[Chri20] C. Christara, The importance of being computationally accurate: the case of COVID-19, unpublished note, Department of Computer Science, University of Toronto, 2020.

[Enri89] W.H. Enright, A new error control strategy for initial value solvers, SIAM J. Numer. Anal., 26, 1989, 588-599.

[HNW87] E. Hairer, S. P. Norsett, G. Wanner, Solving Ordinary Differential Equations I: Nonstiff Problems, Springer Series in Computational mathematics, Springer, 1987.

[Rals62] A. Ralston, Runge-Kutta methods with minimum error bounds, Math. Comp., 16, 1962, 431-437.

[SAP97] L.F. Shampine, R. C. Allen, S. Pruess, Fundamentals of Numerical Computing, Wiley, 1997.

Appendix

This software is available at https://github.com/CYBRPHRK/Research-Thesis.

Optimization.py

```
import scipy.optimize as scope

case = 0

'''
Name: optimize
Description: This function finds the optimal values to minimize the given
                Principal Error Coefficient and provides the results in
                the console.
Parameters:
        f     : f is the name of the function used as Principal Error
              Coefficient.
Returns: None
'''
def optimize(f):
    if (f == E2):
        alpha = [0.1]
    elif(f == E3):
        alpha = [0.01, 0.01]
    elif(f == E4):
        if (case == 1):
            alpha = [0.1, 0.2]
        elif (case == 3):
            alpha = [-0.1]
        elif ((case == 2) or (case == 4) or (case == 5)):
            alpha = [0.1]

    res = scope.minimize(f, alpha, tol=1e-8)
    print ("Message: ", res.message)
    print ("E2: ", res.fun)
    print ("Free Coefficients: ", res.x)

'''
Name: E2
Description: This function represents as the Principal Error Coefficient
                for second order ERK methods. Provided the value of the
                free coefficient (alpha), this function provides the
                Principal Error Coefficient value.
Parameters:
        alpha  : alpha is the list of values for the free coefficients.
Returns:
        result : result is the Principal Error Coefficient value for the
                    given free coefficient.
'''
def E2(alpha):
    b = [1 - (1 / (2 * alpha[0])), 1 / (2 * alpha[0])]
    c = [0, alpha[0]]
    A = [[0, 0], [alpha[0], 0]]
```

```
    csq = [c[0] ** 2, c[1] ** 2]
    bcsq = (b[0] * csq[0]) + (b[1] * csq[1])
    Ac = [(A[0][0] * c[0]) + (A[0][1] * c[1]),(A[1][0] * c[0]) + (A[1][1] *
             c[1])]
    bAc = (b[0] * Ac[0]) + (b[1] * Ac[1])
    result = (((1/2) * (bcsq - (1/3))) ** 2) + ((bAc - (1/6)) ** 2)

    return result

'''
Name: setValuesForThirdOrderCase1
Description: This function sets and returns the values for all the
               coefficients for Case 1 of the third order ERK methods.
Parameters:
        alpha  : alpha is the list of values for the free coefficients.
Returns:
        c2, c3, b1, b2, b3, a31, a32: coefficients for Case 1 of the
                                        third order ERK methods.
'''
def setValuesForThirdOrderCase1(alpha):
    c2 = alpha[0]
    c3 = alpha[1]
    b1 = (2 - (3 * (c2 + c3)) + (6 * c2 * c3)) / (6 * c2 * c3)
    b2 = (c3 - (2/3)) / (2 * c2 * (c3 - c2))
    b3 = ((2/3) - c2) / (2 * c3 * (c3 - c2))
    a31 = (c3 * (c3 - (3 * c2) + (3 * c2 * c2))) / (c2 * ((3 * c2) - 2))
    a32 = (c3 * (c2 - c3)) / (c2 * ((3 * c2) - 2))

    return c2, c3, b1, b2, b3, a31, a32

'''
Names: E3Eq1, E3Eq2, E3Eq3, E3Eq4
Description: These functions compute and return the weighted values of
               the order conditions of Principal Error Coefficient
               of third order ERK methods.
Parameters:
        c  : c are the nodes.
        b  : b are the weights.
        A  : A is the matrix.
Returns:
        weighted values of the order conditions of Principal Error
            Coefficient of third order ERK methods.
'''
def E3Eq1(c, b, A):
    #For Equation 1, find b*c^3
    ccube = [c[0] ** 3, c[1] ** 3, c[2] ** 3]
    bccube = (b[0] * ccube[0]) + (b[1] * ccube[1]) + (b[2] * ccube[2])

    return ((1/6) * (bccube - (1/4)))

def E3Eq2(c, b, A):
    #For Equation 2, find b*c*A*c
    bc = [(b[0] * c[0]), (b[1] * c[1]), (b[2] * c[2])]
    Ac = [((A[0][0] * c[0]) + (A[0][1] * c[1]) + (A[0][2] * c[2])),
          ((A[1][0] * c[0]) + (A[1][1] * c[1]) + (A[1][2] * c[2])),
          ((A[2][0] * c[0]) + (A[2][1] * c[1]) + (A[2][2] * c[2]))]
    bcAc = (bc[0] * Ac[0]) + (bc[1] * Ac[1]) + (bc[2] * Ac[2])

    return (bcAc - (1/8))

def E3Eq3(c, b, A):
    #For Equation 3, find b*A*c^2
```

```
    csq = [c[0] ** 2, c[1] ** 2, c[2] ** 2]
    Acsq = [((A[0][0] * csq[0]) + (A[0][1] * csq[1]) + (A[0][2] * csq[2])),
          ((A[1][0] * csq[0]) + (A[1][1] * csq[1]) + (A[1][2] * csq[2])),
          ((A[2][0] * csq[0]) + (A[2][1] * csq[1]) + (A[2][2] * csq[2]))]
    bAcsq = (b[0] * Acsq[0]) + (b[1] * Acsq[1]) + (b[2] * Acsq[2])

    return ((1/2) * (bAcsq - (1/12)))

def E3Eq4(c, b, A):
    #For Equation 4, find b*A^2*c
    Asq = []
    for i in range (0, len(A)):
        Asq.append([])
        for j in range (0, len(A)):
            Asq[i].append((A[i][0] * A[0][j]) + (A[i][1] * A[1][j]) +
              (A[i][2] * A[2][j]))

    Asqc = [((Asq[0][0] * c[0]) + (Asq[0][1] * c[1]) + (Asq[0][2] * c[2])),
          ((Asq[1][0] * c[0]) + (Asq[1][1] * c[1]) + (Asq[1][2] * c[2])),
          ((Asq[2][0] * c[0]) + (Asq[2][1] * c[1]) + (Asq[2][2] * c[2]))]
    bAsqc = (b[0] * Asqc[0]) + (b[1] * Asqc[1]) + (b[2] * Asqc[2])

    return (bAsqc - (1/24))

'''
Name: E3
Description: This function represents as the Principal Error Coefficient
               for third order ERK methods. Provided the values of the
               free coefficients (alpha), this function provides the
               Principal Error Coefficient value.
Parameters:
        alpha  : alpha is the list of values for the free coefficients.
Returns:
        result : result is the Principal Error Coefficient value for the
                   given free coefficients.
'''
def E3(alpha):
    if ((alpha[0] == 0) or (alpha[0] == 2/3) or (alpha[1] == 0) or (alpha[0]
             == alpha[1])):
        return 1

    c2, c3, b1, b2, b3, a31, a32 = setValuesForThirdOrderCase1(alpha)

    c = [0, c2, c3]
    b = [b1, b2, b3]
    A = [[0, 0, 0],[c2, 0, 0],[a31, a32, 0]]

    #For Equation 1
    eq1 = E3Eq1(c, b, A)

    #For Equation 2
    eq2 = E3Eq2(c, b, A)

    #For Equation 3
    eq3 = E3Eq3(c, b, A)

    #For Equation 4
    eq4 = E3Eq4(c, b, A)

    #E3^2
    result = (eq1 ** 2) + (eq2 ** 2) + (eq3 ** 2) + (eq4 ** 2)

    return result
```

```
'''
Name: setValuesForFourthOrder
Description: This function sets and returns the values for all the
               coefficients for the fourth order ERK methods.
Parameters:
        alpha  : alpha is the list of values for the free coefficients.
Returns:
        c2, c3, c4, b1, b2, b3, b4, a31, a32, a41, a42, a43:
                 coefficients for the fourth order ERK methods.
'''
def setValuesForFourthOrder(alpha):
    if (case == 1):
        c2 = alpha[0]
        c3 = alpha[1]
        c4 = 1
        a31 = (c3 * ((3 * c2) - c3 - (4 * c2 * c2))) / (2 * c2 * (1 - (2 *
             c2)))
        a32 = (c3 * (c3 - c2)) / (2 * c2 * (1 - (2 * c2)))
        a41 = (((c3 ** 2) * ((12 * c2 * c2) - (12 * c2) + 4)) - (c3 * ((12 *
             c2 * c2) - (15 * c2) + 5)) + ((4 * c2 * c2) - (6 * c2) + 2)) /
             ((2 * c2 * c3) * (3 - (4 * (c2 + c3)) + (6 * c2 * c3)))
        a42 = (((-4 * c3 * c3) + (5 * c3) + c2 - 2) * (1 - c2)) / ((2 * c2) *
             (c3 - c2) * (3 - (4 * (c2 + c3)) + (6 * c2 * c3)))
        a43 = ((1 - (2 * c2)) * (1 - c3) * (1 - c2)) / (c3 * (c3 - c2) * (3 -
             (4 * (c2 + c3)) + (6 * c2 * c3)))
        b1 = (1 - (2 * (c2 + c3)) + (6 * c2 * c3)) / (12 * c2 * c3)
        b2 = ((2 * c3) - 1) / ((12 * c2) * (c3 - c2) * (1 - c2))
        b3 = (1 - (2 * c2)) / ((12 * c3) * (c3 - c2) * (1 - c3))
        b4 = (3 - (4 * (c2 + c3)) + (6 * c2 * c3)) / (12 * (1 - c2) * (1 -
             c3))
    elif (case == 2):
        b3 = alpha[0]
        c2 = c3 = 1/2
        c4 = 1
        a31 = ((3 * b3) - 1) / (6 * b3)
        a32 = 1 / (6 * b3)
        a41 = 0
        a42 = 1 - (3 * b3)
        a43 = 3 * b3
        b1 = 1/6
        b2 = (2 / 3) - b3
        b4 = 1/6
    elif (case == 3):
        b3 = alpha[0]
        c2 = 1/2
        c3 = 0
        c4 = 1
        a31 = -1 / (12 * b3)
        a32 = 1 / (12 * b3)
        a41 = (-1/2) - (6 * b3)
        a42 = 3/2
        a43 = 6 * b3
        b1 = (1/6) - b3
        b2 = 2/3
        b4 = 1/6
    elif (case == 4):
        b4 = alpha[0]
        c2 = 1
        c3 = 1/2
        c4 = 1
        a31 = 3/8
        a32 = 1/8
```

```
        a41 = 1 - (1 / (4 * b4))
        a42 = -1 / (12 * b4)
        a43 = 1 / (3 * b4)
        b1 = 1/6
        b2 = 1/6 - b4
        b3 = 2/3
    elif (case == 5):
        c2 = alpha[0]
        c3 = 1/2
        c4 = 1
        a31 = ((4 * c2) - 1) / (8 * c2)
        a32 = 1 / (8 * c2)
        a41 = (1 - (2 * c2)) / (2 * c2)
        a42 = -1 / (2 * c2)
        a43 = 2
        b1 = 1/6
        b2 = 0
        b3 = 2/3
        b4 = 1/6

    return c2, c3, c4, b1, b2, b3, b4, a31, a32, a41, a42, a43

'''
Names: E4Eq1, E4Eq2, E4Eq3, E4Eq4, E4Eq5, E4Eq6, E4Eq7, E4Eq8, E4Eq9
Description: These functions compute and return the weighted values of
               the order conditions of Principal Error Coefficient
               of fourth order ERK methods.
Parameters:
        c  : c are the nodes.
        b  : b are the weights.
        A  : A is the matrix.
Returns:
        weighted values of the order conditions of Principal Error
            Coefficient of fourth order ERK methods.
'''
def E4Eq1(c, b, A):
    #For Equation 1, find b*c^4
    cquad = [c[0] ** 4, c[1] ** 4, c[2] ** 4, c[3] ** 4]
    bcquad = (b[0] * cquad[0]) + (b[1] * cquad[1]) + (b[2] * cquad[2]) +
             (b[3] * cquad[3])

    return ((1/24) * (bcquad - (1/5)))

def E4Eq2(c, b, A):
    #For Equation 2, find b*c^2*A*c
    csq = [c[0] ** 2, c[1] ** 2, c[2] ** 2, c[3] ** 2]
    bcsq = [b[0] * csq[0], b[1] * csq[1], b[2] * csq[2], b[3] * csq[3]]
    Ac = [((A[0][0] * c[0]) + (A[0][1] * c[1]) + (A[0][2] * c[2]) + (A[0][3]
             * c[3])),
          ((A[1][0] * c[0]) + (A[1][1] * c[1]) + (A[1][2] * c[2]) + (A[1][3]
             * c[3])),
          ((A[2][0] * c[0]) + (A[2][1] * c[1]) + (A[2][2] * c[2]) + (A[2][3]
             * c[3])),
          ((A[3][0] * c[0]) + (A[3][1] * c[1]) + (A[3][2] * c[2]) + (A[3][3]
             * c[3]))]

    bcsqAc = (bcsq[0] * Ac[0]) + (bcsq[1] * Ac[1]) + (bcsq[2] * Ac[2]) +
             (bcsq[3] * Ac[3])

    return ((1/2) * (bcsqAc - (1/10)))


def E4Eq3(c, b, A):
```

```
    #For Equation 3, find b*c*A*c^2
    csq = [c[0] ** 2, c[1] ** 2, c[2] ** 2, c[3] ** 2]
    bc = [b[0] * c[0], b[1] * c[1], b[2] * c[2], b[3] * c[3]]
    Acsq = [((A[0][0] * csq[0]) + (A[0][1] * csq[1]) + (A[0][2] * csq[2]) +
             (A[0][3] * csq[3])),
            ((A[1][0] * csq[0]) + (A[1][1] * csq[1]) + (A[1][2] * csq[2]) +
             (A[1][3] * csq[3])),
            ((A[2][0] * csq[0]) + (A[2][1] * csq[1]) + (A[2][2] * csq[2]) +
             (A[2][3] * csq[3])),
            ((A[3][0] * csq[0]) + (A[3][1] * csq[1]) + (A[3][2] * csq[2]) +
             (A[3][3] * csq[3]))]
    bcAcsq = (bc[0] * Acsq[0]) + (bc[1] * Acsq[1]) + (bc[2] * Acsq[2]) +
             (bc[3] * Acsq[3])

    return ((1/2) * (bcAcsq - (1/15)))

def E4Eq4(c, b, A):
    #For Equation 4, find b*c*A^2*c
    bc = [b[0] * c[0], b[1] * c[1], b[2] * c[2], b[3] * c[3]]
    Ac = [((A[0][0] * c[0]) + (A[0][1] * c[1]) + (A[0][2] * c[2]) + (A[0][3]
             * c[3])),
          ((A[1][0] * c[0]) + (A[1][1] * c[1]) + (A[1][2] * c[2]) + (A[1][3]
             * c[3])),
          ((A[2][0] * c[0]) + (A[2][1] * c[1]) + (A[2][2] * c[2]) + (A[2][3]
             * c[3])),
          ((A[3][0] * c[0]) + (A[3][1] * c[1]) + (A[3][2] * c[2]) + (A[3][3]
             * c[3]))]
    AAc = [((A[0][0] * Ac[0]) + (A[0][1] * Ac[1]) + (A[0][2] * Ac[2]) +
             (A[0][3] * Ac[3])),
           ((A[1][0] * Ac[0]) + (A[1][1] * Ac[1]) + (A[1][2] * Ac[2]) +
             (A[1][3] * Ac[3])),
           ((A[2][0] * Ac[0]) + (A[2][1] * Ac[1]) + (A[2][2] * Ac[2]) +
             (A[2][3] * Ac[3])),
           ((A[3][0] * Ac[0]) + (A[3][1] * Ac[1]) + (A[3][2] * Ac[2]) +
             (A[3][3] * Ac[3]))]
    bcAAc = (bc[0] * AAc[0]) + (bc[1] * AAc[1]) + (bc[2] * AAc[2]) + (bc[3] *
             AAc[3])

    return (bcAAc - (1/30))

def E4Eq5(c, b, A):
    #For Equation 5, find b*(A*c)^2
    Ac = [((A[0][0] * c[0]) + (A[0][1] * c[1]) + (A[0][2] * c[2]) + (A[0][3]
             * c[3])),
          ((A[1][0] * c[0]) + (A[1][1] * c[1]) + (A[1][2] * c[2]) + (A[1][3]
             * c[3])),
          ((A[2][0] * c[0]) + (A[2][1] * c[1]) + (A[2][2] * c[2]) + (A[2][3]
             * c[3])),
          ((A[3][0] * c[0]) + (A[3][1] * c[1]) + (A[3][2] * c[2]) + (A[3][3]
             * c[3]))]

    AcAc = [Ac[0] ** 2, Ac[1] ** 2, Ac[2] ** 2, Ac[3] ** 2]
    bAcAc = (b[0] * AcAc[0]) + (b[1] * AcAc[1]) + (b[2] * AcAc[2]) + (b[3] *
             AcAc[3])

    return ((1/2) * (bAcAc - (1/20)))

def E4Eq6(c, b, A):
    #For Equation 6, find b*A*c^3
    ccube = [c[0] ** 3, c[1] ** 3, c[2] ** 3, c[3] ** 3]
    Accube = [((A[0][0] * ccube[0]) + (A[0][1] * ccube[1]) + (A[0][2] *
             ccube[2]) + (A[0][3] * ccube[3])),
```

```
             ((A[1][0] * ccube[0]) + (A[1][1] * ccube[1]) + (A[1][2] *
            ccube[2]) + (A[1][3] * ccube[3])),
             ((A[2][0] * ccube[0]) + (A[2][1] * ccube[1]) + (A[2][2] *
            ccube[2]) + (A[2][3] * ccube[3])),
             ((A[3][0] * ccube[0]) + (A[3][1] * ccube[1]) + (A[3][2] *
            ccube[2]) + (A[3][3] * ccube[3]))]
    bAccube = (b[0] * Accube[0]) + (b[1] * Accube[1]) + (b[2] * Accube[2]) +
              (b[3] * Accube[3])

    return ((1/6) * (bAccube - (1/20)))

def E4Eq7(c, b, A):
    #For Equation 7, find b*A*c*(A*c)
    Ac = [((A[0][0] * c[0]) + (A[0][1] * c[1]) + (A[0][2] * c[2]) + (A[0][3]
              * c[3])),
          ((A[1][0] * c[0]) + (A[1][1] * c[1]) + (A[1][2] * c[2]) + (A[1][3]
              * c[3])),
          ((A[2][0] * c[0]) + (A[2][1] * c[1]) + (A[2][2] * c[2]) + (A[2][3]
              * c[3])),
          ((A[3][0] * c[0]) + (A[3][1] * c[1]) + (A[3][2] * c[2]) + (A[3][3]
              * c[3]))]
    cAc = [c[0] * Ac[0], c[1] * Ac[1], c[2] * Ac[2], c[3] * Ac[3]]
    AcAc = [((A[0][0] * cAc[0]) + (A[0][1] * cAc[1]) + (A[0][2] * cAc[2]) +
              (A[0][3] * cAc[3])),
            ((A[1][0] * cAc[0]) + (A[1][1] * cAc[1]) + (A[1][2] * cAc[2]) +
              (A[1][3] * cAc[3])),
            ((A[2][0] * cAc[0]) + (A[2][1] * cAc[1]) + (A[2][2] * cAc[2]) +
              (A[2][3] * cAc[3])),
            ((A[3][0] * cAc[0]) + (A[3][1] * cAc[1]) + (A[3][2] * cAc[2]) +
              (A[3][3] * cAc[3]))]
    bAcAc = (b[0] * AcAc[0]) + (b[1] * AcAc[1]) + (b[2] * AcAc[2]) + (b[3] *
              AcAc[3])

    return (bAcAc - (1/40))

def E4Eq8(c, b, A):
    #For Equation 8, find b*A^2*c^2
    csq = [c[0] ** 2, c[1] ** 2, c[2] ** 2, c[3] ** 2]
    Acsq = [((A[0][0] * csq[0]) + (A[0][1] * csq[1]) + (A[0][2] * csq[2]) +
              (A[0][3] * csq[3])),
            ((A[1][0] * csq[0]) + (A[1][1] * csq[1]) + (A[1][2] * csq[2]) +
              (A[1][3] * csq[3])),
            ((A[2][0] * csq[0]) + (A[2][1] * csq[1]) + (A[2][2] * csq[2]) +
              (A[2][3] * csq[3])),
            ((A[3][0] * csq[0]) + (A[3][1] * csq[1]) + (A[3][2] * csq[2]) +
              (A[3][3] * csq[3]))]
    AAcsq = [((A[0][0] * Acsq[0]) + (A[0][1] * Acsq[1]) + (A[0][2] * Acsq[2])
              + (A[0][3] * Acsq[3])),
              ((A[1][0] * Acsq[0]) + (A[1][1] * Acsq[1]) + (A[1][2] * Acsq[2])
              + (A[1][3] * Acsq[3])),
              ((A[2][0] * Acsq[0]) + (A[2][1] * Acsq[1]) + (A[2][2] * Acsq[2])
              + (A[2][3] * Acsq[3])),
              ((A[3][0] * Acsq[0]) + (A[3][1] * Acsq[1]) + (A[3][2] * Acsq[2])
              + (A[3][3] * Acsq[3]))]
    bAAcsq = (b[0] * AAcsq[0]) + (b[1] * AAcsq[1]) + (b[2] * AAcsq[2]) +
              (b[3] * AAcsq[3])

    return ((1/2) * (bAAcsq - (1/60)))

def E4Eq9(c, b, A):
    #For Equation 9, find b*A^3*c
    Ac = [((A[0][0] * c[0]) + (A[0][1] * c[1]) + (A[0][2] * c[2]) + (A[0][3]
              * c[3])),
```

```
            ((A[1][0] * c[0]) + (A[1][1] * c[1]) + (A[1][2] * c[2]) + (A[1][3]
               * c[3])),
            ((A[2][0] * c[0]) + (A[2][1] * c[1]) + (A[2][2] * c[2]) + (A[2][3]
               * c[3])),
            ((A[3][0] * c[0]) + (A[3][1] * c[1]) + (A[3][2] * c[2]) + (A[3][3]
               * c[3]))]
    AAc = [((A[0][0] * Ac[0]) + (A[0][1] * Ac[1]) + (A[0][2] * Ac[2]) +
               (A[0][3] * Ac[3])),
           ((A[1][0] * Ac[0]) + (A[1][1] * Ac[1]) + (A[1][2] * Ac[2]) +
               (A[1][3] * Ac[3])),
           ((A[2][0] * Ac[0]) + (A[2][1] * Ac[1]) + (A[2][2] * Ac[2]) +
               (A[2][3] * Ac[3])),
           ((A[3][0] * Ac[0]) + (A[3][1] * Ac[1]) + (A[3][2] * Ac[2]) +
               (A[3][3] * Ac[3]))]
    AAAc = [((A[0][0] * AAc[0]) + (A[0][1] * AAc[1]) + (A[0][2] * AAc[2]) +
               (A[0][3] * Ac[3])),
            ((A[1][0] * AAc[0]) + (A[1][1] * AAc[1]) + (A[1][2] * AAc[2]) +
               (A[1][3] * Ac[3])),
            ((A[2][0] * AAc[0]) + (A[2][1] * AAc[1]) + (A[2][2] * AAc[2]) +
               (A[2][3] * Ac[3])),
            ((A[3][0] * AAc[0]) + (A[3][1] * AAc[1]) + (A[3][2] * AAc[2]) +
               (A[3][3] * Ac[3]))]

    bAAAc = (b[0] * AAAc[0]) + (b[1] * AAAc[1]) + (b[2] * AAAc[2]) + (b[3] *
             AAAc[3])

    return (bAAAc - (1/120))

'''
Name: E4
Description: This function represents as the Principal Error Coefficient
               for fourth order ERK methods. Provided the values of the
               free coefficients (alpha), this function provides the
               Principal Error Coefficient value.
Parameters:
        alpha  : alpha is the list of values for the free coefficients.
Returns:
        result : result is the Principal Error Coefficient value for the
                   given free coefficients.
'''
def E4(alpha):
    if (case == 1):
        if ((alpha[0] <= 0) or (alpha[1] <= 0) or (alpha[0] == 1) or
             (alpha[1] == 1)
            or (alpha[0] == alpha[1]) or (alpha[0] == 1/2) or ((3 - (4 *
             (alpha[0] + alpha[1])) + (6 * alpha[0] * alpha[1])) == 0)):
            return 1
    elif ((case == 2) or (case == 3) or (case == 4) or (case == 5)):
        if (alpha[0] == 0):
            return 1

    c2, c3, c4, b1, b2, b3, b4, a31, a32, a41, a42, a43 =
             setValuesForFourthOrder(alpha)

    c = [0, c2, c3, c4]
    b = [b1, b2, b3, b4]
    A = [[0, 0, 0, 0],[c2, 0, 0, 0],[a31, a32, 0, 0], [a41, a42, a43, 0]]

    #For Equation 1
    eq1 = E4Eq1(c, b, A)

    #For Equation 2
    eq2 = E4Eq2(c, b, A)
```

```
    #For Equation 3
    eq3 = E4Eq3(c, b, A)

    #For Equation 4
    eq4 = E4Eq4(c, b, A)

    #For Equation 5
    eq5 = E4Eq5(c, b, A)

    #For Equation 6
    eq6 = E4Eq6(c, b, A)

    #For Equation 7
    eq7 = E4Eq7(c, b, A)

    #For Equation 8
    eq8 = E4Eq8(c, b, A)

    #For Equation 9
    eq9 = E4Eq9(c, b, A)

    result = (eq1 ** 2) + (eq2 ** 2) + (eq3 ** 2) + (eq4 ** 2) + (eq5 ** 2) +
             (eq6 ** 2) + (eq7 ** 2) + (eq8 ** 2) + (eq9 ** 2)

    return result

'''
Name: displayMenu
Description: This function displays the menu and asks the user to input
               a choice.
Parameters: None
Returns:
        choice : the interger value given by the user.
'''
def displayMenu():
    print ("1. Optimize E2")
    print ("2. Optimize E3")
    print ("3. Optimize E4")
    choice = input("Enter your choice: ")

    return int(choice);

'''
Name: chooseE4Case
Description: This function displays a menu for the cases of the fourth
               order ERK methods and asks the user to input a choice.
Parameters: None
Returns:
        choice : the interger value given by the user.
'''
def chooseE4Case():
    print ("1. Case 1: 0, c2, c3, 1 all distinct,",
           "\nc2≠1/2 and 3 - 4(c2 + c3) + 6*c2*c3 ≠ 0")
    print ("2. Case 2: c2 = c3 = 1/2, b3≠0")
    print ("3. Case 3: c2 = 1/2, c3 = 0, b3≠0")
    print ("4. Case 4: c2 = 1, c3 = 1/2, b4≠0")
    print ("5. Case 5: c2≠0, c3 = 1/2, b2 = 0")
    choice = input("\nEnter your case choice: ")

    return int(choice)

'''
```

```
Name: initializeOptimizer
Description: This function initializes the optimization process for the
                Principal Error Coefficient of user's choice.
Parameters:
        choice  : the interger value given by the user.
Returns: None
'''
def initializeOptimizer(choice):
    global case
    if (choice == 1):
        optimize(E2)
    elif (choice == 2):
        optimize(E3)
    elif (choice == 3):
        case = chooseE4Case()
        if ((case < 1) or (case > 5)):
            initializeOptimizer(choice)
        else:
            optimize(E4)
    else:
        print ("Invalid choice. Please try again.")
        initializeOptimizer(displayMenu())

initializeOptimizer(displayMenu())
```

main.py

```
import EulersMethod as em
import Function as f
import Methods as m
import FileIO.FileIO as FileIO
import config
import HermiteInterpolation as hi

'''
Name: displayMenu
Description: This function displays a menu and asks the user to input
                a choice.
Parameters: None
Returns:
        choice : the integer value given by the user.
'''
def displayMenu():
    print ("1. Specific IVODE on Specific Method")
    print ("2. Specific IVODE on All Methods and Export results to a file")
    choice = input("Enter your choice: ")
    print ("")
    return int(choice)

'''
Name: chooseMenuOption
Description: This function runs the specific function to initiate
                the testing of one or more ERK methods.
Parameters:
        choice : the integer value given by the user.
Returns: None
'''
def chooseMenuOption(choice):
    if (choice == 1):
        specificIVODESpecificMethod()
    elif (choice == 2):
        specificIVODEAllMethods()
    else:
        print ("Invalid Choice.\n")
        chooseMenuOption(displayMenu())

'''
Name: specificIVODESpecificMethod
Description: This function initiates the specific ERK method
                to solve a specific IVODE chosen by the user.
Parameters : None
Returns    : None
'''
def specificIVODESpecificMethod():
    t0, tf, y0 = f.setFormulaValues(f.displayFormulas())
    em.setInitialValues(t0, tf, y0)

    m.displayMethods()

    j = 1
    while(j <= 6):
        if (f.exactExists):
            ee, tt, yy = em.eulersMethod(j)
            order = em.findOrder(ee, j)
            for x in order:
                print (em.dictToString(x))
            print ()
        else:
```

```
            k = 0
            tt, yy = em.eulersMethod(j)
            for y in yy[len(yy) - 1]:
                print ("Steps:", (2 ** (j * (-1))), "\ty[" + str(k) + "]:",
             y)
                k += 1
            print ()
        j = j + 1
    hi.plotHermite()

'''
Name: specificIVODEAllMethods
Description: This function initiates all the ERK methods
               to solve a specific IVODE chosen by the user.
Parameters : None
Returns    : None
'''
def specificIVODEAllMethods():
    t0, tf, y0 = f.setFormulaValues(f.displayFormulas())
    em.setInitialValues(t0, tf, y0)

    config.file = FileIO.FileIO("Test Results/F" + str(f.formulaNumber) +
             ".txt", "w")
    fname = "F" + str(f.formulaNumber) + " " + str(t0) + " " + str(tf)
    for y in y0:
        fname = fname + " " + str(y)
    config.file.write(fname, end='\n\n')

    orders = []
    methodNumber = 1
    i = 1
    while(methodNumber < 10):
        case = i
        methodInfo = "methodNumber: " + str(methodNumber)
        if (methodNumber == 7) or (methodNumber == 9):
            methodInfo = methodInfo + " Case: " + str(case)

        config.file.write(methodInfo, end='')
        m.setMethodValues(methodNumber, True, case)
        j = 1
        while(j <= 6):
            if (f.exactExists):
                ee, tt, yy = em.eulersMethod(j)
                order = em.findOrder(ee, j)
                for x in order:
                    config.file.write(em.dictToString(x))
                config.file.write("")
            else:
                k = 0
                tt, yy = em.eulersMethod(j)
                for y in yy[len(yy) - 1]:
                    config.file.write("Steps: " + str(2 ** (j * (-1))) +
              "\ty[" + str(k) + "]: " + str(y))
                    k += 1
                config.file.write("")
            j = j + 1
        if (f.exactExists):
            orders.append(order)
        if ((methodNumber != 7) and (methodNumber != 9)):
            methodNumber += 1
        else:
            if (((methodNumber == 7) and (i == 3)) or ((methodNumber == 9)
             and (i == 5))):
```

```
                methodNumber += 1
                i = 1
            else:
                i += 1
    if (f.exactExists):
        em.relToMinError(orders)

chooseMenuOption(displayMenu())

del config.file
```

config.py

```
# Object created to be used for fileIO
file = None

# Object to store the list of lists of time intervals for Hermite
interpolation
t = []

# Object to store the list of lists of y values for Hermite interpolation
y = []

# Object to store the list of lists of function values for Hermite
interpolation
f = []

# Object to store the list of function values per Eulers Method call
ffy = []
```

```
import math
import bokeh.plotting as bp
import Methods as m
import Function as f
import config

t0 = tf = 0
eeOld = y0 = []

'''
Name: setInitialValues
Description: This function sets the initial values for a method.
Parameters:
        t       : t is the initial time.
        tfinal  : tfinal is the final time.
        y       : y is the initial value for the given IVODE.
Returns:    None
'''
def setInitialValues(t, tfinal, y):
    global t0, tf, y0
    t0, tf, y0 = t, tfinal, y[:]

'''
Name: eulerMethod
Description: This function computes the approximate solution for an IVODE
             using a
                given ERK method.
Parameters:
        steps   : steps is the parameter provided to compute the stepsize.
Returns:
        if (f.exactExists = True):
            ee  : ee is the list of lists of errors in approximate numerical
                    solutions for the IVODE.
            tt  : tt is the list of points on the domain where the
              approximate
                    numerical solution for the IVODE is computed.
            yy  : yy is the list of approximate numerical solutions for the
              IVODE
                    computed at points (tt) on the domain.
        if (f.exactExists = False):
            tt  : tt is the list of points on the domain where the
              approximate
                    numerical solution for the IVODE is computed.
            yy  : yy is the list of approximate numerical solutions for the
              IVODE
                    computed at points (tt) on the domain.
'''
def eulersMethod(steps):
    # Setting up all the initial values
    t = t0
    tfinal = tf
    y = y0[:]
    h = math.pow(2, (steps * (-1)))
    tt = [t]
    yy = [y[:]]
    ee = []
    config.ffy = []

    # Computing the approximate numerical solution
    while (t < tfinal):
```

```
        fy = m.method(t, y[:], h)
        for i in range(0, len(y)):
            y[i] = y[i] + (h * fy[i])
        t = t + h
        tt.append(t)
        yy.append(y[:])
    m.method(t, y[:], h)
    config.t.append(tt[:])
    config.y.append(yy[:])
    config.f.append(config.ffy[:])

    if (f.exactExists):
        # Computing the error
        for j in range(0, len(yy)):
            e = f.formula(2, tt[j], yy[j])
            for i in range (0, len(e)):
                e[i] = abs(e[i])
            ee.append(e[:])
        return ee, tt, yy
    else:
        return tt, yy

'''
Name: findOrder
Description: This function computes the ratio of the errors and
               order of convergence of a given ERK method.
Parameters:
        ee      : ee is the list of lists of errors in approximate numerical
                  solutions for the IVODE.
        steps   : steps is the parameter provided to compute the stepsize.
Returns:
        orders  : orders is the list of dictionaries which has error(s),
             stepsize,
                    ratio of the errors and order of convergence of the
             method.
'''
def findOrder(ee, steps):
    global eeOld
    i = 0
    orders = []
    for e in ee[len(ee) -1]:
        order = {}
        order["ee[" + str(i) + "]"] = e
        order["Steps"] = math.pow(2, (steps * (-1)))
        if (steps > 1):
            ratio = eeOld[i]/e
            order['eeOld/ee'] = ratio
            if (ratio == 0):
                order['Order'] = 'n/a'
            else:
                order['Order'] = round(math.log(ratio, 2))
        i += 1
        orders.append(order)
    eeOld = ee[len(ee) - 1]
    return orders

'''
Name: relToMinError
Description: This function computes the relative to minimum error for each
             order of
                ERK method and print them in the results text file.
Parameters:
```

```
        orders  : orders is the list of dictionaries which has error(s),
             stepsize,
                    ratio of the errors and order of convergence of the
             method.
Returns:    None
'''
def relToMinError(orders):
    config.file.write("Rel. To Min. Errors:")
    for j in range (0, len(orders[0])):
        minError = min(orders[1][j].get("ee[" + str(j) + "]"),
             orders[2][j].get("ee[" + str(j) + "]"),
                      orders[3][j].get("ee[" + str(j) + "]"))
        orders[1][j]['RelError'] = (orders[1][j].get("ee[" + str(j) + "]")) /
             minError
        orders[2][j]['RelError'] = (orders[2][j].get("ee[" + str(j) + "]")) /
             minError
        orders[3][j]['RelError'] = (orders[3][j].get("ee[" + str(j) + "]")) /
             minError

        minError = min(orders[4][j].get("ee[" + str(j) + "]"),
             orders[5][j].get("ee[" + str(j) + "]"),
                      orders[6][j].get("ee[" + str(j) + "]"),
             orders[7][j].get("ee[" + str(j) + "]"),
                      orders[8][j].get("ee[" + str(j) + "]"))
        orders[4][j]['RelError'] = (orders[4][j].get("ee[" + str(j) + "]")) /
             minError
        orders[5][j]['RelError'] = (orders[5][j].get("ee[" + str(j) + "]")) /
             minError
        orders[6][j]['RelError'] = (orders[6][j].get("ee[" + str(j) + "]")) /
             minError
        orders[7][j]['RelError'] = (orders[7][j].get("ee[" + str(j) + "]")) /
             minError
        orders[8][j]['RelError'] = (orders[8][j].get("ee[" + str(j) + "]")) /
             minError

        minError = min(orders[9][j].get("ee[" + str(j) + "]"),
             orders[10][j].get("ee[" + str(j) + "]"),
                      orders[11][j].get("ee[" + str(j) + "]"),
             orders[12][j].get("ee[" + str(j) + "]"),
                      orders[13][j].get("ee[" + str(j) + "]"),
             orders[14][j].get("ee[" + str(j) + "]"))
        orders[9][j]['RelError'] = (orders[9][j].get("ee[" + str(j) + "]")) /
             minError
        orders[10][j]['RelError'] = (orders[10][j].get("ee[" + str(j) + "]"))
             / minError
        orders[11][j]['RelError'] = (orders[11][j].get("ee[" + str(j) + "]"))
             / minError
        orders[12][j]['RelError'] = (orders[12][j].get("ee[" + str(j) + "]"))
             / minError
        orders[13][j]['RelError'] = (orders[13][j].get("ee[" + str(j) + "]"))
             / minError
        orders[14][j]['RelError'] = (orders[14][j].get("ee[" + str(j) + "]"))
             / minError

    for x in orders:
        for y in x:
            config.file.write(dictToString(y))
        config.file.write("")

'''
Name: dictToString
Description: This function converts the data stored in a dictionary into
             string.
```

```
Parameters:
        dict    : dict is the dictionary in which the data is stored.
Returns:
        dictString  : dictString is the string converted from the dictonary.
'''
def dictToString(dict):
    dictString = ""
    # Fetching data from the dictonary and saving it in the string
    for x in dict:
        dictString = dictString + x + ": " + str(dict.get(x)) + "\t"
    # Returning the string after removing the extra whitespace
    return dictString.strip()
```

```
import ivode as iv
import config

formulaNumber = 0
exactExists = True

'''
Name: displayFormulas
Description: A function to get the formula number along
                with the respective values.
Parameters:
        None
Returns:
        fname   : returns a string with the formula number
                    and values provided by the user
'''
def displayFormulas():
    print ("Simple: f1 t tfinal y0")
    print ("Predator Prey: f2 t tfinal x y alpha beta gamma delta")
    print ("Simple System: f3 t tfinal x y")
    print ("Test F4: f4 t tfinal y0")
    print ("Test F5: f5 t tfinal y0")
    print ("Test F6: f6 t tfinal y0")
    print ("Test F7: f7 t tfinal y0 alpha")
    print ("Sample COVID-19 Model: f8 t tfinal")
    fname = input("\nEnter the formula with values respectively" +
                  " (Use spaces between the values like shown above):\n")
    return fname

'''
Name: setFormulaValues
Description: A function to set formula number and the
                respective values for the formulas accordingly.
Parameters:
        fname   : fname has the formula number as well as the
                    respective values for the formulas to be used
Returns:
        data[1] : The value of t0 for the initial time of the
                    formula
        data[2] : The value of tf for the final time of the
                    formula (tfinal)
        y[0]    : A list of initial values of y at time t0
'''
def setFormulaValues(fname):
    global formulaNumber, exactExists
    y0 = []
    data = fname.split()
    for i in range(1, len(data)):
        data[i] = float(data[i])

    formulaNumber = int (data[0][1:])

    if(formulaNumber == 2):
        exactExists = False
        y0.append(data[3])
        y0.append(data[4])
        iv.setConstants(data[5], data[6], data[7], data[8])
    elif (formulaNumber == 3):
        y0.append(data[3])
        y0.append(data[4])
    elif (formulaNumber == 7):
```

```
        y0.append(data[3])
        iv.setConstants(data[4])
    elif (formulaNumber == 8):
        exactExists = False
        y0 = iv.sampleCOVID19ModelInitializer()
    elif ((formulaNumber == 1) or ((formulaNumber >= 4) and (formulaNumber <=
6))):
        y0.append(data[3])
    else:
        print ("No formula with that name.")
        exit(0)

    return data[1], data[2], y0

'''
Name: formula
Description: A function to get the:
                i = 0: approximate values of y
                i = 1: exact values of y
                i = 2: error values at t with a given y

             by calling the respective formula function
             according to the formula number.
Parameters:
        t   : The value of t after a certain steps
        y   : The list of values of y at step t
Returns:
        for i = 0:
            y[t+h]  : The list of approximate values of y from
                      the respective formula function for the
                      next step t + h
        for i = 1:
            y[t]    : The list of exact values of y from the
                      respective formula function for the
                      step t
        for i = 2:
            e       : The list of error values with a given y
                      from the respective formula function
                      for the step t
'''
def formula(i, t, y):
    if (formulaNumber == 1):
        return iv.simple(i, t, y)
    elif (formulaNumber == 2):
        return iv.predatorPrey(i, t, y)
    elif (formulaNumber == 3):
        return iv.simple_sys(i, t, y)
    elif (formulaNumber == 4):
        return iv.TestF4(i, t, y)
    elif (formulaNumber == 5):
        return iv.TestF5(i, t, y)
    elif (formulaNumber == 6):
        return iv.TestF6(i, t, y)
    elif (formulaNumber == 7):
        return iv.TestF7(i, t, y)
    elif (formulaNumber == 8):
        return iv.sampleCOVID19Model(i, t, y)
```

```
import bokeh.plotting as bp
import Function as f
import config

'''
Names: h00(t), h10(t), h01(t) and h11(t)
Description: The functions given below work as Hermite Basis Polynomials,
                h00(t), h10(t), h01(t) and h11(t).
Parameters:
        t    : The quantity t measures the relative distance across the
subinterval
Returns:
        result for the Hermite Basis Polynomial at t.
'''
def h00(t):
    return ((1 + (2 * t)) * (1 - t)**2)

def h10(t):
    return (t * (1 - t)**2)

def h01(t):
    return ((t**2) * (3 - (2 * t)))

def h11(t):
    return ((t**2) * (t - 1))

'''
Names: h00_d(t), h10_d(t), h01_d(t) and h11_d(t)
Description: The functions given below work as derivatives of Hermite Basis
Polynomials,
                h00(t), h10(t), h01(t) and h11(t).
Parameters:
        t    : The quantity t measures the relative distance across the
subinterval
Returns:
        result for the derivative of Hermite Basis Polynomial at t.
'''
def h00_d(t):
    return (6 * t * (t - 1))

def h10_d(t):
    return (1 + (3 * (t**2)) - (4 * t))

def h01_d(t):
    return (6 * t * (1 - t))

def h11_d(t):
    return ((3 * (t**2)) - (2 * t))

'''
Name: hermite
Description: This function evaluates Hermite form for u_i(t_i + (theta *
h_i))
                and the associated defect.
Parameters:
        tt    : tt is the list of times after each step.
        yy    : yy is the list of lists of y values at the given times at
each step.
        ffy   : ffy is the function value for f(t, y) using the above values.
Returns:
        t     : t is the list of points on the whole domain.
```

```
        uu    : uu is the list of lists of lists of Hermite forms at uniform
points
                for system of equations in each interval. These are known as
                the continuous approximate numerical solutions.
        ffe   : ffe is the list of exact solutions at all the t values.
        delta : delta is the defect associated with the continuous
approximate
                numerical solution.
'''
def hermite(tt, yy, ffy):
    h = tt[1] - tt[0]
    # To store all the t values from start to end
    t = []
    # To store the u value computed at (t_i + (theta * h_i)).
    u = []
    # To store the derivative of u computed at (t_i + (theta * h_i)).
    u_d = []
    # To store the defect at (t_i + (theta * h_i)).
    d = []
    # To store all the u values computed
    uu = []
    # To store all the exact solutions at all the t values
    ffe = []
    # To store all the defect values in variable 'delta'
    delta = []

    for k in range (0, len(yy[0])):
        # For the first value, u_0(t_0)
        u.append((yy[0][k] * h00(0)) + (h * ffy[0][k] * h10(0)) + (yy[1][k] *
h01(0)) + (h * ffy[1][k] * h11(0)))

        # For the first derivative value, u_0'(t_0)
        u_d.append(((yy[0][k] / h) * h00_d(0)) + (ffy[0][k] * h10_d(0)) +
((yy[1][k] / h) * h01_d(0)) + (ffy[1][k] * h11_d(0)))
    # Here fu = f(t_0, u_0(t_0))
    fu = f.formula(0, tt[0], u[:])
    if (f.exactExists):
        # Exact value at t_0
        fe = f.formula(1, tt[0], u[:])
        # Storing the value in the list named 'ffe'
        ffe.append(fe)

    # d_0(t_0) = u_0'(t_0) - f(t_0, u_0(t_0))
    for k in range (0, len(u)):
        d.append(u_d[k] - fu[k])

    # Storing the values in their corresponding lists
    t.append(tt[0])
    uu.append(u)
    delta.append(d)

    # Performing hermite interpolation on all the intervals
    for i in range (0, len(tt) - 1):
        for j in range (1, 11):
            theta = j/10
            u = []
            u_d = []
            d = []
            for k in range (0, len(yy[i])):
                # u_i(t_i + (theta * h_i))
                u.append((yy[i][k] * h00(theta)) + (h * ffy[i][k] *
h10(theta)) + (yy[i + 1][k] * h01(theta)) + (h * ffy[i + 1][k] * h11(theta)))
```

```
                # u_i'(t_i + (theta * h_i))
                # Here u_d denotes the derivative of u
                u_d.append(((yy[i][k] / h) * h00_d(theta)) + (ffy[i][k] *
h10_d(theta)) + ((yy[i + 1][k] / h) * h01_d(theta)) + (ffy[i + 1][k] *
h11_d(theta)))

            # Here fu = f(t_i + (theta * h_i), u_i(t_i + (theta * h_i)))
            fu = f.formula(0, (tt[i] + (theta * h)), u[:])

            if (f.exactExists):
                # Exact value at (t_i + (theta * h_i))
                fe = f.formula(1, (tt[i] + (theta * h)), u[:])
                # Storing the value in the list named 'ffe'
                ffe.append(fe)

            # d_i(t_i + (theta * h_i)) = u_i'(t_i + (theta * h_i)) - f(t_i +
(theta * h_i), u_i(t_i + (theta * h_i)))
            for k in range (0, len(u)):
                d.append(u_d[k] - fu[k])

            # Storing the values in their corresponding lists
            t.append(tt[i] + (theta * h))
            uu.append(u)
            delta.append(d)
    return t, uu, ffe, delta

'''
Name: displayResults
Description: This function performs Hermite interpolation and displays the
                results in the console.
Parameters:
        reset  : reset is used to check if the lists containing the data
require
                    a reset or not. The default value for reset is False.
Returns:    None
'''
def displayResults(reset=False):
    # For a full display of hermite interpolant
    for i in range (0, len(config.t)):
        t, u, fe, d = hermite(config.t[i], config.y[i], config.f[i])
        if (f.exactExists):
            print ("i\t\t\tt\t\t\tu\t\t\tf\t\t\td")
            for i in range (0, len(t)):
                print (i+1, "\t", t[i], "\t", u[i], "\t", fe[i], "\t", d[i])
        else:
            print ("i\t\t\tt\t\t\tu\t\t\td")
            for i in range (0, len(t)):
                print (i+1, "\t", t[i], "\t", u[i], "\t", d[i])
        input("Press Enter to continue")
    if(reset):
        config.t = []
        config.y = []
        config.f = []

'''
Name: plotHermite
Description: This function performs Hermite interpolation and plots those
results
                on the graph. It provides two graphs:
                Hermite Interpolant.html
                Defect.html
Parameters : None
Returns    : None
```

```
'''
def plotHermite():
    # Calling the hermite() function on the data at the smallest stepsize
    t, u, fe, d = hermite(config.t[len(config.t)-1], config.y[len(config.y)-
1], config.f[len(config.f)-1])

    # Creating lists to prepare them for plotting
    t_list = []
    u_list = []
    if (f.exactExists):
        f_list = []
    d_list = []
    for j in range (0, len(u[0])):
        t_list.append(t)
        u_list.append([])
        if (f.exactExists):
            f_list.append([])
        d_list.append([])

    # Preparing the lists for plotting
    for i in range (0, len(t)):
        for j in range (0, len(u[i])):
            u_list[j].append(u[i][j])
            if (f.exactExists):
                f_list[j].append(fe[i][j])
            d_list[j].append(d[i][j])

    # Creating an object to create an HTML file for plotting
    bp.output_file("Plots/Hermite Interpolant.html")

    # Creating a figure to plot in the HTML file
    p = bp.figure(plot_width = 1366, plot_height = 768)

    # Plotting the data
    if (f.exactExists):
        p.multi_line(t_list + t_list, u_list + f_list)
    else:
        p.multi_line(t_list, u_list)

    # Showing the data in the browser
    bp.show(p)

    # Creating an object to create an HTML file for plotting
    bp.output_file("Plots/Defect.html")

    # Creating a figure to plot in the HTML file
    p = bp.figure(plot_width = 1366, plot_height = 768)

    # Plotting the data
    p.multi_line(t_list, d_list)

    # Showing the data in the browser
    bp.show(p)
```

Methods.py

```
import Function as f
import config

methodNumber = alpha = beta = case = 0

def displayMethods():
    print ("1. Forward Euler Method")
    print ("2. Explicit Midpoint Method")
    print ("3. Heun's Second Order Method")
    print ("4. Second Order RK Method")
    print ("5. Heun's Third Order Method")
    print ("6. Ralston's Third Order Method")
    print ("7. Third Order RK Method")
    print ("8. RK4 Method")
    print ("9. FourthOrderRKMethod")
    mname = input("\nEnter the method with values respectively (Use spaces
             between the values like shown above):\n")
    setMethodValues(mname, False)

def setMethodValues(mname, auto, caseNumber=None):
    global methodNumber, case

    methodNumber = int(mname)
    if ((methodNumber < 1) or (methodNumber > 9)):
        print ("No Method with that number.\n")
        displayMethods()
    else:
        if (auto):
            case = caseNumber
            autoChooseCase()
        else:
            userChooseCase()

def autoChooseCase():
    # To fully automate this, the coefficients are assigned with the optimal
             values
    global alpha, beta
    caseInfo = ""
    if (methodNumber == 4):
        alpha = 2/3
        caseInfo = " alpha=" + str(alpha)
    elif (methodNumber == 7):
        if (case == 1):
            alpha = 0.49650476
            beta = 0.75174749
            caseInfo = " c2=" + str(alpha) + " c3=" + str(beta)
        elif (case == 2):
            alpha = 1/8
            caseInfo = " b3=" + str(alpha)
        else:
            alpha = 3/8
            caseInfo = " b3=" + str(alpha)
    elif (methodNumber == 9):
        if (case == 1):
            alpha = 0.35774159
            beta = 0.59148821
            caseInfo = " c2=" + str(alpha) + " c3=" + str(beta)
        elif (case == 2):
            alpha = 0.83316441
            caseInfo = " b3=" + str(alpha)
        elif ((case == 3) or (case == 4)):
```

```
            alpha = 1/6
            caseInfo = " b" + str(case) + "="+ str(alpha)
        else:
            alpha = 1
            caseInfo = " c2=" + str(alpha)
    config.file.write(caseInfo)

def userChooseCase():
    global alpha, beta, case
    if (methodNumber == 4):
        alpha = input("Enter the alpha: ")
        if ("/" in str(alpha)):
            res = alpha.split('/')
            alpha = int(res[0]) / int(res[1])
        else:
            alpha = float(alpha)
    elif (methodNumber == 7):
        print ("Case 1: if c2≠0, 2/3, c3; c3≠0, c2, then enter: 1 c2 c3")
        print ("Case 2: if b3≠0, where c3=0, then enter: 2 b3")
        print ("Case 3: if b3≠0, where c3≠0, then enter: 3 b3")
        choice = input("\nEnter your case choice: ")
        data = choice.split()
        for i in range(1, len(data)):
            if ("/" in str(data[i])):
                res = data[i].split('/')
                data[i] = int(res[0]) / int(res[1])
            else:
                data[i] = float(data[i])

        case = int(data[0])
        if (case == 1):
            alpha, beta = data[1], data[2]
        elif ((case == 2) or (case == 3)):
            alpha = data[1]
        else:
            print ("No case of that choice.")
            exit(0)
    elif (methodNumber == 9):
        print ("Case 1: 0, c2, c3, 1 all distinct,",
               "\nc2≠1/2 and 3 - 4(c2 + c3) + 6*c2*c3 ≠ 0, then enter: 1 c2
              c3")
        print ("Case 2: c2 = c3 = 1/2, b3≠0, then enter: 2 b3")
        print ("Case 3: c2 = 1/2, c3 = 0, b3≠0, then enter: 3 b3")
        print ("Case 4: c2 = 1, c3 = 1/2, b4≠0, then enter: 4 b4")
        print ("Case 5: c2≠0, c3 = 1/2, b2 = 0, then enter: 5 c2")
        choice = input("\nEnter your case choice: ")
        data = choice.split()
        for i in range(1, len(data)):
            if ("/" in str(data[i])):
                res = data[i].split('/')
                data[i] = int(res[0]) / int(res[1])
            else:
                data[i] = float(data[i])

        case = int(data[0])
        if (case == 1):
            alpha, beta = data[1], data[2]
        elif ((case == 2) or (case == 3) or (case == 4) or (case == 5)):
            alpha = data[1]
        else:
            print ("No case of that choice.")
            exit(0)
```

```
def method(t, y, h):
    if (methodNumber == 1):
        return forwardEulersMethod(t, y)
    elif (methodNumber == 2):
        return explicitMidpointMethod(t, y, h)
    elif (methodNumber == 3):
        return HeunsSecondOrderMethod(t, y, h)
    elif (methodNumber == 4):
        return secondOrderRKMethod(t, y, h)
    elif (methodNumber == 5):
        return HeunsThirdOrderMethod(t, y, h)
    elif (methodNumber == 6):
        return RalstonsThirdOrderMethod(t, y, h)
    elif (methodNumber == 7):
        return thirdOrderRKMethod(t, y, h)
    elif (methodNumber == 8):
        return RK4Method(t, y, h)
    elif (methodNumber == 9):
        return FourthOrderRKMethod(t, y, h)

def forwardEulersMethod(t, y):
    fy = f.formula(0, t, y[:])
    config.ffy.append(fy)
    return fy

def explicitMidpointMethod(t, y, h):
    k1 = f.formula(0, t, y[:])
    config.ffy.append(k1)
    yn = []
    for i in range (0, len(k1)):
        yn.append(y[i] + ((h/2) * k1[i]))
    fy = f.formula(0, (t + (h/2)), yn[:])

    return fy

def HeunsSecondOrderMethod(t, y, h):
    k1 = f.formula(0, t, y[:])
    config.ffy.append(k1)
    yn = []
    for i in range (0, len(k1)):
        yn.append(y[i] + (h * k1[i]))
    k2 = f.formula(0, (t + h), yn[:])
    fy = []
    for i in range (0, len(k2)):
        fy.append((1/2) * (k1[i] + k2[i]))

    return fy

def secondOrderRKMethod(t, y, h):
    global alpha
    k1 = f.formula(0, t, y[:])
    config.ffy.append(k1)
    yn = []
    for i in range (0, len(k1)):
        yn.append(y[i] + (h * (alpha * k1[i])))
    k2 = f.formula(0, (t + (alpha * h)), yn[:])
    fy = []
    for i in range (0, len(k1)):
        fy.append(((1 - (1/(2 * alpha))) * k1[i]) + ((1/(2 * alpha)) *
             k2[i]))

    return fy
```

```
def HeunsThirdOrderMethod(t, y, h):
    c2 = 1/3
    c3 = 2/3
    b1 = 1/4
    b2 = 0
    b3 = 3/4
    a31 = 0
    a32 = 2/3
    k1 = f.formula(0, t, y[:])
    config.ffy.append(k1)

    yn = []
    for i in range (0, len(k1)):
        yn.append(y[i] + (h * (c2 * k1[i])))
    k2 = f.formula(0, (t + (c2 * h)), yn[:])

    yn.clear()
    for i in range (0, len(k2)):
        yn.append(y[i] + (h * ((a31 * k1[i]) + (a32 * k2[i]))))
    k3 = f.formula(0, (t + (c3 * h)), yn[:])

    fy = []
    for i in range (0, len(y)):
        fy.append((b1 * k1[i]) + (b2 * k2[i]) + (b3 * k3[i]))

    return fy

def RalstonsThirdOrderMethod(t, y, h):
    c2 = 1/2
    c3 = 3/4
    b1 = 2/9
    b2 = 1/3
    b3 = 4/9
    a31 = 0
    a32 = 3/4

    k1 = f.formula(0, t, y[:])
    config.ffy.append(k1)

    yn = []
    for i in range (0, len(k1)):
        yn.append(y[i] + (h * (c2 * k1[i])))
    k2 = f.formula(0, (t + (c2 * h)), yn[:])

    yn.clear()
    for i in range (0, len(k2)):
        yn.append(y[i] + (h * ((a31 * k1[i]) + (a32 * k2[i]))))
    k3 = f.formula(0, (t + (c3 * h)), yn[:])

    fy = []
    for i in range (0, len(y)):
        fy.append((b1 * k1[i]) + (b2 * k2[i]) + (b3 * k3[i]))

    return fy

def setValuesForThirdOrder(alpha):
    if (case == 1):
        c2 = alpha[0]
        c3 = alpha[1]
        b1 = (2 - (3 * (c2 + c3)) + (6 * c2 * c3)) / (6 * c2 * c3)
        b2 = (c3 - (2/3)) / (2 * c2 * (c3 - c2))
        b3 = ((2/3) - c2) / (2 * c3 * (c3 - c2))
        a31 = (c3 * (c3 - (3 * c2) + (3 * c2 * c2))) / (c2 * ((3 * c2) - 2))
```

```
            a32 = (c3 * (c2 - c3)) / (c2 * ((3 * c2) - 2))
        elif (case == 2):
            c2 = 2/3
            c3 = 0
            b3 = alpha[0]
            b1 = (1/4) - b3
            b2 = 3/4
            a31 = -1 / (4 * b3)
            a32 = 1 / (4 * b3)
        else:
            c2 = c3 = 2/3
            b3 = alpha[0]
            b1 = 1/4
            b2 = (3/4) - b3
            a31 = ((8 * b3) - 3) / (12 * b3)
            a32 = 1 / (4 * b3)

        return c2, c3, b1, b2, b3, a31, a32

    def thirdOrderRKMethod(t, y, h):
        #Here, alpha is used for c2 or b3 and beta for c3
        global alpha, beta

        c2, c3, b1, b2, b3, a31, a32 = setValuesForThirdOrder([alpha, beta])

        '''print ("\nc2 =", c2)
        print ("c3 =", c3)
        print ("b1 =", b1)
        print ("b2 =", b2)
        print ("b3 =", b3)
        print ("a31 =", a31)
        print ("a32 =", a32)'''

        k1 = f.formula(0, t, y[:])
        config.ffy.append(k1)

        yn = []
        for i in range (0, len(k1)):
            yn.append(y[i] + (h * (c2 * k1[i])))
        k2 = f.formula(0, (t + (c2 * h)), yn[:])

        yn.clear()
        for i in range (0, len(k2)):
            yn.append(y[i] + (h * ((a31 * k1[i]) + (a32 * k2[i]))))
        k3 = f.formula(0, (t + (c3 * h)), yn[:])

        fy = []
        for i in range (0, len(y)):
            fy.append((b1 * k1[i]) + (b2 * k2[i]) + (b3 * k3[i]))

        return fy

    def RK4Method(t, y, h):
        k1 = f.formula(0, t, y[:])
        config.ffy.append(k1)
        yn = []
        for i in range (0, len(k1)):
            yn.append(y[i] + ((h / 2) * k1[i]))
        k2 = f.formula(0, (t + (h / 2)), yn[:])
        yn.clear()
        for i in range (0, len(k2)):
            yn.append(y[i] + ((h / 2) * k2[i]))
        k3 = f.formula(0, (t + (h / 2)), yn[:])
```

```
    yn.clear()
    for i in range (0, len(k3)):
        yn.append(y[i] + (h * k3[i]))
    k4 = f.formula(0, (t + h), yn[:])
    fy = []
    for i in range (0, len(y)):
        fy.append((1/6) * (k1[i] + (2 * k2[i]) + (2 * k3[i]) + k4[i]))

    return fy

def FourthOrderRKMethod(t, y, h):
    #Here, alpha is used for c2, b3 or b4 and beta for c3
    global alpha, beta

    if (case == 1):
        c2 = alpha
        c3 = beta
        c4 = 1
        a31 = (c3 * ((3 * c2) - c3 - (4 * c2 * c2))) / (2 * c2 * (1 - (2 *
              c2)))
        a32 = (c3 * (c3 - c2)) / (2 * c2 * (1 - (2 * c2)))
        a41 = (((c3 ** 2) * ((12 * c2 * c2) - (12 * c2) + 4)) - (c3 * ((12 *
              c2 * c2) - (15 * c2) + 5)) + ((4 * c2 * c2) - (6 * c2) + 2)) /
              ((2 * c2 * c3) * (3 - (4 * (c2 + c3)) + (6 * c2 * c3)))
        a42 = (((-4 * c3 * c3) + (5 * c3) + c2 - 2) * (1 - c2)) / ((2 * c2) *
              (c3 - c2) * (3 - (4 * (c2 + c3)) + (6 * c2 * c3)))
        a43 = ((1 - (2 * c2)) * (1 - c3) * (1 - c2)) / (c3 * (c3 - c2) * (3 -
              (4 * (c2 + c3)) + (6 * c2 * c3)))
        b1 = (1 - (2 * (c2 + c3)) + (6 * c2 * c3)) / (12 * c2 * c3)
        b2 = ((2 * c3) - 1) / ((12 * c2) * (c3 - c2) * (1 - c2))
        b3 = (1 - (2 * c2)) / ((12 * c3) * (c3 - c2) * (1 - c3))
        b4 = (3 - (4 * (c2 + c3)) + (6 * c2 * c3)) / (12 * (1 - c2) * (1 -
              c3))
    elif (case == 2):
        b3 = alpha
        c2 = c3 = 1/2
        c4 = 1
        a31 = ((3 * b3) - 1) / (6 * b3)
        a32 = 1 / (6 * b3)
        a41 = 0
        a42 = 1 - (3 * b3)
        a43 = 3 * b3
        b1 = 1/6
        b2 = (2 / 3) - b3
        b4 = 1/6
    elif (case == 3):
        b3 = alpha
        c2 = 1/2
        c3 = 0
        c4 = 1
        a31 = -1 / (12 * b3)
        a32 = 1 / (12 * b3)
        a41 = (-1/2) - (6 * b3)
        a42 = 3/2
        a43 = 6 * b3
        b1 = (1/6) - b3
        b2 = 2/3
        b4 = 1/6
    elif (case == 4):
        b4 = alpha
        c2 = 1
        c3 = 1/2
        c4 = 1
```

```
    a31 = 3/8
    a32 = 1/8
    a41 = 1 - (1 / (4 * b4))
    a42 = -1 / (12 * b4)
    a43 = 1 / (3 * b4)
    b1 = 1/6
    b2 = 1/6 - b4
    b3 = 2/3
elif (case == 5):
    c2 = alpha
    c3 = 1/2
    c4 = 1
    a31 = ((4 * c2) - 1) / (8 * c2)
    a32 = 1 / (8 * c2)
    a41 = (1 - (2 * c2)) / (2 * c2)
    a42 = -1 / (2 * c2)
    a43 = 2
    b1 = 1/6
    b2 = 0
    b3 = 2/3
    b4 = 1/6

k1 = f.formula(0, t, y[:])
config.ffy.append(k1)

yn = []
for i in range (0, len(k1)):
    yn.append(y[i] + (h * (c2 * k1[i])))
k2 = f.formula(0, (t + (c2 * h)), yn[:])

yn.clear()
for i in range (0, len(k2)):
    yn.append(y[i] + (h * ((a31 * k1[i]) + (a32 * k2[i]))))
k3 = f.formula(0, (t + (c3 * h)), yn[:])

yn.clear()
for i in range (0, len(k3)):
    yn.append(y[i] + (h * ((a41 * k1[i]) + (a42 * k2[i]) + (a43 *
          k3[i]))))
k4 = f.formula(0, (t + (c4 * h)), yn[:])

fy = []
for i in range (0, len(y)):
    fy.append((b1 * k1[i]) + (b2 * k2[i]) + (b3 * k3[i]) + (b4 * k4[i]))

return fy
```

ivode.py

```
import math

# global constants for the IVODEs
alpha = beta = gamma = delta = theta = 0

'''
Name: setConstants
Description: This function sets the constants for the chosen IVODE.
Parameters:
        a           : contant for the IVODE.
        b, c, d, e  : contant for the IVODE (optional).
Returns:    None
'''
def setConstants(a, b=None, c=None, d=None, e=None):
    global alpha, beta, gamma, delta, theta
    alpha, beta, gamma, delta, theta = a, b, c, d, e

'''
Names: simple, predatorPrey, simple_sys, TestF4, TestF5, TestF6, TestF7
Description: These functions return the derivative values, exact
                values or associated error for the IVODEs with
                respect to user's choice.
Parameters:
        i  : i is an integer value to return the respective value.
        t  : t is the point on the domain.
        y  : y is the approximate numerical solution for the IVODE.
Returns:
        if (i == 0):
            the IVODE value.
        if (i == 1):
            exact value for the IVODE. (if exists)
        else:
            Error associated with the IVODE. (if exists)
'''
def simple(i, t, y):
    # IVODE
    if (i == 0):
        return [y[0] * (-1)]
    # Exact value for the IVODE
    elif (i == 1):
        return [math.exp((-1) * t)]
    # Error associated with the IVODE
    else:
        return [y[0] - math.exp((-1) * t)]

def predatorPrey(i, t, y):
    # IVODE
    if (i == 0):
        return [((alpha * y[0]) - (beta * y[0] * y[1])), ((delta * y[0] *
             y[1]) - (gamma * y[1]))]

def simple_sys(i, t, y):
    # IVODE
    if (i == 0):
        return [y[1], (y[0] * (-1))]
    # Exact value for the IVODE
    elif (i == 1):
        return [math.sin(t), math.cos(t)]
    # Error associated with the IVODE
    else:
        return [y[0] - math.sin(t), y[1] - math.cos(t)]
```

```
def TestF4(i, t, y):
    # IVODE
    if (i == 0):
        return [(-1/2) * (y[0] ** 3)]
    # Exact value for the IVODE
    elif (i == 1):
        return [(1 / math.sqrt(1 + t))]
    # Error associated with the IVODE
    else:
        return [y[0] - (1 / math.sqrt(1 + t))]

def TestF5(i, t, y):
    # IVODE
    if (i == 0):
        return [-2 * t * (y[0] ** 2)]
    # Exact value for the IVODE
    elif (i == 1):
        return [(1 / (1 + (t ** 2)))]
    # Error associated with the IVODE
    else:
        return [y[0] - (1 / (1 + (t ** 2)))]

def TestF6(i, t, y):
    # IVODE
    if (i == 0):
        return [(1/4) * y[0] * (1 - (y[0] / 20))]
    # Exact value for the IVODE
    elif (i == 1):
        return [(20 / (1 + (19 * math.exp(((-1) * t) / 4))))]
    # Error associated with the IVODE
    else:
        return [y[0] - (20 / (1 + (19 * math.exp(((-1) * t) / 4))))]

def TestF7(i, t, y):
    # IVODE
    if (i == 0):
        return [(-1 * alpha * y[0]) - (math.exp(-1 * alpha * t) *
             math.sin(t))]
    # Exact value for the IVODE
    elif (i == 1):
        return [(math.exp(-1 * alpha * t) * math.cos(t))]
    # Error associated with the IVODE
    else:
        return [y[0] - (math.exp(-1 * alpha * t) * math.cos(t))]

'''
Name: sampleCOVID19ModelInitializer
Description: This function sets the constants for the COVID-19 model
                and returns its initial values.
Parameters: None
Returns:
        initial values of the COVID-19 model.
'''
def sampleCOVID19ModelInitializer():
    setConstants(0.125, 0.9, 0.06, (0.01/365), 37.741e06)

    y03 = 1
    y02 = 103
    return [theta-y03-y02, y02, y03, 0]

'''
Names: sampleCOVID19Model
```

```
Description: This function returns the derivative values for the IVODEs.
Parameters:
        i  : i is an integer value to return the respective value.
        t  : t is the point on the domain.
        y  : y is the approximate numerical solution for the IVODE.
Returns:
        if (i == 0):
            the IVODE value.
'''
def sampleCOVID19Model(i, t, y):
    # IVODE
    if (i == 0):
        y1 = ((-beta * y[0] * y[2]) / theta) + (delta * theta) - (delta *
             y[0])
        y2 = ((beta * y[0] * y[2]) / theta) - ((alpha + delta) * y[1])
        y3 = (alpha * y[1]) - ((gamma + delta) * y[2])
        y4 = (gamma * y[2]) - (delta * y[3])
        return [y1, y2, y3, y4]
```

FileIO/FileIO.py

```
import os

class FileIO:
    def __init__(self, filename, accessMode):
        if accessMode in ['r', 'rb', 'r+', 'rb+', 'w', 'w+', 'wb', 'wb+',
             'a', 'a+', 'ab', 'ab+']:
            if ((accessMode in ['r', 'rb', 'r+', 'rb+']) and not
             (os.path.exists(filename))):
                print ("The given path is not a file, directory or a valid
             symlink.")
            else:
                self.file = open(filename, accessMode)
        else:
            print ("Invalid access mode.")

    def changeAccessMode(self, accessMode):
        if (self.isCreated()):
            self.file.close()
            self.file = open(self.file.name, accessMode)
        else:
            print ("The object is not initialized. Check the file path or the
             accessMode.")

    def read(self):
        if (self.isCreated()):
            if not (self.file.readable()):
                self.changeAccessMode("r")
            return self.file.read()
        else:
            print ("The object is not initialized. Check the file path or the
             accessMode.")
            return "Nothing to read"

    def readLine(self):
        if (self.isCreated()):
            if not (self.file.readable()):
                self.changeAccessMode("r")
            data = self.file.readline()
            return data
        else:
            print ("The object is not initialized. Check the file path or the
             accessMode.")
            return "Nothing to read"

    def write(self, data, end='\n'):
        if (self.isCreated()):
            if not (self.file.writable()):
                self.changeAccessMode("a")
            data = data + end
            self.file.write(data)
        else:
            print ("The object is not initialized. Check the file path or the
             accessMode.")

    def name(self):
        return self.file.name

    def mode(self):
        return self.file.mode

    def isCreated(self):
```

```
    return hasattr(self, 'file')

def __del__(self):
    if (self.isCreated()):
        self.file.close()
```