Implementation of 3 stage Lobatto IIIC into the Assimulo package

Edmund Aristid Lehsten

Bachelor's thesis
2021:K45

Lund University

Faculty of Science
Centre for Mathematical Sciences
Numerical Analysis

Bachelor's Theses in Mathematical Sciences 2021:K45
ISSN 1654-6229

LUNFNA-4039-2021

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

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

Implementation of 3 stage Lobatto IIIC into the Assimulo package

Edmund Aristid Lehsten

2021

Abstract

In recent years, the popularity of discontinuous Galerkin methods has increased. As shown in [19], a result exists that states that the Discontinuous Galerkin space approximations (DG) are equivalent to the Lobatto IIIC Runge-Kutta method. This thesis therefore outlines the adaptation of Hairer's implementation of the Radau IIA Runge-Kutta method to the Lobatto IIIC method, extended with an adaptation of Pinto et al.'s two step error estimation found in [17]. As an alternative to the classical Three staged Radau IIA Runge Kutta method.

Popular Scientific Description

Ordinary differential equations (ODEs) are the backbone of physics, chemistry and biology. However these cannot be solved over a continuous region in time and space, rather they need to be solved on a discrete grid, while iterating over different points in time.

Runge Kutta methods are one of the methods to solve these problems. In this thesis we discuss the implementation of one such method, the Lobatto IIIC, by adapting the implementation of a similar method, the Radau IIA, done by Hairer [14]. Further we expand upon the implementation by including a newer method by pinto et al. for the local error estimation at the different steps [17].

Acknowledgements

I would like to thank my advisor, Robert Klöfkorn, who helped guide me through this project.
My examiner Philipp Birken, for taking the time to thoroughly evaluate this work.
My professor Claus Führer who aided me whenever I was struggling with an aspect of the code in this project.
Lea Miko Versbach, who helped me spot errors that I would not have found otherwise.
My former teachers Tim Hylén and Per Wahlden, through who's tutelage I first developed my appreciation for mathematics.
And finally my friends Anna-Mariya Otsetova, Dustin Lindner Daii and Mirjam Karlsson-Müller, who kept me motivated throughout this work.

Introduction

In numerical mathematics a common field of interest is solving ordinary differential equations. These equations are commonly used when describing how values that depend on each other change over time. Many different methods for solving these problems exist, one group being the family of Runge-Kutta methods. However, it is not possible to solve these problems, for example the temperature in a room, at every single point within that region. Therefore the regions get discritized in space, which means that one picks a discrete number of points within this region and reformulates the problem to describe the relation between points. Another type of discretization is temporal discretization, where instead of picking points in space one picks points in time.

In recent years, discontinuous Galerkin methods have become more popular than previously [19]. These methods allow one to discritize in both space and time. This is benificial since it allows us to prove different properties mathematically, such as, for example entropy stability. However these methods are not as easy to use as the methods that we have for solving ODEs. Luckily a result exists proving that the Lobatto IIIC Runge Kutta method is equivalent to one of hte Discontinuous Galerkin methods, specifically the spatial Discontinuous Galerkin approximation [19]. Hence when we use the Lobatto IIIC method we have the same properties as we have for the Discontinuous Galerkin space approximation. There has not been to much focus on the Lobatto IIIC method since it has lower order than the Radau method. Furthermore the difference between the methods is solely in their coefficient. This work therefore aims to adapt Heirer's implementation for the Radau IIA Runge Kutta method into an implementation of the Lobatto IIIC method [14]. Furthermore we will incorporate newer results from pinto et al. that improve upon the error estimation disccused by Hairer [17].

We begin by giving a formal definition for the Runge-Kutta method in Section 2. Section 2.1 focuses on the initial conversion of the method into a linear algebra setting. In Section 2.2 we focus on the Newton iteration in the method, along with its starting and stopping conditions. Section 2.3 discusses further simplifications that can be done to the method to improve simulation time. In Sections 2.4 and 2.5 we discuss the approximation of the error and how to apply this to dynamically adjust the step size of the method. We then mention a few differences between the coefficients and their corresponding Runge-Kutta methods while also introducing the Lobatto IIIC coefficient in section 2.6
Section 3 focuses on the package we implemented our method in, while Section 4 discusses the numerical results.

Contents

Chapter 1

Prerequisites

Throughout this work we assume general knowledge of numerical linear algebra and numerical analysis. However, in the rest of this section we shall reiterate some of the concepts that will be used heavily throughout the rest of this work.

1.1 Numerical Linear Algebra

We shall begin this section by giving the definition of the Kronecker product which we utilize later throughout this work and then follow up with the LU-decomposition in conjunction with the Newton iteration.

Definition 1.1.1. *The **Kronecker product** (⊗) between two matricise **A** and **B** is defined as*

$$\boldsymbol{A} \otimes \boldsymbol{B} = \begin{bmatrix} a_{1,1}\boldsymbol{B} & \cdots & a_{1,n}\boldsymbol{B} \\ \vdots & \ddots & \vdots \\ a_{m,1}\boldsymbol{B} & \cdots & a_{m,n}\boldsymbol{B} \end{bmatrix}$$

Given a function f, $f : \mathbb{R}^n \to \mathbb{R}^n$, the Newton iteration is a method which attempts to find zeros of the function f. In other words, we obtain an x such that $f(x) = 0$. This method is defined by the following iterative formula.

$$x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})}$$

Remark 1.1.2. *The Newton iteration can be generalized to multiple dimensions by using the Jacobian $\mathscr{J}$ of f instead of f' and solving the following instead iteratively*

$$\mathscr{J}\Delta x^{(k)} = f(x^{(k)}) \tag{1.1}$$

$$x^{(k+1)} = x^{(k)} - \Delta x^{(k)} \tag{1.2}$$

Note the equation (1.1) is a linear system of equations.

If the starting point, $x^{(0)}$, of the Newton iteration is close enough to the wanted solution, we can use something called the **simplified Newton iteration**. The difference between these two methods is that the Jacobian (or derivative) is only evaluated once and then reused for the rest of the steps. The benefit with this is that computing the Jacobian can be costly depending on the function, the simplified method removes this cost as it only computes it once. Despite that the simplified Newton method comes with the trade of that it is only locally linearly convergent instead of being quadratically convergent like the classical method. This means that the domain in which a starting point will converge to a zero can be smaller depending on the function. Further more the simplified Newton method often requires slightly more iterations to converge. However since the computation of the Jacobian can be extremely costly overall the simplified Newton method is still an improvement [7]. To further improve upon this method we introduce the LU-decomposition

Definition 1.1.3. *An **LU-decomposition** of a square matrix A is the decomposition $A = LU$, where L is a lower triangular matrix and U is an upper triangular matrix.*

The benefit of the LU-decomposition becomes apparent when repeatedly solving linear systems of equations. Solving a lower/upper triangular system is done by either forward or backwards substitution and hence is faster to compute than a full system. When solving $Ax = b$ we instead solve for $Ly = b$ and then for $Ux = y$. Both of these are straight forward to compute and hence solving a system given the LU-decomposition of A is more efficient than solving the original system. Therefore, since we want to repeatedly solve $\mathscr{J}\Delta x^{(k)} = f(x^{(k)})$, for different $x^{(k)}$, we can compute the LU-decomposition of $\mathscr{J}$ once and use it for the repeated Newton iterations to make the computations more efficient.

1.2 Ordinary Differential Equations and Numerical Mathematics

This section is based no chapter 12 in [18].
Ordinary differential equations (ODE)s are common in numerical mathematics. Many different methods exist for finding approximate solutions to these problems. The aim of this work is to implement such a method. However, before we can begin discussing the method we will focus on later, some apriori knowledge is required. We begin with the definition of an ODE and initial value problem.

Definition 1.2.1. *An **ordinary differential equation** (ODE) is an equation of the form*

$$y'(x) = f(x, y(x)) \tag{1.3}$$

where $y(x)$ is a function and x is a variable, commonly x is considered to be time.

As with integrating where the solution contains an arbitrary constant, ODEs can have an infinite amount of solutions known as **solution curves**.

To uniquely define just one solution curve, an initial value (x_0, y_0), with $y(x_0) = y_0$, is given. This value is a form of starting point from which we begin drawing our curve.

Definition 1.2.2. *An **initial value problem** is an ODE together with an initial value.*

Commonly a numerical method can only find an approximate solution $\tilde{y}(x)$ of $y(x)$ for a given x. To arrive at a certain point x, most methods take multiple steps calculating $y_n \approx y(x_n)$ for each step, i.e., $x_0 \to x_1 \to x_2 \to \cdots \to x_n = x$. The step size of a step $x_i \to x_{i+1}$ is $h_i := x_{i+1} - x_i$.

To measure the accuracy of these methods we can observe the **global error** of the method. That is we can chooses a point x_n and compute $\tilde{y}(x_n)$ and then compare this result with the solution $y(x_n)$. However, in most cases a true solution in unknown. We can therefore not evaluate the accuracy of the method. To ensure that the method is still sufficiently accurate we instead study a **local error**[*] estimate which compares two methods of different order at the same point, as found in [8]. We denote this error at x_n as T_n: the greater T_n is the greater the local error is. The order of a method is defined as follows

Definition 1.2.3. *A numerical method is said to have* ***order (of accuracy)*** *p, if p is the largest positive integer such that, for any sufficiently smooth solution curve* $(x, y(x))$ *in the solution domain of the initial value problem, there exist constants* K *and* h_0 *such that*

$$|T_n| \leq Kh^p \quad \textit{for } 0 < h \leq h_0$$

for any pair of points $(x_n, y(x_n))$, $(x_{n+1}, y(x_{n+1}))$ *on the solution curve*

Definition 1.2.4. *A numerical method is* ***consistent*** *with the differential equation* (1.3) *if the local error, is such that for any* $\epsilon > 0$ *there exists a positive* $h(\epsilon)$ *for which* $|T_n| < \epsilon$ *for* $0 < h < h(\epsilon)$ *and any pair of points* $\left(x_n, y_n\right), \left(x_{n+1}, y_{n+1}\right)$ *on the solution curve.*

[*]Also known as the Local Truncated Error

Chapter 2

The Runge-Kutta Method

The idea of the Runge-Kutta method is to extend upon the standard Euler method, as defined in [18], for numerical integration. Instead of evaluating the function f once per step, one evaluates it multiple times. This idea of using a multiplicity of evaluations was originally proposed by Runge in 1895, and was developed further by Heun in 1900 and Kutta in 1901. The latter characterized the set of Runge-Kutta methods of order 4 and proposed the first methods of order 5, as Butcher points out in [8].
Given an initial value problem as defined in definition 1.2.2, we can use the Runge-Kutta method to take a step from $x_0 \to x_1 = x_0 + h$ and find $y_1 = f(x_1)$.

Definition 2.0.1. *An s-stage Runge-Kutta method is defined as:*

$$g_i = y_0 + h \sum_{j=1}^{s} a_{i,j} f(x_0 + c_j h, g_j) \quad , i = 1, \ldots, s \tag{2.1a}$$

$$y_1 = y_0 + h \sum_{j=1}^{s} b_j f(x_0 + c_j h, g_j) \tag{2.1b}$$

with $a_{i,j}, b_i, c_i$ being constant coefficients that are method specific (Radau IIA, Lobbato IIIC, etc.), and h being the step size .

Note that h needs to be sufficiently small for the Runge-Kutta method to be stable.

The Runge-Kutta methods can be roughly grouped into two variants, the explicit and the implicit methods, the explicit methods having the further restriction on the coefficient that $a_{i,j} = 0 \quad \forall j \geq i$, this alters the sum in (2.1a) and the method becomes as follows:

Definition 2.0.2. *An s-stage explicit Runge-Kutta method is defined as:*

$$g_i = y_0 + h \sum_{j=1}^{i-1} a_{i,j} f(x_0 + c_j h, g_j), \quad i = 1, \ldots, s \tag{2.2a}$$

$$y_1 = y_0 + h \sum_{j=1}^{s} b_j f(x_0 + c_j h, g_j) \tag{2.2b}$$

with $a_{i,j}, b_i, c_i$ being constant coefficients that are method specific.

We are interested in the Lobatto IIIC method which is an implicit method. We therefore refer to definition 2.0.1 when we talk about the a Runge-Kutta method throughout the rest of this work.

The selection of these constants results from further conditions imposed on the method ensuring it is consistent and converges. However, the creation of these coefficients is beyond the scope of this text. For further reference refer to Butcher [8, section 33 & 340]. Additionally, Butcher introduced the following structure for grouping the coefficients [8]

$$\begin{array}{c|cccc}
c_1 & a_{1,1} & a_{1,2} & \cdots & a_{1,s} \\
c_2 & a_{2,1} & a_{2,2} & \cdots & a_{2,s} \\
\vdots & \vdots & \vdots & & \vdots \\
c_s & a_{s,1} & a_{s,2} & \cdots & a_{s,s} \\
\hline
 & b_1 & b_2 & \cdots & b_s
\end{array}$$

For further convenience we shall group the individual coefficients into a matrix and vectors following the same structure:

$$\begin{aligned}
A &= \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,s} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,s} \\ \vdots & \vdots & & \vdots \\ a_{s,1} & a_{s,2} & \cdots & a_{s,s} \end{bmatrix} \\
b &= (b_1, \cdots, b_s) \\
c &= (c_1, \cdots, c_s)
\end{aligned}$$

2.1 Initial Implementation

The following sections are based on chapter IV.8 from Hairer [14] unless stated otherwise.
To start the implementation of the implicit Runge-Kutta method, we begin by introducing z_i as

$$z_i = g_i - y_0 \tag{2.3}$$

This change is done to reduce round-off errors in floating point calculations[*]. g_i are the stage values, i.e., the values that we would have at times $t + c_i h$ respectively. By looking at the difference we get that z_i are the increments of the stage values. This is desired since the increments should be small when taking small steps. Observe that the terms in (2.3) are all vectors of dimension n. By calculating the increment instead of the actual value we avoid round-off errors.

Incorporating these terms into equation (2.1a) we get

$$z_i = h \sum_{j=1}^{s} a_{i,j} f(x_0 + c_j h, y_0 + z_i) \ \ i = 1, \ldots, s \tag{2.4}$$

[*]When considering all numbers that can be represented by the float point data structure, these numbers are more dense around 0 and less the further away they are.

Note equation (2.4) can be written in vector notation as follows:

$$\mathbf{Z} = \begin{pmatrix} z_0 \\ z_1 \\ \vdots \\ z_s \end{pmatrix} = A \begin{pmatrix} hf(x_0 + c_1 h, y_0 + z_1) \\ hf(x_0 + c_2 h, y_0 + z_2) \\ \vdots \\ hf(x_0 + c_s h, y_0 + z_s) \end{pmatrix} \tag{2.5}$$

Observe that each term z_i is a vector of size n.
Then, assuming A^{-1} exists, we can introduce $(d_1, \ldots, d_s) = (b_1, \ldots, b_s)A^{-1}$, and equation (2.1b) becomes:

$$y_1 = y_0 + \sum_{j=1}^{s} d_j z_j \tag{2.6}$$

Note that in equation (2.6) the d_j are scalars. Furthermore the d_j's are only defined by the method constants and can thus be pre-calculated. This removes the need to determine the inverse of A while using the method.

2.2 Newton's Method

We now use Newton's method to evaluate $\vec{Z}$ in equation (2.5). For this we rearrange the equation to:

$$\vec{Z} - A \begin{pmatrix} hf(x_0 + c_1 h, y_0 + z_1) \\ hf(x_0 + c_2 h, y_0 + z_2) \\ \vdots \\ hf(x_0 + c_s h, y_0 + z_s) \end{pmatrix} = 0 \tag{2.7}$$

The Jacobian of this system is:

$$I - h \begin{pmatrix} a_{1,1} \frac{\delta f}{\delta y}(x_0 + c_1 h, y_0 + z_1) & \cdots & a_{1,s} \frac{\delta f}{\delta y}(x_0 + c_s h, y_0 + z_s) \\ \\ a_{s,1} \frac{\delta f}{\delta y}(x_0 + c_1 h, y_0 + z_1) & \cdots & a_{s,s} \frac{\delta f}{\delta y}(x_0 + c_s h, y_0 + z_s) \end{pmatrix} \tag{2.8}$$

Note that in equation (2.8) each term $\frac{\delta f}{\delta y}(\cdot, \cdot)$ is a matrix of dimension $n \times n$. Moreover these terms can all be approximated as follows:

$$\frac{\delta f}{\delta y}(x_0 + c_i h, y_0 + z_i) \approx \frac{\delta f}{\delta y}(x_0, y_0) = \mathscr{J}$$

This allows us write an approximation to (2.8) as follows:

$$I - hA \otimes \mathscr{J} \tag{2.9}$$

where $\otimes$ is the Kronecker product as in definition 1.1.1. Note that from this point on I always denotes the identity matrix of appropriate dimension.
Thus, the Newton iteration becomes

$$(I - hA \otimes \mathscr{J})\Delta\vec{Z}^k = -\vec{Z}^k + h(A \otimes I)F(\vec{Z}^k) \tag{2.10a}$$
$$\vec{Z}^{k+1} = \vec{Z}^k + \Delta\vec{Z} \tag{2.10b}$$

with $F(\vec{Z}^k) = \big(f(z_0), \ldots, f(z_s)\big)$

2.2.1 Newton Starting Value

There are different alternatives for the starting value of the Newton iteration. Since we are taking a small step and are calculating the increments, we know that they should be small. Hence setting $Z^0 = \mathbf{0}$ (that is the zero vector of suitable size) is a valid choice, and is what we decided to use. However, other options exist, for instance one could take a small initial step using the explicit Euler method and then use that as the starting value. These other methods have the benefit of arriving at the stopping condition (see next section) in fewer iterations with the downside of being slightly more computationally intensive in the setup. However, the reduction in iterations can often compensate for their initial setup expense and, overall, these methods could yield faster solution compared to the basic approach. Our choice was made to simplify the implementation and since the final problem we will test this method on, see section 4, is linear and a single iteration will be sufficient.

2.2.2 Newton Stopping Condition

For the stopping condition to the Newton iteration we again look to Hairer [14, see IV.8]. We begin by estimating the convergence rate

$$\Theta_k = \frac{||\Delta Z^k||}{||\Delta Z^{k-1}||} \tag{2.11}$$

where ΔZ^k is the result of this iteration and ΔZ^{k-1} the result of the previous iteration. Then, we define the stopping condition to stop iterating if

$$\eta_k ||\Delta Z^k|| \leq \kappa \cdot Tol \quad \text{with} \quad \eta_k = \frac{\Theta_k}{1-\Theta_k} \tag{2.12}$$

holds[†].
Once stopped we accept Z^k as the approximation Z^* to Z. Here Tol denotes the desired tolerance of the solution. As mentioned by Hairer, κ is arbitrary and must be chosen by the user, Hairer suggest a value around 10^{-1} or 10^{-2}. To reduce the necessary Jacobian computation, we shall only recompute the Jacobian if the Newton iterations start to diverge, which we define by $\Theta_k \geq 1$.

2.3 Simplification for Implementation

The Runge-Kutta method discussed above has a few drawbacks, mainly in the computational intensity. Currently for each Newton iteration we must compute:

- s evaluations of f
- solve an $n \cdot s \times n \cdot s$-dimensional linear system

[†]Note that this stopping condition for the Newton iteration requires a minimum of two steps. Hairer makes a suggestion on how this could be improved for linear (or near linear) problems where one iteration would be sufficient. We have implemented it in code but left it out of this work as it is of no further interest.

If the diagonalization of A^{-1} exists i.e.

$$T^{-1}A^{-1}T = \Lambda \tag{2.13}$$

then, by pre-multiplying (2.10a) with $(hA^{-1}) \otimes I$ and introducing the term $W^k = (T^{-1} \otimes I)Z^k$, equations (2.10a) and (2.10b) become:

$$(H^{-1}\Lambda \otimes I - I \otimes J)\Delta W^k = -h^{-1}(\Lambda \otimes I)W^k + (T^{-1} \otimes I)F\big((T \otimes I)W^k\big) \tag{2.14a}$$
$$W^{k+1} = W^k + \Delta W^K \tag{2.14b}$$

From this point on we will focus mainly on the Runge-Kutta methods with $s = 3$[‡].
For the ideal case that A^{-1} has three real eigenvalues $\lambda_1, \lambda_2, \lambda_3$ then

$$\Lambda = \begin{bmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{bmatrix} \tag{2.15}$$

This would imply that the matrix in front of ΔW^k in equation (2.14a) would become:

$$\begin{bmatrix} \lambda_1 I - \mathscr{J} & 0 & 0 \\ 0 & \lambda_2 I - \mathscr{J} & 0 \\ 0 & 0 & \lambda_3 I - \mathscr{J} \end{bmatrix} \tag{2.16}$$

Note here that each element is a block matrix of size $n \times n$. Solving this system can be simplified by splitting it into solving three $n \times n$ systems instead of solving one $3n \times 3n$ system. Unfortunately, most for most Runge-Kutta methods their respective A^{-1} matrices do not have three real eigenvalues. Rather they have one real, γ, and a complex pair, $\alpha \pm \beta i$. In this case, the matrix infront of ΔW^k in equation (2.14a), $\big($the equivalent to (2.16)$\big)$ would look as follows:

$$\begin{bmatrix} \gamma I - \mathscr{J} & 0 & 0 \\ 0 & \alpha I - \mathscr{J} & -\beta I \\ 0 & \beta I & \alpha I - \mathscr{J} \end{bmatrix} \tag{2.17}$$

Here again one can split it into smaller sections to help the computer while solving this. Therefore instead of one $3n \times 3n$ system it would come down to one $n \times n$ and one $2n \times 2n$ system. For the $2n \times 2n$ system further tricks exist to aid the computer in solving this. However, that is beyond the scope of this text[§]. Since the left hand side (LHS) of equation (2.14a) does not change throughout the consecutive iterations within one time step we compute the LU-decomposition of these two matrices (the $n \times n$ and $2n \times 2n$) to significantly decrease the solve time of the systems. If the function and Jacobian are relatively easy to evaluate then this decomposition becomes the bottle neck, that is to say the most costly part, of the method. Hence, if the step size does not change one can reuse the decomposition from the previous step.

[‡]This is done since the source text [14] also makes this choice.

[§]for reference refer to Hairer page 122 [14]

2.4 Local Truncation Error

To estimate the local error we use a similar method to Hairer in chapter IV.8 [14]. His method consists of computing an approximate solution $\hat{y}_n \approx y_n$ which is of slightly lower order. Then the difference between these two solutions gives an estimate of the error $err = \hat{y}_n - y_n$.

$$\hat{y}_{n+1} = y_n + h \sum_{i=1}^{3} \hat{b}_i z_i \tag{2.18}$$

Remark 2.4.1. *These error methods are also known as embedded methods since they reuse the z_i terms from the current step.*

It now remains to find coefficients $\hat{b}_i$ such that the method in (2.18) is of the desired order. As mentioned by Pinto et al. in [17] finding embedded methods of order higher than $s-1$ is in most cases not possible. This means that our lower order approximation will be of order 2. One possible choice would be $\left(\hat{b}_1, \hat{b}_2, \hat{b}_3\right) = \left(-\frac{1}{2}, 2, -\frac{1}{2}\right)$ as found in [20]. On the other hand the difference between the order of the two methods is not ideal. We therefore turn to Butcher's idea found in [8], where he suggests to look for a method with $s+1$ stages where the first s stages are identical. With this we would hope to find a method of order 3. Unfortunately, as mentioned, it is unlikely to find such embedded methods for methods of order $\geq 2s-2$. Furthermore, these methods would require more function evaluations which is undesired since they could be costly. Hairer instead suggests using the method of order $s-1$ but adding an explicit step at it and arrive at the following method:

$$\hat{y}_{n+1} = y_n + h\hat{b}_0 f(x_0, y_0) + \sum_{i=1}^{3} \hat{b}_i z_i \tag{2.19}$$

Hairer further suggests the choice of $\hat{b}_0 = \gamma$, with γ being the real eigenvalue of A^{-1}. Through this explicit step the hope is to improve the order up to s. In our case, $s = 3$, the embedded method would then have order 3 while the actual method has order 4. Note, however, that the embedded method still requires a further function evaluation. Consequently, we look at the suggestion by Fabien and Jason in [13]. Their idea was that instead of one step of size h, one makes two of size $h/2$. Then an approximation can be computed as

$$\hat{y}_{n+1} = y_n + \sum_{i=1}^{2s} \hat{b}_i z_i \tag{2.20}$$

where $z_1, \ldots, z_s$ are the stage values for the first step and $z_{s+1}, \ldots, z_{2s}$ are the stage values for the second step. This method has one drawback which is that whenever a step is rejected one must reject the last two steps since they must always be of equal step size. This issue was addressed by Pito et al. [17]. Their idea was to adjust the coefficients b_i depending on the ratio between the last two step sizes.

Assume that the last step $y_{n-1} \to y_n$ with $t_n = t_{n-1} + h_{n-1}$ has been successful. Then for the approximate solution of the next step $y_n \to y_{n+1}$ with $t_{n+1} = t_n + h_n$, we have

$$\hat{y}_{n+1} = y_n + \sum_{i=1}^{3} \delta_{n,i} z_{n-1,i} + \sum_{i=1}^{3} \beta_{n,i} z_{n,i} \tag{2.21}$$

here $z_{n-1,i}$,$z_{n+1,i}$ denote the stage values from $y_{n-1} \to y_n$ and $y_n \to y_{n+1}$ respectively and $\delta_{n,i}, \beta_{n,i}$ both depend on $r_n = \frac{h_n}{h_{n-1}}$. These coefficients can be determined from the following conditions:

$$\beta_n^T e_s = 0, \ \delta_n^T \mathbf{1} = 0 \tag{2.22}$$

$$\delta_n^T A(c-\mathbf{1})^{j-1} + r_n^j \beta_n^T A c^{j-1} = \frac{r^j}{j} \quad j = 1, \dots, \hat{p} \tag{2.23}$$

where e_s is a vector of dimension s with all zeros except for the last term which is 1, $\mathbf{1}$ is an s-dimensional vector filled with 1's and $\hat{p}$ is the desired approximation order. Note that for our case, $s = 3$ an $\hat{p} = 3$, equations (2.22) and (2.23) give us and under determined system. We therefore wish to impose a further condition, to that end we must define the stability function $r(z)$ of a numerical method.

This section on the stability function is based on [15]
We being by defining the linear stability domain of a method.

Definition 2.4.2. *For a numerical method with constant step size h, with results $y_n \approx y(x_0 + h \cdot n)$ applied to the linear ODE*

$$y'(t) = \lambda y(t), \quad t \geq 1, \quad y(0) = 1, \tag{2.24}$$

*the **stability domain** $\mathscr{D}$, of the method, is the set of all numbers $h \cdot \lambda \in \mathbb{C}$ such that the limit* $\lim_{n\to\infty} y_n = 0$.

Remark 2.4.3. *Note that for $Re(\lambda) < 0$ we have that for the exact solution of* (2.24)*, the limit* $\lim_{t\to\infty} y(t) = 0$

When applying a Runge-Kutta method to (2.24), we get that

$$\xi_i = y_0 + h \sum_{j=1}^{s} a_{i,j} \xi_j, \quad i = 1, \dots, s, \tag{2.25a}$$

$$y_{n+1} = y_n + h\lambda \sum_{j=1}^{s} b_j \xi_j, \tag{2.25b}$$

with

$$\xi = \begin{bmatrix} \xi_1 \\ \xi_2 \\ \vdots \\ \xi_s \end{bmatrix} = (I - h\lambda A)^{-1} \mathbf{1} y_n, \tag{2.26}$$

where $\mathbf{1}$ is the vector of dimension s with all ones. We denote by $\mathbb{P}_\alpha$ the set of polynomials of maximum order α. Similarly, by $\mathbb{P}_{\alpha/\beta}$ we denote the set of rational functions $\hat{p}/\hat{q}$ where $\hat{p} \in \mathbb{P}_\alpha$ and $\hat{p} \in \mathbb{P}_\beta$.

Lemma 2.4.4. *For every Runge-Kutta method of order s applied to* (2.24) *there exists $\rho \in \mathbb{P}_{s/s}$ such that*

$$y_n = [\rho(h\lambda)]^n, \quad n = 0, 1, \dots \tag{2.27}$$

Proof. We shall only give a brief outline of the proof here.
From (2.25b),(2.26) and (2.27) we get that

$$\begin{aligned} y_{n+1} &= y_n + h\lambda \sum_{j=1}^{s} b_j \xi_j \\ &= \left[1 + h\lambda b^{\mathrm{T}}(I - h\lambda A)^{-1}\mathbf{1}\right] y_n \\ \Rightarrow \left[\rho(h\lambda)\right]^{n+1} &= \left[1 + h\lambda b^{\mathrm{T}}(I - h\lambda A)^{-1}\mathbf{1}\right] \cdot \left[\rho(h\lambda)\right]^{n} \\ \Rightarrow \rho(h\lambda) &= 1 + h\lambda b^{\mathrm{T}}(I - h\lambda A)^{-1}\mathbf{1} \\ \Rightarrow \rho(z) &= 1 + z b^{\mathrm{T}}(I - zA)^{-1}\mathbf{1}, \quad \text{with } z \in \mathbb{C} \end{aligned} \tag{2.28}$$

The rest of this proof now consists of proving that $\rho(z)$ is indeed an element of $\mathbb{P}_{s/s}$, the proof of which we shall not shown here but can be found in [15]. □

Lemma 2.4.5. *Suppose that an application of a numerical method to the linear ODE* (2.24) *produces a solution sequence* $y_n = \left[\rho(h\lambda)\right]^n$, $n = 0, 1, \ldots$, *where* ρ *is an arbitrary function. Then*

$$\mathscr{D} = \{z \in \mathbb{C} : |\rho(z)| < 1\}. \tag{2.29}$$

Proof. The proof of this follows directly from the definition of $\mathscr{D}$ in Definition 2.4.2 □

As Pinto explains, we can view the embedded method (2.21) as a Runge-Kutta method of $2s$ stages. Therefore, by lemma 2.4.4, a function $\rho(z)$ exists such that $y_n = [\rho(h\lambda)]^n$ for this method. We denote it by $\hat{R}(z)$. We can now impose the further condition on our coefficient in (2.21) that $\lim_{|z|\to\infty} \hat{R}(z) = \hat{R}(\infty) = 0$[¶]. This condition ensures the stability of the embedded method and is enforced through the following equation, as found in [17].

$$\hat{R}(\infty) = 1 - \beta_n^T \mathbf{1} + \frac{(-1)^{s+1}}{s} \delta_n^T A^{-1} \mathbf{1} = 0 \tag{2.30}$$

Solving this system of equations (2.22,2.23,2.30) for Lobatto IIIC coefficients (see section 2.6.2) with respect to $r = r_n$ with the condition that $\hat{p} = 3$ results in the following coefficients for the embedded method:

$$\begin{array}{r|rl} \beta_1 = \frac{12r^3+14r^2+21r+9}{4r^3+4r^2+3r-3} & \delta_1 = & \frac{12r^3+9r^2+3r}{4r^3+4r^2+3r-3} \\ \beta_2 = \frac{16r^3+8r^2-12r-12}{4r^3+4r^2+3r-3} & \delta_2 = & \frac{8r^4-24r^3-36r^2-12r}{4r^3+4r^2+3r-3} \\ \beta_3 = 0 & \delta_3 = & -\delta_1 - \delta_2 \end{array} \tag{2.31}$$

Note that $\beta_1, \beta_2, \delta_1, \delta_2, \delta_3$ all become undefined when $r = 0.5$. Plotting these coefficients we get the graph seen in Figure 2.1.
To avoid errors as r gets close to 0.5 we redefine r to be 0.4 if r is in $[0.4, 0.6]$ and adjust h_{n+1} accordingly before we take our step. This prevents any errors that could appear due to multiplying with large numbers and circumvents dividing by 0. Since this can only occur when the step size is currently decreasing, it just decreases it a bit extra. See the next section for more on the step size adjustment.

[¶]This is also beneficial to us since we do not fulfill Theorem 2.1 in [17] which implies that $\hat{R}(\infty)$ is bounded, hence by requiring this condition we ensure that it stays bounded

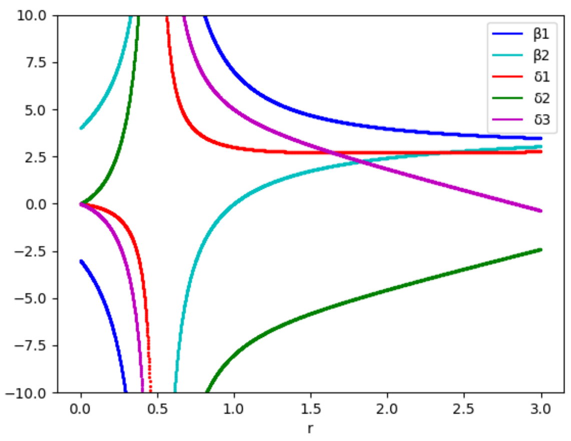


Figure 2.1: plotting the coefficient in (2.31)

Observer that the above error estimate does not work for the initial step, since we cannot calculate r. Thus, for the first step we must use a different method. Pinto et al. suggest solving the above equations with the added restriction $\delta_1 = \delta_2 = \delta_3 = 0$. However, since that would also result in an order 2 method we instead choose to use method (2.19) mentioned before. We now have an approximate error in each term as follows

$$err = \hat{y}_{n+1} - y_{n+1} \tag{2.32}$$

Note that both methods for $\hat{y}_{n+1}$ are of the form $y_n + \sum$. Similarly we have $y_{n+1} = y_n + \sum$. In fact for Lobatto IIIC with $s = 3$ we have $y_{n+1} = y_n + z_3$. We therefore change the error estimates to

$$err = h\gamma f(x_0, y_0) + \sum_{i=1}^{3} \hat{b}_i z_i - z_3 \tag{2.33}$$

for the first step and

$$err = a\left(\sum_{i=1}^{3} \delta_{n,i} z_{n-1,i} + \sum_{i=1}^{3} \beta_{n,i} z_{n,i} - z_3\right) \tag{2.34}$$

for all other steps. The term a is introduced by Pinto et al. and is a free parameter with the only restriction being $a > 0$. We chose a value of $a = 0.05$ which yielded good results, after doing some initial numerical experimentation with the problem discussed in section 4. However, a more thorough investigation into this parameter would be needed to optimize this error estimation and ensure that it works for a general problem.

2.5 Step Size Prediction

For this section we use h_{new} synonymously to h_{n+1} from the previous section on error estimation. The approximate error calculated in (2.32) is per term, we now need to combine these n terms into a single term. For this we use the norm introduced by Hairer in [14] :

$$||err|| = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\frac{err_i}{sc_i}\right)^2} \tag{2.35}$$

with $sc_i = Atol_i + \max(|y_{0,i}|, |y_{1,i}|) * Rtol_i$. The index i refers to the i'th element in the corresponding vectors. $Atol$ and $Rtol$ are the desired absolute and relative tolerances per index in the resulting vector. The difference between this norm and a the two-norm is that this one weights the different indexes, giving one the possibility to be more precise in one index than the other.
Hairer then suggests the following step size control formula[†]

$$h_{new} = fac \cdot h_{old} \cdot ||err||^{-1/4} \tag{2.36}$$

The power of $-1/4$ stems from the order of the approximate solution $\hat{y}_{n+1}$ being 3, which means that the exact solution $\widetilde{y_n} = \hat{y}_n + \mathcal{O}(h^4)$. Note however, that in the first time step our approximate solution $\hat{y}_{n+1}$ is only of order 2 therefore in the first step instead of (2.36) we use

$$h_{new} = fac \cdot h_{old} \cdot ||err||^{-1/3} \tag{2.37}$$

The fac term is dependent on the number of Newton iterations and is defined as

$$fac := 0.9 \times \frac{2k_{\max} + 1}{2k_{\max} + Newt}$$

with $k_{\max}$ being the maximum number of Newton iterations and $Newt$ being the actual number required until the stopping condition was reached. Hairer commented that they noticed best results setting $k_{\max}$ to around 7 or 10.

To not have to recompute the LU-decomposition on every step we shall keep the old step size h_{old} if

$$c_1 h_{old} \leq h_{new} \leq c_2 h_{old} \tag{2.38}$$

Here Hairer suggests a value of 1.0 and 1.2 for c_1 and c_2 respectively. However, in our case, the step size seems to decrease continuously by a small fraction for the first 100 steps. Therefore we chose to set $c_1 = 0.97$. When the step size does not change and the Jacobian does not need to be recomputed then the LU-decomposition can be reused from the previous time step. This significantly reduces the number of LU-decompositions that need to be done.

2.6 The Classes of implicit Runge-Kutta Methods

We begin this section by giving a brief description of Radau and Lobatto method, and then examine the different Lobatto methods in more detail. As mentioned before the Runge-Kutta methods differ only in their choice of coefficients $a_{i,j}, b_i, c_i$. The choice of these coefficients stems

[†]Pito et al. suggest another method for step size control that could be applied as an extension However, is is beyond the scope of this work.

from the restrictions imposed on the method. As Hairer mentions one begins with the simplified assumptions

$$B(p): \sum_{i=1}^{s} b_i c_i^{q-1} = \frac{1}{q} \qquad q = 1, \ldots, p$$

$$C(\eta): \sum_{j=1}^{s} a_{i,j} c_j^{q-1} = \frac{c_i^q}{q} \qquad i = 1, \ldots, s, \;\; q = 1, \ldots, \eta$$

$$D(\xi): \sum_{i=1}^{s} b_i c_i^{q-1} a_{i,j} = \frac{b_j}{q}(1 - c_j^q) \qquad j = 1, \ldots, s, \;\; q = 1, \ldots, \xi$$

We now present, without a proof, a theorem connecting the simplified assumptions to the order of the Runge-Kutta method. A proof may be found in [8]

Theorem 2.6.1. *If the coefficients $b_i, c_i, a_{i,j}$ of a Runge-Kutta method satisfy $B(p), C(\eta), D(\xi)$ with $p \le \eta + \xi + 1$ and $p \le 2\eta + 2$, then the method is of order p.*

One can now use these conditions and aim for maximum order. Doing this results in the Gauss method of order $2s$. However, for this method the c_i are not located at either of the end points. By imposing further restrictions on the c_j terms we get the Radau and Lobatto methods. These methods have this name since they are based on the Radau and Lobatto quadrature formulas. The c_j's where chosen according to the zeros of

$$\mathbf{I}: \frac{d^{s-1}}{dx^{s-1}}\left(x^s(x - 1^{s-1})\right) \tag{2.39}$$

$$\mathbf{II}: \frac{d^{s-1}}{dx^{s-1}}\left(x^{s-1}(x - 1^{s})\right) \tag{2.40}$$

$$\mathbf{II}: \frac{d^{s-1}}{dx^{s-1}}\left(x^{s-1}(x - 1^{s-1})\right) \tag{2.41}$$

Butcher called the Runge-Kutta methods with these c_i's processes of type **I**,**II** and **III**, also known as Radau Left, Radau Right and Lobatto. The Radau left and right have those names since in Radau left the c_0 value is 0 and in Radau right the rightmost (last) c_i value is 1. Which means that in Radau left we have a node at the left endpoint and in Radau right we have one at the right endpoint. Since these both have an extra restriction it is reasonable for their order to be less the optimum, which is known as the Gauss method and has the best possible order of $2s$. In fact they are both of order $2s - 1$. Lobatto on the other hand has an endpoint at both left and right side giving it one more restriction and an order of $2s - 2$ [14].
As mentioned before for this work we are interested specifically in the Lobatto IIIC method.

2.6.1 A, B & C

The following section is based on the text by Laurent O. Jay.[16]
The Lobatto method is not unique even with the choice of the c_i's given by (2.41). However, since the c_i's are defined the b_i's they are also the same for all variants of Lobatto. That means the only coefficients that differ are the $a_{i,j}$'s.

- **A**:
  For the Lobatto IIIA method the coefficients are defined by $C(s)$, together with satisfying $D(s-2)$ and the condition that $a_{s,j} = b_j$ and $a_{a,j} = 0$ for $j = 1, \ldots, s$.

- **B**:
  For the Lobatto IIIB method the coefficients are defined by $D(s)$ along with satisfying $C(s-2)$ and $a_{i,1} = b_1$ along with $a_{i,s} = 0$ for $i = 1,\dots,s$.

- **C**:
  The Lobatto IIIC method is defined by $a_{i,1} = b_1$ and $C(s-1)$. They satisfy $D(s-1)$ and $a_{s,j} = b_j$ for $j = 1,\dots,s$.

The difference between the A,B and C methods is in their stability. When talking about the stability of Runge Kutta methods one generally looks at weather they are A-Stable, B-Stable and/or L-stable. The different stability's look at different properties of the methods to analyse how well different numerical methods approximate the solutions to stiff ODEs. For example A-stability has to do with the stability function defined in (2.28). See [8] for more in-depth explanation on the different stability's. While all three are A-Stable neither A or B is L-stable or B-stable and hence neither is algebraically stable. The Lobatto IIIC method on the other hand is both L-stable and algebraically stable and hence also B-stable.

2.6.2 Lobatto IIIC

As mentioned before we are interested specifically in the Lobatto IIIC method. Additionally we focus on the three staged method, hence the coefficients we use are as follows [8, 14, 16]:

$$\begin{array}{c|ccc}
0 & 1/6 & -1/3 & 1/6 \\
1/2 & 1/6 & 5/12 & -1/12 \\
1 & 1/6 & 2/3 & 1/6 \\
\hline
 & 1/6 & 2/3 & 1/6
\end{array} \tag{2.42}$$

Chapter 3

Integration into Assimulo

Assimulo is a Python package that entails many ODE solvers written in different languages. It aims to provide a high-level interface for a variety of solvers. Thereby providing an easy way to compare different solvers for the same problem without having to redefine the problem in a different language to adjust for the different solvers [2]. We chose to implement the Lobatto IIIC solver as an extension to the Assimulo package for ease in the later comparison process of results and complexity to other solvers. In addition this solution allows for easier reuse of the code later on for other interested parties. We continue discussing some specifics of the implementation, for the full code consult the appendix.

3.1 The solver class

To define our own solver in Assimulo all we are required to do is to define our own class as a child of `Explicit_ODE`. This class should overload the `integrate` method which is called when the solver is asked to solve a problem. We have split our implementation into two classes, the `RKS3CBase` and `Lobatto4ODE` with the latter being a child of the former. This was done so that we could easily implement different Runge-Kutta methods later.

3.1.1 Integrate method

The `integrate` method is defined as `def integrate(self,t,y,tf,opts)` with `t` being the initial time x_0, `y` being the initial value y_0, `tf` being the final time and `opts` being the options. The method is then expected to return the `ID_PY_OK` object along with a list of time stamps and a corresponding list or approximations for each time stamp, `tres` and `yres` respectively in our case. Note that we did not use the options in our implementation. These options could be used to specify that we only wish to receive outputs at certain time points but wish to have a smaller step size, i.e. only store some points. Doing this would help reducing the memory requirements of the entire Runge-Kutta method. The `integrate` method itself is just a loop, taking a step and then storing the results of the new x_n and y_n in the `tres` and `yres` lists respectively. The `step` method is just a helper method which calls the `_newton` method and passes the results back to the `integrate`. The reason that we have the `step` method at all is that the `_newton` sometimes calls itself recursively when the step fails.

3.1.2 Newton Iteration method

The `_netwon` method is the main part of the code. This implements the entire Newton iteration described in section 2.2 together with all the simplifications from section 2.3. To perform the newton iteration described in equation (2.14) we require the left hand side (LHS) and right hand side (RHS) of equation (2.14a) so that we can solve for ΔW^k. The computation of these sides is done in the methods `_compLHS` and `_compRHS` respectively.

The `_compLHS` method constructs the blocks for the block matrix in (2.17). This requires only the Jacobian $\mathscr{J}$ of the problem, which can be passed as a sparse or dense matrix. This method is only called once per newton iteration and only if the step size h, called `h` in the code, has been adjusted. In addition once `_compLHS` is called the result gets passed into the LU-decomposition method. The result of the LU-decomposition is then stored as an instanced variable to be reused in later newton iterations.

The `_compRHS` computes $-h^{-1}(\Lambda \otimes I)W^k + (T^{-1} \otimes I)F\big(Z^k\big)$, see equation (2.14a), which only depends on the current Z^k, W^k and the function f of the ordinary differential equation. This method is called once per newton step, and has three evaluations of the function f. It then returns a single numpy vector of size $3n$

The `_netwon` method also evaluates the error after the Newton iteration terminates with `err_mag = self._normed_aprox_err(zk,y0,y1,n)`. Further it adjusts the step size or restarts the step if the error was to large. The method `_normed_aprox_err` is defined within the Lobatto class and is error approximation method.

Chapter 4

Numerical Results

4.1 Rotating Pulse Problem

To test our implementation we used the rotating pulse problem discussed in Section 5.3 in [5].

$$\frac{\delta C}{\delta t} - \nabla \cdot \left\{ \begin{pmatrix} -4y \\ 4x \end{pmatrix} C - D\nabla C \right\} = 0 \tag{4.1}$$

with the Dirichlet boundary and initial conditions from the analytical solution

$$C(x,y,t) = \frac{2\sigma^2}{2\sigma^2 + 4Dt} \exp\left(-\frac{(\bar{x}-x_c)^2 + (\bar{y}-y_c)^2}{2\sigma^2 + 4Dt} \right) \tag{4.2}$$

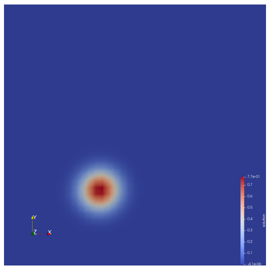
with $\bar{x} = x\cos(4t) + y\sin(4t)$, $\bar{y} = -xsin(4t) + ycos(4t)$. We shift the domain to $\Omega = (0,1)^2$ instead of $(-0.5, 0.5)^2$ and use the parameters $D = 10^{-3}, x_c = 0.25, y_c = 0$ and $2\sigma^2 = 0.004$. The time interval for the simulation is $[0, 0.25]$. This problem describes a rotating heat pulse which cools down over time as can be seen in Figure 4.1 where we show 8 instances in time extending the simulation time to see the effect more clearly. The benefit of using this specific problem is that an analytical solution exists. On that account we are able to directly measure the error between approximation and analytical solution.

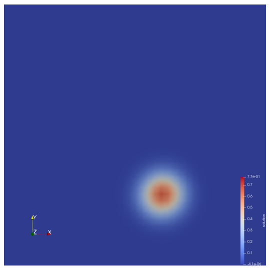
We used an implementation of the problem in the DUNE framework [6, 3, 4, 9, 10, 1, 12]. Then we pass the RHS, f, of our ODE, along with the jacobian $\mathscr{J}$ from DUNE to Assimulo. To define a ODE problem in Assimulo is as simple as calling `Explicit_Problem(rhs,y0)`. Which returns a model object. Here `rhs` is the function, f, in our code and should have inputs `rhs(t,y)`. While the input `y0` is the initial condition, $y_0 = f(x_0)$, of our ODE. Unless an addition parameter for x_0 is passed in it is assumed to be zero. To then find the solution at $x_N \neq x_0$ one first defines a solver, in our case `Lobatto4ODE(model)` which returns a simulator. Here `model` is the model object returned from the Explicit Problem. To then simulate this simulator until x_N we simply call `.simulate(end)` where `end` is x_N. This returns a list of time points and a corresponding list of solutions.

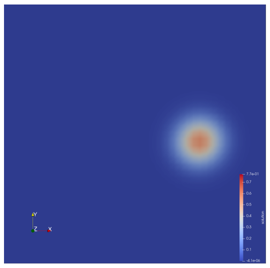
We run the code* with two different spatial resolutions, the first discretization discretizes the space into a grid of 20×20 points and the second having $2 \cdot 20 \times 2 \cdot 20$ points, referred to as *resolution 1* and *resolution 2* respectively from here on. As a point of comparison we will compare the error from our method to the error in the RADAU5 method implemented by Hairer in [14],

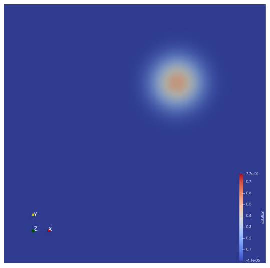
*See appendix for the code used to setup and solve the problem.

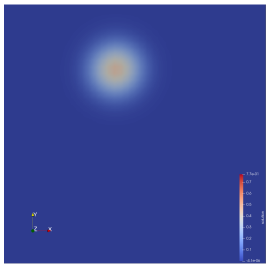
which has already been implemented into the Assimulo package[2]. Since our Lobatto method is of order 4 while the already existing Radau method is of order 5 we expect the Radau method to outperform our method. For a more fair comparison we will also compare the to RungeKutta34 method in Assimulo. This method is of order 4 and as such should behave similar to our method.

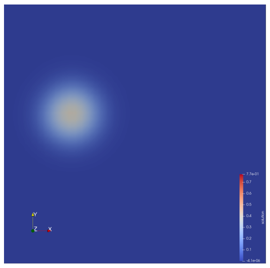

(a) t=0.25

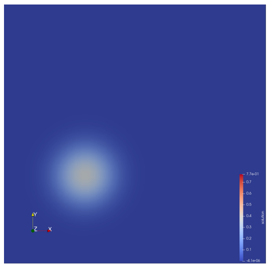

(b) t=0.5


(c) t=0.75


(d) t=1.00


(e) t=1.25


(f) t=1.5


(g) t=1.75

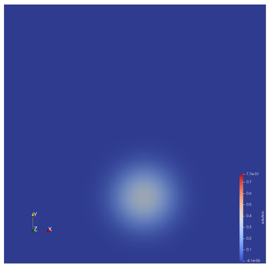

(h) t=2.00

Figure 4.1: A sequence of results from the numerical experiment at different points in time showing the movement of the pulse.

4.2 Results

	Lobatto		Radau5		RungeKutta34	
Resolution	1	2	1	2	1	2
Computation time (sec)	7.98	159	11.36	403	0.69	17.61
Number of function calls	70	76	62	58	380	980
Number of Jacobi evaluations	1	1	1	1	-	-
Number of LU-decompositions	6	9	4	4	-	-
Number of steps	18	18	14	13	76	196
L^2 Error	0.00225	0.000244	0.00226	0.000254	0.00226	0.000251
Average number of function calls per steps	3.888	4.222	4.429	4.462	5.000	5.000

Table 4.1: A comparison between the Lobatto IIIC, Radau IIA and RungeKutta34 method when simulating the rotating pulse problem as described above with a relative and absolute tolerance of 10^{-5}

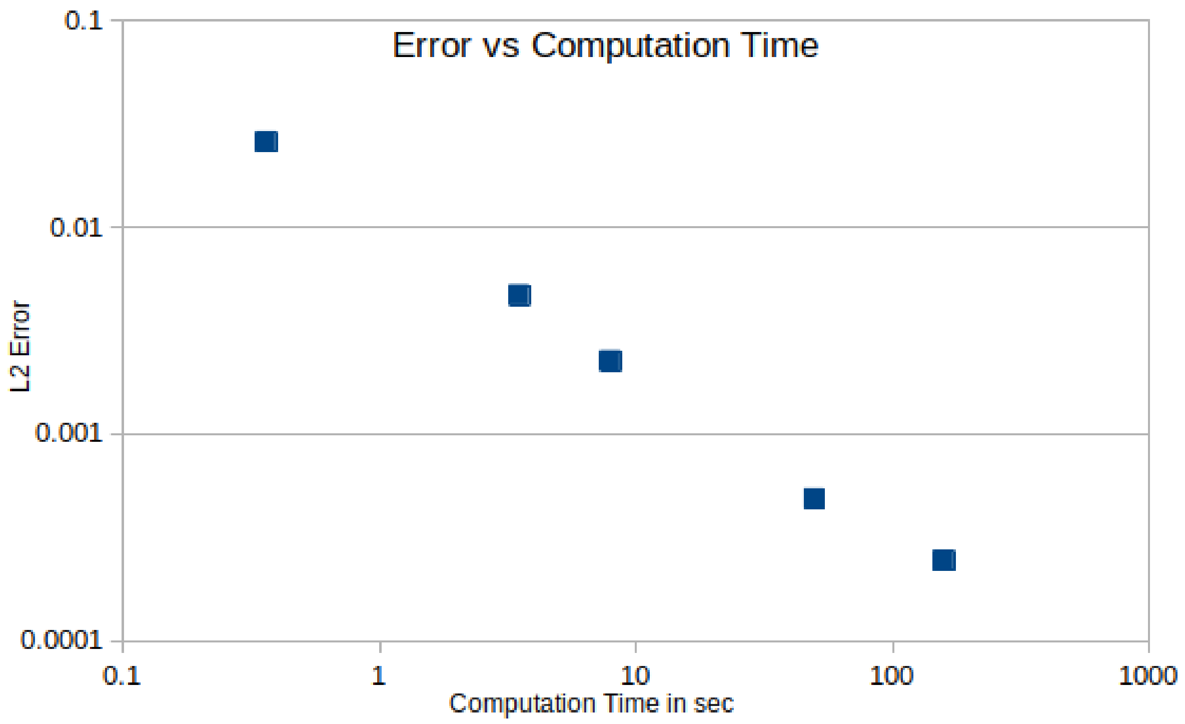


Figure 4.2: Showing the relationship between computation time in seconds and the achieved accuracy of our Lobatto IIIC method. Observe that both axes are in logarithmic scale

Resolution	1	2
Computation time (sec)	3.68	43.7
Number of function calls	60	60
Number of Jacobi evaluations	1	1
Number of LU-decompositions	2	2
Number of steps	16	16
L^2 Error	0.00226	0.000250
Average number of function calls per steps	3.75	3.75

Table 4.2: Comparison between resolutions when running our implementation of the Lobatto IIIC method with a constant step size for the rotating pulse problem as described above

4.3 Comparison of Methods

Table 4.1 shows the difference in the stats gathered while simulating the rotating pulse problem between our Lobatto implementation, Hairers Radau5 and the RungeKutta34 method in Assimulo. We continue by discussing their differences in the following sections.

4.3.1 L^2 Error

The L^2 error of all the methods does not go down to zero, this is because we are taking the norm between our result and a discritization of the exact solution, this discritization introduces a small error. This is also the reason why the error for all methods is nearly the same.

4.3.2 Computational Time

The computational time is the time that our computer clock measures between starting the simulation and finishing. The difference in computational time between the different methods is mainly due to the differences in the implementation of the methods. Making this a bad metric to judge the different methods by. Our Lobatto code for example is written completely in Python while the Radau5 code is written in FORTRAN and we only call it in from Python. Since Python is an interpreted language instead of a compiled one it is slightly slower and therefore will take more time per step. In addition the Radau5 code is unable to use the sparse matrix representation used for the Jacobian and hence for each Jacobian evaluation the matrix must be converted to dense representation first. This might be why for resolution 2 the Radau5 takes longer to compute compared to our Lobatto method. Similarly taking an LU decomposition of a large dense matrices takes longer in comparison to sparse matrices if the matrix is large enough and is filled with significantly many zeros.

We have further investigated the computation time of our Lobatto IIIC method compared to the achieved error, see figure 4.2. To generate this graph we changed the resolution of the discretization. The finer the grid the lower the error however at the cost of computation time.

4.3.3 Number of Function Calls

The number of function calls is directly related to how many steps were taken and how many of those steps failed. As can be seen in all three methods the more steps one needs the more function calls are required. However the Radau5 method has at least one extra function call per step through its error evaluation while the Lobatto method does not. Hence, if we look at the average number of function calls per step we see that our Lobatto method has fewer. Since we only solve for $t = 0.25$ very few steps need to be taken however as the end time increases so dose the number of steps, and as this problem is linear the average number of steps in Lobatto will go to 3 while in Radau it will go to 4.

4.3.4 Number of Jacobian Evaluations

Due to the structure of this problem a single Jacobian evaluation is enough and both our Lobatto and the Radau5 method utilize this.[†]

[†]The RungeKutta34 method probably also uses this considering how fast it is.

4.3.5 Number of LU-decompositions

Our Lobatto method has a few more LU-decompositions than the Radau5 method. The decomposition is recalculated whenever the step size was adjusted. This means that the Radau5 method was slightly faster at determining the required step size[‡].

4.3.6 Number of Steps

The number of steps is directly linked to the step size. Since the Lobatto method is not as good as the Radau method in terms of order, more steps will need to be taken to achieve the same accuracy in the results. Similarly the RungeKutta34 method is, like our Lobatto, order 4. Hence we would expect to see a similar amount of steps. The spatial discritization we used is optimized for the Lobatto points, which is probably why the Lobatto Method requires slightly fewer steps than RungeKutta34.

4.3.7 Overall

Over all our Lobatto method has performed as expected. Being slightly better than the RungeKutta34 but still not as good as the Radau5 method in all metrics other than computation time. However, computation time, as explained is not a good metric since it would require all methods to be implemented in the same programming language with the same amount of optimization. Note further for both our implementation of the Lobatto method and the Radau5 method from Hairer [14] neither of the two significantly increase as the resolution increases, however, the RungeKutta34 method more than doubles the amount of steps it takes.

4.4 Constant step

The interest of this project was sparked due to the result in [19] stating that the Lobatto IIIC method is equivalent to the space time DG approximation. When solving for a space time discritization, the step size is the dicritization in time and is kept constant. We therefore run the before mentioned problem again with a fixed step size of $h = \frac{1}{60}$, and get the results shown in table 4.2. This table shows, that when solving linear problems, manually determining a constant step size can be beneficial. The fact that a lower number of steps still gives an accurate result is probably due to an suboptimal choice of our free parameter a in equation (2.34)

[‡]An idea exists to only recalculate the LU-decomposition when Newton diverges, this was not done here however

Chapter 5

Conclusion

In this work we have outlined the implementation of the Lobatto IIIC Runge-Kutta method as described by Hairer in [14]. We have then done a more in-depth discussion about the different existing error estimates for the local truncated error, and chosen to adapt a method from Radau to Lobatto that Pinto et al. describes in [17].

Comparing our method to the Radau5 method from Hairer the only improvement we see is a lower average amount of function calls per step. However as discussed the Radau5 method has a higher order and hence is expected to outperform our Lobatto IIIC implementation. When comparing our method to the RungeKutta34 method instead, we see that we have significantly fewer steps and hence significantly fewer function calls. Be that as it may, we have only conducted one numerical experiment comparing the methods and further experiments must be conducted with varying ODEs for a thorough comparison.

Further improvements could still be made to our method, such as thorough investigating into a good choice for the free parameter a in equation (2.34) and the multitude of extensions mentioned throughout this work which were excluded due to the limited scope of this study.

Chapter 6

Bibliography

[1] M. Alkämper et al. "The DUNE-ALUGrid Module." In: *Archive of Numerical Software* 4.1 (2016), pp. 1–28. ISSN: 2197-8263. DOI: 10.11588/ans.2016.1.23252. URL: http://dx.doi.org/10.11588/ans.2016.1.23252.

[2] Christian Andersson, Claus Führer, and Johan Åkesson. "Assimulo: A unified framework for {ODE} solvers". In: *Mathematics and Computers in Simulation* 116.0 (2015), pp. 26–43. ISSN: 0378-4754. DOI: http://dx.doi.org/10.1016/j.matcom.2015.04.007.

[3] P. Bastian et al. "A Generic Grid Interface for Parallel and Adaptive Scientific Computing. Part I: Abstract Framework". In: *Computing* 82.2–3 (2008), pp. 103–119. DOI: 10.1007/s00607-008-0003-x. URL: http://dx.doi.org/10.1007/s00607-008-0003-x.

[4] P. Bastian et al. "A Generic Grid Interface for Parallel and Adaptive Scientific Computing. Part II: Implementation and Tests in DUNE". In: *Computing* 82.2–3 (2008), pp. 121–138. DOI: 10.1007/s00607-008-0004-9. URL: http://dx.doi.org/10.1007/s00607-008-0004-9.

[5] Peter Bastian. "Higher Order Discontinuous Galerkin Methods for Flow and Transport in Porous Media". In: *Challenges in Scientific Computing- CISC 2002*. Ed. by Eberhard Bänsh. Springer, 2002, pp. 1–22. ISBN: 978-3-642-19014-8.

[6] Peter Bastian et al. "The Dune framework: Basic concepts and recent developments". In: *Computers Mathematics with Applications* 81 (2021). Development and Application of Open-source Software for Problems with Numerical PDEs, pp. 75–112. ISSN: 0898-1221. DOI: https://doi.org/10.1016/j.camwa.2020.06.007. URL: https://www.sciencedirect.com/science/article/pii/S089812212030256X.

[7] Phillip Birken. *Newton's Method 2*. May 2020.

[8] J. C. Butcher. *The numerical analysis of ordinary differential equations*. John Wile Sons, 1987. ISBN: 0 471 91046 5.

[9] A. Dedner et al. "A Generic Interface for Parallel and Adaptive Scientific Computing: Abstraction Principles and the DUNE-FEM Module". In: *Computing* 90.3–4 (2010), pp. 165–196. DOI: 10.1007/s00607-010-0110-3. URL: http://dx.doi.org/10.1007/s00607-010-0110-3.

[10] A. Dedner et al. "The DUNE-FEM-DG module". In: *Archive of Numerical Software* 5.1 (2017), pp. 21–61. ISSN: 2197-8263. DOI: 10.11588/ans.2017.1.28602. URL: http://journals.ub.uni-heidelberg.de/index.php/ans/article/view/28602.

[11] Andreas Dedner, Robert Kloefkorn, and Martin Nolte. *Python Bindings for the DUNE-FEM module*. Version v2.7.0. Mar. 2020. DOI: 10.5281/zenodo.3706994. URL: https://doi.org/10.5281/zenodo.3706994.

[12] Andreas Dedner et al. *Python Framework for HP Adaptive Discontinuous Galerkin Method for Two Phase Flow in Porous Media*. 2018. arXiv: 1805.00290 [cs.CE].

[13] Brian C. Fabien and P. Frye Jason. "The solution of the inplicit differential equation via a lobatto IIIC Runge-Kutta method". In: *MATHMOD09*. 2009, pp. 2708–2711. ISBN: 978-3-901608-35-3. URL: https://www.argesim.org/fileadmin/user_upload_argesim/ARGESIM_Publications_OA/MATHMOD_Publications_OA/MATHMOD_2009_AR34_35/full_papers/303.pdf.

[14] Ernst Hairer and Gerhard Wanner. *Solving Ordinary Differential Equations II*. 2nd ed. Vol. 2. Springer, 2010, pp. 135–144.

[15] Arieh Iserles. *A First Course in the Numerical Analysis of Differential Equations*. 2nd ed. Cambridge University Press, 2009. ISBN: 978-0-511-50637-6.

[16] O. Jay Laurent. *Lobatto methods*. URL: https://homepage.math.uiowa.edu/~ljay/publications.dir/Lobatto.pdf.

[17] Severiano González Pinto, Domingo Hernández Abreu, and Juan Ignacio Montijano. "Variable Step-Size Control Based on Two-Steps for Radau IIA Methods". In: *ACM Transactions on Mathematical Software* 46.4 (2020). DOI: 10.1145/3408892.

[18] Endre Süli and David Francis Mayers. *An introduction to numerical analysis*. Cambridge University Press, 2014. ISBN: 957-0-521-810267-5.

[19] Lea Miko Versbach. *Multigrid Preconditioners for the Discontinuous Galerkin Spectral Element Method*. 2020. ISBN: 978-91-7895-593-0.

[20] Wikipedia contributors. *List of Runge–Kutta methods — Wikipedia, The Free Encyclopedia*. [Online; accessed 21-May-2021]. 2021. URL: https://en.wikipedia.org/w/index.php?title=List_of_Runge%E2%80%93Kutta_methods&oldid=1019271082.

Chapter 7

Appendix

7.1 Lobatto IIIC Implementation

```python
import sys #for min float number
import numpy as np
import scipy.sparse as sp
import scipy.sparse.linalg as alg
from scipy.sparse.linalg import splu
from abc import ABC, abstractclassmethod
from scipy.linalg import lu_factor,lu_solve

from assimulo.explicit_ode import Explicit_ODE
from assimulo.ode import ID_PY_OK,NORMAL



class RadauError(Exception):
    def __init__(self,expr):
        self.expr = expr
    def __str__(self):
        return self.expr

class RKS3CBase(ABC,Explicit_ODE): #Runge-Kutta S=3 Complex eigen pair Base class



    rtol=1.e-4 #realative tolerance
    atol=1.e-4 #absolute tolerance
    tol =1.e-4 #tollerance for newton itteration
    kappa = 1e-1 #The kappa value used for newton stopping in eq (15)
    maxit=10 #maximum number of newton itterations
    maxsteps=10000000 #maximum number of steps
    r=1 #r is the ratio between the previouse step size and the current

    #used for stoping after single newton itt
    Uround = sys.float_info.min

    #the bounds to decide weather or not to adjust the step size eq (41)
    c1 = 0.95
    c2 = 1.2


    def __init__(self,problem):
        """
        Initiates the solver.

            Parameters::

                problem
                            - The problem to be solved. Should be an instance
                              of the 'Explicit_Problem' class.
        """
        Explicit_ODE.__init__(self, problem) #Calls the base class

        #Solver options
        self.options['h'] = problem.h
        self.h = problem.h
        self._sparceJ = True
        self._dynamic_step = True

        #Statistics
        self.statistics["nsteps"] = 0
        self.statistics["nfcns"] = 0
        self.num_jac_evals = 0
        self.num_lu_decomps = 0

        #method parameter:
        self.y_dim = len(self.problem.y0)
        self.RTol = np.array([self.rtol]*self.y_dim)
        self.ATol = np.array([self.atol]*self.y_dim)
        self._3ATol = np.hstack((self.ATol,self.ATol,self.ATol))
        self._3RTol = np.hstack((self.RTol,self.RTol,self.RTol))

        #internal flags
        self._first = True
        self._recomp_jac = True
        self._recomp_LHS = True
        self._newt_fail_flag = False

        #internal values
        self.J = None
        self._lu = None


    #set and get for sparce Jacobian, need to redifine the lu decompose and
    #lu solve methods here
    def _set_sparce(self,sp):
        self.options["sparce"] = bool(sp)
        if sp:
            def lu_decomp_s(LHS):
                lu1 = splu(LHS[0])
                lu2 = splu(LHS[1])
                return (lu1,lu2)
            self._LUDecomp = lu_decomp_s

            def lu_solve_s(lu,RHS):
                n = RHS.shape[0]//3
                dwk = np.zeros_like(RHS)
                dwk[:n] = lu[0].solve(RHS[:n])
                dwk[n:] = lu[1].solve(RHS[n:])
                return dwk
            self._LUSolve = lu_solve_s
        else:
            def lu_decomp_d(LHS):
                lu1 = lu_factor(LHS[0])
                lu2 = lu_factor(LHS[1])
                return (lu1,lu2)
            self._LUDecomp = lu_decomp_d

            def lu_solve_d(lu,RHS):
                n = RHS.shape[0]//3
                dwk = np.zeros_like(RHS)
                dwk[:n] = lu_solve(lu[0],RHS[:n])
                dwk[n:] = lu_solve(lu[1],RHS[n:])
                return dwk
            self._LUSolve = lu_solve_d
    def _get_sparce(self):
        return self.options["sparce"]
    _sparceJ = property(_get_sparce,_set_sparce)

    #set and get for h parameter, when h changes we also want to recompute the
    #LHS
    def _set_h(self,h):
        self.options["h"] = float(h)
    def _get_h(self):
        return self.options["h"]
    h=property(_get_h,_set_h)


    def integrate(self,t,y,tf,opts):
        """
        _integrates (t,y) values until t > tf
        """
        self.h = min(self.h,abs(tf-t))

        tres = []
        yres = []

        z0 = np.zeros((self.y_dim*3))

        #lists for storeing results
        for i in range(self.maxsteps):
            if t>= tf:
                break

            t,y = self._step(t,y,z0)

            self._first = False

            self.t = t #used for Radau error approximation

            #store the results in array to be returned when done
```

```python
            tres.append(t)
            yres.append(y.copy())


            if self.h > np.abs(tf-t):
                h_old = self.h / self.r
                self.h = np.abs(tf-t)
                self.r = self.h/h_old
                self._recomp_LHS=True

            if abs(self.r-0.5)<0.1:
                #if we are to close to the explosion point, change value
                h_old = self.h/self.r
                self.r = 0.35
                self.h = self.r * h_old
                self._recomp_LHS=True

            if abs(self.h) < self.Uround:
                break

        else:
            raise RadauError(f"final time not reached within maximum number of steps ({self.maxsteps})")
        return ID_PY_OK, tres, yres


    def _step(self,t,y,z0):
        #update statistics
        self.statistics["nsteps"] += 1
        #recomp jacobian if the flag is set
        if self._recomp_jac:
            self.J = self._jacobian(t,y)
            self._recomp_jac = False
            self._recomp_LHS = True

        #newton itteration
        y,h = self._newton(self.J,y.copy(),t,z0)
        return t + h, y

    def _newton(self,J,y0,t,z0=None):

        #handy dimension variable
        n  = self.y_dim

        #setup newton starting value
        zk = z0 if type(z0) is np.ndarray else np.zeros(3*n)
        wk = self._zTOw(zk)

        #newton stopping variables
        ndzk = None #Norm of delta Zk
        theta_k = None #Theta_k

        #setup LHS and lu decompose if needed
        if self._recomp_LHS or self._lu == None:
            LHS = self._compLHS(J)
            self._lu = self._LUDecomp(LHS)
            self.num_lu_decomps += 1
            self._recomp_LHS = False

        for i in range(1,self.maxit):
            #compute the RHS
            RHS = self._compRHS(wk,zk,y0)

            #compute the delta W_k
            dwk = self._LUSolve(self._lu,RHS) #eq (17a)

            #convert delta W_ktodelta Z_k
            dzk = self._wTOz(dwk)

            #increment wk and zk with the corresponding delta values
            wk += dwk #eq (17b)
            zk += dzk

            #---------newton stopping condition-----------

            #for sci we require the max of z_k and z_{k-1} termwise
            diff = dzk.copy()
            diff[diff>0] = 0 #wherever the value has increased we want to keep
            sci = self._3ATol + (zk-diff)*self._3RTol

            #norm of delta zk
            ndzk = self._norm(dzk,sci)

            #declare the eta_k variable so it is outside the if scope
            eta_k = None
            if i == 1 :#first step in newton
                if '_eta' in self.__dict__:
                #one step stopping
                    eta_k = max(self._eta,self.Uround)**.8
                else:
                #first time step, needs at least 2 newton
                    ndzkm1 = ndzk
                    continue
            else:
                #compute theta_k and eta_k
                theta_k = ndzk/ndzkm1 # eq (14)
                eta_k   = theta_k/(1-theta_k) #eq (15)

                #check if we are diverging
                if theta_k > 1:

                    if not self._dynamic_step:
                        raise RadauError(f'newton itteratio diverged')

                    #adjust step size
                    self.h /= 2
                    self.r /= 2

                    #if r is to close to 0.5 adjust it down further,
                    #this is due to explosion of values in error estimation
                    #TODO: move this into the set h function.
                    if (self.r-0.5)<0.1:
                        self.h *= 0.35 /self.r
                        self.r = 0.35

                    #recompute the jacobian
                    self.J = self._jacobian(t,y0)
                    self._recomp_LHS = True

                    #re-run newton
                    return self._newton(self.J,y0,t,z0)

            #test if newton has converged, eq (14)
            if eta_k * ndzk <= self.kappa * self.tol:
                #save old eta for singe step stopping
                self._eta = eta_k

                #new y value
                y1 = self._zTOy(y0,zk)
                #the h value that was used
                h = self.h

                #------determine approximate error---------
                if self._dynamic_step:
                    #approximate error magnitude
                    err_mag = self._normed_aprox_err(zk,y0,y1,n)

                    #if error magnitude > 1 last h will not be changed since
                    #we will rerun this step.
                    h_old = self.h/ ( self.r if err_mag > 1 else 1)

                    #compute new step size
                    h_new = self._adjust_h(err_mag,step_accept = err_mag < 1, itt = i)
                    #test if we should adjust step eq (41):
                    if not (self.c1*self.h <= h_new and h_new <= self.c2 * self.h):
                        #set flag to recompute the LHS in the next itteration
                        self._recomp_LHS = True
                        #calculate the new ratio
                        r = h_new/h_old

                        #special condition for the Lobatto to avoid coefficient
                        #explosion when computeing the error magnitude
                        if abs(r-0.5)<0.1:
                            #if we are to close to the explosion point, change value
                            r = 0.35
                            h_new = r * h_old
                        #save adjusted values
                        self.r = r
                        self.h = h_new
                    else:
                        #if we dont adjust step size
                        h_new = self.h
                        self.r = 1.0


                    if  err_mag > 1:
                        #approximate error is to large, restart with smaller step size
                        return self._newton(J,y0,t,z0)

                    #store z^(k-1) for error estimation in the next step.
                    self.zkm1 = zk.copy()

                #return result and the h used.
                return y1, h

            #update the norm for newton stopping condition
            ndzkm1 = ndzk

        else:
            #if we do not converge within number of newton itteration steps
            if not self._newt_fail_flag:
                self._newt_fail_flag = True
                #rerun with new Jacobian if not done yet
                self.J = self._jacobian(t,y0)
                self._recomp_LHS = True
                #re-run newton
                return self._newton(self.J,y0,t,z0)
            #otherwise raise
            raise RadauError(f'newton itteratio did not converge after {self.maxit} itterations')

    @abstractclassmethod
    def _normed_aprox_err(self,zk,y0,y1,n):
        #abstaract method that each version of this code must impliment seperately
        return np.Inf

    @abstractclassmethod
    def _adjust_h(self,err_mag,TOL = 1e-5,step_accept = True,itt = 0):
        return np.Inf

    def _norm(self,vec,sc):
        #calculates the norm of a vector according to eq (38)
        return np.sqrt(
                    (1/vec.shape[0])*
                    np.sum(
                        (vec/sc)**2
                    )
                )



    def _compRHS(self,wk,zk,y):
        #computes the RHS of eq (17a)

        #dimenson of system
        n = self.y_dim
        #initial term:
        term1 = -np.array(n*[self.gamma/self.h] + \
                        2*n*[(self.alpha)/self.h]) \
                    * wk
        term1[n:2*n] += self.betta/self.h * wk[2*n: ]
        term1[2*n: ] -= self.betta/self.h * wk[n:2*n]

        #function term
        fcn = self.problem.rhs
        f = np.array([fcn(self.t+self.c[0]*self.h,y + zk[:n]),
            fcn(self.t+self.c[1]*self.h,y + zk[n:2*n]),
            fcn(self.t+self.c[2]*self.h,y + zk[2*n:])])
        prod = (self.TI @ f).flatten()

        self.statistics["nfcns"] += 3

        return term1 + prod

    def _compLHS(self,J):
        #construction of the LHS of eq (17a)
        #dimension of system
        n = self.y_dim

        #construct of blocks according to eq (20)
        e1 = np.diag(n*[self.gamma/self.h])-J
        e2 = np.zeros((2*n,2*n))
        np.fill_diagonal(e2,self.alpha/self.h)
        e2[:n,:n] -= J
        e2[n:,n:]  = e2[:n,:n]
        np.fill_diagonal(e2[ :n,n: ],-self.betta/self.h)
        np.fill_diagonal(e2[n: , :n], self.betta/self.h)

        #if sparce then convert to sparce format
        if self._sparceJ:
            e1 = sp.csc_matrix(e1)
            e2 = sp.csc_matrix(e2)
        return (e1,e2)

    def _jacobian(self,t,y):
        """
        calculates the jacobian
        """
        if self.problem_info['jac_fcn']:
            self.num_jac_evals += 1
            J = self.problem.jac(t,y)
            self._recomp_jac = False
            return J
        else:
            raise RadauError('The current implimentation cannot calculate an approximate jacobian yet,' +
                    'hence a jacobian function must be given in the problem')

    def _zTOw(self,z):
        #TODO: this might one returns return directly...
        #convert from z to w
        w = (self.TI@ z.reshape((3,-1))).flatten()
        return w

    def _wTOz(self,w):
        #convert from w to z
        z = (self.T @ w.reshape((3,-1))).flatten()
        return z

    def _zTOy(self,y,z):
        #convert from z to y
        y = y + self.d @ z.reshape((3,-1))
        return y

    def print_statistics(self, verbose=NORMAL):
        """
        Prints the run-time statistics for the problem.
        """
        Explicit_ODE.print_statistics(self, verbose) #Calls the base class
        self.log_message(' Number of J evaluations : ' + str(self.num_jac_evals),    verbose)
        self.log_message(' Number of LU decompos   : ' + str(self.num_lu_decomps),   verbose)
        self.log_message('\n\n\n',verbose)


class Lobatto4ODE(RKS3CBase):

    #-------- Method constants -----------
    # T matrix
    T = np.array([
        [0.4072639531732107,-0.44308062047843144,0.31680119776208315],
        [0.18547209365357897,0.1305271017756723,-0.38187534481524843],
        [0.8942796961362183,0.735153359223255,0.0]
        ])
    # T inverse matrix
    TI= np.array([
        [1.0326372242409414,0.8566688421623923,0.47027336073401904],
        [-1.2561549117980622,-1.0420976007887581,0.7881948366175593],
        [0.07217833861373389,-2.558776912784013,0.49781525580043673],
        ])

    #eigen values gamma, alpha and betta
    gamma = 2.6258168189584676
    alpha = 1.6870915905207662
    betta = -2.508731754924879

    # c,d and b (see paper for reference)
    c = np.array([0.,1/2.,1.])
    d = np.array([0,0,1])
    b = np.array([1/6,2/3,1/6])

    #e used for one step error approximation
    #e = np.array([-.5,2,-.5])-b
    e = np.array([-4.0,0,-1]) #this is e A^-1

    #free parameter a used in error estimation
    a = 5

    def __init__(self,problem):
        RKS3CBase.__init__(self,problem)

        #list for storeing error compairison if flag is set
        self.err_comp = []

    #overwrite z to y conversion to be more efficient
    def _zToy(self,y,z):
        return y + z[2*self.y_dim:]


    #Method to calculate the coefficients for the error estimation
    def coef(self,r):
        if abs(r-0.5)<0.09:
            raise RadauError(f'Some internal error; the relative distance in step size is to close to 0.5

        #TODO: Optimize this method
        #-devisor is always the same
        #-delta3 = -delta1 -delta2
        #-precalculate r^3 and r^2 (this might make it more efficient?)
        r2 = r**2
        r3 = r*r2
        r4 = r*r3
        div = 1/(4*r3 + 4*r2 + 3*r - 3)
        beta1 = (12*r3 + 14*r2 + 21*r + 9)*div
        beta2 = (16*r3 + 8*r2 - 12*r - 12)*div
        beta3 = 0
        delta1=(12*r3 + 9*r2 + 3*r)*div
        delta2=(8*r4 - 24*r3 - 36*r2 - 12*r)*div
        #delta3=(-8*r4 + 12*r3 + 27*r2 + 9*r)*div
        return (beta1,beta2,delta1,delta2,-delta1-delta2)


    def _normed_aprox_err(self,zk,y0,y1,n):
        #calculate scaleing of error in the norm from hairer
        sc = self.ATol + np.maximum(np.abs(y0),np.abs(y1))*self.RTol

        #if we are in the first step
        if self._first or self._do_comp_err:
            #do one step error estimation
            zpart = self.e[0] * zk[:n] + self.e[1] * zk[n:2*n] + self.e[2] * zk[2*n:]
            ydiff = self.h*self.problem.rhs(self.t,y0)/self.gamma + zpart
            self.statistics["nfcns"] += 1

            #normalize error vector
            mag = self._norm(ydiff,sc)

            #if we want to compair the error to the two step error
            if (not self._first):
                beta1,beta2,delta1,delta2,delta3 = self.coef(self.r)
                err_vec_tmp = self.a * (zk[2*n:] - (delta1*self.zkm1[:n] \
                    + delta2*self.zkm1[n:2*n] + delta3*self.zkm1[2*n:] \
                    + beta1 * zk[:n] + beta2 * zk[n:2*n]))
                err = self._norm(err_vec_tmp,sc)
                err_old = mag
                #return err

                self.err_comp.append((err ,mag,self.t,self.h))
                if mag > 1:
                    self.statistics["nfcns"] += 1
                    mag = self._norm(self.h * self.problem.rhs(self.t,y0)/self.gamma + zpart,sc)


            else:
                err = mag
                #if the step fails do more accurate error estimation
                if err > 1:
                    self.statistics["nfcns"] += 1
                    err = self._norm(self.h * self.problem.rhs(self.t,y0)/self.gamma + zpart,sc)
                return err

        else:
            #do two step error estimation
            #calculate coefficient
            beta1,beta2,delta1,delta2,delta3 = self.coef(self.r)
            #determine error vector
            err_vec = self.a * (zk[2*n:] - (delta1*self.zkm1[:n] \
                    + delta2*self.zkm1[n:2*n] + delta3*self.zkm1[2*n:] \
                    + beta1 * zk[:n] + beta2 * zk[n:2*n]))
            #normalize error
            err = self._norm(err_vec,sc)
            return err

    def _adjust_h(self,err_mag,TOL = 1e-5,step_accept = True,itt = 0):
        #function to adjust the step size
        fac = 0.9 * (2*self.maxit+1)/(2*self.maxit+itt)
        return fac * self.h * err_mag ** (-1/4)


if __name__ == '__main__':
    #-----example usage-----
    from assimulo.ode import Explicit_Problem
    import matplotlib.pyplot as plt

    #define the problem
    g : float = 9.81
    l : float = 0.5
    theta : float = np.pi/2.
    def rhs(t,y):
        return np.array([
            y[1],
            np.sin(y[0])*g/l
            ])

    def J(t,y):
        return np.array([
            [0, 1],
            [np.cos(y[0])*g/l,0]
            ])

    #--helper functions for calculating period
    def arith_geo_mean(a,b):
        while True:
            a,b = (a+b)/2, np.sqrt(a*b)
            yield a,b

    from itertools import islice
    def elliptic_integral(k,tol=1e-5,maxiter=100):
        a_0,b_0 = 1.,np.sqrt(1-k**2)
        for a,b in islice(arith_geo_mean(a_0,b_0),maxiter):
            if abs(a-b) < tol:
                return np.pi /(2*a)
        else:
            raise Exception('Algorithm did not converge')

    periode = 4*np.sqrt(l/g) * elliptic_integral(np.sin(theta/2),tol=1e-10)

    #----Initialize Problem
    pend_model = Explicit_Problem(rhs,y0=np.array([theta,0.]))
    pend_model.name = 'Pendulum simulation'
    pend_model.jac = J
    pend_model.h = 0.01

    #----Initialize Solver
#    sim = Radau5ODE(pend_model)
    sim = Lobatto4ODE(pend_model)
    sim.h = 0.016
    sim._do_comp_err = True
    sim.maxit=10
    #----Simulate the solver
    t,y = sim.simulate(periode)
    #----Show results
    sim.plot()
    plt.show()
```

7.2 Experiment implementation

The following code is written by Robert Klöfkorn with the help of [11]

```python
import numpy
import matplotlib
matplotlib.rc( 'image', cmap='jet' )
from matplotlib import pyplot

##########################################################
## Assimulo imports
##########################################################
#import assimulo.solvers as aso
from Lobatto_IIIC_Assimulo import Lobatto4ODE
import assimulo.ode as aode
import assimulo.solvers as aso

##########################################################
## DUNE imports
##########################################################
#from dune.grid import structuredGrid as leafGridView
from dune.grid import cartesianDomain
from dune.alugrid import aluCubeGrid as leafGridView
#from dune.alugrid import aluSimplexGrid as leafGridView
from dune.common import FieldVector
from dune.grid import reader
from dune.fem import parameter
#from dune.fem.space import dgonb as dgSpace
from dune.fem.space import dglagrangelobatto as dgSpace
from dune.fem.operator import molGalerkin as molGalerkin
from dune.fem.function import uflFunction, integrate
from dune.ufl import Constant
from ufl import TestFunction, TrialFunction, SpatialCoordinate, triangle, FacetNormal
from ufl import dx, ds, grad, div, grad, dot, inner, sqrt, exp, conditional, sin, cos
from ufl import as_vector, avg, jump, dS, CellVolume, FacetArea, atan, pi
from dune.femdg.rk import ssp3, euler

# useAssimulo = False
useAssimulo = True

# 3rd order 4-stage Runge-Kutta (R.Alexander)
Stepper = ssp3(4,explicit=False)

# Explicit/Implicit Euler
#Stepper = euler(explicit=False)

time = Constant(0., "time")
dt   = Constant(0.005,"dt")


class SpatialOperator:
    def __init__(self,form,space,cfl=0.45):
        # create method of lines Galerkin operator from PDE form
        self._op = molGalerkin( form )
        self.cfl = cfl
        self.space = space
        # discrete functions uTmp and vTmp
        self.uTmp = space.interpolate([0], name='uTmp')
        self.vTmp = space.interpolate([0], name='vTmp')
        self.localTimeStepEstimate = [dt.value/self.cfl]

    # v = L[u]
    def apply(self, u, v):
        self._op(u,v)
        self.localTimeStepEstimate = [dt.value/self.cfl]
        return

    # jacobian of L[\bar{u}]
    def jacobian(self, ubar):
        from dune.fem.operator import linear as linearOperator
        return linearOperator(self._op, ubar=ubar).as_numpy

    # v = L[u]
    def __call__(self,u,v):
        self.apply(u,v)
        return

    # make current simulation time known to operator
    def setTime(self, t):
        time.value = t
        self._t  = t

    def stepTime(self,t0, dt0):
        global time
        if hasattr(self._op.model, "time"):
            print(f"Setting time to {self._t} + {t0 * dt}")
            self._op.model.time.value = self._t + t0 * dt
        else:
            time.value = self._t + t0 * dt
        # set time to model time if available
        # since time is not in the form we don't need this here
        #if hasattr(self._op.model,"time"):
        #    print("Model has time")
        #elif hasattr(self._op.model,"t"):
        #    print("Model has t")

    def applyLimiter(self, u):
        pass

    ####################################################
    # rhs function for Assimulo forwarding to apply
    ####################################################
    def rhs(self, t, y):
        """ Function that calculates the right-hand-side. Depending on
            the problem and the support of the solver, this function has
            the following input parameters:

            rhs(t,y)      - Normal ODE
        """
        ## set time for PDE operator
        self.setTime(t)
        # copy content of y into uTmp
        self.uTmp.as_numpy[:] = y[:]
        # apply spatial discretization operator L
        self.apply(self.uTmp, self.vTmp)
        # store result in y
        yn = y.copy()
        yn[:] = self.vTmp.as_numpy[:]
        return yn

    def jac(self, t, y, sw=None):
        # copy content of y into uTmp
        self.uTmp.as_numpy[:] = y[:]
        return self.jacobian( self.uTmp ).toarray()

parameter.append({"fem.verboserank": 0})

##########################################################
##
## Spatial discretization
##
##########################################################

domain = cartesianDomain([0, 0], [1, 1], [20, 20])
gridView = leafGridView(domain, dimgrid=2 )
space    = dgSpace(gridView, order=2, storage="fem")

u    = TrialFunction(space)
v    = TestFunction(space)
n    = FacetNormal(space)
he   = avg( CellVolume(space) ) / FacetArea(space)
hbnd = CellVolume(space) / FacetArea(space)
x    = SpatialCoordinate(space)

center  = as_vector([ 0.5,0.5 ])
x0 = x[0] - center[0]
x1 = x[1] - center[1]

ux = -4.0*x1
uy =  4.0*x0

# diffusion factor
epsilon = 0.001

# transport direction and upwind flux
b =  as_vector([ux,uy])
def u0(x,t):
    sig2 = 0.004
    sig2PlusDt4 = sig2+(4.0*eps*t)
    xq = ( x0*cos(4.0*t) + x1*sin(4.0*t)) + 0.25
    yq = (-x0*sin(4.0*t) + x1*cos(4.0*t))
    return (sig2/ (sig2PlusDt4) ) * exp (-( xq*xq + yq*yq ) / sig2PlusDt4 )

# transport direction and upwind flux
#b =  as_vector([1,1])

#def u0(xp,t):
```

```
#    res = 1.
#    for d in range(len(xp)):
#        res *= sin(2.*pi *(xp[d] - t))
#    return res


# upwind (same as LLF in this case)
hatb = (dot(b, n) + abs(dot(b, n)))/2.0

# diffusion factor
eps = Constant(epsilon,"eps")

# penalty parameter for DG scheme
beta = Constant( 10*space.order**2 if space.order > 0 else 1,"beta")

# exact solution
exact = uflFunction(gridView, name="exact", order=3, ufl=u0(x,time))

# d_t u + div( F(u) - eps grad u) = 0
# integration by parts on spatial terms yields
aInternal     = inner(eps*grad(u) -b*u, grad(v)) * dx
advSkeleton   = jump(hatb*u)*jump(v)*dS \
                +(hatb*u + (dot(b,n)-hatb)*exact)*v*ds

# B(u,v) = \int_\Omega grad(u)*grad(v)
#        - \int_\Omega { grad(u) } * [ v ] + [ u ] * { grad(v) }
#        + \int_\Gamma \eta*h^-1 [ u ] * [ v ]
#
# interior skeleton
diffSkeleton  = -eps*inner(jump(u,n),   avg(grad(v)))*dS \
                -eps*inner(avg(grad(u)),jump(v,n))*dS \
                +eps*beta/he*jump(u)*jump(v)*dS
# boundary skeleton
diffSkeleton += -eps*(u-exact)*dot(grad(v),n)*ds \
                -eps*dot(grad(u),n)*v *ds \
                +eps*beta/hbnd*(u-exact)*v*ds
#rhs = eps*8.*pi*pi* inner(exact,v) * dx

# minus since we are solving d_t u = L[u]
# which leads in the simplest form to
# unew = uold + dt * L[uold]
# However, in dune-fem we assume that
# d_t u + L[u] = 0
# therefore we implement -L[u] since we solve
# something like d_t u = -L[u]
form = -(aInternal + advSkeleton)
if abs(epsilon) > 0:
    form -= diffSkeleton
#    form += rhs

## Create right hand side operator
op = SpatialOperator(form, space, cfl=0.25)


error0 = 0.
eoc = 0
```

```python

# 3 EOC loops
for i in range(2):
    t = 0
    op.setTime( t )
    time.value = t

    # interpolate exact solution onto discrete space
    uh = space.interpolate( exact, name="solution")
    #uh.plot()

    # write initial data
    gridView.writeVTK("adv",number=0,celldata=[uh],pointdata=[uh,exact])

    # T
    endTime = 0.25

    # time derivative
    if useAssimulo:
        # uh.plot()
        y0 = numpy.zeros( uh.size )
        y0[:] = uh.as_numpy[:]

        AdvDiff = aode.Explicit_Problem(op.rhs, y0, 0.)
        AdvDiff.name = 'Rotating Pulse'
        AdvDiff.h = dt.value
        AdvDiff.jac = op.jac

        ###Choose which solver to use
        #solver = aso.RungeKutta34(AdvDiff)
        solver = Lobatto4ODE(AdvDiff)
        #solver = aso.Radau5ODE(AdvDiff)
        solver.report_continuously = True

        solver.atol = 1e-5
        solver.rtol = 1e-5
        solver.h = 0.0001
        #Option for our Lobatto method
        #solver._dynamic_step = False


        t, y = solver.simulate(endTime)


        # copy last solution back to uh
        uh.as_numpy[:] = y[-1][:]
    else:
        stepper = Stepper(op, cfl=op.cfl)

        uh_n = uh.copy()
        while t < endTime:
            stepper( uh, dt.value )
            t += dt.value
            op.setTime( t )
            print(f"time = {t}, dt = {dt.value}")

    #safe result
    gridView.writeVTK("adv",number=1,celldata=[uh],pointdata=[uh,exact])

    # compute L2 error
    error1= numpy.sqrt( integrate(gridView,dot(uh-exact,uh-exact),order=6))

    # plot solution
    #uh.plot()
    # exact.plot()

    # compute EOC
    if i > 0:
        eoc = [ numpy.log(error1/error0) / numpy.log(0.5) ]
    print(f"Step {i}: L2-error {error1} | EOC {eoc}")
    error0 = error1

    # refine grid to half grid size (delta x)
    gridView.hierarchicalGrid.globalRefine(1)
    # adjust time step size
    dt.value *= 0.25
```