DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

High-order time stepping schemes for solving partial differential equations with FEniCS

Nikola Wullenweber

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

High-order time stepping schemes for solving partial differential equations with FEniCS

Zeitschrittverfahren hoher Ordnung zur Lösung partieller Differentialgleichungen mit FEniCS

Author:	Nikola Wullenweber
Supervisor:	Benjamin Rodenberg
Examiner:	Prof. Dr. Hans-Joachim Bungartz
Submission Date:	15.07.2021

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.07.2021 Nikola Wullenweber

Abstract

Partial differential equations (PDEs) play an important role in natural sciences; however, they are far from trivial to solve. This work discusses how high-order time stepping schemes can be implemented in Python and applied to automatically solving PDEs with time-dependent boundary conditions. To this end, the Python libraries FEniCS and Irksome are analyzed. FEniCS provides an automated solution of PDEs employing the Finite Element Method, Irksome implements a way of automating Runge-Kutta time-stepping methods which are commonly used for the time discretization step in solving PDEs. However, they are not compatible. The goal of this work is developing a Python implementation of Runge-Kutta methods which is compatible with FEniCS and based on the implementation provided by Irksome. In general, a main challenge when applying Runge-Kutta methods is their high complexity, especially for higher orders. Therefore, this work implements a solution that automates the calculation of PDEs for different methods by simply specifying their butcher tableau. Furthermore, an example application of solving the elastodynamics equation with the generalized-α method is given.

Contents

1 Introduction

The process of solving a scientific problem typically requires a variety of skills and a large depth and breadth of knowledge. Mastering all of these is rarely possible for a single person. However, with the help of various open source projects that assemble different components to solve a problem, a way to facilitate developing a solution is created. This also applies to solving differential equations, which are equations that contain at least one derivative. They play an important role in most natural science and engineering disciplines as they can describe for example movements, currents, curves, models and all kind of other physical problems [1]. As Steven Strogatz, professor of Applied Mathematics at Cornell University, said: "Since Newton, mankind has come to realize that the laws of physics are always expressed in the language of differential equations."

However, most methods and libraries that exist for solving differential equations are designed for ordinary differential equations (ODE), which only depend on one variable. For partial differential equations (PDE) involving various variables and their derivatives, there are less options, especially for time discretization.

In this work, the Finite Element Method (FEM) is used for solving PDEs. FEM was introduced in the 1940s and computes the solution by dividing a computational domain into smaller parts – the so-called finite elements – and then solving these individually. In this thesis, the open source python library FEniCS is employed to solve PDEs with FEM. Additionally, time discretization is necessary to solve time-dependent problems. Methods for this already emerged in the 19th century. Those can calculate analytical as well as numerical solutions. An analytical solution is always exact, while a numerical solution can be an approximation. In 1895, C. Runge developed a generalization of the numerical computation of any solution of a given differential equation whose analytical solution is not known [2]. In 1901 Kutta took up this idea and further developed it, therefore they are called Runge-Kutta methods. One python library implementing this method is called Irksome. However, it is only compatible with Firedrake – another library for solving partial differential equations with FEM. The goal of this work is to implement a version of Irksome which is compatible with FEniCS.

In chapter 2, differential equations will be introduced and the concept of boundary conditions will be explained. Chapter 3 explains different time-stepping methods and their working. In chapter 4, FEniCS and Firedrake combined with Irksome will shortly

be introduced. Subsequently, chapter 5 shows how the 1D-advection equation and the heat equation can be implemented in FEniCS with different time stepping methods using the concept of Irksome. Thereafter, chapter 6 gives a further example application of solving the elastodynamics equation with the generalized-α method. Chapter 7 describes the results and analyzes the different methods in terms of implementation, accuracy, time and complexity. Lastly, chapter 8 summarizes the findings of this work and gives an outlook on future work.

2 Differential Equations

This chapter explains the concept of differential equations. In principle, a differential equation can be understood as an equation that includes at least one derivative, which can depend on different variables. The highest order of the derivative corresponds to the order of the differential equation. A main distinction is made between ODEs and PDEs. In an ODE, the unknown function depends only on one variable whereas in a PDE, it depends on several variables [3]. This will be explained in more detail in the next two sections. Subsequently, section 2.3 introduces boundary conditions which can specify the solution of a differential equation. Thereafter, three examples of PDEs will be introduced in section 2.4. To validate that a PDE is solved as expected, the method of manufactured solution (MMS) can be used. This will be explained shortly at the end of this chapter with the example of the heat equation in section 2.5.

2.1 Ordinary Differential Equations

As explained above, an ODE is a differential equation which only depends on one variable. The equation

$$F(x, y(x), y^{(1)}(x), ..., y^{(n)}(x)) = 0$$

is an ODE of n-th order for the unknown function y(x), where F is given [4]. As ODEs only depend on one variable, they are in general relatively straightforward to solve. Therefore, there already exist various methods which is why the focus of this work lies on partial differential equations.

2.2 Partial Differential Equations

A PDE is a differential equation which depends on more than one variable. The implicit form of PDE for a function u which depends on two variables x and y is

$$F(x, y, u(x, y), \frac{\partial u(x, y)}{\partial x}, \frac{\partial u(x, y)}{\partial y}, ..., \frac{\partial^2 u(x, y)}{\partial x \partial y}, ...) = 0 \tag{2.1}$$

PDEs depend on various variables; therefore, they are a lot harder to solve. While it is very difficult to find a solution to a PDE, it is easy to check whether a given function is a solution to a partial differential equation [1]. Nevertheless, different methods for solving PDEs have been developed. Among the most well-known is the finite element method (FEM) which is used in this work. As a further example, the method of lines (MOL) is introduced at the end of this subsection.

Finite Element method

One of the most common ways of solving PDEs is using the finite element method. It is a numerical analysis technique for approximating a wide variety of engineering problems. The main idea is to model a solution region by replacing it with an assemblage of discrete elements [5]. This idea was first introduced in the work of Courant [6] in 1943. He used linear approximation over sub regions with values specified at discrete points, which can be seen as the node points of a mesh of elements. However, the name finite element method was first used in 1960 by Clough in [7].

Principles of FEM

The finite element method reduces a continuum to elements, the so called finite elements, and expresses the unknown field variables in terms of approximating functions within each element. These approximating functions are also called interpolation functions and define the field variable throughout the assemblage of elements. Therefore, the degree of the approximation depends on the size and number of elements as well as the selected interpolation functions [5]. The solving of a problem follows a step-by-step process as explained in [5]:

- Discretize the continuum
- Select interpolation functions
- Find element properties
- Assemble the element properties to obtain the system equations
- Impose the boundary conditions
- Solve the system equations

First, the continuum must be divided into subdomains, the so-called elements. Next, interpolation functions are selected, which each represent the variation of the field variable over the element. Third, element properties must be found. Subsequently,

element properties are assembled in order to find the properties of the overall system. This is done by combining the matrix equations expressing the behaviour of the elements. Therefore, a matrix equation for the entire system is formed which has the same form as the equations of the individual elements except that it contains more terms as it includes all the nodes. Fifth, the boundary conditions are imposed. Most systems need boundary conditions in order to have a single solution, which will be explained further in section 2.3. Therefore, these known values on the boundaries are added to the system equations. Last, the system is solved by solving a set of linear or nonlinear algebraic equations.

Although the finite element method was developed in the 1940s, it still remains the dominant method for solving continuum problems today.

Method of lines

MOL is another approach to solve PDEs. Its basic idea is to replace the spatial derivatives in the PDE with algebraic approximations. This lets the spatial derivatives no longer be stated explicitly in terms of the spatial independent variable. Therefore only the initial value variable remain for solving the equation. That way, a system of ODEs can be created that approximates the original PDE. Then, any algorithm can be applied for solving the ODEs and get a numerical approximation of the PDE [8].

2.3 Boundary Conditions

Since the solutions of differential equations are usually not unique, they are often specified by a set of boundary conditions or initial conditions. A boundary condition describes the behaviour of a function on the boundary of its area of definition. An initial condition specifies the state of the system at the initial time $t = 0$. It should be noted that not all boundary conditions lead to a solution; therefore, it is important to use conditions that make physical sense. The maximum order of the derivative in the boundary condition must be one order less than the order of the differential equation [9]. The five types of boundary conditions often used are the Dirichlet boundary condition, the von Neumann boundary condition, the Robin's boundary condition, the mixed boundary condition and the Cauchy boundary Condition.

The Dirichlet boundary condition specifies the value of the unknown function u on its boundary. For a PDE as described in 2.1, it can be written as

$$u(x) = g(x), \quad x \in \delta\Omega$$

where Ω is the area of definition and the function $g : \delta\Omega \to \mathbb{R}$ describes the solutions of the function on the boundary.

A second main type are the von Neumann boundary conditions. These specify the values of the derivative for the boundaries of the definition area and can be defined for a PDE as

$$\frac{\partial u(x)}{\partial n} = g(x), \quad x \in \delta\Omega$$

where Ω is the area of definition, g(x) is a given scalar function and n is the normal to the boundary $\delta\Omega$.

Robin's boundary condition is a weighted combination of the ones introduced above. It employs a linear combination of the values of a function and its derivative on the boundary of the definition area. It is defined as

$$au(x) + b\frac{\partial u(x)}{\partial n} = g(x), \quad x \in \delta\Omega$$

where a and b are non-zero constants, Ω is the definition area, u is the unknown function and $\frac{\partial u(x)}{\partial n}$ is the normal derivative at the boundary.

When using mixed boundary conditions, different types of boundary conditions are applied in different parts of the domain. Cauchy boundary conditions are put both on the unknown field and its derivative. In contrary to the Robin condition, this implies the imposition of two constraints[1].

2.4 Examples of PDEs

In the following, three well-known PDEs, the one-dimensional advection equation(1D-advection equation) , the heat equation as well as the elastodynamics equation will be introduced. They are also used in the code examples of this work.

1D-advection equation

The advection problem is defined in the interval $[\alpha, \beta]$ and the PDE and its boundary conditions are defined as

$$\begin{aligned} \frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} &= 0, \quad t \in (0, T), \quad c > 0 \\ u(0, x) &= u_0(x) \\ u(\alpha) &= u(\beta) = 0 \end{aligned}$$

The advection equation can be solved analytically, which will later be used for calculating the error of the solution computed with time discretization methods [10]. Its solution is

[1] https://www.simscale.com/docs/simwiki/numerics-background/what-are-boundary-conditions/

$$u(t,x) = g(x - ct)$$

Heat equation

The heat equation is one of the most common problems in thermodynamics. It describes the stationary distribution of heat in a body to a time-dependent problem. For a two-dimensional spatial domain Ω, the model reads [11]

$$\frac{\partial u}{\partial t} = \bigtriangledown^2 u + f \quad in \quad \Omega \times (O,T], \tag{2.2}$$

$$u = u_D \quad on \quad \delta\Omega \times (0,T], \tag{2.3}$$

$$u = u_0 \quad at \quad t = 0. \tag{2.4}$$

Both examples make use of Dirichlet boundary conditions.

Elastodynamics equation

Elastonynacmics is part of continuum mechanics and covers the propagation of waves in elastic media [12]. The elastodynamic equation, which describes the wave movement of an elastic medium, reads the following:

$$\bigtriangledown\sigma + \rho b = \rho \frac{\partial^2 u}{\partial t^2}$$

u is the displacement vector field, the time derivation $\frac{\partial^2 u}{\partial t^2}$ is the acceleration, ρ the material density, b a given body force and σ the stress tensor related to the displacement through a constitutive equation [13]. For further details about the equations [13] can be read. In this work, the elastodynamic equation is solved in 6 using the generalized-α method as time discretization as it allow to add numerical damping.

2.5 Method of manufactured solution

MMS is is a procedure to determine if a piece of software will accurately calculate a solution to a prescribed numerical order[2]. This is done by constructing a problem with a known analytical solution so that it can be easily checked if the computed solution is correct. So for example, if the heat equation was solved with a first order time stepping method, one would need to create a test problem, that can easily be solved and has a

[2] https://mfix.netl.doe.gov/doc/vvuq-manual/main/html/mms/overview.html

linear variation in time. The FEniCS tutorial [11] suggests the following for the heat equation

$$u = 1 + x^2 + \alpha y^2 + \beta t$$

with two arbitrary parameter α and β. This function guarantees, that the computed values at the nodes will be exact regardless of the size of the elements and δt, as long as it is used with uniformly partitioned mesh [11]. By inserting this into the heat equation 2.2 one can find that

$$f(x, y, t) = \beta - 2 - 2a$$

Also, the initial value of the equation must be given by

$$u_0 = 1 + x^2 + \alpha y^2$$

and the boundary value by

$$u_D = 1 + x^2 + \alpha y^2 + \beta t$$

As the analytical solution of the problem is known, this can be used to calculate possible errors in the computed solution. To validate higher order methods, one can also adapt the problem to higher orders in time. So for example a polynomial order of 16 can be reached with

$$u = 1 + x^2 + \alpha y^2 + \beta t^{16}$$

and the corresponding equations

$$\begin{aligned} f(x, y, t) &= 15^{16} - 2 - 2a \\ u_0 &= 1 + x^2 + \alpha y^2 + 16t^{15} \\ u_D &= 1 + x^2 + \alpha y^2 + \beta t^{16} \end{aligned}$$

3 Time stepping methods

In the following, different methods to discretize the time in order to solve partial differential equations are presented. First, the Runge-Kutta methods are introduced. Their subdivision into explicit and implicit methods is explained with the aid of examples. To this end, the Butcher tableau, the explicit Euler method, the Heun's method and the RK4 method are introduced. Next, three examples for implicit Runge-Kutta methods – the Radau IIa methods, the Lobatto IIIc methods as well as the Gauss-Legendre methods – are discussed. Subsequently, the generalized-α method is introduced as an alternative to Runge-Kutta methods.

As many different problems can be described by the initial value problem for first order ODEs, the following equations will be considered for section 3.1 [14].

$$\frac{\partial y}{\partial t} = f(t, y), \quad t \in (0, T), \tag{3.1}$$

$$y(0) = y_0 \tag{3.2}$$

A numerical time stepping method calculates a sequence $y_0, y_1, y_2,...$ such that y_k approximates the solution of $y(t_0 + k\delta t)$. δt is the stepsize, so that $\delta t * k$ is the total time span T.

3.1 Runge-Kutta Methods

The Runge–Kutta methods are a family of implicit and explicit iterative methods used in time discretization for the approximate solutions of differential equations and belong to the class of one-step integrators [15]. They were first introduced by Carl Runge in 1893 [2] and Wilhelm Kutta in 1901 [16]. The main idea behind the Runge-Kutta methods is that the calculation of the solution is performed in a certain number of stages. Even though the order of the error does not necessarily equal the number of stages, it usually applies that increasing the number of stages also increases the order of the error. Runge-Kutta methods can be divided into two groups: explicit and implicit approaches. The family of implicit methods has only been known for about 50 years and has become much more important than the explicit methods as they guarantee more stability [17]. Although implicit methods require higher computational costs, they

are usually the superior choice for solving most scientific problems. These problems, where the cost is worth paying, are called stiff problems. Therefore, non-stiff problems are usually efficiently solved with explicit Runge-Kutta methods and stiff problems with implicit ones [15].

In the following, the difference between explicit and implicit methods will be shown. In section 3.1.1, the concept of Butcher tableaux will also be explained.

3.1.1 Explicit Runge-Kutta Methods

In order to approximate the individual solutions, the interval $[0, T]$ is divided into N time steps with a step size δt. For each point $y(n+1)$ with $t_{n+1} = t_n + \delta t$

$$y(n+1) = y(n) + \delta t \sum_{i=1}^{S} b_i k_i$$

applies with S being the number of stages and k_i being

$$\begin{aligned}
k_1 &= f(t_n, y_n) \\
k_2 &= f(t_n + c_2\delta t, y_n + \delta t a_(21 k_1) \\
k_3 &= f(t_n + c_3\delta t, y_n + \delta t(a_{31}k_2 + a_{32}k_2) \\
&... \\
k_s &= f(t_n + c_S\delta t, y_n + \delta(t a_{s1}k_1 + ... + a_{s,s-1}k_{s-1})
\end{aligned}$$

The Coefficients a_{ij}, b_i and c_i are determined by the particular method, but it usually holds that $c_i = \sum_{j=1}^{i-1} a_{ij}$. Additionally, a Runge-Kutta method is consistent only if $\sum_{i=1}^{s} b_i = 1$.

Butcher Tableau

With the coefficients a_{ij}, b_i and c_i , each Runge-Kutta method can be defined by a so called Butcher Tableau, which has the following form:

$$\begin{array}{c|cccc}
c_1 & a_{11} & a_{12} & \cdots & a_{1s} \\
c_2 & a_{11} & a_{12} & \cdots & a_{2s} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
c_s & a_{s1} & a_{s2} & \cdots & a_{ss} \\
\hline
 & b_1 & b_2 & \cdots & b_s
\end{array}$$

With those, a Runge-Kutta method can easily be specified by only giving a tableau.

The simplest explicit Runge-Kutta method is the explicit Euler methods, which has only one stage. Furthermore, Heun's method will be explained as it is an improved Euler method. On the other hand, the most used explicit Runge-Kutta method is the RK4 method, which has four stages and is of fourth order.

Explicit Euler method

The most known explicit Runge-Kutta method is the explicit Euler method, which proposes the following equation for solving the initial value problem 3.1 [15]

$$y(n+1) = y(n) + \int_{t_n}^{t_{n+1}} f(t, y(t))dt$$

Its Butcher Tableau is given by

0	0
	1

The explicit Euler method is a first-order method and was introduced by Leonhard Euler in 1768 [18]. It is often used as the basis of other, more complex methods.

Heun's Method

Heun's method, which is often referred to as an improved Euler method has the Butcher Tableau

0	0	0
1	1	0
	1/2	1/2

It is a two stage method and considers the interval spanned by the tangent line segment for the calculation of each time step.

RK4 method

The RK4 method is the most common Runge-Kutta method and is therefore also referred to as classical Runge-Kutta method. It has four stages and the following Butcher Tableau:

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

As the RK4 method is a fourth-order method, its local truncation error is on the order of $O(t^5)$ and its total accumulated error on the order of $O(t^4)$.

3.1.2 Implicit Runge-Kutta Methods

In the following, the family of implicit Runge-Kutta methods are explained more precisely using equations 3.1 and 3.2. In order to approximate the individual solutions, the interval [0,T] is again divided into N time steps with a step size δt. Each point $y(n+1)$ can be calculated with

$$y(n+1) = y(n) + \delta t \sum_{i=1}^{S} b_i k_i$$

and k_i

$$k_i = f(t_n + c_i h, y_n + h \sum_{j=1}^{S} a_{ij} k_j)$$

As above, a_{ij}, c_i and b_i are in principle chosen arbitrarily, but so that the resulting methods have a given accuracy. S describes the number of stages [19]. In general, implicit methods have higher computational costs. However, since they are more stable, they are used more often. The code examples in 5 use the following three families of implicit methods.

Radau IIa methods

Radau IIa methods are time discretization methods for solving stiff differential equations.

The most known method of this family is the implicit Euler method, which is also called backward Euler method. It is the implicit counterpart of the explicit Euler method and is therefore the simplest implicit method. Its Butcher Tableau is

1	1
	1

Therefore, it computes the approximation using

$$y_{n+1} = y_n + \delta t f(y_{n+1}, t_{n+1})$$

It is a first order method with a local truncation error of $O(t^2)$ and an error at specific time t of $O(t)$. Because of its simplicity it is often used in numerical analysis and scientific computing.

The third order Radau IIa method has the Butcher Tableau

1/3	5/12	-1/12
1	3/4	1/4
	3/4	1/4

There even exists a 17th order Radau IIa method which was introduced in 2010 and can be further seen in [20].

Lobatto IIIc methods

The LobattoIIIC methods are discontinuous collocation methods, which all have the order $(2s-2)$. Also, all of them have the coefficients $c_0 = 0$ and $c_i = 1$. For example the second order method which consequently has two stages has the Butcher Tableau

0	1/2	-1/2
1	1/2	1/2
	1/2	1/2

It is also called trapezoidal Rule and has a local truncation error of $O(t^3)$ and an error at specific time t of $O(t^2)$.

Another example is the fourth-order Lobatto IIIc method with the butcher tableau

0	1/6	-1/3	1/6
1/2	1/6	5/12	- 1/12
1	1/6	2/3	1/6
	1/6	2/3	1/6
	-1/2	2	-1/2

It has two stages, a local truncation error of $O(t^5)$ and an error at specific time of $O(t^4)$.

Gauss-Legendre methods

Last, the family of Gauss-Legendre methods is introduced. They are collocation methods based o the points of the Gauss-Legendre quadrature. All of them have a order of $2s$, so the order is twice as high as the number of stages. However, the computational cost is rather excessive, therefore they are rarely used [21].

The two order Gauss-Legendre is also called implicit midpoint rule and has the Butcher tableau

1/2	1/2
	1

The name midpoint rule has emerged from the fact, that the solution is always approximated at the midpoint between t_n and $t_n + 1$.

Gauss-legendre only need two stages to get a fourth order method. The corresponding butcher tableau is

$$\begin{array}{c|cc} \frac{1}{2} - \frac{1}{6}\sqrt{3} & \frac{1}{4} & \frac{1}{4} - \frac{1}{6}\sqrt{3} \\ \frac{1}{2} + \frac{1}{6}\sqrt{3} & \frac{1}{4} + \frac{1}{6}\sqrt{3} & \frac{1}{4} \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$$

As one can see, the butcher tableau is rather complicated compared to the of the Lobatto IIIc and the Radau IIa methods. But, for problems, where computational costs are not most important, Gauss-Legendre methods are a good choice, as for example a 16th order method can be implemented with only eight stages.

3.2 Generalized-α method

This section introduces the generalized-α method which can also be seen as an extension of the Newmark-beta method.The generalized-α method possesses numerical damping that can be controlled by the user [22]. Even though higher frequencies have been found to improve the convergence of iterative equation solvers, those often lead to less accuracy or excessive algorithmic damping in low frequency modes. The generalized-α methods avoids this while giving a second order accuracy in time and provides unconditional stability. Also, the algorithmic parameters can be defined in terms of the desired amount of high-frequency dissipation. Therefore, it is widely used in engineering and science [23]. In the following, it is shown how the generalized-α method is used for solving the semi-discrete initial value problem for non-linear structural dynamics.

$$M\frac{\partial^2 u}{\partial t^2} + C\frac{\partial u}{\partial t} + S(u(t)) = F(t)$$

M and C are the mass and damping matrices, $S(u(t))$ describes the vector of non-linear internal forces, $F(t9)$ is the vector of applied loads and $u(t)$ is the displacement vector [24]. To solve the problem, a function $u = u(t)$ which satisfies the Equation for all $t \in [0, T], \quad t > 0$ with the given initial value conditions must be determined.

$$u(0) = d, \quad u'(0) = v$$

The generalized-α method solves the equation at intermediate time between t_n and t_{n+1} as follows

$$[M]\{u_{n+1-\alpha_m}\} + [C]\{u_{n+1-\alpha_f}\} + [K]\{u_{n+1-\alpha_f}\} = \{F(t_{n+1-\alpha_f})\} \tag{3.3}$$

For $\alpha_f = \alpha_m = 0$, equation 3.3 describes the Newmark method. Usually, parameters $\alpha_m, \alpha_f \leq \frac{1}{2}$ are used.

4 Frameworks

The two open source libraries FEniCS and Firedrake that are used in this work will be introduced in the following. They both provide an automatic solution of partial differential equations. However, there are several differences between them, which will be analyzed in this chapter. The focus lies on FEniCS as it is the library used in the programmed part of this work. In order to solve partial differential equations with Runge-Kutta methods, Firedrake can be combined with the package Irksome.

4.1 FEniCS

FEniCS is a collection of open-source packages and allows the automated solution of differential equations with the finite element method. It can be used with C++ and Python on the operating systems Linux, OS X, Unix and Windows. However, in this work, FEniCS was used in Python on a Windows system. It consists of the following core components [25]:

- UFL (Unified Form Language) defines an interface for choosing finite element spaces and expressions for weak forms in a close to mathematical notation.
- FIAT (FInite element Automatic Tabulator) allows an automatic generation of functions for a wide range of finite element families.
- FFC (FEniCS Form Compiler) is used to create a low-level C++ Code from the high-level description of the form. It is needed for the C++ Interface.
- DOLFIN is the C++/Python interface of FENiCS and provides the problem solving environment.
- UFC (Unified Form-assembly Code) is the interface between FFC and DOLFIN.
- INSTANT is a module that allows to incline C++ code in Python

FEniCS contains an ensemble of functions and tools. For further documentation, please refer to [11]. Nevertheless, some of the FEniCS functions need to be discussed in this work for the code to be understandable. First, DOLFIN allows to create a mesh with a

single line of code. E.g. a uniform finite element mesh over the square unit $[0,1]x[0,1]$ divided into $8x8$ rectangles which are each divided into a pair of triangles would be created with

```
mesh = UnitSquareMesh(8, 8)
```

To define functionspaces and functions, FEniCS provides classes. A functionspace is created with 3 arguments, the first holding the mesh, the second the type of element and the third the degree of the finite element. For the type of element, FEniCS supports all simplex element families and the notation defined in the Periodic Table of the Finite Elements [26]. For functions defined on the functionspace, the classes Function(), TrialFunction and TestFunction() are provided. With these, the unknown function u and the testfunction v can be defined. The following code example shows how a functionspace of the standard Lagrange family with its corresponding functions can be implemented with FEniCS [11].

```
V = FunctionSpace(mesh, 'P', 1)
u = TrialFunction(V)
v = TestFunction(V)
```

For solving a PDE, an additional option to specify Boundary conditions must be provided. Usually, Dirichlet boundary conditions are used, which can be initiated with three arguments: first, the functionspace they are defined on, second, an expression u_D defining which points belong on the boundary, and third, a function that defines which points belong on the boundary [11]. A simple time-dependent Dirichlet boundary condition can be written as

```
def boundary(x, on_boundary):
    return on_boundary
V = FunctionSpace(msh, "CG", 1)
u_D = Expression('x[0]-c*t', degree=1, c=c,t=0)
bc = DirichletBC(V, u_D, boundary)
```

C++ syntax is used to define the expression u_D. It can depend on the variables $x[i]$ which correspond to the coordinates of the mesh. The function boundary checks if x is on the boundary of the defined area. One can also define more complex boundary functions to, for example, restrict the domain where the boundary condition is applied to only a part of the boundary. We refer to [11] for examples.

The variational problem can be defined either in its standard or in its abstract form. This is done with the unified form language which is, as described above, part of the FEniCS project. It defines discrete variational form and functional in a close to pen-and-paper notation. It has a set of operators and atomic expressions that help express

variational forms and functionals. Expressing forms in the following generalized format is possible

$$a(v;w) = \sum_{k=1}^{n_c} \int_{\Omega_k} I_k^c(v;w)dx + \sum_{k=1}^{n_e} \int_{\delta\Omega_k} I_k^e(v;w)ds + \sum_{k=1}^{n_i} \int_{\Gamma_k} I_k^i(v;w)dS$$

In UFL, integrals are expressed by multiplication with a measure representing the integral. Thus, a multiplication with dx is the integral over the interior of the domain Ω, with ds over the boundary $\delta\Omega$ of Ω and with dS over the set of interior facets Γ. Also, UFL has a way to express spatial derivatives. This can be done with

```
f = Dx(v, i)
```

which describes the scalar derivative in spatial direction x_i. Further information on how to define forms with UFL can be found in the UFL documentation [27]. As an example, the abstract form of the heat equation solved with the implicit Euler method can be written in UFL as

```
F = u*v*dx + dt*dot(grad(u), grad(v))*dx - (u_n + dt*f)*v*dx
a, L = lhs(F), rhs(F)
```

The derivation of this term is explained in detail in chapter 5.

Furthermore, it is possible to define finite elements with UFL. It is notated as

```
element = FiniteElement(family, cell, degree)
```

where family is one of the possible element families, cell is the element domain and degree the polynomial degree. Information on how a cell is defined and which families are available can as well be found in the UFL documentation. For elements there exist the classes FiniteElement, VectorElement, TensorElement, MixedElement and EnrichedElement in FEniCS.

Subsequently, the time-stepping with FEniCS is performed as

```
u = Function(V)
t = 0
for n in range(num_steps):
    # Update current time
    t += dt
    u_D.t = t
    # Solve variational problem
    solve(a == L, u, bc)
    # Update previous solution
    u_n.assign(u)
```

This section described the basic functions of FEniCS that are used in this work. A complete documentation can be found in [28].

4.2 Firedrake

Firedrake is another tool for automating the numerical solution of partial differential equations. Even though it adopts the domain-specific language for the finite element method of the FEniCS project, it has a pure Python runtime-only implementation centred on the composition of several existing and new abstractions for particular aspects of scientific computing. Additionally, it does not use the interface DOLFIN, but PyOP2, which is a parallel, unstructured mesh computation framework. The main optimisations include factorising mixed function spaces, transforming and vectorising inner loops, as well as intrinsically supporting block matrix operations [29]. Firedrake adds to the in FEniCS implemented separation of concerns between employing the finite element method and implementing it a separation within the implementation layer between the local discretization of mathematical operators and their parallel execution over the mesh. This results in a more compact code base, as the core Firedrake code only has 5000 lines of executable code [29]. However, the framework used in the code examples is FEniCS, so the usage of Firedrake is not further introduced, but can be seen in [29].

In this work, we try to adapt the Firedrake package Irksome in a way that it also works with FEniCS. The package implements a way of automating Runge-Kutta time-stepping for finite Element methods. It was first introduced in 2020 by Patrick E. Farrell, Robert C. Kirby and Jorge Marchena-Menendez [19]. The Project itself is restricted to Runge-Kutta methods, which cover various orders of accuracy. Instead of being a blackbox model of a Runge-Kutta method, it relies on a UFL manipulation approach.

The package Irksome has four main classes: *ButcherTableaux*, *deriv*, *stepper* and *getForm*. *ButcherTableaux* provides an interface for creating the corresponding Butcher Tableau to the used Runge-Kutta method. It is initialized with the Arguments A, b, btilde, c and order, of which A, b and c contain the weights of the Runge-Kutta method, btilde the weights for an embedded lower order method, if present, and order the formal order of accuracy of the method. While theoretically every Runge-Kutta method can be implemented by passing its Butcher tableau, Irksome provides implementations of many classical methods, like Gauss-Legendre, Lobatto IIIc and Radau IIa [19].

The class *deriv* handles the time derivative which is used in Irksome instead of coding the time discretization method. The time derivative is represented with $Dt(u)$. This allows to use the semi-discrete form of any PDE according to the methods of lines introduced in section 2.2. In chapter 5 $Dt(u)$ is used in the code examples to represent code, which does not yet have a time-stepping method implemented. However, in our implementation of Irksome in needs to be replaced by the used time-stepping method.

Stepper provides an interface to handle the main class of Irksome *getForm*. It takes an instance F of a UFL form as input which describes the semi-discrete problem

$F(t, u; v) == 0$, a Butcher tableau representing the Runge-Kutta method, the time values t and dt, a function u0 containing the current state of the problem to be solved, and, optionally, boundary conditions and solver parameters. At each time step, the function $stepper.advance(self)$ is called, which performs the solving of the problem.

The main functionality of Irksome is implemented in the function *getForm*. The function takes the UFL form for the time-dependent variational form, a Butcher tableau, the current time, a UFL coefficient and, optionally, boundary conditions as an input. It returns the UFL for the coupled, multi-stage method and the boundary conditions.

5 Implementation of Runge-Kutta Methods

As Irksome currently only works with the Firedrake package, the goal of this work was to adjust the code in order to work with FEniCS as well. In the following, it will be shown how Irksome automates the solution with Runge-Kutta methods using the examples of the 1D-advection equation and the heat equation. Hereafter, the developed solution, which automatically adapts the code to different Runge-Kutta methods for a given butcher tableau, is introduced.

5.1 1D-Advection equation

The code was developed with two different PDEs, of which one is the 1D-advection equation and the other the heat equation.

1D-advection equation

As described in chapter 2, the advection problem is the following

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0, \quad t \in (0, T), c > 0$$
$$u(0, x) = u_0(x)$$
$$u(\alpha) = u(\beta) = 0$$

The advection equation can be solved analytically, which will later be used for calculating the error of the solution computed with time discretization methods [10]. Its solution is

$$u(t, x) = g(x - ct)$$

Usually, with the Irksome extension $Dt(u)$ describing the time discretization, the equation can be written in UFL in its semi-deiscrete form as

```
F = u*v*dx + u_n*v*dx - c*dt*Dt(u)*v*dx
```

However, a time stepping method to replace $Dt(u)$ is necessary for the solution. In the following, it will be discussed how this can be done with the one stage Radau IIa

method, the so-called implicit Euler method, and thereafter with Heun's method. Thereby, it will also be elaborated in more detail how Runge-Kutta methods can be implemented.

Solution with the implicit Euler method

First, the 1d-advection equation was solved in the usual way of solving an equation in FEniCS with the implicit Euler method as introduced in the FEniCS tutorial [11]. To this end, the time derivative is first discretized by a finite difference approximation yielding a sequence of stationary problems which are then each turned into a variational form. Let u^n mean u at time level n. Then, the 1D-advection equation at time level $n+1$ can be written as:

$$(\frac{\partial u}{\partial t})^{n+1} = -c\frac{\partial u^{n+1}}{\partial x}$$

Using the backward Euler time discretization, the time-derivative can be approximated with

$$(\frac{\partial u}{\partial t})^{n+1} \approx \frac{u^{n+1} - u^n}{\delta t}$$

This results in a sequence of spatial problems for u^{n+1}:

$$u^{n+1} + u^n - c\delta t\frac{\partial u^{n+1}}{\partial x} = 0$$

For the finite element method, the equation then has to be multiplied by a testfunction v and integrated by parts. In the following, the symbol u will be used for u^{n+1}

$$F_{n+1}(u;v) = \int_\alpha^\beta (uv + u^n v - c\delta t v\frac{\partial u}{\partial x})dx = 0$$

In UFL, this can be written as

```
F = u*v*dx + u_n*v*dx - dt*c*Dx(u,1)*v*dx
```

with v being the testfunction, u the trialfunction, u_n the known solution of the previous step and dt the time step size. The $Dt()$ from before was hereby replaced by the implicit Euler method.

Solution with the implicit Euler method based on Irksome

However, for implementing Runge-Kutta methods, the variational form needs to update stages rather than updating the value of the previous calculated time step [19]. To this end, Irksome provides an automatic transformation from the semidiscrete form F to its

Runge-Kutta variational form. The concept of this will be shown in the following using the examples Backward Euler method and Heun's method. As described above, the implicit Euler method has the Butcher Tableau

$$\begin{array}{c|c} 1 & 1 \\ \hline & 1 \end{array}$$

Therefore, k_0 and u_0 can be defined as

$$k_1 = f(t_n, u_n)$$
$$u_0 = u_n + \delta t f(t_n, u_n)$$

In UFL, this can be written as:

```
k1 = Function(V)
v0 = TestFunction(V)
u0 = u_n + dt*Constant(1.0)*k0
F = u*v0*dx+u0*v*dx - dt*Dx(u0,1)*v0*k0*dx
```

u_n is a given function in a functionspace V containing the solution at time n and *dt* is the time step size.

Solution with Heun's Method based on Irksome

We have generalized the implicit Euler method. Now we can use any Runge-Kutta method, e.g. Heun. For this we still need some modifications, which are explained in the example. The 1D-advection equation was solved with the Heun's method which is also known as the explicit midpoint rule. Heun's method has the Butcher Tableau

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1 & 0 \\ \hline & 1/2 & 1/2 \end{array}$$

Accordingly, the Runge-Kutta equations can be defined as:

$$k_1 = f(t_n, u_n)$$
$$k_2 = f(t_n + \delta t, u_n + \delta t k_1)$$
$$u_0 = u_n$$
$$u_0 = u_n + \delta t k_1$$

The corresponding UFL code is given by

```
E = V.ufl_element() * V.ufl_element()
Vbig = FunctionSpace(V.mesh(),E)
u = TrialFunction(V)
k = Function(Vbig)
k0, k1 = split(k)
v0, v1 = TestFunctions(Vbig)
u0 = u_n + dt*Constant(0.0)*k0 + dt*Constant(0.0)*k1
u1 = u_n + dt*Constant(1.0)*k0 + dt*Constant(0.0)*k1
F = u*v0*dx + u*v1*dx + u0*v0dx + u1*v1*dx
    - dt*Dx(u0,1)*v0*k0*dx - dt*Dx(u1,1)*v1*k1*dx
```

As above, u_n is a given function in a Functionspace V containing the solution at time n and dt is the time step size. As Heun's method is an implicit method, solving PDEs with it is not very reasonable. However, here, it was used with the purpose of showing the functionality with Irksome.

5.2 Heat equation

For the development of the simplified version of Irksome, the example of the heat equation was also used. Although the 1D-advection equation is very simple, as it is only a first order equation and the analytical solution is very easy to calculate, the heat equation is more suitable for our purposes. This is due to the fact that it can be solved accurately and the analytical solution can be adapted very easily to the respective needs. However, the heat equation is of second order and has a right side, which makes it a bit more complicated.

The heat equation was first implemented using the implicit Euler method based on the FEniCS tutorial, and then using the Lobatto IIIc method based on a code proposed in the Irksome paper [19].

Heat equation

The heat equation is given by the following as can be seen in Chapter 2 and in [11]

$$\frac{\partial u}{\partial t} = \bigtriangledown^2 u + f \quad in \quad \Omega \times (0, T], \tag{5.1}$$

$$u = u_D \quad on \quad \delta\Omega \times (0, T], \tag{5.2}$$

$$u = u_0 \quad at \quad t = 0. \tag{5.3}$$

With the Irksome extension $Dt()$, one would express this in UFL in its semi-discrete form as [19]

```
F = inner (Dt(u), v) * dx + inner (grad(u), grad(v)) * dx
```

As our version of Irksome does not support $Dt()$ it will be replaced with different time-stepping methods in the next subsections.

In the following examples, let the boundaries u_D be as described in section 2.5 using the MMS

```
u_D = Expression('1 + x[0]*x[0] + alpha*x[1]*x[1] + beta*t',
                 degree=2, alpha=alpha, beta=beta, t=0)
```

with $alpha$ and $beta$ being arbitrary parameters. With this definition of u_D, the problem is a first order problem.

Solution with the one stage Radau IIa method

The FEniCS tutorial [11] explains in further detail how the time derivative of the heat equation is first discretized into a sequence of stationary problems, which are then each turned into a variational form. This step is not included here, since it can be read in detail in [11].

For the backward Euler time stepping method, the resulting equation is

$$F_{n+1}(u;v) = \int_\Omega (uv + \delta t \bigtriangledown u \bigtriangledown v - (u^n + f^{n+1}v)dx$$

where v is the testfunction, δt the time step size and the symbol u is used for u_{n+1}. In UFL, this is written as

```
F = u*v*dx + dt*dot(grad(u), grad(v))*dx - (u_n + dt*f)*v*dx
```

with v being the testfunction, u the trialfunction, u_n the known solution of the previous step and dt the time step size. As the backward Euler method is a first order method, it can only solve first order problems without errors beyond computational errors.

Solution with the second order Lobatto IIIc method based on Irksome

Next, the two dimensional heat equation is solved using the second order Lobatto IIIc method as described in [19]. The second order Lobatto IIIc method has the Butcher Tableau

0	1/2	-1/2
1	1/2	1/2
	1/2	1/2

As it has two stages, the solution at each point $n+1$ can be computed with

$$y_{n+1} = y_n + \delta t b_1 k_1 + \delta t b_2 k_2 \tag{5.4}$$

with

$$k_1 = f(t_n + c_1 \delta t, y_n + \delta t a_{11} k_1 + \delta t a_{12} k_2)$$
$$k_2 = f(t_n + c_2 \delta t, y_n + \delta t a_{21} k_1 + \delta t a_{22} k_2)$$

and

$$\tilde{u}_1 = y_n + +\delta t a_{11} k_1 + \delta t a_{12} k_2 \tag{5.5}$$
$$\tilde{u}_2 = y_n + \delta t a_{21} k_1 + \delta t a_{22} k_2 \tag{5.6}$$

The right hand side (RHS) of the heat equation can be calculated with equation 5.1 by subtracting the time derivative on both sides

$$rhs = u - \frac{\partial u}{\partial t} \tag{5.7}$$

With this, the weak form of the solution u is

$$< \frac{\partial u}{\partial t}, v > + < \bigtriangledown u, \bigtriangledown v > = < rhs, v >$$

Therefore, the weak form for each stage k_i must be

$$< k_i, v_i > + < \bigtriangledown \tilde{u}_i, \bigtriangledown v_i > = < rhs(t_n + c_i \delta t), v$$

Consequently, the complete variational form for the heat equation solved with a two stage Runge-Kutta method is

$$\begin{aligned} 0 = &< k_1, v_1 > + < \bigtriangledown u_1, \bigtriangledown v_1 > - < rhs_1, v_1 > + \\ &< k_2, v_2 > + < \bigtriangledown u_2, \bigtriangledown v_2 > - < rhs_2, v_2 > \end{aligned}$$

One challenge for the implementation arose particularly at the implementation of the RHS as it must include the derivative and also be adapted to the number of stages. This was resolved as follows

```
f = ns * [None]
for i in range(ns):
    f[i] = Expression('beta - 2 - 2*alpha', degree=2,
                      alpha=alpha, beta=beta, t=0)
    f[i].t = t + bt.c[i] * dt
```

As can be seen, the in section 2.5 according to the MMS calculated *u_D* was used. The functions u and v as well as the Form F are then defined as

```
k0, k1 = TrialFunctions(Vbig)
v0, v1 = TestFunctions(Vbig)

u0 = u_ini + A[0][0] * dt * k0 + A[0][1] * dt * k1
u1 = u_ini + A[1][0] * dt * k0 + A[1][1] * dt * k1

F = (inner(k0 , v0) * dx + inner(grad(u0), grad(v0)) * dx)
    + (inner(k1, v1) * dx + inner(grad(u1), grad(v1)) * dx)-
    f[1] * v1 * dx - f[0] * v0 * dx
```

where *u_ini* is the interpolation of u_D with V and describes the solution at t=0.

In this example, it also needs to be taken into account that the boundary conditions need to be a time derivation of *u_D* for the reason that the solution is found by computing the stages *k*0 and *k*1. Therefore, the boundary conditions are defined as

```
du_Ddt = ns * [None]
bc = []
for i in range(ns):
    du_Ddt[i] = Expression('beta', degree=2, alpha=alpha,
                           beta=beta, t=0)
    du_Ddt[i].t = t + bt.c[i] * dt
    bc.append(DirichletBC(Vbig.sub(i), du_Ddt[i], boundary))
```

The expression *beta* is the time derivative of *u_D* as explained in section 2.5. The two stage Lobatto IIIc method is a second order method and can therefore solve second order problems without errors besides computational ones.

5.3 Code

This subsection explains how the simplified FEniCS version of Irksome was implemented and how it is based on the original Irksome code. For this, the heat equation is used again.

As described in section 4.2 the functionality of Irksome is mainly implemented by the function *getForm*, which performs the UFL manipulation and also adopts the boundary condition to the corresponding Runge-Kutta method. The class *stepper* handles the interaction with getForm.

However, our implemented version only uses the file *ButcherTableaux*; nevertheless, the code is also based on the other Irksome classes, especially *getForm*. It can be

divided into three parts. First, the general conditions for solving the equation are set up, next, the variational form is created and, finally, the equation itself is solved.

Setting up benchmark scenario

To begin with, all the parameters are defined. These are the final time, current time, number of time steps, time step size and additional parameters that may be needed for solving the functions. In our example of the heat equation, these are the parameters alpha and beta, which are used for the initial conditions.

```
T = 2.0             # final time
t = 0               # current time
num_steps = 20      # number of time steps
dt = T / num_steps # time step size
alpha = 3           # parameter alpha
beta = 1.2          # parameter beta
```

Now, the geometry is defined. For our example, a mesh with a size of $8 * 8$ is chosen. However, any mesh can be used.

```
nx = ny = 8
msh = UnitSquareMesh(nx, ny)
```

Finally, the Butcher tableau is selected. The Irksome file ButcherTableaux is used for this. It represents many different Runge-Kutta methods. The required number of stages can be passed by a parameter. For example, the following code would initialize the two-stage LobattoIIC method.

```
bt =  LobattoIIIC(2)
num_stages = bt.num_stages
A = bt.A
b = bt.b
c = bt.c
```

Creating variational form

To solve the equation, the variational form must be created. To this end, a functionspace must be initialized. Since the equation is solved with a Runge-Kutta method in several stages, it must have the same dimension as the number of stages. This was realised with the help of a mixed element.

```
# Create mixed function space depending on number of stages
V = FunctionSpace(msh, "P", 1)
if(num_stages==1):
    Vbig=V
else:
    mixed = MixedElement(num_stages*[V.ufl_element()])
    Vbig = FunctionSpace(V.mesh(), mixed)
```

However, if there is only one stage, FEniCS cannot use a mixed element. Due to this, there is a need for the if-clause.

To have a unique solution, initial conditions must be defined. The parameters alpha and beta are used for this. Also, as the heat equation is time dependent, the initial condition is time dependent.

```
# Define initial condition
u_D = Expression('1 + x[0]*x[0] + alpha*x[1]*x[1] + beta*t*t', degree=2,
                 alpha=alpha, beta=beta, t=0)
u_ini = interpolate(u_D, V)
```

Next, the boundary conditions are set up. Since the individual stages k_i are calculated as the solution, this must be a time derivative of *u_D*. The calculation for the corresponding number of stages is performed as follows

```
def boundary(x, on_boundary):
    return on_boundary
du_Ddt = num_stages * [None]
bc = []
for i in range(num_stages):
    du_Ddt[i] = Expression('2*beta*t', degree=3, alpha=alpha,
                           beta=beta, t=0)
    du_Ddt[i].t = t
    for j in range (i-1):
        du_Ddt[i].t = du_Ddt[i].t + bt.c[j] * dt
    if(num_stages==1):
        bc.append(DirichletBC(Vbig, du_Ddt[i], boundary))
    else:
        bc.append(DirichletBC(Vbig.sub(i), du_Ddt[i], boundary))
```

In FEniCS, trialfunctions as well as testfunctions can always be defined with the functions *TestFunction*() and *TrialFunction*() by simply passing a functionspace as the argument. However, if multidimensional functionspaces are used as they are

needed for solving equations with Runge-Kutta methods, that have more than one stage, there also need to be multiple test- and trialfunctions. This can be implemented by splitting the functions that have been defined over the expanded functionspace. As a result, the number of test- and trialfunctions equals the number of stages f of the used Runge-Kutta method.

```
k = TrialFunction(Vbig)
v = TestFunction(Vbig)
ks = split(k)
vs = split(v)
```

Afterwards, the solutions for each individual stage are calculated. Doing this is equivalent to equations 5.5 and 5.6 and implemented with the help of two for-loops. The values in the matrix A from the Butcher Tableaux are also used as explained in Chapter 3.

```
u = num_stages * [None]
for i in range(num_stages):
    uhelp = u_ini
    for j in range (num_stages):
        uhelp = uhelp + A[i][j] * dt * ks[j]
    u[i] = uhelp
```

In order to solve the equation, the right-hand side of the equation must also be defined, since it is not equal to zero. Since it is calculated as in 5.7, this can be realised by the following code. Again, the derivative of *u_D* must be employed.

```
f = num_stages * [None]
for i in range(num_stages):
    f[i] = Expression('2*beta*t- 2 - 2*alpha', degree=2,
                      alpha=alpha, beta=beta, t=0)
    f[i].t = t + c[i] * dt
rh = 0
for i in range(num_stages):
    rh = rh + f[i]* vs[i]*dx
```

Finally, the weak Runge-Kutta Form is assembled from the calculated parts.

```
F = 0
for i in range(num_stages):
    F = F + heatreplace(vs[i],u[i],ks[i])
F = F - rh
a, L = lhs(F), rhs(F)
```

For this, there exists the function *replace*. It takes the arguments v, u and k and then returns the left side of the heat equation with the passed parameters. Theoretically, this function can be replaced with any other function for different equations which makes an adaption of the code to other problems easy. However, for the heat equation the function *replace* is defined as follows

```
def replace( v, u, k):
    L = (inner(k , v) * dx + inner(grad(u), grad(v)) * dx)
    return L
```

One of the advantages of the concept of Irksome is that one must only define the semi-discrete form of the PDE as it is used in the method of lines. This is the simplified space discretization without any time discretization.

Solving equation

The last step is to solve the problem itself. The unknown stages are the values that need to be solved. Therefore, they are defined as a function over the expanded functionspace. For solving the equation, a for-loop over the number of time steps is used and the individual stages are calculated at each time. Subsequently, the solution must be put together, as calculated in 5.4. Also, the time in the boundary conditions and the RHS must be updated in each step, as well as the time itself. Finally, a reference solution is computed by interpolating the initial condition V. This helps calculating the error.

```
# Unknown: stages k
k = Function(Vbig)
for n in range(num_steps):
    # Update BCs and rhs wrt current time.
    for i in range(num_stages):
        du_Ddt[i].t = t + bt.c[i] * dt
        f[i].t = t + bt.c[i] * dt
    # Compute solution for stages
    solve(a == L, k, bc)
    # Assemble solution from stages
    if(num_stages==1):
        u_sol_help = u_ini+ dt*bt.b[0]*k
    else:
        u_sol_help = u_ini
        for i in range (num_stages):
            u_sol_help = u_sol_help + dt*bt.b[i]*k.sub(i)
```

```
u_sol = project(u_sol_help,V)
u_ini.assign(u_sol)
# Update time and compute reference solution
t += dt
u_D.t = t
u_ref = interpolate(u_D, V)
```

To estimate the error, the L2-norm is calculated. For an n-dimensional vector, this is defined as

$$|x| = \sqrt{\sum_{k=1}^{n} |x_k|^2}$$

The different Runge-Kutta methods compute the expected results with expected errors for higher order equations. Further derivation on the results can be found in Chapter 7.

6 Generalized-α method

This chapter describes how the in section 2.4 presented elastodynamics equation was implemented in FEniCS using the generalized-α method. The code is based on the example [1] introduced in [13]. In order to examine the functioning of the generalized-α method in more detail, the sample code[2] was supplemented by an error calculation. Hereby the decrease of the total energy loss in the time period of 4.0 seconds was calculated for different different time steps $[0.25, 0.125, 0.0625, 0.03125]$. As can be seen in Figure 6.1, the total energy loss decreases with increasing time t, thus this implementation of the generalized-α method is not energy conserving. This is indicating numerical damping, which is according to [13] as expected. Also, it can be seen that with decreasing time step size, the total energy loss increases.

[1] https://comet-fenics.readthedocs.io/en/latest/_sources/demo/elastodynamics/demo_elastodynamics.py.rst.txt
[2] https://comet-fenics.readthedocs.io/en/latest/demo/elastodynamics/demo_elastodynamics.py.html#time-discretization-using-the-generalized-alpha-method

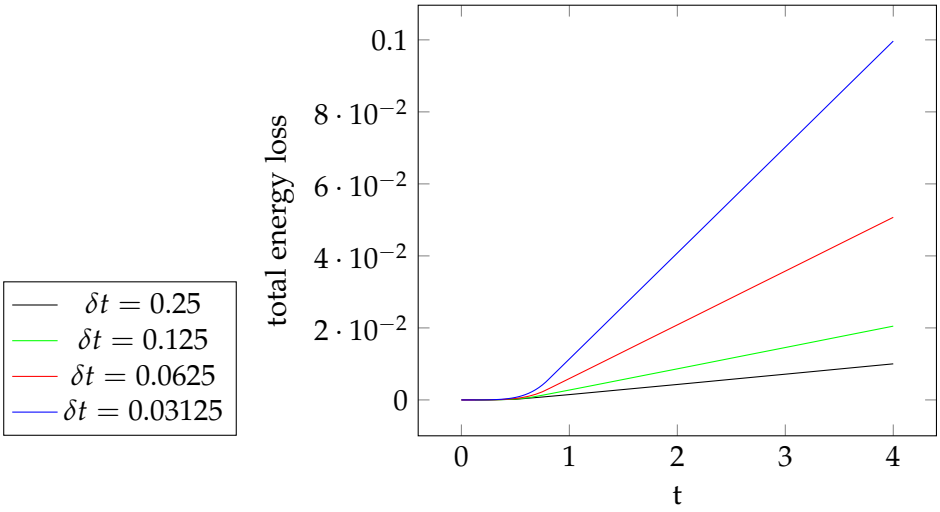

Figure 6.1: Total energy loss of the elastodynamics equation

7 Results

This chapter shows how the different time stepping methods behave for PDEs of different polynomial degree. In chapter 5 it was shown how to implement different time-stepping methods for the heat equation. Now the MMS as explained in section 2.2 will be used to fit the polynomial degree of the test problem to the order of the time-stepping methods and to calculate the error. All of the results in this chapter are based on the following setup to solve the heat equation

- maximum simulation time T = 2.0
- largest timestep size $\delta t_0 = 1/4$
- considered timestep sizes δt_i = [1/4, 1/8, 1/16, ..., 1/1024, 1/2048]

In 7.1 the results in general are introduced. Thereafter, a convergence study is performed in 7.2.

7.1 General results

The implemented code for solving PDEs with Runge-Kutta methods behaves according to the order of the used method. Because of the usage of Runge-Kutta methods we achieved to solve heat equation with a very high order RHS. So, it was for example possible to solve the heat equation, with a polynomial order of the RHS of 16 with an eight stage Gauss-Legendre method.

We tested this with a series of Gauss-Legendre methods of orders 1-8 and a large time step size of $\delta t = 0.03125$ solving the heat equation with right-hand side terms of order 2-16. The initial condition was adapted to the orders of the time-stepping method according to the MMS. So for example the initial condition u_D of 16th order was defined as:

```
u_D = Expression('1 + x[0]*x[0] + alpha*x[1]*x[1] +
       beta*t*t*t*t*t*t*t*t*t*t*t*t*t*t*t*t',
       degree=2, alpha=alpha, beta=beta, t=0)
```

The results can be seen in the following table

Number of stages	Polynomial order of RHS	Error
1	2	1.098402742047217e-14
2	4	7.45602465414041e-15
3	6	7.228898649093617e-15
4	8	7.612449719588713e-15
5	10	7.90655177936233e-15
6	12	7.975739059056074e-15
7	14	7.793098923555298e-15
8	16	8.098984644410854e-15

As all of the the errors are quite small, they most likely originated from computational errors, therefore the methods work according to their orders.

7.2 Convergence study

The FEniCS version of Irksome can also be used to perform a convergence study. The heat equation as implemented in chapter 5 is again used as a benchmark scenario.

There were several convergence studies done. For each study, one each of the three families of implicit Runge-Kutta methods presented was used. The notation of the graphs used is L(i) for the Lobatto IIIc method with i stages, R(i) for the Radau IIa method with i stages and G(i) for the Gauss-Legendre method with i stages.

Third order heat equation

First, a heat equation problem with a third order RHS was solved using different Runge-Kutta methods. The applied initial condition is the following

```
u_D = Expression('1 + x[0]*x[0] + alpha*x[1]*x[1] + beta*t*t*t',
        degree=2, alpha=alpha, beta=beta, t=0)
```

The results can be seen in Figure 7.1. G(1) is the second order Gauss-Legendre methods, which has one stage, L(3) is the fourth order Lobatto IIIc method with three stages and R(1) the first order Radau IIa method with one stage which is the implicit Euler method. All methods behave as expected and reach the order that is expected by theory. The errors of the fourth order Lobatto IIIc method are so small that they most likely originated from computational errors. This also explains, why they are bigger for smaller time-steps, as those require more computational work. Therefore, Lobatto IIIc also behaves as expected.

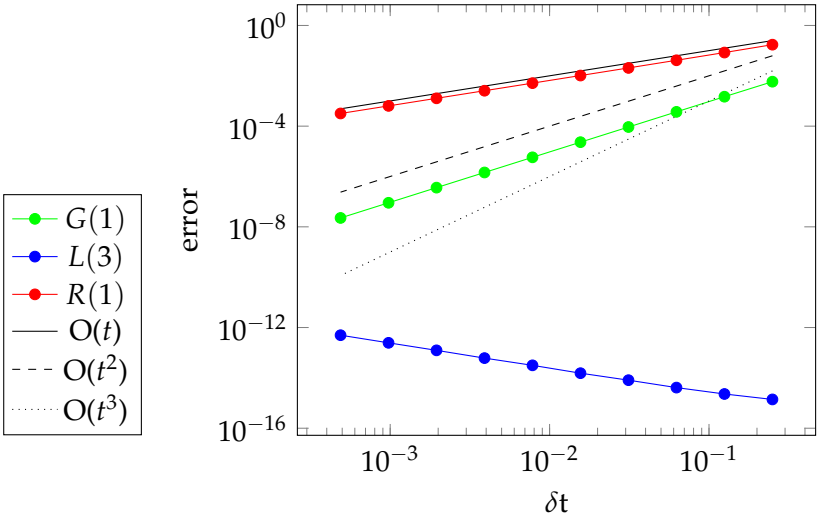

Figure 7.1: Methods behave as expected for a heat equation with a third order RHS

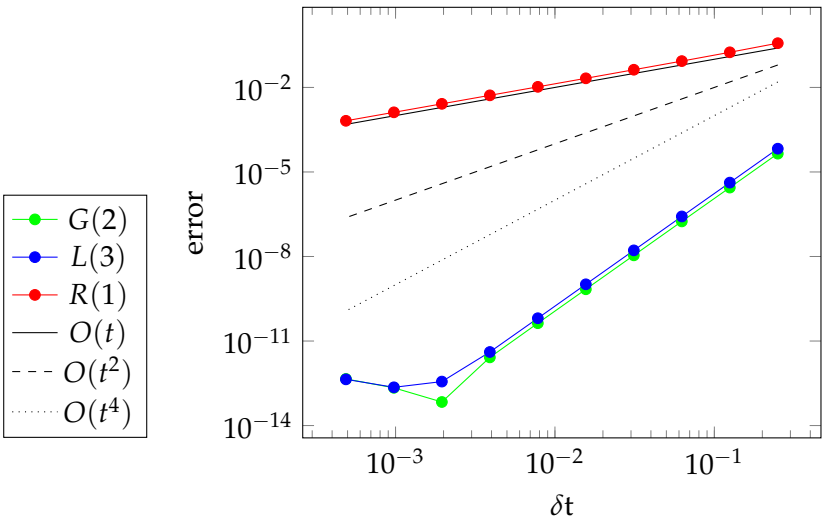

Figure 7.2: Methods behave as expected for a heat equation with a fifth order RHS

Fifth order heat equation

Another convergence study was performed with a heat equation with a fifth order RHS. For this, *u_D* was changed to:

```
u_D = Expression('1 + x[0]*x[0] + alpha*x[1]*x[1] + beta*t*t*t*t*t',
        degree=2, alpha=alpha, beta=beta, t=0)
```

This was solved with the three staged Lobatto IIIc method, the implicit Euler method and the two stage Gauss-Legendre method. As can be seen in 7.2, the results almost meet the expectations. Only at very small time steps, there are a few deviations from the usual error curves for the fourth order methods L(3) and G(2). This can be explained by the fact, that those errors are extremely small and therefore in the area of computational errors.

Seventh order heat equation

Last, a convergence study was done with a heat equation with a seventh order RHS and the following initial condition for *u_D*

```
u_D = Expression('1 + x[0]*x[0] + alpha*x[1]*x[1] + beta*t*t*t*t*t*t*t',
        degree=2, alpha=alpha, beta=beta, t=0)
```

For the time discretization the Lobatto IIIc method, the Radau IIa method and the Gauss-Legendre method are all used with three stages. This means L(3) is a fourth order methods, R(3) a fifth order method and G(3) a sixth order method. As one can see in 7.3, the graph only behaves as expected with larger time steps. G(3) has the largest deviations, as it also has the largest order. This again is due to computational errors.. Therefore, the graphs correspond as expected to the respective error order.

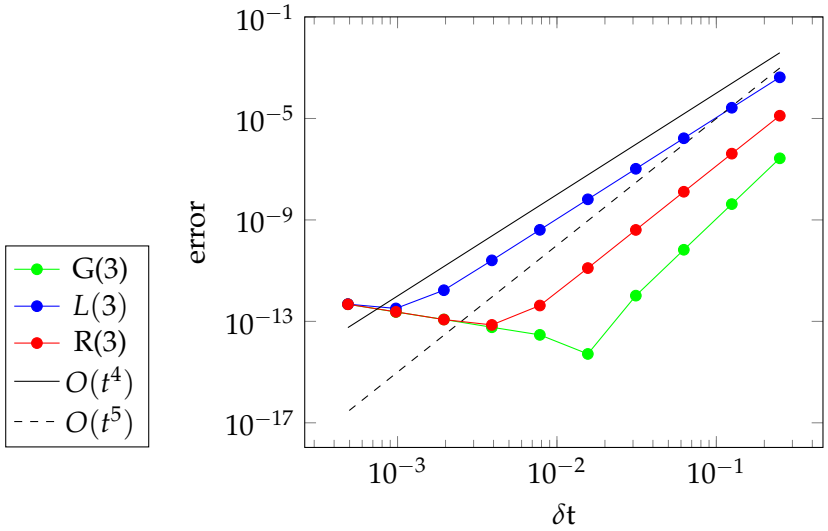

Figure 7.3: Methods behave as expected for a heat equation with a third order RHS

8 Conclusion and Outlook

The following sections summarize the conducted work and give an outlook on further research.

8.1 Summary of the Work

As it is a long-standing critique of fully implicit Runge-Kutta methods, especially for PDEs, that they require a very large algebraic solve for all stages concurrently [19], it is very beneficial to have a program like Irksome. Originally, it was planned to develop an equivalent version of Irksome, which is compatible with FEniCS. However, due to several differences between Firedrake and FEniCS, this would have been beyond the scope of this thesis. Therefore, this work provides a simplified version. Thereby, it is possible to implement Runge-Kutta methods of several families simply by specifying the respective Butcher tableau. In chapter 7 it has been shown that these methods work up to the 16th order. We used a simple heat equation example to verify correct behavior through convergence studies. It showed that the methods behave according to their order in convergence studies. The 1D-advection equation and the heat equation were used to develop the code. Also, we implemented the elastodynamics equation with the generalized-α method to give a further example of how PDEs are solved with time-stepping methods.

Thanks to projects like Irksome, it is becoming more and more easy to solve PDEs efficiently and accurately with a relatively intuitive code compared to the problem itself. This is precisely why it is important to offer this simplification to the widest more than one finite element toolboxes. This work is meant to transfer an automated Runge-Kutta method for differential equations to FEniCS.

8.2 Outlook

In this work, we have seen that the concepts of Irksome can be transferred to FEniCS. Naturally, this does not only apply for FEniCS, but for finite element toolboxes in general. As the implementation of time stepping schemes can be complicated, time-consuming and error-prone, simple ways to implement efficient methods for time

discretization are needed. This was done with Irksome [19] and also with our proposed code for Runge-Kutta methods. However, it would be good if the in this paper proposed code could be further evaluated as it was only tested with the heat equation so far. Irksome on the other hand gives more examples of solving different PDEs in [19]. It is presumably possible to extend the FEniCS code to the functionalities of Irksome. For some PDEs the generalised-α method might be the better choice as it allows numerical dumping, which can be controlled by the user. But it is only a second order method and can therefore not be used for an analytical solution of higher order differential equations. However, it can still approximate those. In [23] a third-order generalized-alpha method is devised and analyzed. To achieve this, extra parameters were assigned to the higher-order terms of a Taylor series representation. Nevertheless, in the current state of research, the generalised-α method still only exists up to the third order.

Since partial differential equations are used in the solution of most physical problems, research about solving them is essential. This thesis showed that Runge-Kutta methods can be generalized not only for Irksome, but also for other finite element toolboxes like FEniCS.

List of Figures

Bibliography

[1] C. Karpfinger, *Höhere Mathematik in Rezepten*. Springer Spektrum, 2014 (cit. on pp. 1, 4).

[2] C. Runge, *Ueber die numerische Auflösung von Differentialgleichungen*. Math. Annalen, Band 46, S. 167–178, 1894 (cit. on pp. 1, 9).

[3] V. Barbu, *Differential Equations*. Springer, 2016 (cit. on p. 3).

[4] E. Miersemann, *Partial Differential Equations, Lecture Notes*. Department of Mathematics, Leipzig University, 2012 (cit. on p. 3).

[5] V. Jagota, A. P. Sethi, and K. Kumar, *Finite Element Method: An Overview*. Walailak Journal of Science & Technology, 2013 (cit. on p. 4).

[6] R. Courant, *Variational methods for the solution of problems of equilibrium and vibrations*. Bull. Am. Math. Soc., 1943 (cit. on p. 4).

[7] R. W. Clough, *The finite element method in plane stress analysis*. Proceedings of the second ASCE Conference on Electronic Computation, Pittsburgh, 1960 (cit. on p. 4).

[8] S. Hamdi, W. W. Schiesser, and G. W. Griffiths, *Method of lines*. Scholarpedia, 2(7):2859, 2007 (cit. on p. 5).

[9] N. Walet, *Partial Differential Equations*. University of Manchester, 2021 (cit. on p. 5).

[10] M. H. Holmes, *Introduction to Numerical Methods in Differential Equations*. Springer, 2007 (cit. on pp. 6, 21).

[11] H. P. Langtangen and A. Logg, *Solving PDEs in Python– The FEniCS Tutorial Volume I*. Springer, 2017 (cit. on pp. 7, 8, 16, 17, 22, 24, 25).

[12] M. Ostoja-Starzewski, *Ignaczak equation of elastodynamics*. Mathematics and Mechanics of Solids2019, Vol. 24(11) 3674–3713, 2018 (cit. on p. 7).

[13] J. Chung and G. Hulbert, *Time-integration of elastodynamics equation*. Numerical tours of Computational Mechanics using FEniCS, 1993 (cit. on pp. 7, 33).

[14] I. Faraǵo, *Note on the convergence of the implicit Euler method*. MTA-ELTE "Numerical Analysis and Large Networks" Research Group aźany P. s. 1/c, 1117 Budapest, Hungary, 2013 (cit. on p. 9).

[15] E. Hairer, *Runge–Kutta Methods, Explicit, Implicit*. Encyclopedia of Applied and Computational Mathematics (pp.1282-1285), 2015 (cit. on pp. 9–11).

[16] W. Kutta, *Beitrag zur näherungsweisen Integration totaler Differentialgleichungen*. Z. Math. Phys., Band 46, S. 435–453, 1901 (cit. on p. 9).

[17] J. C. Butcher, *Practical rungekutta methods for scientific computation*. ANZIAM J.50(2009), 333–342doi:10.1017/S1446181109000030, 2009 (cit. on p. 9).

[18] L. Euler, *Institutiones calculi integralis*. Institutiones calculi integralis, 1768 (cit. on p. 11).

[19] P. E. Farrell, R. C. Kirby, and J. Marchena-Menendez, *Irksome: Automating Runge–Kutta time-stepping for finite element methods.*, 2020 (cit. on pp. 12, 19, 22, 24, 25, 40, 41).

[20] J. Martín-Vaquero, *A 17th-order Radau IIA method for package RADAU. Applications in mechanical systems*. Computers Mathematics with Applications, Volume 59, Issue 8, Pages 2464-2472, 2010 (cit. on p. 13).

[21] M. Herrmann and M. Saravi, *A First Course in Ordinary Differential Equations*. Springer, 2014 (cit. on p. 13).

[22] J. Chung and G. Hulbert, *A time integration algorithm for structural dynamics with improved numerical dissipation: the generalized-alpha method*. J. Appl. Mech., 60 (2) (1993), pp. 371-375, 1993 (cit. on p. 14).

[23] P. Behnoudfar, Q. Deng, and V. M.Calo, *High-order generalized-alpha method*. Applications in Engineering Science Volume 4, December 2020, 100021, 2020 (cit. on pp. 14, 41).

[24] S. Erlicher, L. Bonaventura, and O. Burs, *The analysis of the Generalized-a method for non-linear dynamic problems*. Computational Mechanics, Springer Verlag, 2002, 28, pp.83-104.10.1007/s00466-001-0273-z, 2002 (cit. on p. 14).

[25] V. D. Nguyen, *High Performance Finite Element Methods*. KTH Royal Institute of Technology, 2018 (cit. on p. 16).

[26] D. N. Arnold and A. Logg, *Periodic table of the finite elements*. SIAM News, 2014 (cit. on p. 17).

[27] ——, *Unified Form Language Documentation*. FEniCS Project Revision, 2021 (cit. on p. 18).

[28] A. Logg, K.-A. Mardal, and G. Wells, *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012 (cit. on p. 18).

[29] F. Rathgeber, D. A. Ham, and L. M. et al, *Firedrake: automating the finite element method by composing abstractions*. ACM Transactions on Mathematical Software · January 2015, 2015 (cit. on p. 19).