Gryphon - a Module for Time Integration of Partial Differential Equations in FEniCS

Knut Erik Skare

Master of Science in Physics and Mathematics
Submission date: June 2012
Supervisor: Anne Kværnø, MATH

Norwegian University of Science and Technology
Department of Mathematical Sciences

Sammendrag

Denne oppgaven tar sikte på å implementere tidsintegratorer i FEniCS-rammeverket. Mer spesifikt går oppgaven ut på å velge egnede tidsintegratorer, implementere disse og verifisere at de virker ved å anvende dem på et utvalg relevante testproblemer. Dette arbeidet resulterte i en modul til FEniCS som fikk navnet Gryphon. Oppgaven er delt inn i fire deler.

Del I bygger et teoretisk rammeverk som motiverer hvorfor ESDIRK-metoder (Singly Diagonally Implicit Runge-Kutta method with an Explicit first stage) er gode løsere for systemer av stive ordinære differensialligninger (ODEer). Det vil også bli vist hvordan en ESDIRK metode kan brukes til å løse tidsavhengige partielle differensialligninger (PDEer) ved å løse det semi-diskretiserte systemet som oppnås ved å først anvende en endelig elementmetode. Vi vil begrense oss til PDEer som enten semi-diskretiseres til et rent ODE-system eller et differensial-algebraisk system (DAE) av indeks 1.

Del II tar for seg implementasjonen av Gryphon, med fokus på nytteverdi og kodestruktur.

Del III tar for seg numeriske eksperimenter på ESDIRK-metodene som er implementert i Gryphon. Eksperimentene vil etablere konvergens og gi kjøretidsresultater for ulike ESDIRK-metoder. Vi vil også se at L-stabilitet er en nyttig egenskap når en jobber med stive ligninger, ved å sammenligne en ESDIRK metode med trapesmetoden. Det blir også verifisert at skrittlengde-kontrollerne implementert i Gryphon oppfører seg som forventet. Som testproblemer vil vi se på varmeligningen, Fisher-Kolmogorov-ligningen, Gray-Scott-ligningene, Fitzhugh-Nagumo-ligningene og Cahn-Hilliard-ligningen.

Del IV er en brukermanual for Gryphon hvor alle parameterne brukeren kan endre vil bli forklart. Manualen inneholder også kode for å løse varmeligningen, Gray-Scott-ligningene og Cahn-Hilliard-ligningen, for å hjelpe leseren i gang med å løse egne problemer.

Abstract

This thesis aims to implement time integrators in the FEniCS framework. More specifically, the thesis focuses on selecting suitable time integrators, implement these and verify that the implementation works by applying them to various relevant test problems. This work resulted in a module for FEniCS, named Gryphon. The thesis is divided into four parts.

The first part builds a theoretical framework which will motivate why singly diagonally implicit Runge-Kutta methods with an explicit first stage (ESDIRKs) should be considered for solving stiff ordinary differential equations (ODEs). It will also be shown how an ESDIRK method can be utilized to solve time dependent partial differential equations (PDEs) by solving the semidiscretized system arising from first applying a finite element method. We will restrict our attention to PDEs which either give rise to a pure ODE system or a DAE (differential-algebraic equation) system of index 1.

The second part discusses the implementation of Gryphon, focusing on why such a module is useful and how the source code is structured.

The third part is devoted to numerical experiments on the ESDIRK solvers implemented in Gryphon. The experiments will establish convergence and give some run-time statistics for various ESDIRK schemes. We will also see that L-stability is a favorable trait when working with stiff equations, by comparing an ESDIRK method to the trapezoidal rule. It will also be verified that the step size selectors implemented in Gryphon behaves as expected. As test problems we consider the heat equation, the Fisher-Kolmogorov equation, the Gray-Scott equations, the Fitzhugh-Nagumo equations and the Cahn-Hilliard equations.

The fourth part is a user manual for Gryphon. All the parameters which can be changed by the user are explained. The manual also includes example code for solving the heat equation, the Gray-Scott equations and the Cahn-Hilliard equation, to get the reader starting on solving their own problems.

Preface

This thesis completes my master's degree in Industrial Mathematics at the Norwegian University of Science and Technology (NTNU). The work has been carried out at the Department of Mathematical Sciences during the spring semester of 2012 under the supervision of Anne Kværnø.

I would like to thank Anne for giving me an interesting assignment which allowed me to combine my passion for programming with a mathematical application, and for always having an open door to help and discussions.

I would also like to thank Anders Logg, Marie Rognes and Benjamin Kehlet at Simula Research Laboratory in Oslo for valuable input on my work and for giving me insight in how to use FEniCS.

Finally I would like to thank my friends and family for providing me with both moral support and distractions when needed. A special thanks goes to my class mate Olav Møyner for many a useful discussion on programming and numerics.

Knut Erik Skare
June 11, 2012
Trondheim

Contents

I Theoretical Background **6**

1.1 Introduction 7

1.2 Applying a Runge-Kutta method 7

1.3 Stiff Equations 8

1.4 Stability Properties 9

1.5 Classification of Runge-Kutta Methods 11

1.6 Runge-Kutta pairs 14

1.7 Adaptive Step Size Selection 15

1.8 Differential Algebraic Equations 19

1.9 Applying a Finite Element Method 20

II Implementation **23**

2.1 Introduction 24

2.2 The FEniCS Project 24

2.3 The Gryphon Module 26

2.3.1 The ESDIRK class . . . 29
2.4 A Code Example . . . 31

III Experiments 34

3.1 Introduction . . . 35
3.2 Test Cases . . . 35
3.2.1 Case 1: The Heat Equation . . . 36
3.2.2 Case 2: The Fisher-Kolmogorov Equation . . . 36
3.2.3 Case 3: The Gray-Scott Model . . . 38
3.2.4 Case 4: A FitzHugh-Nagumo Reaction-Diffusion model . . . 39
3.2.5 Case 5: The Cahn-Hilliard equation . . . 42
3.3 Verification of Convergence . . . 42
3.4 Verification of Constructive Step Size Selection . . . 46
3.5 Run Time Statistics . . . 46
3.6 Comparison to Trapezoidal Rule . . . 47

IV Gryphon User Manual 55

4.1 Introduction . . . 56
4.2 Handling Explicit Time Dependency . . . 56
4.3 Solver: ESDIRK . . . 56
4.3.1 Parameters . . . 57
4.3.2 Example Output . . . 62
4.4 Example Problems . . . 63

A Appended Code and Documentation 71

Part I

Theoretical Background

1.1 Introduction

The purpose of this part is to give an introduction to what a Runge-Kutta (RK) method is and how it can be applied to solve ordinary differential equations (ODEs) and index 1 differential algebraic equations (DAEs). It will also be shown how Runge-Kutta methods can be extended to solve partial differential equations (PDEs) by applying them to a semidiscretized system. Different classes of Runge-Kutta methods are presented alongside with their advantages and disadvantages related to computational cost, order and stability properties. This will build the theoretical framework which in turn will motivate why singly diagonally implicit Runge-Kutta methods with an explicit first stage (ESDIRKs) should be considered when working with stiff equations.

1.2 Applying a Runge-Kutta method

A Runge-Kutta method is a one-step time integration scheme which can be applied to approximate a system of ODEs. Given the following system,

$$\frac{\mathrm{d}}{\mathrm{d}t}y(t) = f(t,y), \qquad y(0) = y_0, \qquad y \in \mathbb{R}^m,$$

an s-stage Runge-Kutta method is applied by setting

$$\begin{aligned} y_{n+1} &= y_n + \Delta t \sum_{i=1}^{s} b_i \dot{Y}_i, \qquad \dot{Y}_i = f(t_n + c_i \Delta t, Y_i) \\ Y_i &= y_n + \Delta t \sum_{j=1}^{s} a_{ij} \dot{Y}_j, \qquad i = 1, \ldots, s, \end{aligned}$$

where a_{ij}, b_i and c_i are coefficients which define the method applied and Δt is the step size. It is common to characterize a Runge-Kutta method by a table called the Butcher tableau. The structure of such a tableau can be seen in table 1.1.

$$\begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \dots & a_{1s} \\ c_2 & a_{21} & a_{22} & \dots & a_{2s} \\ \vdots & \vdots & \vdots & & \vdots \\ c_s & a_{s1} & a_{s2} & \dots & a_{ss} \\ \hline & b_1 & b_2 & \dots & b_s \end{array}$$

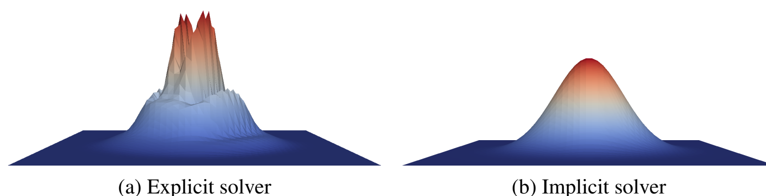
Table 1.1: A Butcher tableau used to characterize Runge-Kutta methods.

When deciding upon a Runge-Kutta method to solve a problem, we are first and foremost interested in efficiency. We want to solve our problem within reasonable time to a certain accuracy without using too much computing power. Depending on the nature of our problem, we may be forced to use computationally demanding implicit methods or we may get satisfactory results using relatively cheap, explicit methods.

1.3 Stiff Equations

To give a precise mathematical definition to the phenomena of stiff equations, has shown to be a difficult task. Instead it is helpful to talk about qualitative features which can help us decide whether or not our problem is stiff. In general, stiff problems are known to cause poor performance in explicit Runge-Kutta solvers, meaning that different solvers may give different answers or that the solution grows exponentially. The remedy for this is to use implicit solvers which tend to handle such problems a lot better. This behavior can be linked to stability properties which will be discussed in the next section.

Another source for stiffness comes from semidiscretizing a PDE with high order spatial derivatives. The stiffness of the problem is then related to the eigenvalues of the corresponding ODE/DAE system differing in several orders of magnitude as the spatial discretization becomes more refined.



(a) Explicit solver (b) Implicit solver

Figure 1.1: Solution of the heat equation.

A visual example of how a stiff problem behaves when subjected to an implicit method versus an explicit method, can be seen in figure 1.1. The problem solved is the heat equation with homogeneous Dirichlet boundary conditions and a simple Gauss pulse as initial condition. The implicit solver is able to maintain the smooth solution while the explicit one gets oscillations which eventually causes the solution to grow unbounded. It should be noted that the behavior in figure 1.1a takes place after just five time steps. After five more, the solution have diverged into meaningless data. For larger step sizes, the effect manifests quicker.

1.4 Stability Properties

The material for this section was found in [HW10, IV.3], where a more elaborate presentation on the following topics can be found.

When discussing stability properties of a Runge-Kutta method, it is useful to consider how it behaves applied to a linear test problem given as

$$\frac{\mathrm{d}}{\mathrm{d}t} y = \lambda y \tag{1.1}$$

where λ is often called the stiffness parameter. This equation is often referred to as the Dahlquist test equation, and any Runge-Kutta method (explicit or implicit) applied to it, can be written as

$$y_{n+1} = R(\lambda \Delta t) y_n, \qquad R(\lambda \Delta t) = \frac{P(\lambda \Delta t)}{Q(\lambda \Delta t)}, \qquad P, Q \in \mathbb{P}.$$

The function $R(\lambda \Delta t)$, commonly referred to as the stability function, can tell us a great deal about the stability properties of the method in question. It takes the form of a rational function defined in the complex plane and can be written as

$$R(z) = \frac{P(z)}{Q(z)} = 1 + z b^T (I - zA)^{-1} \mathbb{1}, \qquad z = \lambda \Delta t, \tag{1.2}$$

where A and b constitute the Butcher tableau and $\mathbb{1}$ is a vector of just ones. This stability function also satisfies the following expression

$$R(z) = \frac{\det(I - zA + z\mathbb{1}b^T)}{\det(I - zA)} \tag{1.3}$$

which is an easier expression to analyze than (1.2). The stability region, which is where our method is expected to produce stable solutions, is defined to be the areas where $|R(z)| < 1$. If this is a very narrow region and our problem is very stiff due to $\lambda \ll 0$, we may have to choose unreasonably small step sizes in order for our solution to remain stable. An example of this behavior can be seen by considering the explicit Euler method applied to (1.1). The stability function will in this case amount to

$$R(z) = 1 + z.$$

The stability region, where $|R(z)| < 1$, will amount to a circle of radius 1 centered in the point $(-1, 0)$ in the negative complex half plane. We now see that if we have a very large, negative value for λ, we have to choose a correspondingly small value for Δt in order to stay within the stability region since we get the bound

$$|R_{euler}(\lambda \Delta t)| < 1 \Longrightarrow 0 < \Delta t < -\frac{2}{\lambda}, \qquad \lambda < 0,$$

which is an increasingly small interval as $|\lambda|$ grows. Methods which do not suffer from this behavior are said to be A-stable, which are methods where $\mathbb{C}^-$ is contained in the stability region. This criterion is satisfied if and only if

$$|R(iy)| < 1, \qquad y \in \mathbb{R}$$

and

$$R(z) \text{ is analytic for } \mathrm{Re}(z) < 0.$$

A slightly weaker criterion than A-stability is $A(\alpha)$ stability. This concept arose from the observation that methods which are not completely A-stable, still may be decent methods. A method is said to be $A(\alpha)$ stable if

$$S_\alpha = \{z \text{ such that } |\arg(z)| \leq \alpha,\ z \neq 0\}$$

is contained in the stability region.

We may however still run into problems if we let z approach the real axis with a very large negative real value (very stiff problem). In this situation, $|R(z)|$ is very close to one which will cause bad convergence for our method due to the stiff parts being damped out very slowly. This motivates the concept of L-stability which is defined to be A-stable methods which also satisfy

$$\lim_{z \to \infty} R(z) = 0.$$

To further understand why L-stable methods are beneficial, we will quote some results from [HW10, IV.15]. A more sophisticated test equation than the Dahlquist equation (1.1), is the Prothero-Robinson equation given as

$$y' = \lambda(y - \gamma(x)) + \gamma'(x), \qquad y(x_0) = \gamma(x_0), \qquad \mathrm{Re}(\lambda) < 0, \tag{1.4}$$

with analytical solution $y(x) = \gamma(x)$. The constant λ is still the stiffness parameter. Applying a Runge-Kutta to (1.4) yields

$$Y_i = y_0 + \Delta t \sum_{j=1}^{s} a_{ij}[\lambda(Y_i - \gamma(x_0 + c_i\Delta t)) + \gamma'(x_0 + c_i\Delta t)], \tag{1.5}$$

$$y_1 = y_0 + \Delta t \sum_{i=1}^{n} b_i[\lambda(Y_i - \gamma(x_0 + c_i\Delta t)) + \gamma'(x_0 + c_i\Delta t)]. \tag{1.6}$$

By inserting the analytical solution for y, we get

$$\gamma(x_0 + c_i\Delta t) = \gamma(x_0) + \Delta t \sum_{j=1}^{s} a_{ij}\gamma'(x_0 + c_i\Delta t) + \Delta_{i,\Delta t}(x_0), \tag{1.7}$$

$$\gamma(x_0 + \Delta t) = \gamma(x_0) + \Delta t \sum_{i=1}^{s} b_i\gamma'(x_0 + c_i\Delta t) + \Delta_{0,\Delta t}(x_0), \tag{1.8}$$

where $\Delta_{i,\Delta t}(x_0), \Delta_{0,\Delta t}(x_0)$ are known as the numerical defect. By doing Taylor expansion of the above statements it can be shown that

$$\Delta_{0,\Delta t}(x_0) = \mathscr{O}(\Delta t^{p+1}), \qquad \Delta_{i,\Delta t}(x_0) = \mathscr{O}(\Delta t^{q+1}),$$

where p is the order and q is the stage order of the Runge-Kutta method in question. We can now express the error by taking the difference between (1.5)-(1.6) and (1.7)-(1.8):

$$y_1 - \gamma(x_0 + \Delta t) = R(z)(y_0 - \gamma(x_0)) - zb^T(I - zA)^{-1}\Delta_{\Delta t}(x_0) - \Delta_{0,\Delta t}(x_0)$$

where $z = \lambda\Delta t$, $R(z)$ is the stability function, $\Delta_{\Delta t}(x_0) = (\Delta_{1,\Delta t}(x_0), \ldots, \Delta_{s,\Delta t}(x_0))$. If we replace x_0 with x_n we get the recursion

$$y_{n+1} - \gamma(x_{n+1}) = R(z)(y_n - \gamma(x_n)) + \delta_{\Delta t}(x_n) \tag{1.9}$$

where

$$\delta_{\Delta t}(x_n) = -zb^T(I - zA)^{-1}\Delta_{\Delta t}(x_n) - \Delta_{0,\Delta t}(x_n).$$

While we can not control the latter term of (1.9), the first term can be controlled by imposing L-stability ($R(\infty) = 0$), making the term vanish asymptotically.

1.5 Classification of Runge-Kutta Methods

This section will present some classes of Runge-Kutta methods. We will focus on which problems the methods can be applied to versus computational complexity. The material regarding DIRK/SDIRK methods was found in [KNO96].

Explicit Runge-Kutta methods (ERKs)

An explicit Runge-Kutta method is best characterized by the Butcher tableau being lower triangular and thus take the form found in table 1.2. These methods have very low computational cost since all the stage values can be expressed explicitly. To perform a time step we only have to evaluate the right hand side in different points, rather than having to solve linear/nonlinear equations. The drawback of using explicit RK methods can be seen by considering the stability function. It takes the form of a polynomial,

$$R(z) = P(z) = 1 + \mathscr{O}(z),$$

implying that these methods can never be A-stable. From a practical viewpoint, this is enough to classify explicit Runge-Kutta methods as poor candidates when working with stiff equations, and the methods should not be expected to perform well if extended to PDE solvers. There do however exist techniques where explicit methods can be used to give satisfactory results. See [EJL03] for an example.

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

Table 1.2: Butcher tableau for an explicit Runge-Kutta method.

Implicit Runge-Kutta methods (IRKs)

An implicit Runge-Kutta method has a Butcher tableau for which one or more stage value is implicitly dependent on any of the other. In the extreme case of a full implicit Runge-Kutta method (FIRK), all the stage values are implicitly dependent on all the other, meaning that performing one time step will involve solving s coupled systems of equations.

These methods can however be constructed to incorporate stability properties like A- and L-stability, making them able to handle stiff problems.

Diagonally Implicit Runge-Kutta methods

A subclass of the implicit Runge-Kutta methods are the diagonally implicit Runge-Kutta methods, or DIRKs. The Butcher tableau for this class of methods takes the form found in table 1.3, where at least one of the diagonal elements a_{ii} must be nonzero. Compu-

$$
\begin{array}{c|cccc}
c_1 & a_{11} & & & \\
c_2 & a_{21} & a_{22} & & \\
\vdots & \vdots & \vdots & \ddots & \\
c_s & a_{s1} & a_{s2} & \dots & a_{ss} \\
\hline
 & b_1 & b_2 & \dots & b_s
\end{array}
$$

Table 1.3: Butcher tableau for a diagonally implicit Runge-Kutta method.

tationally speaking, these methods are less demanding than a full implicit Runge-Kutta method since each stage is only dependent on itself and previous stages, if we start by solving from the top.

Singly Diagonally Implicit Runge-Kutta methods

A subclass of DIRK methods known as SDIRK methods (Singly Diagonally Implicit Runge-Kutta methods) have the property that

$$a_{ii} = \gamma, \quad i = 1, \dots, s.$$

These methods have a significant computational advantage over DIRK methods since the Newton matrix will be the same for all stages. This can be seen by realizing that solving for each stage value will be a problem on the form

$$Y_i - \Delta t \gamma \dot{Y}_i = y_n + \Delta t \sum_{i=1}^{i-1} a_{ij} \dot{Y}_i,$$

for which the Newton matrix is

$$I - \Delta t \gamma f_y,$$

where f is the right hand side of your problem. Successful realizations of SDIRK methods include SIMPLE by Nørsett and Thomsen [NT84] as well as SDIRK4 by Hairer and Wanner [HW10].

The main restrictions on SDIRK methods is their relatively low order (at most order $s+1$ for A-stable methods) as well as suffering from order reduction when applied to very stiff problems.

Singly Diagonally Implicit Runge-Kutta methods with an Explicit First Stage

A singly diagonally implicit Runge-Kutta method with an explicit first stage (ESDIRK) has the same format as an SDIRK method with the exception that the first row in the Butcher tableau is equal to zero. The structure of this table can be found in table 1.4. This class of methods have been studied by Anne Kværnø [Kvæ04], and in her article several ESDIRK methods are developed and presented as adaptive Runge-Kutta pairs of order p and $p-1$. Each pair gives rise to two different methods depending on whether or not local extrapolation is used.

To differentiate between the advancing method and the error estimating method, the hat-symbol will be used to mark the error estimating methods, i.e. $R(x)$ is the stability function of the advancing method while $\hat{R}(x)$ is the stability function of the error estimating method.

The methods presented in [Kvæ04] are constructed according to the following criteria:

$$
\begin{array}{c|cccccccc}
0 & 0 & & & & & & & \\
c_2 & a_{21} & \gamma & & & & & & \\
c_3 & a_{31} & a_{32} & \gamma & & & & & \\
\vdots & \vdots & & & \ddots & & & & \\
\vdots & \vdots & & & & \ddots & & & \\
c_{s-2} & a_{s-2,1} & a_{s-2,2} & a_{s-2,3} & \cdots & \cdots & \gamma & & \\
1 & a_{s-1,1} & a_{s-1,2} & a_{s-1,3} & \cdots & \cdots & a_{s-1,s-2} & \gamma & \\
1 & a_{s1} & a_{s2} & a_{s3} & \cdots & \cdots & a_{s,s-2} & a_{s,s-1} & \gamma.
\end{array}
$$

Table 1.4: Butcher tableau for a singly diagonally implicit Runge-Kutta method with an explicit first stage.

1. Stiff accuracy in both the advancing and the error estimating methods.
2. $R(\infty) = 0$, and $|\hat{R}(\infty)|$ as small as possible, at least less than one.
3. A-stability, or at least $A(\alpha)$ stability for both methods.
4. As high stage order as possible.

Stiff accuracy in the advancing method is a well known remedy for the problem of order reduction SDIRK methods may experience. It is however not as common to incorporate stiff accuracy in the error estimator, causing the order of the error estimate to be lower than expected. In this sense, the methods developed by Kværnø stands out by requiring stiff accuracy in the error estimator as well.

A-stability together with the requirement $R(\infty) = 0$ makes the methods L-stable which is beneficial when working with very stiff problems. Computationally speaking we naturally inherit all the benefits of SDIRK methods in addition to the local error estimate being reduced to

$$le = Y_{s-1} - Y_s.$$

1.6 Runge-Kutta pairs

We will now discuss two strategies for selecting step size when integrating ODE/DAE systems over a given time interval. The simplest strategy is to specify a fixed step size which the program will use until it reaches the end of the domain. This can be said to be a reasonable approach if the behavior of your problem in the desired time interval is

relatively uniform. If this is not the case, allowing your method to change time steps adaptively becomes advantageous. We can save computational power by doing large time steps when the solution is slowly varying and use that to take smaller time steps when the solution is varying more rapidly.

In order to determine the size of the next time step, the program needs an estimate for the local error in the current time step. The user can then specify a tolerance which the new step size is adjusted according to. To allow for such functionality, Runge-Kutta methods are often presented in pairs of order p and $p-1$. The idea is that one method is used as an advancing method while the other one is used as a reference to measure the error against. If the method of highest order is used to advance the solution, we do what is known as local extrapolation. The Butcher tableau for a Runge-Kutta pair is written as

$$\begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \dots & a_{1s} \\ \vdots & \vdots & \vdots & & \vdots \\ c_s & a_{s1} & a_{s2} & \dots & a_{ss} \\ \hline & b_1 & b_2 & \dots & b_s \\ & b_1^* & b_2^* & \dots & b_s^* \end{array},$$

where b_i refers to the coefficients of the advancing method and b_i^* refers to the coefficients of the error measuring method. The local error (denoted by le) can then be estimated by

$$le = y_n - y_n^* = \Delta t \sum_{i=1}^{s} (b_i - b_i^*) f(Y_i). \tag{1.10}$$

A special case for this estimate occurs when both the advancing method and the error estimating method are stiffly accurate. The estimate for the local error (1.10) simply reduce to

$$le = y_n - y_n^* = \Delta t \sum_{i=1}^{s} (a_{s,i} - a_{s-1,i}) f(Y_i) = Y_s - Y_{s-1}.$$

1.7 Adaptive Step Size Selection

This section will consider adaptivity in the framework of adaptive Runge-Kutta pairs. Two different step size selectors will be discussed.

Consider a Runge-Kutta method with advancing method of order p applied to a problem with step size Δt. Asymptotic theory sates that the local error in that step is approxi-

mately equal to

$$le_n \approx \varphi_n \cdot \Delta t_n^p \tag{1.11}$$

where φ_n is some unknown quantity. It is then common to make the prediction[1] that

$$\hat{\varphi}_n = \varphi_{n-1},$$

that is, the distribution in φ_n is varying slowly over the course of the time stepping. If we now want to select a new step size corresponding to a user specified tolerance tol, we get the same relation,

$$tol \approx \varphi_{n+1} \cdot \Delta t_{n+1}^p. \tag{1.12}$$

Since we assume that φ_n is constant, we can divide (1.12) by (1.11) and get

$$\frac{tol}{le_n} \approx \left(\frac{\Delta t_{n+1}}{\Delta t_n}\right)^p \Rightarrow \Delta t_{n+1} \approx \Delta t_n \cdot \left(\frac{tol}{le_n}\right)^{1/p}. \tag{1.13}$$

To compensate for this oversimplified model, it is common to include what is called a *pessimistic factor* (denoted $\mathscr{P}$) in (1.13) in the following way

$$\Delta t_{n+1} = \mathscr{P} \cdot \Delta t_n \left(\frac{tol}{le_n}\right)^{1/p}, \qquad \mathscr{P} \in [0,1]. \tag{1.14}$$

This factor, which can be viewed as a measure on how trustworthy we consider the step size selector to be, must be adjusted according to experimental results on a particular problem. It is common to start with $\mathscr{P} = 0.8$, but if the program still rejects/accepts a lot of steps it should be decreased/increased accordingly. A nice feature by selecting step sizes in this way, is that the step size is automatically increased/decreased if the local error is smaller/greater than the specified tolerance.

There are however cases where the assumption $\varphi_n \approx \varphi_{n-1}$ can be said to be a poor approximation. In his article, Kjell Gustafsson [Gus94], who has studied control-theoretic techniques for step size selection, lists the following examples where the assumption may fail:

- The properties of the differential equation change along the solution.
- The step size is nonzero during the integration and is not necessarily the lowest-order term that dominates in the error expression. Consequently the error may behave as if p is larger than expected in (1.11).

[1] Note that predicted values are denoted with a hat-symbol, so that $\hat{\varphi}$ means the predicted value for φ.

- Some implicit methods lose convergence order when applied to stiff problems (order reduction), causing p in (1.11) to be smaller than expected.

To capture the variation in φ, Gustafsson uses a model which assumes a linear trend in $\log\varphi$:

$$\log\hat{\varphi}_n = \log\varphi_{n-1} + \nabla\log\varphi_{n-1}, \qquad \nabla\log\varphi_{n-1} = \log\varphi_{n-1} - \log\varphi_{n-2}.$$

The expression for $\log\hat{\varphi}_n$ can be rewritten in the following way

$$\log\hat{\varphi}_n = (2 - q^{-1})\log\varphi_{n-1}$$

where q is the forward shift operator[2]. By taking the logarithm of (1.11) and considering a step size satisfying $le_n = tol$, we get

$$\log\Delta t_n = \frac{1}{p}(\log tol - \log\hat{\varphi}_n)$$

By inserting the expression for $\log\hat{\varphi}_n$ we get

$$\log\Delta t_n = \frac{1}{p}(2 - q^{-1})(\log tol - \log\varphi_{n-1}), \tag{1.15}$$

where we have used that tol is constant (and thus unaffected by the shift operator). Finally we have that

$$\log\varphi_{n-1} = \log le_{n-1} - p\log\Delta t_{n-1},$$

which inserted in (1.15) gives

$$\log\Delta t_n = \frac{1}{p}(2 - q^{-1})(\log tol - \log le_{n-1} + p\log\Delta t_{n-1})$$

By applying the shift operators and multiplying out the parenthesis, we arrive at

$$\log\Delta t_n = \frac{1}{p}(\log tol - 2\log le_{n-1} + \log le_{n-2}) + 2\log\Delta t_{n-2} - \log\Delta t_{n-1},$$

which, by removing the logarithms, transforms into

$$\Delta t_n = \frac{\Delta t_{n-1}}{\Delta t_{n-2}}\left(\frac{tol \cdot le_{n-2}}{le_{n-1}^2}\right)^{1/p}\Delta t_{n-1},$$

[2]The forward shift operator is defined as $q(\varphi_{n-1}) = \varphi_n$.

which is a step size controller based on the two most recent values of le and Δt. Before describing the actual implementation of the step size controller, we first have to consider what to do when we get rejected steps. If a step is rejected, we have to restart the step size selector since it requires information from two consecutive successful steps. This is done by using the standard asymptotic step size selector (1.14). In the case of two or more consecutive rejects, it is possible that the value of p is lower than expected (so called order reduction). When this occurs, we can exploit the fact that we have several measurements in the same timestep, which satisfy $\varphi_n = \varphi_{n-1}$, to form the following prediction for p:

$$\frac{le_n}{le_{n-1}} = \left(\frac{\Delta t_n}{\Delta t_{n-1}}\right)^p \Rightarrow \hat{p} = \frac{\log(le_n/le_{n-1})}{\log(\Delta t_n/\Delta t_{n-1})}.$$

For robustness, Gustafsson suggests that predictions outside the range $[0.1, p]$ should be rejected. We now have all the components needed to present an outline of the complete step size algorithm which can be found in Algorithm 1.

Algorithm 1 Gustafsson step size selector

1: **if** current step is accepted **then**
2:     **if** first step **or** fist after consecutive rejects **or** restricted timestep **then**
3:         $\Delta t_{n+1} \leftarrow \left(\frac{tol}{le_n}\right)^{1/p} \Delta t_n$ ▷ Asymptotic step size selector to restart.
4:     **else**
5:         $\Delta t_{n+1} \leftarrow \frac{\Delta t_n}{\Delta t_{n-1}} \left(\frac{tol \cdot le_{n-1}}{le_n^2}\right) \Delta t_n$ ▷ Gustafsson step size selector.
6:     **end if**
7: **else**
8:     **if** consecutive rejects **then**
9:         $\hat{p} \leftarrow \frac{\log(le_n/le_{n-1})}{\log(\Delta t_n/\Delta t_{n-1})}$
10:         $\hat{p} \leftarrow \texttt{Restrict}(\hat{p})$
11:         $\Delta t_{n+1} \leftarrow \left(\frac{tol}{le_n}\right)^{1/\hat{p}} \Delta t_n$
12:     **else**
13:         $\Delta t_{n+1} \leftarrow \left(\frac{tol}{le_n}\right)^{1/p} \Delta t_n$
14:     **end if**
15: **end if**
16: $\Delta t_{n+1} \leftarrow \texttt{Restrict}(\Delta t_{n+1})$

The `Restrict`-method should be designed to cap predictions of $\hat{p}$ to the interval $[0.1, p]$ and to make sure that the timestep Δt_{n+1} is not increased/decreased too much. If a

timestep is restricted, the algorithm should regard this as a restart.

1.8 Differential Algebraic Equations

Consider the following system of implicit differential equations:

$$F\left(\frac{\mathrm{d}}{\mathrm{d}t}y, y, t\right) = F(y', y, t) = 0. \tag{1.16}$$

If the Jacobian $\frac{\partial F}{\partial y'}$ is nonsingular, we can, by the implicit function theorem, solve (1.16) for y' and get a system of ordinary differential equations on the form

$$y' = F(y, t).$$

If this is not the case, the function y will have to satisfy some algebraic constraints and we have what is called a *differential algebraic equation* (DAE). In order to measure how far a DAE system is from being an ODE system, we can assign the system an *index*. It is however not a trivial task to come up with a single definition on what this index should be, different definitions are suitable for different problems. For our purpose, we will consider the definition given by Brenan, Campbell and Petzold [BCP89, Chapter 2.2] known as the *differentiation index*:

Definition 1 *The minimum number of times that all or the part of the system*

$$F(y', y, t) = 0$$

must be differentiated with respect to t in order to determine y' as a continuous function of y, t, is the differentiation index of $F(y', y, t)$.

When solving a DAE system, selecting initial conditions is not quite as straightforward as when solving an ODE system. For a well posed system of ODEs, a set of initial conditions uniquely determines a solution. For a general high index DAE, on the other hand, finding a set of consistent initial conditions may be very hard due to (potentially numerous) complicated algebraic constraints.

In her article, Kværnø [Kvæ04] states that the ESDIRK methods (presented in section 1.5 of this document) can be directly applied to semi-explicit systems of index 1. This will amount to the following system,

$$\begin{aligned} y' &= f(y, z), \\ 0 &= g(y, z), \end{aligned} \tag{1.17}$$

where we assume g_z to be nonsingular. We now show that this system indeed has index 1 according to Definition 1. Differentiating the second component of (1.17), with respect to t, yields

$$\frac{\partial g(y,z)}{\partial t} = g_y y'(y,z) + g_z z'(y,z) = g_y f(y,z) + g_z z'(y,z).$$

We can now solve for z' and formulate the following pure ODE problem:

$$\begin{aligned} y' &= f(y,z), \\ z' &= -g_z^{-1}(g_y f)(x,y). \end{aligned}$$

Since we needed to carry out one differentiation, we say that this system has index 1. By the same definition, a pure ODE system has index 0.

When solving a semi-explicit system on the form (1.17), we could, since g_z^{-1} is assumed to be nonsingular, solve $g(y,z)$ for z and insert into the first equation to get the pure ODE-system

$$y' = f(y, G(y)), \qquad z = G(y).$$

This is mathematically equivalent to applying a Runge-Kutta method in the following way:

$$\begin{aligned} Y_i &= y_n + \Delta t \sum_{j=1}^{s} a_{ij} f(Y_i, Z_i), & 0 &= g(Y_i, Z_i), \\ y_{n+1} &= y_n + \Delta t \sum_{i=1}^{s} b_i f(Y_i, Z_i), & 0 &= g(y_{n+1}, z_{n+1}). \end{aligned}$$

for $i = 1, \ldots, s$. It is possible to avoid the step of solving for z_{n+1} by applying a stiffly accurate Runge-Kutta method for which

$$y_{n+1} = Y_s \Rightarrow z_{n+1} = Z_s.$$

1.9 Applying a Finite Element Method

We will now show how ESDIRK methods can be extended to solve PDEs. Our strategy will be to use the ESDIRK method to handle time dependencies, and a finite element method to handle the spatial dependencies of our problem.

Say that we want to solve a PDE given as

$$u(x,y) - \nabla^2 u(x,y) - f(u(x,y)) = 0, \qquad (x,y) \in \Omega, \tag{1.18}$$
$$u(x,y) = 0, \qquad (x,y) \in \partial\Omega,$$

where the function f is assumed to be nonlinear. This way of formulating a PDE problem is known as a strong formulation, where we require that our solution u satisfies (1.18) in every point (x,y), and that it belongs to the space

$$H_0^2(\Omega) = \left\{ v : \int_\Omega v^2 \mathrm{d}\Omega < \infty, \ \int_\Omega |\nabla v^2| \mathrm{d}\Omega < \infty, \ \int_\Omega |\nabla^2 v^2| \mathrm{d}\Omega < \infty, \ v|_{\partial\Omega} = 0 \right\}.$$

We can relax the regularity conditions by multiplying (1.18) by a function v coming from some function space $\hat{V}$ to get

$$uv - \nabla^2 uv - f(u)v = 0.$$

If we now integrate this equation over the domain Ω, and do integration by parts on the diffusion term, we end up with

$$\int_\Omega uv \mathrm{d}\Omega + \int_\Omega \nabla u \nabla v \mathrm{d}\Omega - \int_\Omega f(u) v \mathrm{d}\Omega = 0. \tag{1.19}$$

We now define $u \in V$ to be a weak solution of (1.18) if it satisfies (1.19) for all $v \in \hat{V}$. This is imposed for all $v \in \hat{V}$ since our choice of v is completely arbitrary. The space V is known as the trial space and is the space where the weak solution is located, while the space $\hat{V}$ is known as the test space. The functions u and v are thus referred to as trial and test functions. In our example the two spaces coincides with the space $H_0^1(\Omega)$ due to the homogeneous Dirichlet boundary conditions. In general, the space V is the space of functions with the required regularity which also satisfies the Dirichlet boundary conditions of our problem. A finite element method seeks to approximate the weak solution of (1.18) by constructing the finite dimensional subspaces $V_h \subset V$ and $\hat{V}_h \subset \hat{V}$ and use those to construct an approximation to (1.19). In our example, the two spaces are the same and will be referred to as just V_h. Let now V_h be defined as

$$V_h = \mathrm{span}\{\varphi_1, \varphi_2, \ldots, \varphi_m\},$$

where all the functions φ_i are linearly independent functions selected from V. Which functions we choose to span this subspace depends on which kind of element we want to use. Our numerical approximation to the solution u, denoted by u_h, can can be written as a linear combination of the functions spanning V_h as such

$$u_h = \sum_{i=1}^{m} U_i \varphi_i, \qquad \varphi_i \in V_h,$$

where $U = [U_1, U_2, \ldots, U_m]$ are the degrees of freedom we need to solve for. If we insert this approximation into (1.19) we get

$$\sum_{i=1}^{m} \left(U_i \int_{\Omega} [\varphi_i \hat{\varphi}_j + \nabla \varphi_i \cdot \nabla \hat{\varphi}_j] \mathrm{d}\Omega \right) - \int_{\Omega} f \left(\sum_{i=1}^{m} U_i \varphi_i \right) \hat{\varphi}_j \mathrm{d}\Omega = 0, \qquad j = 1, \ldots, m,$$

which is a system of nonlinear equations in U since f is assumed to be nonlinear. By solving this system, we can construct our approximation $u_h = \sum_{i=1}^{m} U_i \varphi_i$.

We now show how applying a finite element method to a PDE can give rise to a system of DAEs. Consider the Cahn-Hilliard system (which will be studied in some detail in the experiments chapter) with $w = w(x, y)$ and $z = z(x, y)$:

$$\begin{aligned} \frac{\mathrm{d}w}{\mathrm{d}t} &= \nabla^2 z, \\ 0 &= z - f(w) - \nabla^2 w. \end{aligned}$$

The weak formulation of this problem amounts to finding $w \times z \in V \times V$ such that

$$\begin{aligned} \int_{\Omega} \frac{\mathrm{d}w}{\mathrm{d}t} v \mathrm{d}\Omega &= -\int_{\Omega} \nabla z \nabla v \mathrm{d}\Omega, \\ 0 &= \int_{\Omega} (z - f(w)) q \mathrm{d}\Omega + \int_{\Omega} \nabla w \nabla q \mathrm{d}\Omega, \end{aligned}$$

for all $v \times q \in \hat{V} \times \hat{V}$ where $V, \hat{V}$ are appropriate finite element spaces. By applying a finite element method we seek the approximations $w_h \times z_h$ such that

$$\begin{aligned} \int_{\Omega} \frac{\mathrm{d}w_h}{\mathrm{d}t} v \mathrm{d}\Omega &= -\int_{\Omega} \nabla z_h \nabla v \mathrm{d}\Omega, \\ 0 &= \int_{\Omega} (z_h - f(w_h)) q \mathrm{d}\Omega + \int_{\Omega} \nabla w_h \nabla q \mathrm{d}\Omega, \end{aligned}$$

for all $v \times q \in \hat{V}_h \times \hat{V}_h$ where $\hat{V}_h \subset V_h$ and $V_h \subset V$. By inserting

$$w_h = \sum_{i=1}^{m} W_i \varphi_i, \qquad z_h = \sum_{i=1}^{m} Z_i \varphi_i, \qquad q = v = \hat{\varphi} \in \hat{V},$$

we get the following system of equations:

$$\sum_{i=1}^{m} \int_{\Omega} \frac{\mathrm{d}W_i}{\mathrm{d}t} \varphi_i \hat{\varphi}_j \mathrm{d}\Omega = \sum_{i=1}^{m} Z_i \int_{\Omega} \nabla \varphi_i \cdot \nabla \hat{\varphi}_j \mathrm{d}\Omega,$$

$$0 = \sum_{i=1}^{m} \left(Z_i \int_{\Omega} \varphi_i \hat{\varphi}_j \mathrm{d}\Omega \right) + \int_{\Omega} f \left(\sum_{i=1}^{m} W_i \varphi_i \right) \hat{\varphi}_j \mathrm{d}\Omega + \sum_{i=1}^{m} \left(W_i \int_{\Omega} \nabla \varphi_i \cdot \nabla \hat{\varphi}_j \mathrm{d}\Omega \right),$$

for $j = 1, \ldots, m$. This is now a DAE system of index 1 (with W_i as ODE-components and Z_i as algebraic components) which can be solved directly by applying one of the ESDIRK methods described in section 1.5.

Part II

Implementation

2.1 Introduction

This part aims to give a presentation of my work resulting in the Gryphon module. First, a short introduction to FEniCS with special focus on the components relevant for the development of Gryphon, will be given. Next, the need for a module like Gryphon will be motivated through an example. The focus will be on how the work flow of solving a time dependent PDE with FEniCS compares to the work flow of solving the same problem with FEniCS and Gryphon. This part will not go into details regarding the source code, instead we will focus on how I wanted Gryphon to work and how I used object orientation to achieve a clean program architecture. For details on the source code, the reader is encouraged to consult the attached documentation (see appendix A for details).

2.2 The FEniCS Project

Before discussing implementation details concerning Gryphon, we will give a short introduction to underlying framework, namely the FEniCS project. A brief summary of the project can be found in the "about" section on the FEniCS web page[3] where it is stated that:

> *The FEniCS Project is a collaborative project for the development of innovative concepts and tools for automated scientific computing, with a particular focus on automated solution of differential equations by finite element methods.*

We will now dive into some key features in FEniCS in order to build some terminology which will be useful later. For a more complete coverage of the topics presented, the reader is encouraged to download the FEniCS book [LMW11] from `http://fenicsproject.org/book/`.

UFL (Unified Form Language) UFL is the language used to specify problems in FEniCS. It is described in the following way by its authors on the UFL web page[4]:

> *The Unified Form Language (UFL) is a domain specific language for declaration of finite element discretizations of variational forms. More precisely,*

[3] `http://fenicsproject.org/about`
[4] `http://launchpad.net/ufl`

it defines a flexible interface for choosing finite element spaces and defining expressions for weak forms in a notation close to mathematical notation.

The elegance of this language is best shown with an example. Say that we would like to solve the Laplace equation given as

$$\nabla^2 u = 0$$

on some domain Ω. The weak form of this equation amounts to finding $u \in V$ such that

$$a(u, v) = 0,$$
$$-\int_{\Omega} \nabla u \nabla v \, d\Omega = 0,$$

for all $v \in \hat{V}$ for appropriate trial/test spaces $V, \hat{V}$. This problem can be described by the following UFL code where we have used first order Lagrange elements as basis functions on a 2D domain:

```
element = FiniteElement("Lagrange", triangle , 1)

v = TestFunction(element)
u = TrialFunction(element)

a = -inner(grad(u)*grad(v))*dx
```

Listing 2.1: UFL for code defining the Laplace equation.

As this code snippet shows, UFL indeed provides a very clean and intuitive way of defining weak forms by using familiar names like `inner` for inner product and `grad` for the gradient operator. UFL also supports symbolic differentiation which is a tremendous advantage when working with nonlinear PDEs, since we are no longer required to estimate the Jacobian matrix used when doing Newton iterations.

FFC (The FEniCS Form Compiler) and UFC (Unified Form-assembly Code) The FEniCS Form Compiler is responsible for interpreting UFL and translate it into UFC. Simply put, UFC is C++ code responsible for assembling the systems of equations described in UFL. The way FFC is used depends on which FEniCS API you are working with.

If you are working with the C++-API, the common approach is to define your problem in UFL, send the UFL-file to FFC to get a UFC-object, and then include the header file of the UFC-object in your C++ program.

If you are working with the Python-API, the UFL/UFC handling is more seamless. UFL can be imported as a module in Python, allowing users to create and manipulate UFL

forms while the script is running. The UFC generation can be postponed until it is specifically requested by a FEniCS component, like for instance a linear/nonlinear solver. This is made possible by a module called *Instant*, which make FFC support just-in-time (JIT) compilation. Instant also supports caching so that already generated UFC code can be reused if possible.

For details on the concepts presented above, the reader is encouraged to consult [LMW11] chapter 11 for FFC, chapter 14 for Instant, chapter 16 for UFC and chapter 17 for UFL.

2.3 The Gryphon Module

We will start by motivating why a module like Gryphon is useful when working with time dependent PDEs. This will be done by showing how the work flow for solving a time dependent in FEniCS, without Gryphon, can be improved upon. As working example we will consider the Heat equation with a source term given as

$$\begin{aligned} \frac{\partial u}{\partial t} &= \nabla^2 u + (\beta - 2 - 2\alpha), \quad && u \in \Omega, \\ u &= 1 + x^2 + y^2 + \beta t, && u \in \partial\Omega, \\ u &= 1 + x^2 + y^2, && t = 0, \end{aligned} \tag{2.20}$$

on the domain $\Omega = (x,y) \in [0,1] \times [0,1]$ with $\alpha = 3$ and $\beta = 1.2$ with $u = u(x,y)$. This example was found in the FEniCS tutorial in the section "time-dependent problems" [Pro12b]. Because of the time derivative, we can not solve the problem in FEniCS in its current form, we have to apply some sort of time integrator. In the FEniCS tutorial, the backward Euler method is applied, giving rise to the new problem:

$$\begin{aligned} \frac{u_n - u_{n-1}}{\Delta t} &= \nabla^2 u_n + (\beta - 2 - 2\alpha), \quad && u_n \in \Omega, \\ u_n &= 1 + x^2 + y^2 + \beta t, && u_n \in \partial\Omega, \\ u_0 &= 1 + x^2 + y^2, \end{aligned}$$

where Δt is the step size in time and u_n refers to the numerical solution in time step n. Rearranging the first expression yields

$$u_n - \Delta t \nabla^2 u_n = u_{n-1} + \Delta t(\beta - 2 - 2\alpha).$$

The linear variational problem corresponding to each time step amounts to finding $u_n \in V$ such that

$$a(u_n, v) = \ell(v),$$
$$\int_\Omega (u_n v + \Delta t \nabla u_n \nabla v) \mathrm{d}\Omega = \int_\Omega (u_{n-1} + \Delta t(\beta - 2 - 2\alpha)) \mathrm{d}\Omega,$$

for all $v \in \hat{V}$ where $V, \hat{V}$ are appropriate test/trial spaces.

The problem is now in a form suitable for FEniCS, but we still have to write additional code for doing the actual time stepping process. In the simplest case, this can be achieved by a while-loop which solves the variational problem for each time step and updates the solution $u_{n+1} \leftarrow u_n$. If we want to utilize more advanced tools for the time stepping, like for instance adaptive step size control, the required code becomes considerably more involved.

Additional complications arise if we want to use a more advanced time integrator than the backward Euler method. Reformulating your initial problem to a form suitable for FEniCS, and then writing the code for solving each time step, may not be straightforward. If you are experimenting with different equations this work flow will most likely lead to a lot of boilerplate code which is both tedious and hard to maintain.

In the end we have distanced ourselves from the actual problem we were trying to solve in the first place. Gryphon aims to bridge this gap by the following philosophy:

> *Just as FEniCS provides an easy and intuitive way of defining and solving differential equations, Gryphon should provide an easy and intuitive way of applying and customizing a time integrator to solve a time dependent PDE.*

In other words, I wanted Gryphon to be a tool which captured the feel of working in a FEniCS environment, where the main focus is on the problem you are trying to solve. This goal gave rise to the work flow presented in figure 2.2, where the already described work flow for solving a time dependent PDE in FEniCS is compared to solving the same problem in FEniCS with Gryphon. Gryphon adapts the already existing parameter system from FEniCS, allowing users to familiarize themselves with the available parameters by calling

```
info(Gryphon_object.parameters,verbose=True)
```

All the available parameters are explained in the user manual found in part IV.

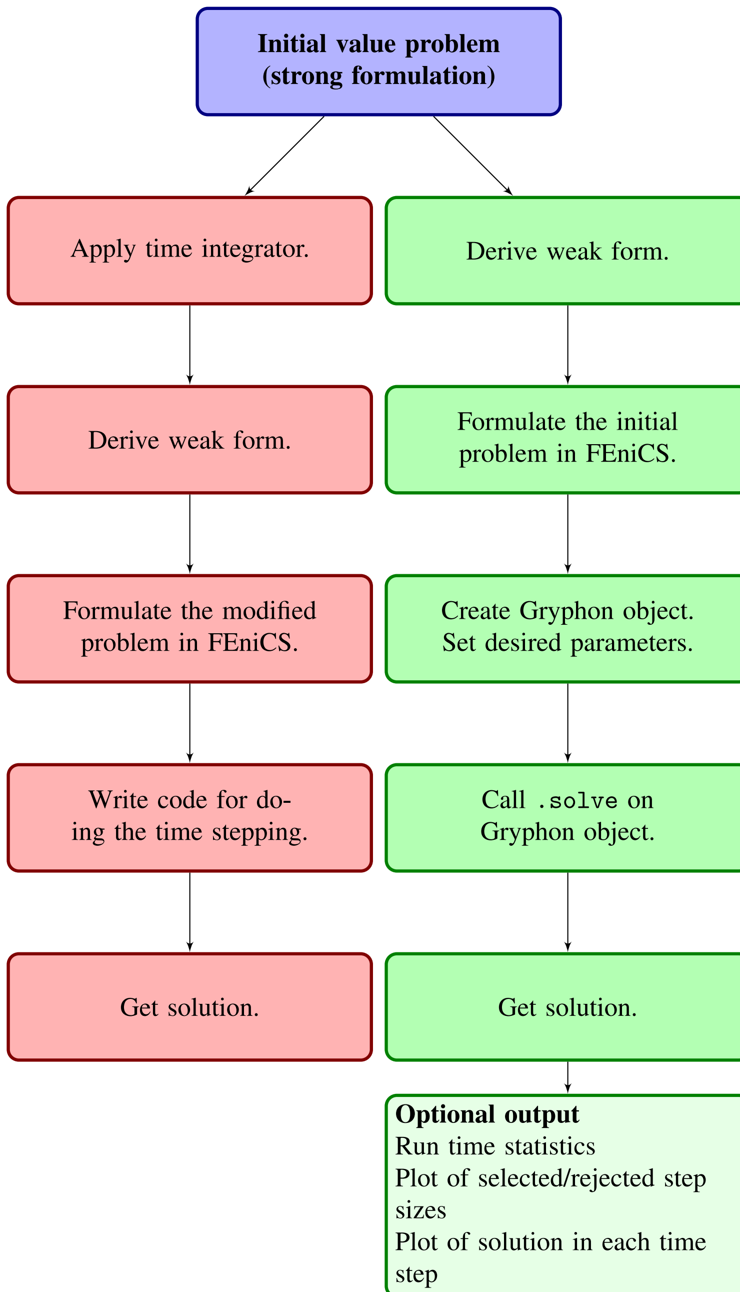


Figure 2.2: Work flow for solving a time dependent PDE in FEniCS without Gryphon (red) and with Gryphon (green).

To give a better understanding on how Gryphon works, we will give a short summary of the program architecture. When developing Gryphon, I sought to follow the don't repeat yourself (DRY) principle, which is a principle formulated by Andy Hunt and Dave Thomas stating that *"Every piece of knowledge must have a single, unambiguous, authoritative representation within a system"* [HT99]. This gave rise to the class hierarchy found in figure 2.3. To elaborate on the reason behind this structure, each class will be presented with its intended usage.

`gryphon_base`: This class is the highest superclass in the Gryphon hierarchy. Its constructor is responsible for assigning a variety of class variables to be used in both the `gryphon_toolbox`-class and the `time_integrator`-class. It is also contains methods for input verification, printing of program progress and error handling. In short, it is designed to be a platform for which to build tools for doing time integration, and for the time integrators themselves.

`gryphon_toolbox`: This class is intended to contain tools found useful when implementing a time integrator. This resonates well with the DRY principle in the sense that we collect tools relevant for several time integrators in one place, which makes for easy maintenance. Notable tools in this class includes step size selectors and methods for output generation (run time statistics, plot of accepted/rejected step sizes).

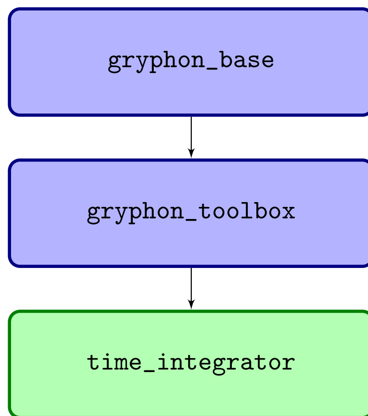


Figure 2.3: Class hierarchy in Gryphon

`time_integrator`: This final class layer is intended to contain the realization of some time integrator. Each class in this layer must contain the code for augmenting a user specified UFL form with the code amounting to applying the desired time integrator. As of today, the only class of methods realized in Gryphon are ESDIRK methods. To show how other time integrators can be implemented, we will take a closer look at how the ESDIRK class is implemented, with the intent that the methodology can be copied.

2.3.1 The ESDIRK class

The ESDIRK class represents the realization of the ESDIRK methods developed by Anne Kværnø presented in section 1.5. It is capable of solving systems of PDEs which

either semidiscretize into a pure ODE system or a DAE system of index 1. The constructor expects to receive UFL forms representing the right hand side of a system of PDEs on the form

$$Mu' = rhs$$

where M may be singular.

`Constructor`: The constructor is responsible for passing on the user given data to the `gryphon_toolbox` class which then passes the data to the `gryphon_base` class for input verification and initialization.

`getLinearVariationalForms`: This method is responsible for creating the UFL-forms arising from applying an ESDIRK method to the user given right hand side. This method assumes the problem to be linear and creates a variational problem on the form: Find $u_n \in V$ such that

$$a(u_n, v) = \ell(v)$$

for all $v \in \hat{V}$ where a is bilinear in u_n and v and ℓ is linear in v.

`getNonlinearVariationalForms`: This method is responsible for creating the UFL-forms arising from applying an ESDIRK to the user given right hand side. This method assumes the problem to be nonlinear and creates a variational problem on the form: Find $u_n \in V$ such that

$$F(u_n; v) = 0$$

for all $v \in \hat{V}$ where F is linear in v.

`solve`: This method is responsible for initializing and performing the time stepping loop. The process is outlined in algorithm 2, but a more detailed description can be found by inspecting the attached documentation.

Algorithm 2 `ESDIRK.solve`

1: Get Butcher-tableau and create s `Function`-objects to store the stage values
2: **if** linear problem **then**
3: Call `getLinearVariationalForms`
4: **else**
5: Call `getNonlinearVariationalForms`
6: **end if**
7: **while** in the time stepping loop **do**
8: Set first stage value equal to previous time step
9: Solve for the next $s-1$ stage values
10: Estimate local error
11: **if** Local error in current time step $<$ tolerance **then**
12: Accept step, calculate new step size.
13: **else**
14: Reject step, calculate new step size.
15: **end if**
16: **end while**
17: Generate specified output and terminate program

2.4 A Code Example

We will round off this part by showing the code required to solve the problem introduced in section 2.3:

$$\begin{aligned} \frac{\partial u}{\partial t} &= \nabla^2 u + (\beta - 2 - 2\alpha), && u \in \Omega, \\ u &= 1 + x^2 + y^2 + \beta t, && u \in \partial\Omega, \\ u &= 1 + x^2 + y^2, && t = 0, \end{aligned}$$

on the domain $\Omega = (x,y) \in [0,1] \times [0,1]$ with $\alpha = 3$ and $\beta = 1.2$ with $u = u(x,y)$. Listing 2.2 shows the code for solving (2.20) using FEniCS alone. The code was constructed by following the FEniCS tutorial. Listing 2.3 shows the code for solving (2.20) using FEniCS in combination with Gryphon. The code snippets can also be found in the attached files `sol_FEniCS.py` and `sol_FEniCS_Gryphon.py`. See appendix A for details.

```
from dolfin import *

# Create mesh and define function space
nx = ny = 40
mesh = UnitSquare(nx, ny)
V = FunctionSpace(mesh, "Lagrange", 1)

# Define boundary conditions
alpha = 3; beta = 1.2
u0 = Expression("1 + x[0]*x[0] + alpha*x[1]*x[1] + beta*t",
                alpha=alpha, beta=beta, t=0)

class Boundary(SubDomain): # define the Dirichlet boundary
    def inside(self, x, on_boundary):
        return on_boundary

boundary = Boundary()
bc = DirichletBC(V, u0, boundary)

u_1 = interpolate(u0, V) # Initial condition

dt = 0.3 # time step

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(beta - 2 - 2*alpha)
a = u*v*dx + dt*inner(grad(u), grad(v))*dx
L = (u_1 + dt*f)*v*dx

A = assemble(a)  # assemble only once, before the time stepping
b = None         # necessary for memory saving assemeble call

# Compute solution
u = Function(V)  # the unknown at a new time level
T = 1.8          # total simulation time
t = dt
while t <= T:
    print 'time =', t
    b = assemble(L, tensor=b)
    u0.t = t
    bc.apply(A, b)
    solve(A, u.vector(), b)

    t += dt
    u_1.assign(u)

plot(u_1, interactive=True) # Plot solution in final time step
```

Listing 2.2: FEniCS code for solving (2.20) without Gryphon.

```
from dolfin import *
from gryphon import ESDIRK

# Create mesh and define function space
nx = ny = 40
mesh = UnitSquare(nx, ny)
V = FunctionSpace(mesh, 'Lagrange', 1)

# Define boundary conditions
alpha = 3; beta = 1.2
u0 = Expression('1 + x[0]*x[0] + alpha*x[1]*x[1] + beta*t',
                alpha=alpha, beta=beta, t=0)

class Boundary(SubDomain): # define the Dirichlet boundary
    def inside(self, x, on_boundary):
        return on_boundary

boundary = Boundary()
bc = DirichletBC(V, u0, boundary)

# Initial condition
w = Function(V)
w = interpolate(u0, V)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(beta - 2 - 2*alpha)
rhs = -inner(grad(u), grad(v))*dx + f*v*dx

T = [0,1.8] # Set time interval

# Use ESDIRK solver
solverObject = ESDIRK(T,w,rhs,bcs=bc,tdfBC=[u0])
solverObject.solve()

# Plot solution in final time step
plot(solverObject.u, interactive=True)
```

Listing 2.3: FEniCS code for solving (2.20) using Gryphon.

Part III

Experiments

3.1 Introduction

This part is devoted to testing of the implementation of the ESDIRK methods developed by Anne Kværnø [Kvæ04]. We will follow the naming convention used in her article[5], namely that the methods will be denoted as ESDIRK $p/p-1$ followed by a for methods using local extrapolation $(y_{n+1} = Y_s)$ or b for methods where $(y_{n+1} = Y_{s-1})$. The goal of the experiments is to

- verify that the methods converge correctly
- verify that the global error is reduced according to user specified tolerance for the time integration
- compare run time and CPU time for different problems and tolerances
- verify that the solver can reproduce pattern behavior in reaction-diffusion models and that the stepsize selectors are well behaved

We will restrict our attention to the methods realized in the Gryphon module, namely ESDIRK4/3*a*, ESDIRK4/3*b*, ESDIRK3/2*a* and ESDIRK3/2*b*.

All the experiments presented in this chapter were performed using FEniCS 1.0.0 under Python 2.7.1+ on a computer running Ubuntu 11.04 (64-bit version). The computer was equipped with an Intel Core i7 Quad @ 2.80 GHz processor and 16 GB of internal memory. The CPU-time was measured using the built-in Python module `time.clock()` while wall time was measured using `time.time()`. Plots of the test cases were produced using ESDIRK4/3*a* unless otherwise stated.

In the following sections, u will mean $u(x,y,t)$. The arguments will be written out explicitly where it is convenient for the reader. Systems of nonlinear/linear equations was solved using the FEniCS Newton/LU-solver with default parameters[6] unless otherwise specified.

3.2 Test Cases

In order to verify the correctness of the implementation of the ESDIRK methods, several test problems will be considered. For each test case, the spatial discretization, the time

[5] Note that Y_i refer to the i-th stage value while y_n refer to the n-th time step.

[6] The convergence criterion for the Newton solver is set to `relative` with relative/absolute tolerance equal to $10^{-9}/10^{-10}$. The LU-solver is set to not reuse factorization.

domain and the reason for selecting that problem will be presented.

3.2.1 Case 1: The Heat Equation

As the first test case, the heat equation with source term given as

$$\frac{\partial u}{\partial t} = D_u \nabla^2 u + 10 \sin\left(\frac{\pi}{2} t\right) \exp\left[-\frac{(x-0.7)^2 + (y-0.5)^2}{0.01}\right], \qquad u \in \Omega,$$

with diffusion coefficient $D_u = 0.1$ and boundary/initial conditions given as

$$u(0,y,t) = t, \qquad u(1,y,t) = 0, \qquad u(x,y,0) = 0,$$

on the spatial domain $\Omega = [0,1] \times [0,1]$ and time domain $t \in [0,1]$ was considered. The spatial domain was discretized using first order Lagrange elements on a 49×49 grid, resulting in $50^2 = 2500$ nodes. The source term was selected to give a time dependent localized contribution near the boundary $x = 1$. It is scaled so that the contribution is significant on the specified time interval. The correct diffusion behavior can be seen by inspecting plots of the solution found in figure 3.4.

This somewhat simple example is still useful to study since it shows that the solver is able to handle PDEs which are explicitly dependent on time, both inside the domain and on the boundary. It will also be shown that the solver is able to exploit the linearity of the problem, resulting in increased performance compared to nonlinear problems.

3.2.2 Case 2: The Fisher-Kolmogorov Equation

For the second case, the Fisher-Kolmogorov equation given as

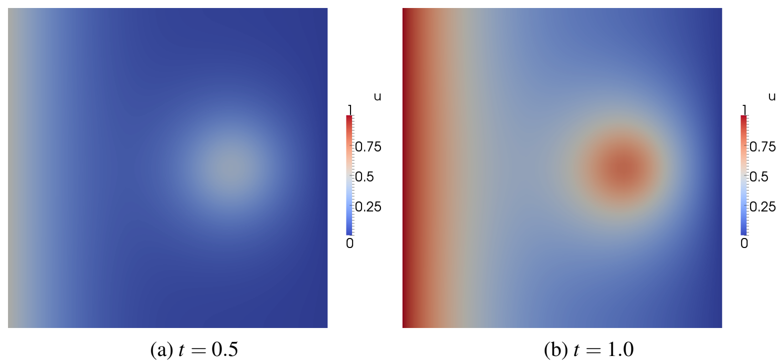
$$\frac{\partial u}{\partial t} = D_u \nabla^2 u + u(1-u), \qquad u \in \Omega,$$

with initial condition given as

$$u(x,y,0) = \exp[-8x]$$

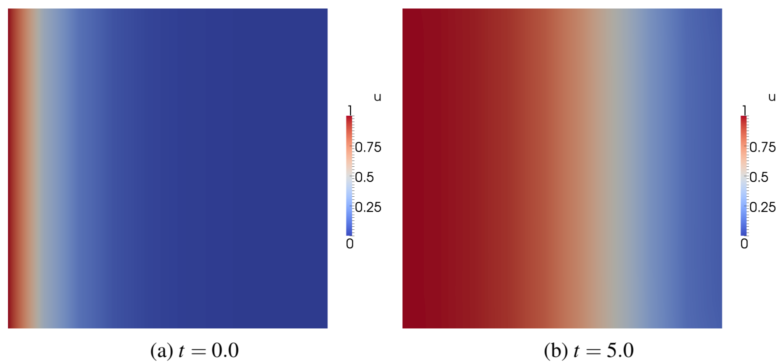
on the spatial domain $\Omega = [0,1] \times [0,1]$ and time domain $t \in [0,5]$ was considered. The spatial domain was discretized using first order Lagrange elements on a 49×49 grid, resulting in $50^2 = 2500$ nodes. For the initial condition specified, the solution assumes a traveling wave front which travels across the spatial domain until the solution $u(x,y,t) = 1$ is reached. This can be seen in figure 3.5 where snapshots of the solution is shown.

Even though this case assumes a rather unproblematic solution, it is still a useful example since it will, by comparison to case 1, show how the solver performs when applied to nonlinear problems.



(a) $t = 0.5$ (b) $t = 1.0$

Figure 3.4: Plot of solution for Case 1: The Heat Equation.



(a) $t = 0.0$ (b) $t = 5.0$

Figure 3.5: Plot of solution for Case 2: The Fisher-Kolmogorov Equation.

3.2.3 Case 3: The Gray-Scott Model

The Gray-Scott model is a nonlinear reaction-diffusion system which consist of a coupled system of two partial differential equations given as

$$\frac{\partial u}{\partial t} = D_u \nabla^2 u - uv^2 + F(1-u), \tag{3.21}$$

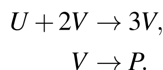
$$\frac{\partial v}{\partial t} = D_v \nabla^2 v + uv^2 - (F+k)v, \tag{3.22}$$

where

$$D_u, D_v, F, k \in \mathbb{R}.$$

These equations were studied on the spatial domain $\Omega = [0,2] \times [0,2]$ which was discretized using first order Lagrange elements on a 49×49 grid. Homogeneous Neumann conditions was used for the boundary.

The model describe the interaction between two chemicals that diffuse, react and get replenished at different rates according to the chemical reaction equation (3.2.3).



$$U + 2V \rightarrow 3V,$$
$$V \rightarrow P.$$

The parameters in (3.21)-(3.22) have the following interpretation.

- u and v are the concentrations of the two chemicals in question,
- D_u and D_v are the diffusion rates of u and v,
- k represents the rate of conversion of V to P,
- F represents the rate of the process that feeds U and drains U, V and P.

Altering these parameters cause the Gray-Scott model to produce vastly different solutions with patterns that vary in a highly nonlinear fashion. By this we mean that the solution can exhibit time intervals where there is rapid change in the solution, and time intervals where the solution is more or less stationary. Numerically, these are interesting cases to study since they can be used to verify that the adaptive step size selection is performed in a constructive manner, that is, small step sizes for rapid change and big step sizes for more stationary conditions.

For the numerical experiments we will consider two cases of pattern formation.

1. Bubble patterns (see figure 3.6)
2. Dot patterns (see figure 3.7)

The parameters for these two cases can be found in table 3.5. For the bubble pattern, the parameters are selected to replicate the results of Robert P. Munafo [Mun12], who has done a very extensive study of the Gray-Scott model for different parameters on his web site. The parameters for the dot pattern are selected to replicate the results of W. Hundsdorfer and J. G. Verwer [HV03, p.22]. As initial condition for the dot pattern,

$$u(x,y,0) = 1 - 2v(x,y,0),$$

$$v(x,y,0) = \begin{cases} 0.25\sin^2(4\pi x)\sin^2(4\pi y) & \text{if } 0.75 \leq x,y \leq 1.25, \\ 0 & \text{elsewhere.} \end{cases}$$

As initial condition for the bubble pattern, $v = 1$ on squares of size 0.1×0.1 distributed randomly on the domain, and 0 otherwise. The other component $u = 1 - v$.

It should be noted that the solution in figure 3.7 fail to be symmetric about the xy-axes. This error is believed to be caused by the low order spatial discretization (first order Lagrange elements). This hypothesis is supported by the fact that when using second order Lagrange elements (not shown here), the resulting solution became more symmetric.

Pattern	$\mathbf{D_u}$	$\mathbf{D_v}$	F	k
Dot	$8.0 \cdot 10^{-5}$	$4.0 \cdot 10^{-5}$	0.024	0.060
Bubble	$2.0 \cdot 10^{-4}$	$1.0 \cdot 10^{-4}$	0.098	0.057

Table 3.5: Parameters used in the Gray-Scott model.

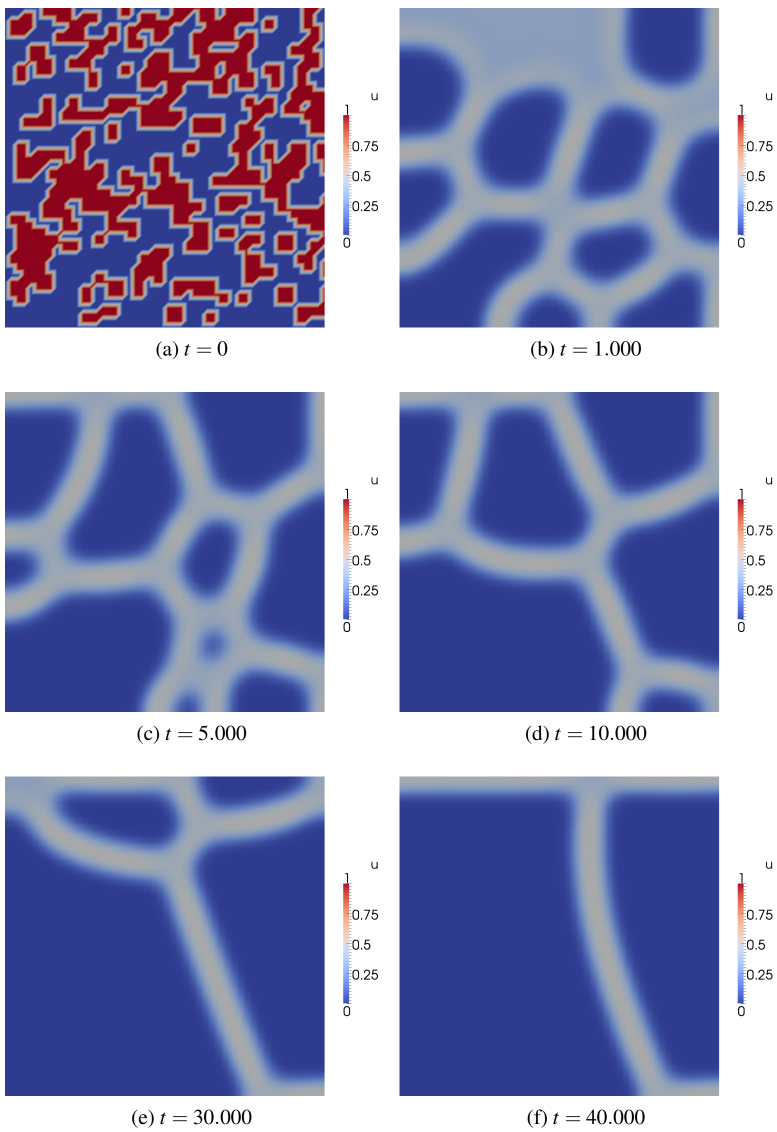
3.2.4 Case 4: A FitzHugh-Nagumo Reaction-Diffusion model

A FitzHugh-Nagumo type reaction-diffusion system can be written as the coupled system of two nonlinear partial differential equations:

$$\frac{\partial u}{\partial t} = D_u \nabla^2 u + f(u) - \sigma v + \kappa,$$
$$\tau \frac{\partial v}{\partial t} = D_v \nabla^2 v + u - v,$$

where

$$D_u, D_v, \tau, \sigma, \kappa \in \mathbb{R}.$$

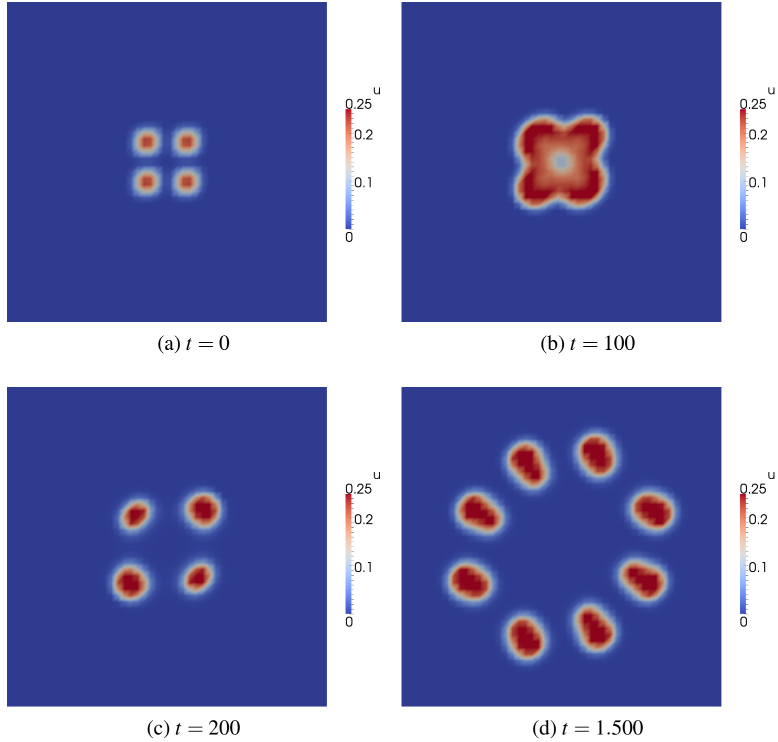


(a) $t = 0$ (b) $t = 1.000$

(c) $t = 5.000$ (d) $t = 10.000$

(e) $t = 30.000$ (f) $t = 40.000$

Figure 3.6: Plot of solution (v-component) for Case 3: The Gray-Scott Model, bubble pattern.



(a) $t = 0$ (b) $t = 100$

(c) $t = 200$ (d) $t = 1.500$

Figure 3.7: Plot of solution (v-component) for Case 3: The Gray-Scott Model, dot pattern.

We will consider a special case of this system where

$$f(u) = \lambda u - u^3, \qquad \lambda \in \mathbb{R}$$

on the spatial domain $\Omega = [0,1] \times [0,1]$. The parameters used for the numerical experimentation can be found in table 3.6 and are selected to reproduce a repeating target pattern behavior. To better the symmetric behavior of the solution, second order Lagrange elements was used to discretize the domain Ω on a 29×29 grid. In order to capture several periods of the solution, the time domain was set to $t \in [0,100]$. As initial condition

$$u(x,y,0) = \exp\left[-\frac{(x-0.5)^2 + (y-0.5)^2}{0.02}\right], \qquad v(x,y,0) = 0,$$

was used, while homogeneous Neumann conditions was used for the boundary. Snapshots of the solution can be found in figure 3.9. Note that these plots show the solution on the extended domain $\hat{\Omega} = [0,5] \times [0,5]$ to illustrate that the solution consist of traveling waves propagating from the center of the domain (where the initial condition is centered). By using second order Lagrange elements we get a better spatial behavior (the wave fronts propagate roughly the same way in all directions). There do however seem to be some noise on the edges which is believed to be caused by a Gibbs effect. This case will be used to verify that the step size selectors behave correctly throughout several waves.

3.2.5 Case 5: The Cahn-Hilliard equation

The Cahn-Hilliard equation is used to model how the mixture of two binary fluids can separate and form domains pure in each component. The equation reads

$$\frac{\partial u}{\partial t} = \nabla \cdot M \left(\nabla \left(\frac{\mathrm{d}f}{\mathrm{d}u} + \lambda \nabla^2 u \right) \right) = 0$$

where $M, \lambda \in \mathbb{R}$. This fourth order PDE can be rewritten to the following system of PDEs,

$$\begin{aligned} \frac{\partial u}{\partial t} &= M \nabla^2 v, \\ 0 &= v - \frac{\mathrm{d}f}{\mathrm{d}u} - \lambda \nabla^2 u, \end{aligned}$$

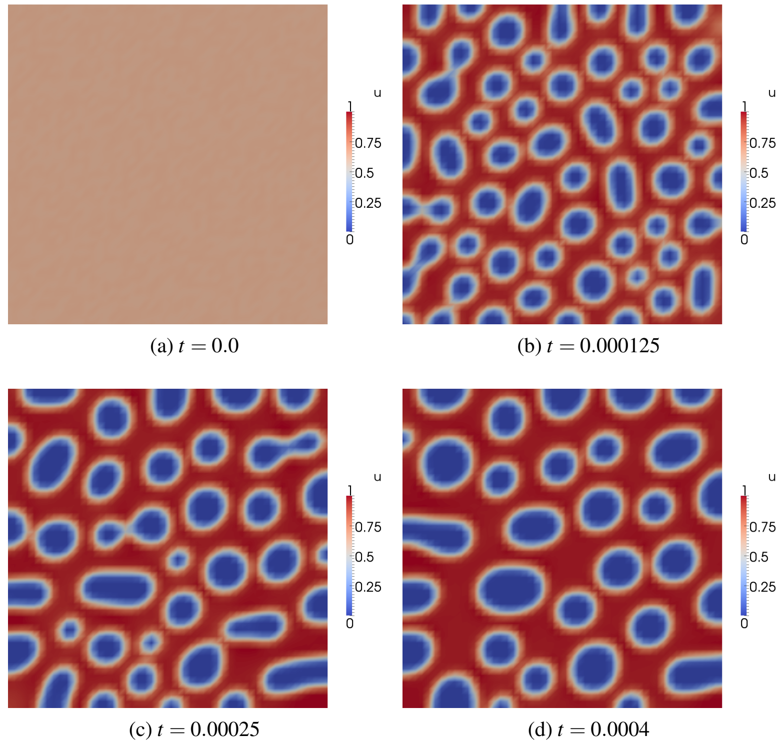
which, when discretized using a finite element method will result in a DAE system of index 1. We will consider the case when $M = 1.0$, $\lambda = 1 \cdot 10^{-2}$ and $f = 100u^2(1-u^2)$ on the spatial domain $\Omega = [0,1] \times [0,1]$ with homogeneous Neumann boundary conditions. In time, we will consider the domain $t \in [0, 4 \cdot 10^{-4}]$. The following initial conditions will be used:

$$u(x,y) = 0.63 + 0.02 \cdot (0.5 - \chi), \qquad \chi \sim Unif[0,1], \qquad v(x,y) = 0.$$

The parameters, initial conditions and domains are chosen to replicate the results by the authors of the FEniCS project[Pro12a]. Plots of the solution can be found in figure 3.8.

3.3 Verification of Convergence

When discussing convergence, it is first and foremost the semidiscretized system of ODEs arising from applying a finite element method to the spatial components of a PDE, we will investigate. For such a system, the following experiment was performed.



(a) $t = 0.0$

(b) $t = 0.000125$

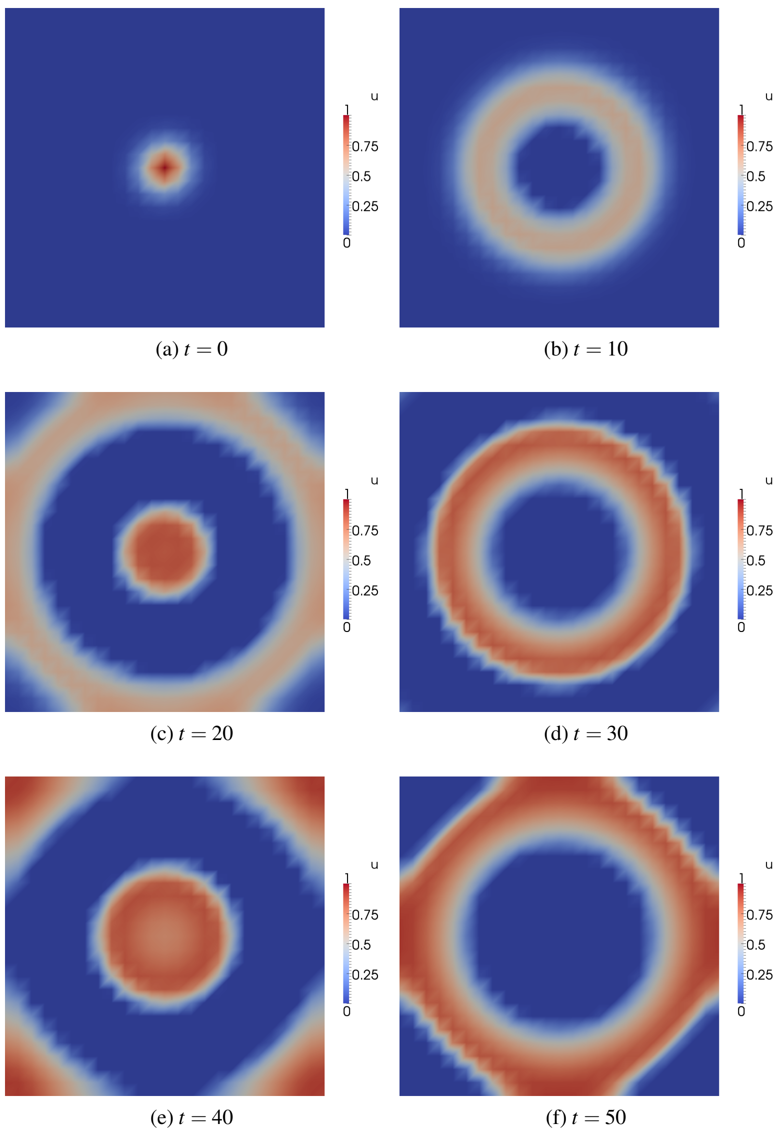
(c) $t = 0.00025$

(d) $t = 0.0004$

Figure 3.8: Plot of solution (u-component) for Case 5: The Cahn-Hilliard equation.

Pattern	$\mathbf{D_u}$	$\mathbf{D_v}$	λ	τ	σ	κ
Target	0.000964	0.0001	0.9	4.0	1.0	0.0

Table 3.6: Parameters used in the FitzHugh-Nagumo Reaction-Diffusion model.

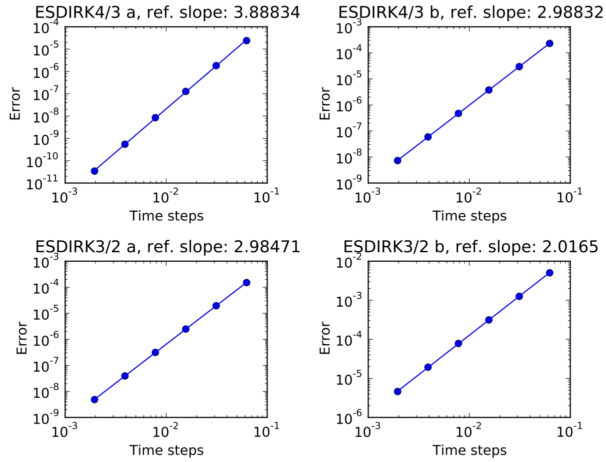


(a) $t = 0$ (b) $t = 10$

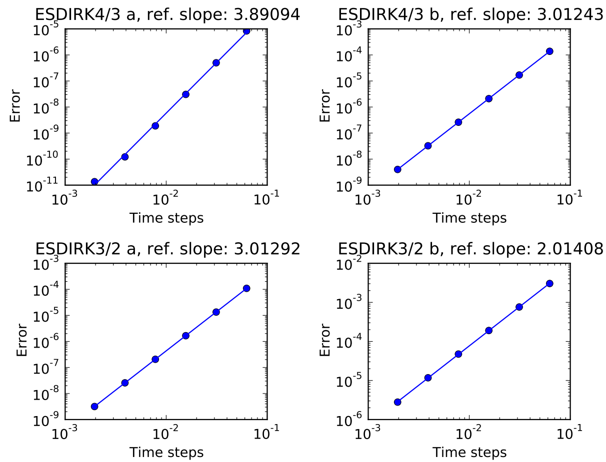
(c) $t = 20$ (d) $t = 30$

(e) $t = 40$ (f) $t = 50$

Figure 3.9: Plot of solution (u-component) for Case 4: A FitzHugh-Nagumo Reaction-Diffusion model, target pattern.



(a) Case 1: The Heat Equation



(b) Case 2: The Fisher-Kolmogorov Equation

Figure 3.10: Convergence verified for Case 1 (linear problem) and Case 2 (nonlinear problem).

For different time dependent PDEs, an ESDIRK method was used to integrate the system in time with a very small, fixed step size. This was used as an approximation to the exact solution to the semidiscretized system. The same system was then integrated over

the same time domain with a variety of greater step sizes. These solutions was then compared with the highly refined solution in order to get an estimate for the global error. When using an ESDIRK method whose advancing method has order p, we should expect that the error, when plotted in a loglog-plot, assumes a linear relation with slope p.

Case 1: The Heat Equation and Case 2: The Fisher-Kolmogorov Equation will be considered to verify that the ESDIRK implementation converges correctly for both the linear and nonlinear problems.

For both cases, ESDIRK43a with step size $\Delta t = 2^{-11}$ was used to produce the approximation to the exact solution of the semidiscretized system. This solution was then compared with solutions produced using time steps $\Delta t = 2^{-i}$ for $i = 4, \ldots, 9$. For the nonlinear case, the convergence criterion for the Newton iterations was set to near machine precision in an attempt to eliminate significant error contributions. The results of these experiments can be found in figure 3.10a and 3.10b which show that the convergence is as expected.

3.4 Verification of Constructive Step Size Selection

Gryphon comes equipped with two different step size selectors (see chapter 1.7 for details). The following experiment sought to investigate how the two step size selectors performed compared to each other. To test this, each step size selector was applied to Case 3: The Gray-Scott model with parameters selected to produce the bubble-pattern and Case 4: A FitzHugh-Nagumo Reaction-Diffusion model. Output from the ESDIRK module can be found in figure 3.11 and 3.12 and in table 3.8. From these figures we see that the standard step size selector is unable to handle dramatic step size reductions without rejecting every other step. The step size selector by Gustafsson, on the other hand, is able to handle this situation better, which is as expected.

3.5 Run Time Statistics

The following experiment sought to verify that the implemented ESDIRK methods behaved as expected when subjected to linear/nonlinear problems for different tolerances and step size selectors. Test Case 1: The Heat Equation and Test Case 2: The Fisher-Kolmogorov Equation will be revisited. We of course expect the solver to perform better with respect to run time on the linear problem, than the nonlinear. It is also of interest to verify that the global error is well behaved with respect to the user specified time stepping tolerance. We would like our solver to behave such that a way that the global error

is reduced proportional to a reduction in the tolerance.

Both the test cases were discretized on a 49×49 grid using first order Lagrange elements (50^2 nodes). The convergence criterion for the time integration was set to `absolute`. As stepsize selectors, both the standard and the Gustafsson selector was tested with pessimistic factor $\mathscr{P} = 0.90$. To generate the reference solution, ESDIRK4/3*a* was used with convergence criterion for the time integration set to `absolute` with absolute tolerance set to 10^{-8}.

Table 3.9/3.10 and 3.11/3.12 shows the results for Case 1 and Case 2 using the standard/Gustafsson stepsize selector. As these tables show, it is clearly advantageous to use local extrapolation since it provides a more accurate solution. The reduction in global error is as desired for both the linear and nonlinear case using both the standard and the Gustafsson stepsize selector. For ESDIRK3/2*b*, the amount of rejected steps using the Gustafsson stepsize selector is significantly higher than for any other ESDIRK-method. This effect can be explained by the fact that the stability function for the error estimate of this method fails to satisfy $|\hat{R}(\infty)| < 1$[7].

3.6 Comparison to Trapezoidal Rule

The trapezoidal rule is a popular second order, *A*-stable, Runge-Kutta method. It has a relatively simple Butcher tableau which can be found in table 3.7. As this table sug-

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

Table 3.7: The Butcher tableau for the Trapezoidal rule.

gests, the method consists of one explicit stage and one implicit, which is significantly less computationally demanding then for example ESDIRK4/3*a* which has one explicit stage and four implicit. The following experiment sought to investigate whether or not an ESDIRK method was able to produce as accurate solutions as the trapezoidal rule, when allowed to use approximately the same amount of CPU time. We will compare performance on a ODE problem and a DAE problem. As ODE problem, case 3: The Gray-Scott model was used with parameters selected to produce the dot pattern (see table 3.5) on the time domain $t \in [0, 200]$ and $t \in [0, 1000]$. As DAE problem, case 5: The

[7] Kværnø has noted this in her article where she suggests that the estimate for local error could be stabilized by multiplying it with $(I - \Delta t \gamma_0 J)^{-1}$ where γ_0 is some constant and J is the Jacobian of the equation. This "trick" originates from [HW10, Section IV.8].

Cahn-Hilliard equation was used on the time domain $t \in [0, 4 \cdot 10^{-4}]$.

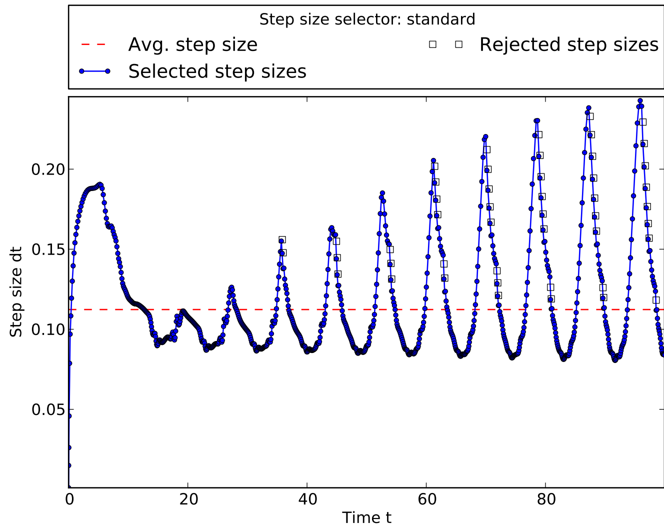
The experiment was carried out in the following way.

- Produce a highly refined solution to be used when approximating the global error.
- Solve the problem with the trapezoidal rule for various fixed time steps.
- Solve the problem with an ESDIRK method for various tolerances.

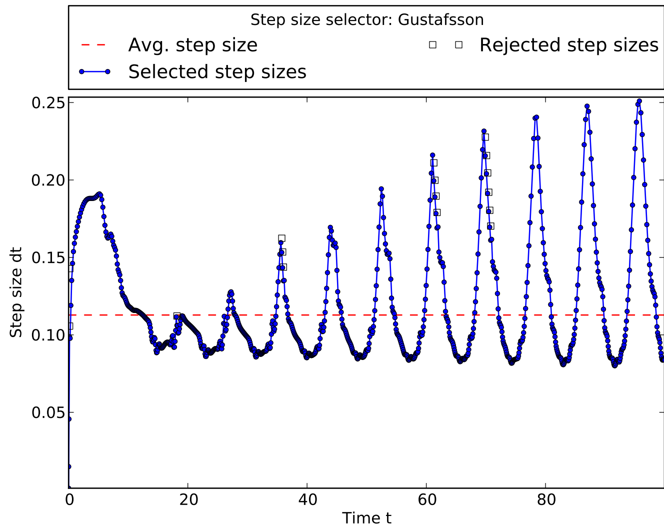
The highly refined solution was, in both cases, produced using ESDIRK4/3*a* with convergence criterion set to "absolute" and absolute tolerance set to 10^{-8}. The tolerance for the Newton solver was set to near machine precision.

The results of this experiment for case 3 can be found in table 3.13, 3.14, 3.15 and 3.16 where CPU time (CPU), wall time (WTM), maximum global error (MGE), number of timesteps performed, number of function evaluations (F) and number of Jacobian evaluations (J) are shown. As these tables show, the Trapezoidal rule is able to compete with ESDIRK4/3*a* for crude tolerances in terms of CPU-time/global error. This is however not the case when we look at a larger time interval ($T = [0, 1000]$) and require increased accuracy. ESDIRK4/3*a* then clearly performs better and is able to produce a solution which is ten times more accurate than what the Trapezoidal rule is able to produce, using approximately half of the CPU/wall time (compare last row in table 3.15/3.16).

The results of this experiment for case 5 can be found in table 3.17 and 3.18 and we see that ESDIRK clearly outperforms the Trapezoidal Rule. This behavior can be explained by the fact that the specified initial data fail to be consistent. As a result of this, the Trapezoidal Rule, which is not L-stable, will get a significant contribution to the error in the first time step (see equation 1.9).

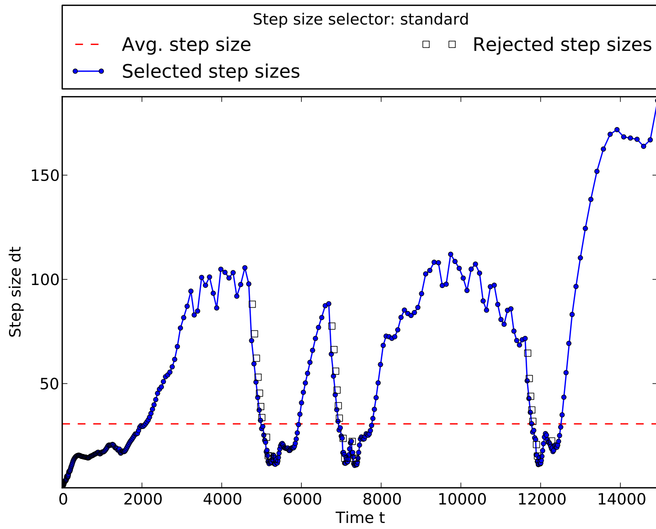


(a) Using standard step size selector.

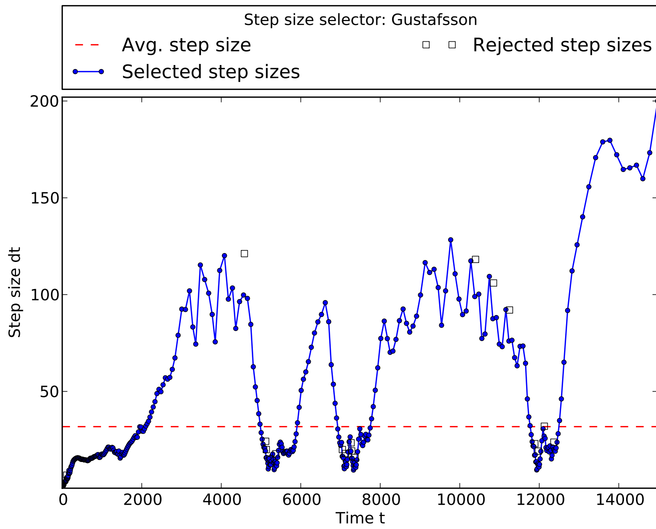


(b) Using Gustafsson step size selector.

Figure 3.11: Selected step sizes for Case 4: A FitzHugh-Nagumo Reaction-Diffusion model.



(a) Using standard step size selector.



(b) Using Gustafsson step size selector.

Figure 3.12: Selected step sizes for Case 3: The Gray-Scott model with bubble pattern parameters.

CPU/wall time	1451.69/0:23:38
No.accepted/rejected steps	891 (94.99%)/47 (5.01%)
Convergence criterion	absolute
Absolute/relative tolerance	0.0001/1e-06
Pessimistic factor	0.95
Step size selector	standard
Function/Jacobian calls	11324/7572
t_{min}/t_{max}	0.001/0.24282
t_{mean}/t_{var}	0.112319/0.00111999

(a) Results using ESDIRK43a on domain (0,50). Using standard step size selector, Case 4 with target pattern.

CPU/wall time	1390.95/0:22:42
No.accepted/rejected steps	887 (98.23%)/16 (1.77%)
Convergence criterion	absolute
Absolute/relative tolerance	0.0001/1e-06
Pessimistic factor	0.95
Step size selector	Gustafsson
Function/Jacobian calls	10915/7303
t_{min}/t_{max}	0.001/0.25098
t_{mean}/t_{var}	0.112813/0.00120819

(b) Results using ESDIRK43a on domain (0,50). Using Gustafsson step size selector, Case 4 with target pattern.

CPU/wall time	507.13/0:08:05
No.accepted/rejected steps	488 (94.21%)/30 (5.79%)
Convergence criterion	absolute
Absolute/relative tolerance	0.0001/1e-06
Pessimistic factor	0.85
Step size selector	standard
Function/Jacobian calls	7810/5738
t_{min}/t_{max}	0.001/185.788
t_{mean}/t_{var}	30.67/1268.62

(c) Results using ESDIRK43a on domain (0,15000). Using standard step size selector, Case 3 with bubble pattern.

CPU/wall time	483.97/0:07:44
No.accepted/rejected steps	472 (96.33%)/18 (3.67%)
Convergence criterion	absolute
Absolute/relative tolerance	0.0001/1e-06
Pessimistic factor	0.85
Step size selector	Gustafsson
Function/Jacobian calls	7465/5505
t_{min}/t_{max}	0.001/200
t_{mean}/t_{var}	31.8009/1396.81

(d) Results using ESDIRK43a on domain (0,15000). Using Gustafsson step size selector, Case 3 with bubble pattern.

Table 3.8: Run time statistics complementing figure 3.11 and 3.12.

Method	TOL	CPU	REJ	ACC	MGE
ESDIRK43a	10^{-2}	1.41	0	11	8.56972e-05
	10^{-3}	2.47	0	20	9.51104e-06
	10^{-4}	4.37	0	36	1.40821e-06
	10^{-5}	7.74	0	63	1.47382e-07
	10^{-6}	13.56	0	112	1.82645e-08
ESDIRK43b	10^{-2}	1.16	0	9	3.13776e-04
	10^{-3}	2.00	0	16	7.12935e-05
	10^{-4}	3.60	0	29	1.32331e-05
	10^{-5}	6.48	2	51	2.31339e-06
	10^{-6}	11.34	2	91	4.20631e-07
ESDIRK32a	10^{-2}	1.24	0	14	4.14390e-05
	10^{-3}	2.52	0	29	4.34604e-06
	10^{-4}	5.23	0	61	4.89570e-07
	10^{-5}	10.96	0	128	5.19361e-08
	10^{-6}	23.67	0	274	5.22892e-09
ESDIRK32b	10^{-2}	1.01	0	11	7.62485e-04
	10^{-3}	2.04	0	23	1.76441e-04
	10^{-4}	4.17	1	47	3.92444e-05
	10^{-5}	8.73	1	100	8.50711e-06
	10^{-6}	18.30	1	212	1.83872e-06

Table 3.9: Case 1: The Heat Equation, standard stepsize selector.

Method	TOL	CPU	REJ	ACC	MGE
ESDIRK43a	10^{-2}	1.39	2	9	9.40667e-05
	10^{-3}	2.11	1	16	1.17102e-05
	10^{-4}	3.89	1	31	1.47787e-06
	10^{-5}	6.99	0	58	1.77092e-07
	10^{-6}	12.85	0	107	1.77143e-08
ESDIRK43b	10^{-2}	1.36	2	8	4.54981e-04
	10^{-3}	2.00	2	14	7.95169e-05
	10^{-4}	3.32	1	26	1.52189e-05
	10^{-5}	5.98	2	47	2.46546e-06
	10^{-6}	10.78	2	87	4.31213e-07
ESDIRK32a	10^{-2}	1.18	1	12	4.29667e-05
	10^{-3}	2.38	1	26	4.65070e-06
	10^{-4}	4.93	0	57	5.06088e-07
	10^{-5}	10.65	0	124	5.20707e-08
	10^{-6}	23.07	0	269	5.26237e-09
ESDIRK32b	10^{-2}	1.30	3	11	8.34979e-04
	10^{-3}	2.51	7	22	1.80384e-04
	10^{-4}	5.01	12	47	3.85343e-05
	10^{-5}	10.10	18	99	8.35449e-06
	10^{-6}	21.53	38	213	1.79763e-06

Table 3.10: Case 1: The Heat Equation, Gustafsson stepsize selector

Method	TOL	CPU	REJ	ACC	MGE
ESDIRK43a	10^{-2}	6.85	0	17	1.57370e-03
	10^{-3}	10.63	0	27	1.31066e-04
	10^{-4}	14.79	0	46	1.13467e-05
	10^{-5}	26.21	1	82	1.02636e-06
	10^{-6}	43.63	2	147	9.74272e-08
ESDIRK43b	10^{-2}	6.04	0	15	2.53641e-03
	10^{-3}	8.68	0	22	8.32247e-04
	10^{-4}	11.80	0	37	1.41376e-04
	10^{-5}	21.39	1	66	2.42187e-05
	10^{-6}	35.05	1	117	4.19581e-06
ESDIRK32a	10^{-2}	6.40	0	22	5.03430e-04
	10^{-3}	10.16	0	44	4.59702e-05
	10^{-4}	21.12	1	92	4.39966e-06
	10^{-5}	42.77	1	198	4.32061e-07
	10^{-6}	87.64	2	425	4.16067e-08
ESDIRK32b	10^{-2}	5.16	1	17	4.27114e-03
	10^{-3}	7.83	0	34	9.30985e-04
	10^{-4}	16.71	1	72	2.02401e-04
	10^{-5}	33.32	2	154	4.38003e-05
	10^{-6}	68.12	2	329	9.46558e-06

Table 3.11: Case 2: The Fisher-Kolmogorov Equation, standard stepsize selector.

Method	TOL	CPU	REJ	ACC	MGE
ESDIRK43a	10^{-2}	5.89	0	14	1.94109e-03
	10^{-3}	10.08	1	24	1.54366e-04
	10^{-4}	13.80	0	43	1.25726e-05
	10^{-5}	25.31	1	79	1.07417e-06
	10^{-6}	42.71	2	143	9.87964e-08
ESDIRK43b	10^{-2}	5.85	0	14	2.90268e-03
	10^{-3}	8.77	2	20	8.74712e-04
	10^{-4}	11.20	0	35	1.53068e-04
	10^{-5}	20.33	1	63	2.51366e-05
	10^{-6}	34.35	1	115	4.27497e-06
ESDIRK32a	10^{-2}	6.26	1	20	5.75900e-04
	10^{-3}	9.43	0	41	4.95075e-05
	10^{-4}	20.65	1	90	4.52694e-06
	10^{-5}	42.03	1	195	4.36589e-07
	10^{-6}	87.05	2	423	4.05236e-08
ESDIRK32b	10^{-2}	6.44	6	17	4.13258e-03
	10^{-3}	9.89	9	34	9.14550e-04
	10^{-4}	19.36	13	71	1.99556e-04
	10^{-5}	39.18	28	154	4.28894e-05
	10^{-6}	79.14	64	333	9.25742e-06

Table 3.12: Case 2: The Fisher-Kolmogorov Equation, Gustafsson stepsize selector.

TOL	CPU	WTM	MGE	# steps	F	J
10^{-2}	48.53	0:00:46	4.51007e-04	38	529	377
10^{-3}	75.44	0:01:11	4.79235e-05	61	831	587
10^{-4}	104.40	0:01:39	5.67949e-06	101	1198	794
10^{-5}	172.10	0:02:43	6.15685e-07	174	1992	1296
10^{-6}	305.42	0:04:49	8.26995e-08	304	3527	2311

Table 3.13: Case 3, ESDIRK4/3a on domain $t \in [0, 200]$.

Δt	CPU	WTM	MGE	# steps	F	J
1.0	46.17	0:00:43	9.95809e-05	200	600	400
0.8	57.48	0:00:54	6.37574e-05	250	750	500
0.4	114.19	0:01:47	1.59479e-05	500	1500	1000
0.2	164.67	0:02:35	5.44435e-06	1000	2401	1401
0.1	265.94	0:04:11	1.66880e-06	2000	4215	2215

Table 3.14: Case 3, Trapezoidal Rule on domain $t \in [0, 200]$.

TOL	CPU	WTM	MGE	# steps	F	J
10^{-2}	105.21	0:01:39	6.64926e-04	73	1121	829
10^{-3}	172.84	0:02:44	1.02003e-04	128	1903	1391
10^{-4}	263.65	0:04:09	1.21198e-05	226	2940	2036
10^{-5}	417.86	0:06:36	1.26448e-06	408	4800	3168
10^{-6}	766.28	0:12:05	1.25009e-07	741	8771	5807

Table 3.15: Case 3, ESDIRK4/3a on domain $t \in [0, 1000]$.

Δt	CPU	WTM	MGE	# steps	F	J
2.5	93.20	0:01:27	9.05509e-04	400	1208	808
0.8	286.49	0:04:29	9.30402e-05	1250	3750	2500
0.4	493.48	0:07:43	2.41657e-05	2500	6757	4257
0.2	645.63	0:10:09	7.43637e-06	5000	10401	5401
0.1	1234.43	0:19:24	2.04581e-06	10000	20215	10215

Table 3.16: Case 3, Trapezoidal Rule on domain $t \in [0, 1000]$.

TOL	CPU	WTM	MGE	# steps	F	J
10^{-2}	405.34	0:06:19	6.80848e-02	325	5106	3806
10^{-3}	864.13	0:13:27	5.13463e-03	686	10764	8020
10^{-4}	1292.01	0:20:08	3.79484e-04	1427	17460	11752
10^{-5}	2525.29	0:39:22	2.78449e-05	2870	34436	22956

Table 3.17: Case 5, ESDIRK4/3a on domain $t \in [0, 4 \cdot 10^{-4}]$

Δt	CPU	WTM	MGE	# steps	F	J
1e-06	118.63	0:01:50	6.63918e+00	400	1587	1187
1e-07	821.53	0:12:44	2.48727e-02	4000	12000	8000
5e-08	1640.64	0:25:25	1.13210e-02	8000	24000	16000
1e-08	8194.87	2:07:01	2.08466e-03	40000	120000	80000

Table 3.18: Case 5, Trapezoidal Rule on domain $t \in [0, 4 \cdot 10^{-4}]$

Part IV

Gryphon User Manual

4.1 Introduction

This part serves as a user manual to the Gryphon module. Gryphon is intended for use under FEniCS 1.0.0 with the Python API. This document expects that the reader is familiar with the FEniCS framework.

4.2 Handling Explicit Time Dependency

In order for the Gryphon to recognize explicit time dependent expressions, the variable representing time in an expression **must** be named `t`. If we want to represent the function

$$f(x,y,t) = x + y + t,$$

we can do it in two ways. The simplest way is to write

```
f = Expression("x[0] + x[1] + t",t=0)
```

or you could define a class

```
class f(Expression):
  def eval(self, values,x):
    values[0] = x[0] + x[1] + self.t
```

Note that the variable `t` must be assigned an initial value.

4.3 Solver: ESDIRK

The ESDIRK class represents the realization of a collection of singly diagonally implicit Runge-Kutta methods with an explicit first stage. It is able to handle systems of PDEs which either semidiscretize into a pure ODE system or a DAE system of index 1.

In order to use an ESDIRK solver you first have to import it as such:

```
from gryphon import ESDIRK
```

The constructor for the ESDIRK object have three required arguments and four optional keyword arguments. The header for the constructor is as follows:

```
def __init__(self,T,u,f,g=[],bcs=[],tdf=[],tdfBC=[]):
```

The arguments have the following interpretations:

T: This argument should be an array with two elements defining the start and end of the time domain you wish to integrate over. For example, `T = [0,1]`.

u: This argument represent the initial condition used for the time integration. It should be given as a `Function`-object defined on the spatial mesh for which to integrate. When the solver terminates, this object will contain the solution in the final time step.

f: This argument should be a `Form`-object representing the ODE-component of your problem. If you are solving a system with more than one ODE-component, you can pass a list of `Form`-objects corresponding to each ODE-component in the system. An example of this can be found at the end of this document where the Gray-Scott model is solved.

g: This argument should be a `Form`-object representing the DAE-component of your problem. If you are solving a system with more than one DAE-component, you can pass a list of `Form`-objects corresponding to each DAE-component in the system.

tdf: If any of the `Form`-objects in `f` or `g` contains explicit time dependent functions, they must be passed to the constructor via this argument.

bcs: This argument should be a list of `DirichletBC`-objects or `PeriodicBC`-objects defining the Dirichlet boundary conditions of your problem.

tdfBC: If any of the boundary conditions specified in `f` or `g` are explicitly dependent on time, they must be passed to the constructor via this argument.

4.3.1 Parameters

Gryphon inherits the parameter system included in the FEniCS framework. This allows any user familiar with FEniCS to quickly get an overview over the available parameters for a ESDIRK-object by calling

```
info(ESDIRK_object.parameters, verbose=True)
```

By default, this will return the following output (some columns are omitted)

```
<Parameter set "gryphon" containing 3 parameter(s)>
  gryphon    |     type      value
  ---------------------------------
  drawplot   |     bool      false
  method     |    string   ESDIRK43a
  verbose    |     bool      false

  <Parameter set "output" containing 3 parameter(s)>
    output       |     type        value
    ------------------------------------
    path         |   string   outputData
    plot         |     bool        false
     statistics  |     bool        false

  <Parameter set "timestepping" containing 10 parameter(s)>
    timestepping                 |     type      value
    --------------------------------------------------
    absolute_tolerance           |  double      1e-07
    adaptive                     |    bool       true
    convergence_criterion        |  string   absolute
    dt                           |  double       0.15
    dtmax                        |  double         15
    dtmin                        |  double      1e-14
     inconsistent_initialdata    |    bool      false
    pessimistic_factor           |  double        0.8
    relative_tolerance           |  double      1e-06
     stepsizeselector            |  string   standard
```

It should be noted that the parameter sets "output" and "timestepping" are nested under the parameter set "gryphon". This means that if we for instance would like to turn off adaptive time stepping, turn on plotting and save statistics, we have to write the following:

```
ESDIRK_object.parameters["timestepping"]["adaptive"] = False
ESDIRK_object.parameters["drawplot"] = True
ESDIRK_object.parameters["output"]["statistics"] = True
```

The various parameters have the following interpretations:

Parameter set: gryphon

method (Default value: `"ESDIRK43a"`) This parameter defines which ESDIRK method the program should use when doing the time stepping. Available methods to choose from are ESDIRK43a, ESDIRK43b, ESDIRK32a, ESDIRK32b. For details on these methods, see table 4.19

Name	Order	Implicit Stages	Local Extrapolation
ESDIRK4/3a	4	4	Yes
ESDIRK4/3b	3	4	No
ESDIRK3/2a	3	3	Yes
ESDIRK3/2b	2	3	No

Table 4.19: Details on available ESDIRK methods

drawplot (Default value: `False`) If this parameter is set to `True`, the program will display a plot of the solution in each time step using the built in `plot`-function included in the FEniCS framework. Note that the plot is initialized with the keyword argument `rescale=True`.

verbose (Default value: `False`) If this parameter is set to `True`, the program will output a progress bar showing the progress of the time stepping. Example output can be found in listing 4.4. Upon completion, it will also cause the program to print the same statistics as found in listing 4.5 to screen.

```
 |====>..............  | 27.2%  t=2.722  Completion in ~ 0:03:42
 |====>..............  | 27.3%  t=2.728  Completion in ~ 0:03:44
 |====>..............  | 27.3%  t=2.733  Completion in ~ 0:03:30
 |====>..............  | 27.4%  t=2.739  Completion in ~ 0:03:22
 |====>..............  | 27.4%  t=2.744  Completion in ~ 0:03:36
|=====>.............  | 27.5%  t=2.750  Completion in ~ 0:03:23
|=====>.............  | 27.6%  t=2.755  Completion in ~ 0:03:37
```

Listing 4.4: Example output from `verbose=True`

The estimated run time of the program is calculated as

$$\text{Run time} \approx \frac{\text{Wall time in previous iteration}}{\text{Average of selected time steps}} \cdot (t_{end} - t_n) \tag{4.23}$$

where t_{end} is the end of the time domain and t_n is the current time in time step n. Because this estimate is dependent on the average of currently selected time steps, the estimate is not shown before the time stepping process has proceeded 1%.

Parameter set: output

path (Default value: `outputData`) This parameter defines the path to a folder where the program may save output, relative to the current working directory. Usually, this will result in a folder `outputData` being created in the same folder as the script you are running.

plot (Default value: `False`**)** If this parameter is set to `True`, the program will export a plot of the solution in each time step to VTK-format. If you are solving a system of PDEs, each component will be saved separately. The plots will be saved in the following folder:

```
$ current_working_directory/savepath/plot/
```

statistics (Default value: `False`**)** If this parameter is set to `True`, the program will output statistics to an `.tex`-file and a `ascii`-file as well as saving a plot of the selected step sizes. Example output can be seen below in table 4.21, listing 4.5 and figure 4.13. All the data will be saved to the path defined by the parameter `path`.

Parameter set: timestepping

adaptive (Default value: `True`**)** This parameter indicates whether the program should use adaptive step size selection. If set to `False`, fixed time step integration will be used where the fixed time step is defined by the parameter `dt`.

convergence_criterion (Default value: `"absolute"`**)** When deciding whether or not to accept a time step, the estimated local error can be subject to one of two user specified criteria; "absolute" or "relative". If `convergence_criterion = "absolute"`, the step is accepted if

$$\|e_n\|_2 \leq \texttt{absolute_tolerance}.$$

If `convergence_criterion = "relative"` the step is accepted if

$$\|e_n\|_2 \leq \max\{\texttt{relative_convergence} \cdot \|u_n\|_2, \texttt{absolute_convergence}\}.$$

In the above statements, e_n is the estimate for the local error and u_n is the numerical solution in time step n.

absolute_tolerance (Default value: `1e-7`**)** See **convergence_criterion** for more details.

relative_tolerance (Default value: `1e-6`**)** See **convergence_criterion** for more details.

dt (Default value: one thousandth of the time domain) This parameter defines the initial time step used by the program. If the parameter `adaptive` is set to `True`, this is the time step used by the program through the entire domain.

dtmax (Default value: one tenth of the time domain) This parameter defines the greatest allowable time step which the program can use.

dtmin (Default value: `1e-14`) This parameter defines the smallest allowable time step which the program can use. The program will terminate if this boundary is reached (not sure about this yet).

inconsistent_initialdata (Default value: `False`) If the initial data for solving a system of PDEs (which semidiscretize into a DAE system) is inconsistent, the program can attempt to take a very small time step in order to get consistent data. If successful, the program will continue with the initial time step specified by `dt`.

pessimistic_factor (Default value: `0.8`) This parameter defines the pessimistic factor used in the adaptive step size selection. Allowable range for this parameter is $[0,1]$. See **stepsizeselector** for more details.

stepsizeselector (Default value `standard`) This parameter allows the user to select which time step selecting algorithm to use. The available methods are listed in table 4.20.

Method	Expression
`gustafsson`	$\Delta t_{n+1} = \frac{\Delta t_n^2}{\Delta t_{n-1}} \left(\frac{\mathscr{P} \cdot tol \cdot le_{n-1}}{le_n^2} \right)^{1/p}$
`standard`	$\Delta t_{n+1} = \Delta t_n \left(\frac{\mathscr{P} \cdot tol}{le_n} \right)^{1/p}$

Table 4.20: Available time step algorithms.

The `gustafsson` step size selector is developed by K. Gustafsson [Gus94].

4.3.2 Example Output

If the parameter `savestatistics=True` for an ESDIRK-object, the program will, if it terminated successfully, produce a LaTeX/ASCII-table with some relevant run time statistics as well as a plot of the selected time steps. Examples of this output can be seen in table 4.21, listing 4.5 and figure 4.13. The statistics were generated by running the heat equation example presented in the next section.

```
*******************************************
    ODE solver terminated successfully !
*******************************************
Method used:  ESDIRK43a
Domain:    [0,1]
CPU-time:  11
Walltime:   0:00:13
Step size selector :  standard
 Pessimistic factor :  0.8
Convergence criterion :  absolute
Absolute tolerance :  1e-07
 Relative tolerance :  1e-06
Number of steps accepted: 193 (98.97%)
Number of steps rejected :  2 (1.03%)
Maximum step size selected: 0.0118945
Minimum step size selected :  0.000177563
Mean step size :   0.00517595
Variance in step sizes : 1.40139e-05
```

Listing 4.5: Example output from `savestatistics=True`.

CPU/wall time	11/0:00:13
No.accepted/rejected steps	193 (98.97%)/2 (1.03%)
Convergence criterion	absolute
Absolute/relative tolerance	1e-07/1e-06
Pessimistic factor	0.8
Step size selector	standard
t_{min}/t_{max}	0.000177563/0.0118945
t_{mean}/t_{var}	0.00517595/1.40139e-05

Table 4.21: Results using ESDIRK43a on domain [0,1].

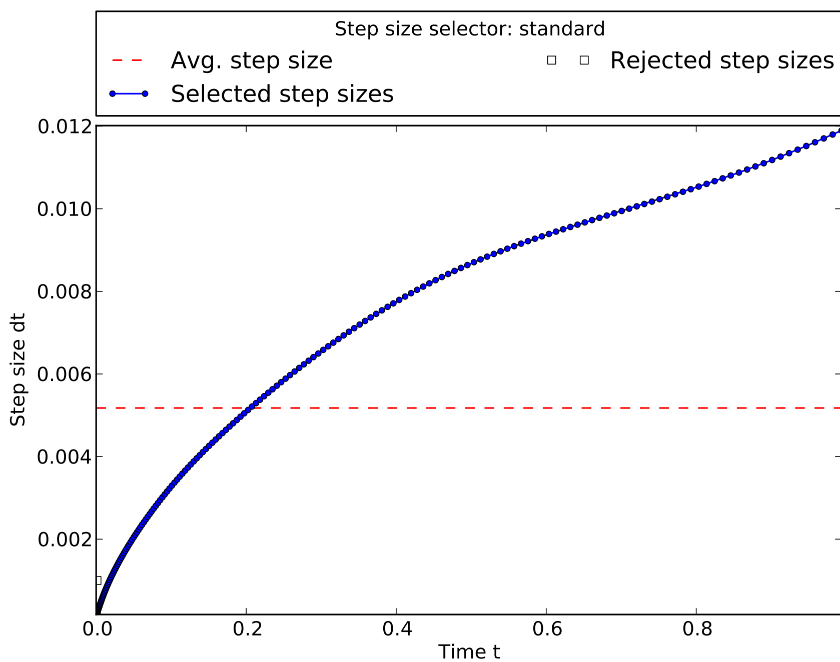


Figure 4.13: Example output from `savestatistics=True`.

4.4 Example Problems

This section will show some code examples for solving time dependent partial differential equations. It is assumed that the reader is familiar with the FEniCS framework as well as being able to derive the weak solution of a PDE.

The Heat Equation

Consider the heat equation with a source term given as

$$\frac{\partial u}{\partial t} = D_u \nabla^2 u + 10 \sin\left(\frac{\pi}{2} t\right) \exp\left[-\frac{(x-0.7)^2 + (y-0.5)^2}{0.01} \right], \qquad u \in \Omega,$$

with diffusion coefficient $D_u = 0.1$ and boundary/initial conditions given as

$$u(0,y,t) = t, \qquad u(1,y,t) = 0, \qquad u(x,y,0) = 0,$$

on the spatial domain $\Omega = [0,1] \times [0,1]$ and time domain $t \in [0,1]$. The FEniCS code for solving this problem using the ESDIRK module can be found in listing 4.6.

```
from gryphon import ESDIRK
from dolfin import *

# Define spatial mesh, function space, trial/test functions
mesh = UnitSquare(29,29)
V = FunctionSpace(mesh,"Lagrange",1)
u = TrialFunction(V)
v = TestFunction(V)

# Define diffusion coefficient and source inside domain
D = Constant(0.1)
domainSource = Expression("10*sin(pi/2*t)*exp(-((x[0]-0.7)*(x[0]-0.7) + (x[1]-0.5)*(x
    [1]-0.5))/0.01)",t=0)

# Define right hand side of the problem
rhs = -D*inner(grad(u),grad(v))*dx + domainSource*v*dx

# Definie initial condition
W = Function(V)
W.interpolate(Constant(0.0))

# Define left and right boundary
def boundaryLeft(x,on_boundary):
  return x[0] < DOLFIN_EPS

def boundaryRight(x,on_boundary):
  return 1.0 - x[0] < DOLFIN_EPS

boundarySource = Expression("t",t=0)
bcLeft  = DirichletBC(V,boundarySource,boundaryLeft)
bcRight = DirichletBC(V,0.0,boundaryRight)

# Define the time domain
T = [0,1]

# Create the ESDIRK object
obj = ESDIRK(T,W,rhs,bcs=[bcLeft,bcRight],tdfBC=[boundarySource],tdf=[domainSource])

# Turn on some output and save run time
# statistics to sub folder "HeatEquation"
obj.parameters["verbose"] = True
obj.parameters["drawplot"] = True
obj.parameters["output"]["path"] = "HeatEquation"
obj.parameters["output"]["statistics"] = True

# Solve the problem
obj.solve()
```

Listing 4.6: FEniCS code for solving the heat equation with time dependent source term and time dependent boundary.

The Gray-Scott model

The Gray-Scott model is a nonlinear reaction-diffusion system which consist of a coupled system of two partial differential equations given as

$$\frac{\partial u}{\partial t} = D_u \nabla^2 u - uv^2 + F(1-u),$$
$$\frac{\partial v}{\partial t} = D_v \nabla^2 v + uv^2 - (F+k)v,$$

where

$$D_u, D_v, F, k \in \mathbb{R}.$$

We want to solve this system using the parameters $D_u = 8.0 \cdot 10^{-5}$, $D_v = 4.0, \cdot 10^{-5}$, $F = 0.024$, $k = 0.060$ on the spatial domain $\Omega = [0,2] \times [0,2]$ over the time domain $t \in [0,100]$, with homogeneous Neumann boundary conditions and initial conditions given as

$$u(x,y,0) = 1 - 2v(x,y,0),$$
$$v(x,y,0) = \begin{cases} 0.25 \sin^2(4\pi x) \sin^2(4\pi y) & \text{if } 0.75 \leq x,y \leq 1.25, \\ 0 & \text{elsewhere.} \end{cases}$$

The FEniCS code for solving this problem using the ESDIRK module can be found in listing 4.7.

```
from gryphon import ESDIRK
from dolfin import *
from numpy import power,pi,sin

class InitialConditions(Expression):
  def eval(self, val, x):
    if between(x[0],(0.75,1.25)) and between(x[1],(0.75,1.25)):
      val[1] = 0.25*power(sin(4*pi*x[0]),2)*power(sin(4*pi*x[1]),2)
      val[0] = 1 - 2*val[1]
    else:
      val[1] = 0
      val[0] = 1
  def value_shape(self):
    return (2,)

# Define mesh, function space and test functions
mesh = Rectangle(0.0, 0.0, 2.0, 2.0, 49, 49)
V = FunctionSpace(mesh, "Lagrange", 1)
ME = V*V
q1,q2 = TestFunctions(ME)

# Define and interpolate initial condition
W = Function(ME)
W.interpolate(InitialConditions())
u,v = split(W)
# Define parameters in Gray-Scott model
Du  = Constant(8.0e-5)
Dv  = Constant(4.0e-5)
F   = Constant(0.024)
k   = Constant(0.06)
# Define the right hand side for each of the PDEs
F1 = (-Du*inner(grad(u),grad(q1)) - u*(v**2)*q1 + F*(1-u)*q1)*dx
F2 = (-Dv*inner(grad(v),grad(q2)) + u*(v**2)*q2 - (F+k)*v*q2)*dx
# Define the time domain
T = [0,100]
# Create the solver object and adjust tolerance
obj = ESDIRK(T,W,[F1,F2])
obj.parameters["timestepping"]["absolute_tolerance"] = 1e-3
# Turn on some output and save run time
# statistics to sub folder "GrayScott"
obj.parameters["verbose"] = True
obj.parameters["drawplot"] = True
obj.parameters["output"]["path"] = "GrayScott"
obj.parameters["output"]["statistics"] = True
# Suppress some FEniCS output
set_log_level(WARNING)
# Solve the problem
obj.solve()
```

Listing 4.7: FEniCS code for solving the Gray-Scott model.

The Cahn-Hilliard equation

The Cahn-Hilliard equation is used to model how the mixture of two binary fluids can separate and form domains pure in each component. The equation can be written as the following system

$$\begin{aligned} \frac{\partial u}{\partial t} &= M\nabla^2 v, \\ 0 &= v - \frac{\mathrm{d}f}{\mathrm{d}u} - \lambda \nabla^2 u, \end{aligned}$$

which, when discretized, will result in a DAE system of index 1. We will consider the case when $M = 1.0$, $\lambda = 1 \cdot 10^{-2}$ and $f = 100u^2(1-u^2)$ on the spatial domain $\Omega = [0,1] \times [0,1]$ with homogeneous Neumann boundary conditions. In time, we will consider the domain $t \in [0, 4 \cdot 10^{-4}]$. The following initial conditions will be used

$$u(x,y) = 0.63 + 0.02 \cdot (0.5 - \chi), \qquad \chi \sim Unif[0,1], \qquad v(x,y) = 0.$$

The FEniCS code for solving this problem using the ESDIRK module can be found in listing 4.8

```
from gryphon import ESDIRK
from dolfin import *
import random

# Initial conditions
class InitialConditions (Expression):
  def __init__(self):
      random.seed(2 + MPI.process_number())
  def eval( self , values , x):
      values [0] = 0.63 + 0.02*(0.5 - random.random())
      values [1] = 0.0
  def value_shape(self):
      return (2,)

# Create mesh and define function spaces
mesh = UnitSquare(49, 49)
V = FunctionSpace(mesh, "Lagrange", 1)
ME = V*V

q,v = TestFunctions(ME)

# Define and interpolate initial condition
u   = Function(ME)
u. interpolate ( InitialConditions ())

c,mu = split(u)
c = variable(c)
f    = 100*c**2*(1-c)**2
dfdc = diff(f, c)
lmbda = Constant(1.0e-02)

# Weak statement of the equations
f = -inner(grad(mu), grad(q))*dx
g = mu*v*dx - dfdc*v*dx - lmbda*inner(grad(c), grad(v))*dx

T = [0,5e-5] # Time domain

myobj = ESDIRK(T,u,f,g=g)
myobj.parameters['timestepping']['absolute_tolerance'] = 1e-2
myobj.parameters['timestepping'][' inconsistent_initialdata '] = True
myobj.parameters['verbose'] = True
myobj.parameters['drawplot'] = True

# Suppress some FEniCS output
set_log_level(WARNING)

# Solve the problem
myobj.solve()
```

Listing 4.8: FEniCS code for solving the the Cahn-Hilliard equation.

Summary of Results

After considering various Runge-Kutta schemes for solving stiff ODEs, the ESDIRK methods developed by Anne Kværnø were found to be suitable. The idea of using these methods as time integrators when solving time dependent partial differential equations, gave rise to the Gryphon module.

The code structure of Gryphon focuses on modularity. Adding a new time integrator scheme only requires writing the code for augmenting a user-specified problem (specified in UFL) with the code amounting to applying the desired time integrator, and the code for performing the time stepping loop. Tools like time stepping algorithms are already available and can be reused. The source code for the ESDIRK solvers are well documented (see attachment A) and aims to serve as guidance for people who wants to extend the framework with their own time integrators.

The ESDIRK methods have been tested on various time dependent PDEs, verifying that the methods converge correctly and are able to compete with other time integrators (Trapezoidal rule), when it comes to numerical accuracy versus run time. It has also been verified that the step size selectors behave as expected by subjecting them to reaction-diffusion problems.

Numerous code examples have also been presented, hopefully motivating the reader to try and use Gryphon for solving his/her own problems.

Appendix A

Appended Code and Documentation

Together with this document you should have received an attached zip file called `gryphon_appendix.zip`. This file contains the following:

`documentation/`

This folder contains the documentation for Gryphon. To start, open the file `Gryphon.html` in any web browser (Firefox/Chrome looks good).

`source/`

This folder contains the source code for the Gryphon module. In order to use Gryphon, this folder must be included in your Python-path.

`examplecode/`

This folder contains the Python code for the examples found in the Gryphon user manual.

`reportcode/`

This folder contains the Python code for the example presented in section II.2.4.

Bibliography

[BCP89] K.E. Brenan, S.L. Campbell, and L.R. Petzold, *Numerical solution of initial-value problems in differential-algebraic equations*, Classics in Applied Mathematics, Society for Industrial and Applied Mathematics, 1989.

[EJL03] K. Eriksson, C. Johnson, and A. Logg, *Explicit time-stepping for stiff ODEs*, SIAM Journal on Scientific Computing **25** (2003), 1142.

[Gus94] K. Gustafsson, *Control-theoretic techniques for stepsize selection in implicit Runge-Kutta methods*, ACM Transactions on Mathematical Software (TOMS) **20** (1994), no. 4, 496–517.

[HT99] A. Hunt and D. Thomas, *The pragmatic programmer: From journeyman to master*, Pearson Education, 1999.

[HV03] W.H. Hundsdorfer and J.G. Verwer, *Numerical solution of time-dependent advection-diffusion-reaction equations*, vol. 33, Springer Verlag, 2003.

[HW10] E. Hairer and G. Wanner, *Solving ordinary differential equations. ii: Stiff and differential-algebraic problems.*, Springer Series in Computational Mathematics 14. Berlin: Springer. xvi, 614 p., 2010.

[KNO96] A. Kværnø, SP Nørsett, and B. Owren, *Runge-Kutta research in Trondheim*, Applied numerical mathematics **22** (1996), no. 1, 263–277.

[Kvæ04] A. Kværnø, *Singly diagonally implicit Runge-Kutta methods with an explicit first stage*, BIT Numerical Mathematics **44** (2004), no. 3, 489–502.

[LMW11] A. Logg, K.A. Mardal, and G. Wells, *Automated solution of differential equations by the finite element method: The fenics book*, 2011.

[Mun12] R. Munafo, *Reaction-diffusion by the Gray-Scott model: Pearson's parameterization*, `http://mrob.com/pub/comp/xmorphia/`, April 2012.

[NT84] S.P. Nørsett and P.G. Thomsen, *Embedded SDIRK-methods of basic order three*, BIT Numerical Mathematics **24** (1984), no. 4, 634–646.

[Pro12a] The FEniCS Project, *Cahn-hilliard equation*, `http://fenicsproject.org/documentation/dolfin/1.0.0/python/demo/pde/cahn-hilliard/python/documentation.html`, May 2012.

[Pro12b] ———, *Time-dependent problem, tutorial*, `http://fenicsproject.org/documentation/tutorial/timedep.html`, May 2012.