Numerical Methods For Solution of Differential Equations

Tobias Ritschel

Kongens Lyngby 2013
B.Sc.-2013-16

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Building 303B, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk B.Sc.-2013-16

Summary

Runge-Kutta methods are used to numerically approximate solutions to initial value problems, which may be used to simulate, for instance, a biological system described by ordinary differential equations. Simulations of such system may be used to test different control strategies and serve as an inexpensive alternative to real-life testing.

In this thesis a toolbox is developed in `C` and `Matlab` containing effective numerical Runge-Kutta methods. Testing thse methods on the time it takes to simulate a system for a large set of parameters showed that some methods performed well in some cases whereas others were faster in others such that not one method outbested the others.

Carrying out the same tests using parallel simulations showed that a speed-up of between 11 and 12 is possible in `Matlab` and in `C` when using 12 processes, and that different implementations of parallel simulations in `C`, are suitable for different number of processes.

In all aspects, simulations were obtained much faster in `C` compared to `Matlab`.

Resumé

Runge-Kutta metoder bruges til at approksimere løsninger til begyndelses-værdi problemer numerisk, hvilket eksempelvis kan bruges til at simulere et biologisk system beskrevet af ordinære differentialligninger. Disse simuleringer kan bruges til at teste forskellige kontrolstrategier og være et mindre krævende alternativ til fysiske tests.

I dette projekt udvikles en toolbox in `C` og `Matlab` med effektive numeriske Runge-Kutta metoder. Tests af den tid det tager metoderne at simulere et system for et stort sæt parametre viste at visse metoder var hurtige i en slags situationer og andre var hurtigere i andre situtioner således at ikke éen metode var de andre overlegen.

Ved at udføre de samme tests med parallelle simuleringer kunne det ses at det er muligt at udføre simuleringerne 11 til 12 gange hurtigere i `Matlab` og i `C` ved brug af 12 processer og at forskellige implementeringer af parallelle simuleringer i `C`, er passende afhængigt af antallet af processer.

Simuleringer blev i alle tilfælde udført meget hurtigere i `C` i forhold til `Matlab`.

Preface

This thesis was prepared at Department of Applied Mathematics and Computer Science, the Technical University of Denmark in fulfillment of the requirements for acquiring the B.Sc. degree in Mathematics & Technology.

The thesis concerns numerical methods for solving initial value problems and documents the Runge-Kutta toolbox created during the project. The main focus is on implementation of the numerical methods in `C` and `Matlab` and on the runtimes of the implementations on the two platforms. The simulations which are timed, will be implemented in both sequential and parallel.

I would like to thank John Bagterp Jørgensen for guidance throughout the project and many helpful comments on the thesis and Carsten Völcker for help with implementation of the methods. I would also like to thank Bernd Dammann for help with MPI and LAPACK in `C`.

Lyngby, December 2013

Tobias Ritschel

Contents

Chapter 1

Introducton and Purpose

The topics that are treated in this project is presented in Section 1.1. These are, the type of differential equations to which solutions are approximated, which methods will be treated for obtaining these approximations and why it is important to consider conservation properties.

The purpose and goals of this project are presented in Section 1.2, namely what methods are implemented, what should be their interface and how will the methods be tested.

1.1 Introduction

This project is concerned with the computation of numerical approximations to the solution of initial value problems (IVPs) of the sort

$$\frac{\mathrm{d}}{\mathrm{d}t}g(x(t)) = f(t, x(t)), \quad x(t_0) = x_0, \tag{1.1}$$

where $g : \mathbb{R}^n \to \mathbb{R}^n$, $f : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$, $x(t) \in \mathbb{R}^n$. $g(x(t))$ may be nonlinear in $x(t)$ and $f(t, x(t))$ may be nonlinear in t and $x(t)$.

Equation (1.1) is a generalization of the standard form of ordinary differential equations, which is,

$$\frac{\mathrm{d}}{\mathrm{d}t}x(t) = f(t, x(t)), \quad x(t_0) = x_0. \tag{1.2}$$

We will investigate methods for solving both problem (1.1) and (1.2). Both problems may be stiff and methods for both the stiff and the non-stiff case are treated.

Explicit methods are preferred over implicit methods when the IVP is non-stiff because of lower computational cost. In the non-stiff case we use the Euler method, the Classical Runge-Kutta, the Runge-Kutta-Fehlberg and the Dormand-Prince method. In the stiff case implicit methods may produce accurate solutions using far larger steps than an explicit method of equivalent order, would. In the stiff case we use ESDIRK23. We will also treat the modifications needed for these methods to approximate solutions to (1.1).

When approximating the solutions to an IVP for many sets of parameters, the computations may be carried out sequentially or in parallel, in both `Matlab` and `C`. The runtime on each of these platforms are tested on simulations of a fed batch fermenter. The model of this fermenter is described in Chapter 6. These runtime tests are carried out with both fixed and adaptive step size, requiring both low and high precision, and using a different number of processes to carry out the parallel simulations. Low and high precision are defined in Section 8.1.

1.1.1 Conservation Properties

Transforming a problem in the form of (1.1) into (1.2) poses some trouble regarding the conservation which the differential equation describes, e.g. mass or energy. For example, using the explicit Euler for (1.1) gives the following approximation.

$$\frac{g(x_{n+1}) - g(x_n)}{h} = f(t_n, x_n),$$

where the conservation described in the differential equations is conserved from step to step. Using the chain rule, (1.1) may be rewritten as,

$$\frac{\mathrm{d}}{\mathrm{d}t} g(x(t)) = \frac{\partial}{\partial x} g(x(t)) \frac{\mathrm{d}}{\mathrm{d}t} x(t) = f(t, x(t)), \tag{1.3}$$

where

$$\frac{\partial}{\partial x} = \begin{bmatrix} \frac{\partial}{\partial x_1} & \frac{\partial}{\partial x_2} & \cdots & \frac{\partial}{\partial x_n} \end{bmatrix} = \nabla_x^T.$$

Using the explicit Euler method on the problem in this form gives the approximation

$$\frac{\partial}{\partial x} g(x_n) \cdot \frac{x_{n+1} - x_n}{h} = f(t_n, x_n) \tag{1.4}$$

Here the conservation described by the differential equation is not conserved. This introduces further error besides the one introduced by the method. This occurs because the chain rule is only valid in the limit $h \to 0$ when $\frac{\mathrm{d}}{\mathrm{d}t} x(t)$ is discretized from (1.3) to (1.4).

The last step of the transformation from (1.1) to (1.2), is to isolate $\frac{\mathrm{d}}{\mathrm{d}t} x(t)$ such that (1.4) becomes,

$$\begin{aligned} \frac{\mathrm{d}}{\mathrm{d}t} x(t) &= \left[\frac{\partial}{\partial x} g(x(t)) \right]^{-1} f(t, x(t)) \\ &= F(t, x(t)), \end{aligned}$$

where $F(t, x(t))$ is now the right hand side in (1.2).

1.2 Purpose

The purpose of this project is to develop a toolbox in `C` and `Matlab` containing effective numerical Runge-Kutta methods and to document the implementation of these methods. The subpurposes of this project are,

1. implement the following Runge-Kutta methods for (1.2)
    - The Explicit Euler method
    - The Classic Runge-Kutta method, RK4
    - The Runge-Kutta-Fehlberg method, RKF45
    - The Dormand-Prince method, DOPRI54
    - the ESDIRK23 method
2. modify the methods in 1. for (1.1)
3. compare implementations on
    - runtime for fixed step size
    - runtime for low precision
    - runtime for high precision
4. compare simulation runtimes in `Matlab` to simulation runtimes in `C`
5. compare runtime of sequential simulations to those of parallel simulations
6. compare runtime of different implementations of parallel simulations

Low and high precision are defined in Section 8.1. The implementations of these methods have different interfaces. Any method has one interface in `Matlab` and a different one in `C`. Furthermore, the unmodified explicit methods require only $f(t, x(t)$ from (1.2). The unmodified implicit methods need also $\frac{\partial}{\partial x} f(t, x(t))$. The modified versions of these also need $g(x(t))$ and $\frac{\partial}{\partial x} g(x(t))$. In the following Sections, the interfaces for the implemented methods are described.

The comparison between the Runge-Kutta methods will be done by timing the simulations used in the fed batch fermenter problem in Chapter 6. This includes comparing the runtime for a given method when using different fixed step sizes and using adaptive step size, with either low or high accuracy. These comparisons will be done for simulations in both `Matlab` and `C` and using both sequential and parallel simulations.

1.3 Matlab Interfaces

The four different `Matlab` interfaces are shown in Listings 1.1 to 1.4.

Listing 1.1: Matlab-interface for the unmodified explicit methods.

```
function [t,x] = <ERK>(
fun,
tspan,
x0,
AbsTol, RelTol,
varargin)
```

Listing 1.2: Matlab-interface for the unmodified implicit methods.

```
function [t,x] = <IRK>(
fun,
Jac,
tspan,
x0,
AbsTol, RelTol,
varargin)
```

Listing 1.3: Matlab-interface for the modified explicit methods.

```
function [t,x] = <ERKMod>(
fun,
gfun,
gJac,
tspan,
x0,
AbsTol, RelTol,
varargin)
```

Listing 1.4: Matlab-interface for the modifed implicit methods.

```
function [t,x] = <IRKMod>(
fun,
Jac,
gfun,
gJac,
tspan,
x0,
AbsTol, RelTol,
varargin)
```

`<ERK>` is either the Euler method, RK4, RKF45 or DOPRI54. `<IRK>` is ESDIRK23. `<ERKMod>` and `<IRKMod>` are the modified versions of these.

In the `Matlab` implementations, the methods return a vector containing the discrete time values, `t`, and a two-dimensional array containing the approximation of the solution to the IVP in the discrete time values, `x`. `t` is identical to`tspan` supplied by the user if using fixed step size, i.e. if `tspan` has more than two elements.

The inputs are the function handles `fun, Jac, gfun` and `gJac` which return $f(t, x(t))$ and $g(x(t))$ from (1.1) or (1.2), and the Jacobi matrices of these. `fun` and `gfun` should return column vectors. If the method should use fixed step size, `tspan` should contain the equidistant time values, and else, the initial and final time values. `x0` contains the initial conditions. `AbsTol` and `Reltol` are the absolute and relative tolerances, and `varargin` may be used for any parameters which should be passed to the function handles.

1.4 C Interfaces

The four different `C` interfaces are shown in Listings 1.5 to 1.8.

Listing 1.5: C-interface for the unmodified explicit methods.

```
<ERK>(
ODEModel_t* fun,
const int nx, const int nt,
const double *tspan,
const double *x0
const double *AbsTol, const double RelTol,
const void   *params,
double *t,
double *x)
```

Listing 1.6: C-interface for the unmodified implicit methods.

```
<IRK>(
ODEModel_t* fun,
ODEModel_t* Jac,
const int nx, const int nt,
const double *tspan,
const double *x0
const double *AbsTol, const double RelTol,
const void   *params,
double *t,
```

```
10 double *x)
```

Listing 1.7: C-interface for the modified explicit methods.

```
1  <ERKMod>(
2  ODEModel_t* fun,
3  ODEModel_t* gfun,
4  ODEModel_t* gJac,
5  const int nx, const int nt,
6  const double *tspan,
7  const double *x0
8  const double *AbsTol, const double RelTol,
9  const void   *params,
10 double *t,
11 double *x)
```

Listing 1.8: C-interface for the modifed implicit methods.

```
1  <IRKMod>(
2  ODEModel_t* fun,
3  ODEModel_t* Jac,
4  ODEModel_t* gfun,
5  ODEModel_t* gJac,
6  const int nx, const int nt,
7  const double *tspan,
8  const double *x0
9  const double *AbsTol, const double RelTol,
10 const void   *params,
11 double *t,
12 double *x)
```

As for the `Matlab` interfaces, `<ERK>` is either the Euler method, RK4, RKF45 or DOPRI54. `<IRK>` is ESDIRK23 and, `<ERKMod>` and `<IRKMod>` are the modified versions of the above.

The `C` implementations have somewhat the same input and output as the `Matlab` versions, however, the function handles in `Matlab` are implemented as function pointers in `C`. These are of the type `ODEModel_t`, which is implemented as

Listing 1.9: ODEModel_t type.

```
1 typedef void ODEModel_t(
2 const double t, const double *x,
3 const void *params,
4 double *f);
```

`t` is a time scalar, `x` is a pointer to an array, `params` is a pointer to an array of any type, containing parameters. This is used in the same way `varargin` is used in `Matlab`. `f` is a pointer to an array where the function evaluation is stored and is effectively the output of the function. The `const` in the types indicates that these variables are not going to be overwritten or changed during the function call. Moreover it is a useful way of indicating which is input and which is output.

`gfun` and `gJac` do not use t as $g(x(t))$ does not depend on time explicitly, however they do have `t` as input, such that they may be passed as the same type of function as `fun` and `Jac`.

`nx` is the same as n in (1.1) and (1.2) and is the number of variables. In `Matlab` this is omitted since it is `length(x0)`, however this feature is not available in `C`. Likewise for `nt`, which in `Matlab` is `length(tspan)`. If `nt` $= 2$, the method uses adaptive step size, and if it is larger, it uses `nt` steps with fixed step size.

Chapter 2

Runge-Kutta Methods

This Chapter introduces the subclasses of Runge-Kutta methods used for numerically approximating solutions to IVPs in Section 2.2. The IVPs have either of the two forms

$$\frac{\mathrm{d}}{\mathrm{d}t}g(x(t)) = f(t, x(t)), \quad x(t_0) = x_0,$$

and

$$\frac{\mathrm{d}}{\mathrm{d}t}x(t) = f(t, x(t)), \quad x(t_0) = x_0,$$

Sections 2.3 to 2.7 introduce the methods which are implemented in the Runge-Kutta Toolbox. All of these methods approximates the solution of IVPs in the form (1.2).

The modifications needed for the methods in the toolbox to approximate solutions to (1.1) are dicussed in Section 2.8.

The modified methods and ESDIRK23 require Newton iterations, which are described in Section 2.9.

2.1 Introduction of Numerical Methods for Initial Value Problems

The simplest method for approximating solutions to initial value problems of the form, (1.2) is the forward Euler method, also known as the explicit Euler method. This may also be the most intuitive to derive. The idea is based on the expression of the forward derivative as a limit.

$$\frac{\mathrm{d}}{\mathrm{d}t}x(t) = \lim_{h\to\infty} \frac{x(t+h) - x(t)}{h} = f(t, x(t)).$$

The forward Euler method simply comes from exchanging this limit with an adequately small fixed step size, h. This gives

$$\frac{x(t+h) - x(t)}{h} \approx f(t, x(t))$$

such that

$$x(t+h) \approx x(t) + hf(t, x(t)).$$

The Euler method is written as the step update

$$x_{n+1} = x_n + hf(t_n, x_n),$$

where x_n is the approximation of $x(t_n)$ and t_n is the n'th time value. This will approximate the solution when applied repeatedly from some initial time with an initial condition, up to some predefined end time. Each step depends only on the previous step and the right-hand-side function.

The initial value problem is solved over a finite time interval which is split up into a number of discrete time values in which the solution, $x(t)$, is approximated. One can either choose a number of equidistant time values, a fixed step size or an arbitrary set of time values in which one wants an approximation. The time values may also be picked by a step size controller which calculates each step size based on an error estimate.

The forward Euler method is an example of an explicit one-step one-stage method. An explicit method approximates the solution using only approximated solution values at earlier times. A method can also be implicit which means that the expression for the step x_{n+1} also depends on x_{n+1} itself. Take for example the backward Euler method which is defined as

$$x_{n+1} = x_n + hf(t_{n+1}, x_{n+1}). \tag{2.1}$$

In general, the function $f(t, x(t))$ is nonlinear and cannot be solved for x_{n+1} analytically. The step may instead be obtained by using Newton iterations to

approximate the solution, x_{n+1}, to (2.1). This can be done quite effectively since the initial guess may be picked as the approximation in the previous step which, for small step sizes, will be relatively close to the approximation in the following step. These Newton iterations are also used in the modified methods.

Depending on the nature of the particular problem it may be very effective to use an implicit method over an explicit. These problems are referred to as stiff problems. A stiff problem is characterised by LeVeque [2007] as one where $\frac{\partial}{\partial x} f(t, x(t))$ is much larger in norm value than $\frac{\mathrm{d}}{\mathrm{d}t} x(t)$. One way of checking for this is to calculate the stiffness ratio, which is the ratio between the largest eigenvalue of the Jacobian matrix $f'(t, x(t))$ and the minimum eigenvalue.

$$\frac{\max |\lambda_p|}{\min |\lambda_p|}$$

If this value is large the problem may very well be stiff, however there is no implication since a scalar problem always has a stiffness ratio of one but may also be stiff. Likewise, the value may be large even though the problem is not very stiff at all.

Elden [2010] defines a problem as stiff if the solution contains both slow and very fast processes. This could be a system describing chemical kinetics in which some chemical reactions are much faster than others.

2.2 Subclasses of Runge-Kutta Methods

The numerical methods used in this project are of the class known as Runge-Kutta methods. These are stage based single step methods. A general Runge-Kutta method with r stages has the form

$$T_i = t_n + c_i h \tag{2.2a}$$

$$X_i = x_n + h \sum_{j=1}^{r} a_{ij} f\left(T_j, X_j\right) \tag{2.2b}$$

$$x_{n+1} = x_n + h \sum_{j=1}^{r} b_j f\left(T_j, X_j\right), \tag{2.2c}$$

where $i = 1 \ldots r$ and the coefficients a_{ij}, b_j and c_j are specific for each method. These coefficients are usually collected in a scheme called a Butcher tableau

$$\begin{pmatrix} 0 & 0 & 0 \\ a_{21} & 0 & 0 \\ a_{31} & a_{32} & 0 \end{pmatrix} \quad \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \quad \begin{pmatrix} a_{11} & 0 & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \quad \begin{pmatrix} \gamma & 0 & 0 \\ a_{21} & \gamma & 0 \\ a_{31} & a_{32} & \gamma \end{pmatrix}$$

(a) ERK (b) FIRK (c) DIRK (d) SDIRK

$$\begin{pmatrix} 0 & 0 & 0 \\ a_{21} & \gamma & 0 \\ a_{31} & a_{32} & \gamma \end{pmatrix}$$

(e) ESDIRK

Figure 2.1: The different subclasses of Runge-Kutta methods.

which has the general form.

$$\begin{array}{c|c} \mathbf{c} & \mathbf{A} \\ \hline & \mathbf{b} \end{array} = \begin{array}{c|ccc} c_1 & a_{11} & \cdots & a_{1r} \\ \vdots & \vdots & & \vdots \\ c_r & a_{r1} & \cdots & a_{rr} \\ \hline & b_1 & \cdots & b_r \end{array} . \tag{2.3}$$

Here $\mathbf{A}$ is a square matrix and $\mathbf{b}$, $\mathbf{c}$ are vectors. There are several subclasses of Runge-Kutta methods, all specified by the structure of the $\mathbf{A}$ matrix. These will be described in detail in the following Subsections. The $\mathbf{A}$ matrices for 3-stage cases are shown in Figure 2.1.

As will be discussed in a later Section, the local truncation error may be estimated as the difference between an approximationof order p and one of order $p+1$. When this is done, the Butcher tableau is extended in the following way, where $\hat{b}$ are the coefficient for the higher order method.

$$\begin{array}{c|c} \mathbf{c} & \mathbf{A} \\ \hline & \mathbf{b} \\ \hline & \hat{\mathbf{b}} \end{array} = \begin{array}{c|ccc} c_1 & a_{11} & \cdots & a_{1r} \\ \vdots & \vdots & & \vdots \\ c_r & a_{r1} & \cdots & a_{rr} \\ \hline & b_1 & \cdots & b_r \\ \hline & \hat{b}_1 & \cdots & \hat{b}_r \end{array} . \tag{2.4}$$

The Butcher Tableaus for the methods implemented in the Runge-Kutta Toolbox are listed in Sections 2.3 through 2.7.

2.2.1 Explicit Runge-Kutta Methods

The explicit Runge-Kutta methods, or ERK methods, are the simplest to implement since each stage depends only on previous stages and there is no need for solving nonlinear equations. These have the form

$$T_i = t_n + c_i h \tag{2.5a}$$

$$X_i = x_n + h \sum_{j=1}^{i-1} a_{ij} f\left(T_j, X_j\right) \tag{2.5b}$$

$$x_{n+1} = x_n + h \sum_{j=1}^{r} b_j f\left(T_j, X_j\right), \tag{2.5c}$$

where the index in the sum in the expression for X_i now only runs from 1 to $i-1$.

2.2.2 Fully Implicit Runge-Kutta Methods

The fully implicit Runge-Kutta methods are the most general since any coefficient of the $\mathbf{A}$ matrix may be non-zero, which means that every internal stage, may depend on all of the other internal stages, including itself.

This means that in each stage, a system of nr nonlinear equations has to be solved, where n is the dimension of the problem and r is the number of stages. The Fully Implicit Runge-Kutta methods, or FIRK for short, have the form shown in (2.2).

2.2.3 Diagonally Implicit Runge-Kutta Methods

These methods are characterised by having zero elements in the strictly upper triagonal part of the $\mathbf{A}$ matrix. Hence each stage depends only on the previous stages and itself. This means that a sequence of r implicit systems, each of size n needs to be solved, rather than nr for FIRK methods. The diagonally implicit

Runge-Kutta methods, or DIRK methods, have the form

$$T_i = t_n + c_i h \tag{2.6a}$$

$$X_i = x_n + h \sum_{j=1}^{i} a_{ij} f\left(T_j, X_j\right) \tag{2.6b}$$

$$x_{n+1} = x_n + h \sum_{j=1}^{r} b_j f\left(T_j, X_j\right). \tag{2.6c}$$

The internal stages may be written as

$$\begin{aligned} X_i &= x_n + h \sum_{j=1}^{i} a_{ij} f\left(T_j, X_j\right) \\ &= h a_{ii} f\left(T_i, X_i\right) + \psi_i, \end{aligned}$$

where

$$\psi_i = x_n + h \sum_{j=1}^{i-1} a_{ij} f\left(T_j, X_j\right).$$

This ψ_i need only to be calculated once for each step. The method now uses Newton iterations, as described in Section 2.9, where the residual function for the i'th stage is

$$R_i(X_i) = X_i - h a_{ii} f\left(T_i, X_i\right) - \psi_i = 0.$$

and the Jacobian of this residual function is

$$\frac{\partial}{\partial x} R_i(X_i) = I - h a_{ii} \frac{\partial}{\partial x} f\left(T_i, X_i\right),$$

where $I \in \mathbb{R}^{n \times n}$ is the identity matrix. Each Newton iteration is then calculated as

$$X_i^{k+1} = X_i^k - \Delta X_i^k,$$

where ΔX_i is the solution to the system

$$\left[\frac{\partial}{\partial x} R_i(X_i^0)\right] \Delta X_i^k = R_i(X_i^k). \tag{2.7}$$

Notice that the Jacobi matrix is evaluated in the initial guess rather than in the current iteration. This is discussed further in Section 2.9.

2.2.4 Singly Diagonal Implicit Runge-Kutta Methods

This subclass is characterized by all the diagonal elements in the **A** matrix being identical. When solving the system of equations in (2.7), LU-factorizations are used. For DIRK methods this has to be done for every stage. However, for the singly diagonal implicit methods, or SDIRK methods for short, the LU-factorization may be reused for every stage, which lowers the computational cost. These methods have the form

$$T_i = t_n + c_i h \tag{2.8a}$$

$$X_i = x_n + h\sum_{j=1}^{i-1} a_{ij} f\left(T_j, X_j\right) + h\gamma f\left(T_i, X_i\right) \tag{2.8b}$$

$$x_{n+1} = x_n + h\sum_{j=1}^{r} b_j f\left(T_j, X_j\right). \tag{2.8c}$$

2.2.5 Explicit Singly Diagonal Implicit Runge-Kutta Methods

The ESDIRK methods have the same properties as the SDIRK methods, except that the first stage is the current step, i.e. $a_{11} = 0$. The remaining elements in the diagonal are still identical. Since the first stage is also the current step, the evaluation of the right hand side, in the previous step, may be reused. The form is

$$T_1 = t_n \tag{2.9a}$$

$$X_1 = x_n \tag{2.9b}$$

$$T_i = t_n + c_i h, \quad i \geq 2 \tag{2.9c}$$

$$X_i = x_n + h\sum_{j=1}^{i-1} a_{ij} f\left(T_j, X_j\right) + h\gamma f\left(T_i, X_i\right), \quad i \geq 2 \tag{2.9d}$$

$$x_{n+1} = x_n + h\sum_{j=1}^{r} b_j f\left(T_j, X_j\right). \tag{2.9e}$$

2.3 Explicit Euler

The explicit Euler method is a one-stage first order ERK method. The Butcher Tableau for the explicit Euler method is,

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

This Butcher Tableau translates into the simple equation

$$x_{n+1} = x_n + hf(t_n, x_n)$$

2.4 The Classical Runge-Kutta Method

This method, RK4, is a four-stage, fourth order method. Its Butcher Tableau is

$$\begin{array}{c|cccc} 0 & & & & \\ \frac{1}{2} & \frac{1}{2} & & & \\ \frac{1}{2} & 0 & \frac{1}{2} & & \\ 1 & 0 & 0 & 1 & \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array}$$

The RK4 method has been used widely in the precomputer era, as the coefficients are simple enough to compute the approximations by hand.

It may be noticed that each stage depends only on the previous stage. This is not a general feature of the explicit Runge-Kutta methods.

2.5 The Runge-Kutta-Fehlberg Method

RKF45 is an embedded ERK method, which means that it uses embedded error estimation. The method uses a fourth- and fifth-order method in the embedded error estimation and it has six stages which is the minimum number of stages

needed for a fifth order method. It has the extended Butcher Tableau

$$\begin{array}{c|cccccc}
0 & & & & & & \\
\frac{1}{4} & \frac{1}{4} & & & & & \\
\frac{3}{8} & \frac{3}{32} & \frac{9}{32} & & & & \\
\frac{12}{13} & \frac{1932}{2197} & -\frac{7200}{2197} & \frac{7296}{2197} & & & \\
1 & \frac{439}{216} & -8 & \frac{3680}{513} & -\frac{845}{4104} & & \\
\frac{1}{2} & -\frac{8}{27} & 2 & -\frac{3544}{2565} & \frac{1859}{4104} & -\frac{11}{40} & \\
\hline
 & \frac{25}{216} & 0 & \frac{1408}{2565} & \frac{2197}{4104} & -\frac{1}{5} & 0 \\
\hline
 & \frac{16}{135} & 0 & \frac{6656}{12825} & \frac{28561}{56430} & -\frac{9}{50} & \frac{2}{55}
\end{array}$$

The error estimate simply uses the difference between the b and the $\hat{b}$ coefficients, rather than calculating both approximations.

2.6 The Dormand-Prince Method

DOPRI54 is, like the RKF45 method, an embedded ERK method. It also uses Runge-Kutta methods of orders four and five, however, it has seven stages, one more than RKF45. It has the following Butcher Tableau

$$\begin{array}{c|ccccccc}
0 & & & & & & & \\
\frac{1}{5} & \frac{1}{5} & & & & & & \\
\frac{3}{10} & \frac{3}{40} & \frac{9}{40} & & & & & \\
\frac{4}{5} & \frac{44}{45} & -\frac{56}{15} & \frac{32}{9} & & & & \\
\frac{8}{9} & \frac{19372}{6561} & -\frac{25360}{2187} & \frac{64448}{6561} & -\frac{212}{729} & & & \\
1 & \frac{9017}{3168} & -\frac{355}{33} & \frac{46732}{5247} & \frac{49}{176} & -\frac{5103}{18656} & & \\
1 & \frac{35}{384} & 0 & \frac{500}{1113} & \frac{125}{192} & -\frac{2187}{6784} & \frac{11}{84} & \\
\hline
 & \frac{5179}{57600} & 0 & \frac{7571}{16695} & \frac{393}{640} & -\frac{92097}{339200} & \frac{187}{2100} & \frac{1}{40} \\
\hline
 & \frac{35}{384} & 0 & \frac{500}{1113} & \frac{125}{192} & -\frac{2187}{6784} & \frac{11}{84} & 0
\end{array}$$

As can be seen, the coefficients for the fifth order method are identical to those in the seventh stage. This means that the function evaluation $f(T_7, X_7)$ used in the error estimate, can be reused in the following step as $f(T_1, X_1)$, which means that DOPRI54 needs no more function evaluations than RKF45, even though it has one more stage.

2.7 ESDIRK23

ESDIRK23 is also an embedded method, which uses three-stage implicit Runge-Kutta methods of order two and three. It has the Butcher Tableau

$$\begin{array}{c|ccc} 0 & & & \\ 2\gamma & \gamma & \gamma & \\ 1 & \frac{1-\gamma}{2} & \frac{1-\gamma}{2} & \gamma \\ \hline & \frac{1-\gamma}{2} & \frac{1-\gamma}{2} & \gamma \\ \hline & \frac{6\gamma-1}{12\gamma} & \frac{1}{12\gamma(1-2\gamma)} & \frac{1-3\gamma}{3\cdot(1-2\gamma)} \end{array}$$

where $\gamma = 1 - \frac{1}{\sqrt{2}}$. As for DOPRI54, the last stage is also the approximation in the subsequent step. The residual function for each internal stage is

$$R_i(X_i) = X_i - h\gamma f(T_i, X_i) - \psi_i,$$

where

$$\begin{aligned} \psi_2 &= x_n + ha_{21}f(T_1, X_1) \\ \psi_3 &= x_n + h(a_{31}f(T_1, X_1) + a_{32}f(T_2, X_2)). \end{aligned}$$

2.8 Modified Runge-Kutta Methods

Using Runge-Kutta methods to approximate the solution to IVPs in the form

$$\frac{\mathrm{d}}{\mathrm{d}t}g(x(t)) = f(t, x(t)), \quad x(t_0) = x_0,$$

requires a little more work, in the sense that each stage X_i needs to be solved for, using Newton iterations. A general Runge-Kutta method for (1.1) with r stages has the form

$$T_i = t_n + c_i h \tag{2.10a}$$

$$G_i = g_n + h \sum_{j=1}^{r} a_{ij} f\left(T_j, X_j\right) \tag{2.10b}$$

$$g(X_i) = G_i \tag{2.10c}$$

$$g_{n+1} = x_n + h \sum_{j=1}^{r} b_j f\left(T_j, X_j\right) \tag{2.10d}$$

$$g(x_{n+1}) = g_{n+1}, \tag{2.10e}$$

where g_n is the approximation of $g(x(t_n))$. For explicit methods, Newton iterations are then used to solve $g(X_i) = G_i$ and $g(x_{n+1}) = g_{n+1}$ for X_i and x_{n+1} respectively.

For implicit methods, which already require Newton iterations to obtain each stage, the difference lies in the residual function, and the additional evaluations of $g(x(t))$ and $\frac{\partial}{\partial x}g(x(t))$. For these modified implicit methods, the work required for each step may not be more than for the unmodified implicit methods.

The procedure of Newton iterations is discussed further in Section 2.9.

2.9 Newton Iterations

This Section covers the Newton iterations used in ESDIRK23 and in the modified methods. See Chapter 4 of [Elden, 2010] for more on numerical methods for solving nonlinear equations. The system of equations, which requires to be solved in the modified methods is,

$$g(x(t)) = g, \tag{2.11}$$

where g is a computed approximation of $g(x(t))$, and this expression needs to be solved for $x(t)$. Newtons method is used to find roots of a non-linear function, hence, solving the problem

$$R(x(t)) = 0.$$

In the context of implicit methods, this is called the residual function. These iterations use an initial guess, x_0, from which a new guess is computed as,

$$x^{k+1} = x^k - \Delta x^k, \tag{2.12}$$

where Δx^k is the solution to the linear system of equations,

$$\frac{\partial}{\partial x} R(x^k) \Delta x^k = R(x^k).$$

This procedure is repeated until the norm of the residual function is sufficiently small. Notice that superscript indicates step in Newton iterations and that subscript is step in the numerical methods. Newton iterations may be used to solve the non-linear system of equations in (2.11) by setting

$$\begin{aligned} R(x(t)) &= g(x(t)) - g, \\ \frac{\partial}{\partial x} R(x(t)) &= \frac{\partial}{\partial x} g(x(t)). \end{aligned}$$

The iteration update may be approximated by letting Δx_k be the solution to,

$$\frac{\partial}{\partial x} R(x^0) \Delta x^k = R(x^k). \tag{2.13}$$

Provided that the initial guess x_0 is reasonably close to the real solution, the approximation may work very well.

Using this approximation reduces the number of LU-factorizations to one per step and only the backward and forward substitutions are required in each step. The procedure of Newton's method used in this project is shown in Algorithm 1.

Data: $x^0, R(x(t))$
Result: x_{n+1}, α
initial guess, x^0;
evaluate $R\left(x^0\right)$;
evaluate $\frac{\partial}{\partial x}R\left(x^0\right)$;
check for convergence, $||R\left(x^0\right)||_\infty < \tau$;
set $k = \alpha = 0$;
while *not converged, diverged and convergence is not slow* **do**
    solve equation (2.13) for Δx^k;
    calculate x^{k+1} by equation (2.12);
    evaluate $R\left(x^{k+1}\right)$;
    $\alpha = \max\left(\alpha, \frac{||R\left(x^{k+1}\right)||_\infty}{||R(x^k)||_\infty}\right)$;
    check for convergence, $R\left(x^{k+1}\right) < \tau$;
    check for slow convergence, $k > k_{max}$;
    check for divergence, $\alpha > 1$;
    increment k by 1;
end
set $x_{n+1} = x^k$;

Algorithm 1: Newton iterations.

2.10 Summary

This Chapter has described the five subclasses of Runge-Kutta methods, explicit methods, implicit methods, diagonally implicit methods, singly diagonally implicit methods and explicit singly diagonally explicit methods and their advantages.

The methods which are implemented in the Runge-Kutta Toolbox has been described in the sense of their Butcher Tableaus and other properties the methods may have.

Generally, Runge-Kutta methods are meant for IVPs in the form (1.2), and the modifications required for the methods to approximate solutions to IVPs of the form (1.1) have been described in Section 2.8.

The modified methods, and ESDIRK23, require Newton iterations which have been described in Section 2.9.

Chapter 3

Adaptive Step Size

This Chapter concerns step size control for numerical methods for approximating solutions to IVPs of the form (1.1) and (1.2).

Sections 3.1 and 3.2 discuss two methods for estimating the local truncation error and Section 3.3 describes the norm which is used in the Runge-Kutta Toolbox.

Sections 3.4 and 3.5 discuss two step size controllers and the use of these is presented in Section 3.6 where to algorithms for updating the step and step size are described.

3.1 Step Doubling

This section describes the step doubling procedure for an explicit Runge-Kutta method approximating the solution to an IVP of the form (1.2). The concept is to take a full step of step size h, and a double step, consisting of two steps, of step size $h/2$. Let the full step be

$$\begin{aligned} T_i &= t_n + c_i h \\ X_i &= x_n + h \sum_{j=1}^{i-1} a_{ij} f\left(T_j, X_j\right) \\ x_{n+1} &= x_n + h \sum_{j=1}^{r} b_j f\left(T_j, X_j\right). \end{aligned}$$

Then the double step is

$$\begin{aligned} \hat{T}_i^{n+\frac{1}{2}} &= t_n + c_i \frac{h}{2} \\ \hat{X}_i^{n+\frac{1}{2}} &= x_n + \frac{h}{2} \sum_{j=1}^{i-1} a_{ij} f\left(\hat{T}_j^{n+\frac{1}{2}}, \hat{X}_j^{n+\frac{1}{2}}\right) \\ \hat{x}_{n+\frac{1}{2}} &= x_n + \frac{h}{2} \sum_{j=1}^{r} b_j f\left(\hat{T}_j^{n+\frac{1}{2}}, \hat{X}_j^{n+\frac{1}{2}}\right) \end{aligned}$$

$$\begin{aligned} \hat{T}_i^{n+1} &= \left(t_n + \frac{h}{2}\right) + c_i \frac{h}{2} \\ \hat{X}_i^{n+1} &= \hat{x}_{n+1}^{n+\frac{1}{2}} + \frac{h}{2} \sum_{j=1}^{i-1} a_{ij} f\left(\hat{T}_j^{n+1}, \hat{X}_j^{n+1}\right) \\ \hat{x}_{n+1} &= \hat{x}_{n+1}^{n+\frac{1}{2}} + \frac{h}{2} \sum_{j=1}^{r} b_j f\left(\hat{T}_j^{n+1}, \hat{X}_j^{n+1}\right). \end{aligned}$$

The local truncation error of x_{n+1} may then be estimated as

$$e = x_{n+1} - \hat{x}_{n+1}. \tag{3.3}$$

This is described more concisely in Algorithm 2. This procedure uses three steps, but only requires two function evaluations. It is, however, more expensive than the embedded error estimate, described in Section 3.2.

If using an implicit Runge-Kutta method or a modified method, the three calculations require Newton iterations for each stage, which means that three steps may be very expensive compared to a single step.

1 evaluate $f(t_n, x_n)$;
2 calculate x_{n+1} using step size, h;
3 calculate $\hat{x}_{n+\frac{1}{2}}$ using step size, $\frac{h}{2}$;
4 evaluate $f(t_n + \frac{h}{2}, \hat{x}_{n+\frac{1}{2}})$;
5 calculate $\hat{x}_{n+1}$ using step size $\frac{h}{2}$;
6 estimate error as $e = x_{n+1} - \hat{x}_{n+1}$;

Algorithm 2: Step doubling.

3.2 Embedded Error Estimation

Embedded Runge-Kutta methods use embedded error estimation. This Section describes the procedure for explicit Runge-Kutta methods for approximating the solution to an IVP of the form (1.2). The concept is similar to that of step doubling. For a p order Runge-Kutta method, the local truncation error may be estimated using a $p+1$ order Runge-Kutta method. The advantage of this method over step doubling is that it is essentially free, since the higher order method is designed to have the same coefficients, A and c.

$$
\begin{aligned}
T_i &= t_n + c_i h \\
X_i &= x_n + h \sum_{j=1}^{i-1} a_{ij} f\left(T_j, X_j\right) \\
x_{n+1} &= x_n + h \sum_{j=1}^{r} b_j f\left(T_j, X_j\right), \\
\hat{x}_{n+1} &= x_n + h \sum_{j=1}^{r} \hat{b}_j f\left(T_j, X_j\right)
\end{aligned}
$$

Here, x_{n+1} is the p'th order approximation and $\hat{x}_{n+1}$ is the $p+1$'th order approximation. The only difference between these two methods is the b coefficients, i.e. $b_j \neq \hat{b}_j$ for at least one value of j. Like for step doubling, the error estimate of x_{n+1} is

$$
\begin{aligned}
e = x_{n+1} - \hat{x}_{n+1} &= \left(x_n + h \sum_{j=1}^{r} b_j f\left(T_j, X_j\right) \right) - \left(x_n + h \sum_{j=1}^{r} \hat{b}_j f\left(T_j, X_j\right) \right) \\
&= h \sum_{j=1}^{r} \left(b_j - \hat{b}_j \right) f\left(T_j, X_j\right) \\
&= h \sum_{j=1}^{r} d_j f\left(T_j, X_j\right). \qquad (3.5)
\end{aligned}
$$

There is no need for calculating $\hat{x}_{n+1}$ since the error estimate depends only on the coefficients d, and the function evaluations, which have already been obtained during the calculations of the stages, the error estimate is simply an inexpensive sum.

In practice one may use $\hat{x}_{n+1}$ as the advancing method, since for small step sizes, the LTE is assumed to be even smaller for this approximation. Then the calculation of x_{n+1} may be omitted and the expense is the same.

3.3 Maximum Norm

The Runge Kutta Toolbox uses the maximum-norm, which requires both an absolute and a relative tolerance. In the implementation of the Runge-Kutta Toolbox methods, these tolerances are supplied by the user.

The norm is used, both for the error estimates, described in the previous Sections, and in the Newton iterations. The norm of the error estimates is shown below, with the terminology of the error estimation Sections.

$$||e||_\infty = \max_{i\in[1,n]} \left\{ \frac{|e_i|}{AbsTol_i + |(x_{n+1})_i| \cdot RelTol} \right\}. \tag{3.6}$$

e is the error estimate for x_{n+1} and e_i is the i'th component of e. Likewise for the other vector components. $AbsTol$ is a user-specified absolute tolerance vector and $RelTol$ is a user-specified relative tolerance scalar.

In the Newton iterations the norm is

$$||R(x^k)\,||_\infty = \max_{i\in[1,n]} \left\{ \frac{|R(x^k)_i|}{AbsTol_i + |(g_{n+1})_i| \cdot RelTol} \right\}, \tag{3.7}$$

where g_{n+1} is the approximation of $g(x(t_{n+1}))$, and $R(x^k)$ is the residual function evaluated in the current iteration.

3.4 Asymptotic Controller

The asymptotic controller is derived from the expression of the local truncation error. For sufficiently small step sizes h, the local truncation error is dominated by the leading term, which also determines the order of a method. For a p'th order method the local truncation error, E, is approximately

$$E \approx F(x_{n+1})h^p, \tag{3.8}$$

where $F(x_{n+1})$ is some function dependent on the current step, but not on the step size. This is usually some function including derivatives of $x(t)$, evaluated in t_{n+1}. h is the step size. Strictly, this is only valid in the asymptotic limit where h goes to zero.

Assume that the step size h was used to take one step and for estimating the local truncation error. Given some tolerance, ϵ, we want to find the step size $\hat{h}$, which would have produced an estimated local truncation error, ϵ, hence it should satisfy,

$$\epsilon = F(x_{n+1})\hat{h}^p. \tag{3.9}$$

The ratio between these two error estimates is,

$$\frac{\epsilon}{E} \approx \frac{F(x_{n+1})\hat{h}^p}{F(x_{n+1})h^p} = \left(\frac{\hat{h}}{h}\right)^p. \tag{3.10}$$

The step size, $\hat{h}$ which would have produced an error estimate of ϵ is

$$\hat{h} = h\left(\frac{\epsilon}{E}\right)^{\frac{1}{p}}. \tag{3.11}$$

If the solution, $x(t)$, is smooth in a neighborhood around x_{n+1} it may be expected that using this step size in the subsequent step, will satisfy the tolerance, ϵ. In the sense of current and next step size, (3.11) is,

$$h_{n+1} = h_n\left(\frac{\epsilon}{E_{n+1}}\right)^{\frac{1}{p+1}} \tag{3.12}$$

where h_n is the previous step size, ϵ is the tolerance, usually 0.8 or 0.9. E_{n+1} is the estimated error of the next step and p is the order of the method.

3.5 PI Controller

The PI step size controller is a little more advanced, and takes into account both the current and the previous step size. For explicit methods the step size update is calculated as,

$$h_{n+1} = h_n \left(\frac{\epsilon}{E_{n+1}}\right)^{k_I} \left(\frac{E_n}{E_{n+1}}\right)^{k_p} \tag{3.13}$$

$$k_I = \frac{0.4}{p+1} \tag{3.14}$$

$$k_p = \frac{0.3}{p+1}. \tag{3.15}$$

For implicit methods the PI step update is

$$h_{n+1} = h_n \left(\frac{h_n}{h_{n-1}}\right) \left(\frac{\epsilon}{E_{n+1}}\right)^{k_I} \left(\frac{E_n}{E_{n+1}}\right)^{k_p} \tag{3.16}$$

$$k_I = \frac{1}{p+1} \tag{3.17}$$

$$k_p = \frac{1}{p+1}. \tag{3.18}$$

This is different from the PI controller for explicit methods because of the factor $\left(\frac{h_n}{h_{n-1}}\right)$, and the numerator in k_I and k_p is 1 instead of 0.4 and 0.3, respectively. See [Engsig-Karup et al., 2012] for more step size controllers and norms, and more on error estimation.

3.6 Control Algorithms

3.6.1 Step Size Control for Explicit Runge Kutta Methods

For the unmodified explicit methods, the step size control is as shown in Algorithm 3.

```
1  if the error estimate is sufficiently small then
2  |  update step;
3  |  if first step then
4  |  |  adjust step size with asymptotic controller;
5  |  else
6  |  |  adjust step size with PI controller;
7  |  end
8  |  store error for use in next accepted step;
9  else
10 |  adjust step size with asymptotic controller;
11 end
```

Algorithm 3: Step size control for methods where approximations are obtained without any use of Newton iterations.

Since there are no Newton iterations, the step size control only depends on the error estimate. In case the error is too large or if the iteration is at its first step, the asymptotic step size controller is used. In the latter case, this is simply because there is no measure of the error in the previous step, which is needed in the PI step size controller.

Updating the step in line 2, means updating time as $t_{n+1} = t_n + h$, updating the approximation as either x_{n+1} or $\hat{x}_{n+1}$. Furthermore, evaluations of $f(t_{n+1}, x_{n+1})$ and $g(x_{n+1})$ may be updated too, if computed values may be reused.

3.6.2 Step Size Control for the Modified Euler's Method and ESDIRK23

For the methods which use Newton iterations, i.e. the modified methods and ESDIRK23, the step size control is as shown in Algorithm 4.

```
if all Newton iterations converged then
    if the error estimate is sufficiently small then
        update step;
        if first step then
            adjust step size with asymptotic controller;
        else
            adjust step size with PI controller;
        end
        store error for use in next accepted step;
    else
        adjust step size with asymptotic controller;
    end
    if α_ref / α < 1 then
        restrict step size with α_ref / α;
    end
else if any Newton iteration diverged then
    restrict step size with max(1/2, α_ref / α);
else
    if α_ref / α < 1 then
        restrict step size with min(1/2, α_ref / α);
    else
        restrict step size with 1/2;
    end
end
```

Algorithm 4: Step size control for methods where approximations are obtained using Newton iterations.

If *all Newton iterations converged*, the procedure is very similar to that of Algorithm 3, except that the step size is restricted according to the ratio $\frac{\alpha_{ref}}{\alpha}$ if this is smaller than 1. In the case where all Newton iterations converged, the ratio can be no smaller than α_{ref}.

α_{ref} is effectively any value between 0.2 and 0.5, the choice used in the implementations is 0.4. α is, as described in Section 2.9, the maximum ratio between the residual in two subsequent Newton iterations.

The asymptotic controller is the same for both the modified and the unmodified methods, and for both explicit and implicit methods. As mentioned in Section 3.5 the PI controller is different for the implicit methods, but as the asymptotic controller, it is the same for both the modified and unmodified methods.

If *any Newton iteration diverged* the step size is restricted by the smaller of $\frac{\alpha_{ref}}{\alpha}$ and $\frac{1}{2}$. If no Newton iterations converged or diverged, they all suffered from slow convergence, i.e. too many iterations. In this case the step size is restricted by the largest of $\frac{1}{2}$ and $\frac{\alpha_{ref}}{\alpha}$. The restriction may be no larger than 1.

As for the step size control algorithm for the unmodified explicit methods, the step update in line 3 is $t_{n+1} = t_n + h$, updating the approximation as either x_{n+1} or $\hat{x}_{n+1}$ and possibly function evaluations as well.

3.7 Summary

This Chapter has described the two methods of error estimation, step doubling and embedded error estimation. When using adaptive step size, the step update is accepted or failed based on the norm of the error, and the norm used has also been described. The step size may be adjusted in each step using a step size controller. The asymptotic controller and the PI controller are described and so is the entire procedure of updating the step and step size both for the unmodified explicit methods, and the modified methods and ESDIRK23.

The Euler method and RK4 use step doubling to estimate the error whereas RKF45, DOPRI54 and ESDIRK23 are embedded methods, which use embedded error estimation. The asymptotic controller is used to update the step size after the first step and if the error estimate of a step is too large in norm value. Otherwise the PI controller is used.

For the modified methods and ESDIRK23 the procedure of updating the step and step size also takes into account whether the Newton iterations converged, diverged or suffered from slow convergence. The step size is also restricted depending on the norm values of the residuals in the Newton iterations.

Chapter 4

Implementation of Numerical Methods

This Chapter describes the implementation of the Runge-Kutta methods described in Sections 2.3 through 2.7, which numerically approximate solutions to IVPs of the form

$$\frac{\mathrm{d}}{\mathrm{d}t}x(t) = f(t, x(t)), \quad x(t_0) = x_0,$$

and

$$\frac{\mathrm{d}}{\mathrm{d}t}g(x(t)) = f(t, x(t)), \quad x(t_0) = x_0,$$

The implementation of both the modified and unmodified versions of each method is described. The methods are similar in many ways, however each has details which sets it apart from the others.

4.1 Euler

Both the modified and the unmodified Euler method is implemented as shown in Algorithm 5. The unmodified method uses Algorithm 6 in line 7 and 12 and Algorithm 3 in line 15. The modified method uses the alternative.

Data: $f(t, x(t))$, $g(x(t))$, $\frac{\partial}{\partial x} g(x(t))$, initial and final time t_0 and t_f, number of steps N, initial conditions x_0, absolute tolerance, relative tolerance, parameters
Result: time and approximation vectors are returned or manipulated

```
1  initialization;
2  check if the method should use fixed step size;
3  if using fixed step size then
4  |  calculate step size, h = (t_f − t_0)/(N − 1);
5  |  for every step do
6  |  |  update time step;
7  |  |  calculate x_{n+1} using Algorithm 6 or 7;
8  |  end
9  else
10 |  while final time is not exceeded do
11 |  |  check if final time is exceeded by step size;
12 |  |  calculate x_{n+1} and x̂_{n+1} using Algorithm 2 and either 6 or 7;
13 |  |  estimate error as x_{n+1} − x̂_{n+1};
14 |  |  calculate norm of error using the norm (3.6);
15 |  |  update step and step size using either Algorithm 3 or 4;
16 |  end
17 end
18 return number of steps;
```

Algorithm 5: Algorithm for both the modified and unmodified Euler method.

If the method uses fixed step size, each step is simply calculated until the final time is reached. Each step uses one function evaluation. If the method is used with adaptive step size, the step doubling described in Algorithm 2, is used for estimating the error.

In the step update algorithm $\hat{x}_{n+1}$ is used as the approximation. In the modified Euler method, $g(\hat{x}_{n+1})$ is evaluated in the step function and updated together with $\hat{x}_{n+1}$. Note that the double step approximation is used as the advancing step.

4.1.1 The Unmodified Euler Step Function

The unmodified Euler method is the simplest of the methods in the Runge-Kutta Toolbox. Its step function is shown in Algorithm 6, and simply calculates the Euler step. Prior to each call of this function should be a right-hand-side evaluation. This function evaluation is omitted in the step function because it saves function evaluations when using double stepping. The double stepping could be put into each step function, but would be futile when using fixed step size, where no error estimate is needed.

Data: $f(t, x(t))$, $f(t_n, x_n)$, t_n, x_n, h
Result: x_{n+1}
1 $x_{n+1} = x_n + hf(t_n, x_n)$;

Algorithm 6: The unmodified Euler step function.

4.1.2 The Modified Euler Step Function

The modified Euler step function is shown in Algorithm 7. The α returned by the Newton iterations is also the one returned by the step function. When using double stepping, it is important to make sure whether all three Newton iterations converged or if any of them diverged, even though one may expect the two steps using half step size to converge if the full step did.

Data: $f(t, x(t))$, $g(x(t))$, $\frac{\partial}{\partial x} g(x(t))$,
$f(t_n, x_n), g(x_n), t_n, x_n, h, AbsTol, RelTol$
Result: x_{n+1}, g_{n+1}, α
1 calculate $g_{n+1} = g(x_n) + hf(t_n, x_n)$;
2 set $x^0 = x_n$;
3 set $R(x(t)) = g(x(t)) - g_{n+1}$;
4 use Algorithm 1 to obtain x_{n+1};

Algorithm 7: The modified Euler step function.

4.2 Classical Runge-Kutta

Both the modified and the unmodified classical Runge-Kutta method is implemented as shown in Algorithm 8. The unmodified method uses Algorithm 9 in line 9 and 14 and Algorithm 3 in line 17. The modified method uses the alternative.

Data: $f(t, x(t))$, $g(x(t))$, $\frac{\partial}{\partial x} g(x(t))$, initial and final time t_0 and t_f, number of steps N, initial conditions x_0, absolute tolerance, relative tolerance, parameters
Result: time and approximation vectors are returned or manipulated

```
1  initialization;
2  define A, b and c;
3  check if the method should use fixed step size;
4  if using fixed step size then
5  |  calculate step size, h = (t_f − t_0)/(N − 1);
6  |  for every step do
7  |  |  update time step;
8  |  |  evaluate f(t_n, x_n);
9  |  |  calculate x_{n+1} using Algorithm 9 or 10;
10 |  end
11 else
12 |  while final time is not exceeded do
13 |  |  check if final time is exceeded by step size;
14 |  |  calculate x_{n+1} and x̂_{n+1} using Algorithms 2 and either 9 or 10;
15 |  |  estimate error as x_{n+1} − x̂_{n+1};
16 |  |  calculate norm of error using the norm (3.6);
17 |  |  update step and step size using either Algorithm 3 or 4;
18 |  end
19 end
20 return number of steps;
```

Algorithm 8: Algorithm for both the modified and unmodified classical Runge-Kutta method.

Except for the calculation of x_{n+1} and $\hat{x}_{n+1}$, this method is essentially the same as the Euler method. For the modified RK4 method, the step update in Algorithm 4 also updates $\hat{x}_{n+1}$ and $g(\hat{x}_{n+1})$. Note that the double step approximation is used as the advancing step.

4.2.1 The Unmodified RK4 Step Function

For Euler's method the Butcher Tableau was so simple that it did not need any representation in the implementation. The Classical Runge-Kutta Method, or RK4 for short, however, has arrays representing the b and c vectors and the A-matrix. The step function is shown in Algorithm 9.

Although this may be implemented with an actual for-loop, writing out each stage explicitly saves a few calculations, whereever there are zero coefficients. We see that in one iteration where step doubling is used, the RK4 method uses 11 function evaluations whereas Euler's method used only two. The higher order of this method should make up for this by admitting larger steps.

Data: $f(t, x(t))$, $f(t_n, x_n)$, t_n, x_n, h, A, b, c
Result: x_{n+1}
set $X_1 = x_n$;
set $f_1 = f(t_n, x_n)$;
for $i = 2 \dots 4$ **do**
    calculate $X_i = x_n + h \sum_{j=1}^{i-1} a_{i,j} f_i$;
    evaluate $f_i = f(t_n + c_i h, X_i)$;
end
calculate $x_{n+1} = x_n + h \cdot \sum_{i=1}^{4} b_i f_i$;

Algorithm 9: The unmodified RK4 step function.

4.2.2 The Modified RK4 Step Function

The modified RK4 step function is shown in Algorithm 10. Special for the modified RK4 step function is that Newton iterations are used to obtain each internal stage. Each of these iterations use the previous stage as initial guess, and for the approximation x_{n+1} the last stage is used as initial guess. The residual function, $R(x(t))$ varies through the stages, however, the Jacobi matrix, $\frac{\partial}{\partial x} R(x(t))$, is the same for all Newton iterations in this step function.

Data: $f(t, x(t))$, $g(x(t))$, $\frac{\partial}{\partial x} g(x(t))$,
$f(t_n, x_n), g(x_n), t_n, x_n, h, AbsTol, RelTol, A, b, c$
Result: x_{n+1}, g_{n+1}, α
1 set $X_1 = x_n$;
2 set $f_1 = f(t_n, x_n)$;
3 **for** $i = 2 \dots 4$ **do**
4 calculate $G_i = g_n + h \sum_{j=1}^{i-1} a_{i,j} f_i$;
5 set $x^0 = X_{i-1}$;
6 set $R(x(t)) = g(x(t)) - G_i$;
7 use Algorithm 1 to obtain X_i and α_i;
8 evaluate $f_i = f(t_n + c_i h, X_i)$;
9 **end**
10 calculate $g_{n+1} = g_n + h \cdot \sum_{i=1}^{4} b_i f_i$;
11 set $x^0 = X_4$;
12 set $R(x(t)) = g(x(t)) - g_{n+1}$;
13 use Algorithm 1 to obtain x_{n+1};

Algorithm 10: The modified RK4 step function.

4.3 Runge-Kutta-Fehlberg

The modified and the unmodified Runge-Kutta-Fehlberg method is implemented as shown in Algorithm 11. The unmodified method uses Algorithm 12 in line 9 and 15 and Algorithm 3 in line 17. The modified method uses the alternative.

Unlike the Euler method and RK4, this is an embedded method using embedded error estimation. This is done in the step functions described in the Subsections below. For the modified RKF45 method, the step update in Algorithm 4 also updates $\hat{x}_{n+1}$ as well as $g(\hat{x}_{n+1})$. Note that the fifth order method is used as the advancing method.

4.3.1 The Unmodified RKF45 Step Function

Like RK4, this method has array representations of the Butcher Tableau, however this also needs representations of $\hat{b}$ and d used in the error estimate. The step function is implemented as shown in Algorithm 12. This is very similar to that of RK4, except for the last line which calculates the error estimate. Hence the error estimate is simply returned by the step function, and there is no need for step doubling. Because of this there is actually no need to have the right-hand-side evaluated outside the step function, however, for consistency, this is

Data: $f(t, x(t))$, $g(x(t))$, $\frac{\partial}{\partial x} g(x(t))$, initial and final time t_0 and t_f, number of steps N, initial conditions x_0, absolute tolerance, relative tolerance, parameters
Result: time and approximation vectors are returned or manipulated
initialization;
define A, b, $\hat{b}$, c and d;
check if the method should use fixed step size;
if *using fixed step size* **then**
    calculate step size, $h = \frac{t_f - t_0}{N-1}$;
    for *every step* **do**
        update time step;
        evaluate $f(t_n, x_n)$;
        calculate x_{n+1} using Algorithm 12 or 13;
    end
else
    while *final time is not exceeded* **do**
        check if final time is exceeded by step size;
        evaluate $f(t_n, x_n)$;
        calculate $\hat{x}_{n+1}$ and e using either Algorithm 12 or 13;
        calculate norm of error using the norm (3.6);
        update step and step size using either Algorithm 3 or 4;
    end
end
return number of steps;

Algorithm 11: Algorithm for both the modified and unmodified Runge-Kutta-Fehlberg method.

not changed.

In comparison RKF45 uses 6 function evaluations per iteration where RK4 used 11 and Euler used 2 because of the double stepping. It should also be noticed that the advancing step uses $\hat{b}$ instead of b, which means that the method is expected to be of fifth order.

Data: $f(t, x(t))$, $f(t_n, x_n), t_n, x_n, h, A, \hat{b}, c, d$
Result: $\hat{x}_{n+1}, e_{n+1}$
set $X_1 = x_n$;
set $f_1 = f(t_n, x_n)$;
for $i = 2 \dots 6$ **do**
  calculate $X_i = x_n + h \sum_{j=1}^{i-1} a_{i,j} f_i$;
  evaluate $f_i = f(t_n + c_i h, X_i)$;
end
calculate $\hat{x}_{n+1} = x_n + h \sum_{i=1}^{6} \hat{b}_i f_i$;
calculate $e_{n+1} = h \sum_{i=1}^{6} d_i \cdot f_i$;

Algorithm 12: The unmodified RKF45 step function.

4.3.2 The Modified RKF45 Step Function

The modified RKF45 is much like the modified RK4 and so is its step function which is shown in Algorithm 13.

Also for the RKF45 step function, Newton iterations are used to obtain each internal stage, and these also use the previous stage as initial guess and the last stage as initial guess for $\hat{x}_{n+1}$. Also in this step function the residual function, $R(x(t))$ varies through the stages, and the Jacobi matrix, $\frac{\partial}{\partial x} R(x(t))$, is the same for all Newton iterations in this step function. The embedded error estimate is still virtually free. Important to notice is that the higher order method is used as the advancing method.

Data: $f(t, x(t))$, $g(x(t))$, $\frac{\partial}{\partial x} g(x(t))$,
$f(t_n, x_n), g(x_n), t_n, x_n, h, AbsTol, RelTol, A, \hat{b}, c, d$
Result: $\hat{x}_{n+1}$, e_{n+1}, $\hat{g}_{n+1}$, α
1 set $X_1 = x_n$;
2 set $f_1 = f(t_n, x_n)$;
3 **for** $i = 2 \dots 6$ **do**
4 calculate $G_i = g_n + h \sum_{j=1}^{i-1} a_{i,j} f_i$;
5 set $x^0 = X_{i-1}$;
6 set $R(x(t)) = g(x(t)) - G_i$;
7 use Algorithm 1 obtain X_i;
8 evaluate $f_i = f(t_n + c_i h, X_i)$;
9 **end**
10 calculate $\hat{g}_{n+1} = g_n + h \sum_{i=1}^{6} \hat{b}_i f_i$;
11 set $x^0 = X_6$;
12 set $R(x(t)) = g(x(t)) - \hat{g}_{n+1}$;
13 use Algorithm 1 to obtain $\hat{x}_{n+1}$;
14 calculate $e_{n+1} = h \sum_{i=1}^{6} d_i \cdot f_i$;

Algorithm 13: The modified RKF45 step function.

4.4 Dormand-Prince

Both the modified and the unmodified Dormand-Prince method is implemented as shown in Algorithm 14. The unmodified method uses Algorithm 15 in line 9 and 14 and Algorithm 3 in line 16. The modified method uses the alternative.

Like RKF45, this is an embedded method using embedded error estimation. This is done in the step functions described in the Subsections below. For the modified DOPRI54 method, the step update in Algorithm 4 updates both $\hat{x}_{n+1}$, $f(t_{n+1}, \hat{x}_{n+1})$ and $g(\hat{x}_{n+1})$. Note that the fifth order method is used as the advancing method and that there are no function evaluations outside the step functions, except at the first step.

4.4.1 The Unmodified DOPRI54 Step Function

The Dormand-Prince method, DOPRI54, has array representations of A, b, $\hat{b}$, c and d from the Butcher Tableau just like RKF45 did. The DOPRI54 step function is implemented as shown in Algorithm 15. The step function does not need b or $\hat{b}$ as only the latter is used and it is also the last row of A. This

Data: $f(t, x(t))$, $g(x(t))$, $\frac{\partial}{\partial x} g(x(t))$, initial and final time t_0 and t_f, number of steps N, initial conditions x_0, absolute tolerance, relative tolerance, parameters
Result: time and approximation vectors are returned or manipulated

```
initialization;
define A, b, b̂, c and d;
evaluate f(t_0, x_0);
check if the method should use fixed step size;
if using fixed step size then
    calculate step size, h = (t_f − t_0)/(N − 1);
    for every step do
        update time step;
        calculate x_{n+1} using Algorithm 15 or 16;
    end
else
    while final time is not exceeded do
        check if final time is exceeded by step size;
        calculate x̂_{n+1} and e using either Algorithm 15 or 16;
        calculate norm of error using the norm (3.6);
        update step and step size using either Algorithm 3 or 4;
    end
end
return number of steps;
```

Algorithm 14: Algorithm for both the modified and unmodified Dormand-Prince method.

also means that the output $f(t_{n+1}, \hat{x}_{n+1})$ can be input in the next iteration as $f(t_n, x_n)$. Hence DOPRI54 uses six function evaluations per iteration just like RKF45, even though DOPRI54 has one more stage.

Data: $f(t, x(t))$, $f(t_n, x_n)$, t_n, x_n, h, A, c, d
Result: $\hat{x}_{n+1}$, e_{n+1}, $f(t_{n+1}, \hat{x}_{n+1})$
set $X_1 = x_n$;
set $f_1 = f(t_n, x_n)$;
for $i = 2 \dots 7$ **do**
    calculate $X_i = x_n + h \sum_{j=1}^{i-1} a_{i,j} f_i$;
    evaluate $f_i = f(t_n + c_i h, X_i)$;
end
set $\hat{x}_{n+1} = X_7$;
set $f(t_{n+1}, \hat{x}_{n+1}) = f_7$;
calculate $e_{n+1} = h \cdot \sum_{i=1}^{7} d_i f_i$;

Algorithm 15: DOPRI54 step function.

4.4.2 The Modified DOPRI54 Step Function

The modified DOPRI54 step function is shown in Algorithm 16. DOPRI54 has one more internal stage than RKF45, however it does not need to solve $g(\hat{x}_{n+1}) = \hat{g}_{n+1}$. Like for RKF45, the initial guess for each stage is the previous stage. DOPRI54 also uses the higher order method as the advancing method. It also outputs $\hat{f}_{n+1}$.

Data: $f(t, x(t))$, $g(x(t))$, $\frac{\partial}{\partial x} g(x(t))$, $f(t_n, x_n)$, $g(x_n)$, t_n, x_n, h, $AbsTol$, $RelTol$, A, c, d
Result: $\hat{x}_{n+1}$, e_{n+1}, $\hat{f}_{n+1}$, $\hat{g}_{n+1}$, α
1 set $X_1 = x_n$;
2 set $f_1 = f(t_n, x_n)$;
3 **for** $i = 2 \dots 7$ **do**
4 calculate $G_i = g_n + h \sum_{j=1}^{i-1} a_{i,j} f_i$;
5 set $x^0 = X_{i-1}$;
6 set $R(x(t)) = g(x(t)) - G_i$;
7 use Algorithm 1 to obtain X_i;
8 evaluate $f_i = f(t_n + c_i h, X_i)$;
9 **end**
10 set $\hat{x}_{n+1} = X_7$;
11 set $f_{n+1} = f_7$;
12 set $\hat{g}_{n+1} = G_7$;
13 calculate $e_{n+1} = h \cdot \sum_{i=1}^{7} d_i f_i$;

Algorithm 16: DOPRI54 step function.

4.5 ESDIRK23

Both the modified and the unmodified ESDIRK23 method is implemented as shown in Algorithm 17. The unmodified method uses Algorithm 18 in line 8 and 13. The modified method uses the alternative.

Like RKF45 and DOPRI54, ESDIRK23 is an embedded method using embedded error estimation, which is done in the step functions described in the Subsections below. For the unmodified ESDIRK23 method, only x_{n+1} and $f(t_{n+1}, x_{n+1})$ is updated in Algorithm 4. For the modified ESDIRK23 method, the step update also updates $g(x_{n+1})$. Note that the second order method is used as the advancing method instead of the third order and that there are no function evaluations outside the step functions except in the first step. Note also that the step update procedure is the same for both the modified and the unmodified ESDIRK23 method.

4.5.1 The Unmodified ESDIRK23 Step Function

The ESDIRK23 method has array representations of b, $\hat{b}$, c and d from the Butcher Tableau. Since the A-matrix only has two distinct elements, and the last row has the same coefficients as b, the matrix itself need not be stored.

Data: $f(t, x(t))$, $\frac{\partial}{\partial x} f(t, x(t))$, $g(x(t))$, $\frac{\partial}{\partial x} g(x(t))$, initial and final time t_0 and t_f, number of steps N, initial conditions x_0, absolute tolerance, relative tolerance, parameters

Result: time and approximation vectors are returned or manipulated

initialization;
define A, b, $\hat{b}$, c and d;
check if the method should use fixed step size;
if *using fixed step size* **then**
    calculate step size, $h = \frac{t_f - t_0}{N-1}$;
    for *every step* **do**
        update time step;
        calculate x_{n+1} using Algorithm 15 or 19;
    end
else
    while *final time is not exceeded* **do**
        check if final time is exceeded by step size;
        calculate x_{n+1} and e using either Algorithm 18 or 19;
        calculate norm of error using the norm (3.6);
        update step and step size using 4;
    end
end
return number of steps;

Algorithm 17: Algorithm for both the modified and unmodified Dormand-Prince method.

Euler steps are used as initial guess for each internal stages and it uses Newton iterations to obtain the approximation in the internal stages. Every time the residual function, $R(x)$, is evaluated in the Newton iterations, so is $f(t, x(t))$.

If the first Newton iterations diverge or converges slowly, the while-loop in the second Newton iterations will not initialize.

Data: $f(t, x(t))$, $\frac{\partial}{\partial x} f(t, x(t))$, $f(t_n, x_n)$, t_n, x_n, h, $AbsTol$, $RelTol$, b, c, d, γ, $AbsTol$, $RelTol$
Result: x_{n+1}, e_{n+1}, f_{n+1}, α

set $X_1 = x_n$;
set $f_1 = f(t_n, x_n)$;
evaluate $J = \frac{\partial}{\partial x} f(t_n, x_n)$;
calculate $\frac{\partial}{\partial x} R(x) = I - h\gamma J$;
for $i = 2 \ldots 3$ **do**
    calculate $\phi_i = x_n + h \sum_{j=1}^{i-1} Ai, jf_j$;
    calculate $x^0 = x_n + c_i h f_1$;
    set $R(x(t)) = x(t) - h\gamma f(t_n + c_i h, x(t)) - \phi_i$;
    use Algorithm 1 to obtain X_i;
end
set $x_{n+1} = X_3$;
set $f_{n+1} = f_3$;
calculate $e = h \cdot \sum_{i=1}^{3} d_i f_i$;

Algorithm 18: The unmodified ESDIRK23 step function.

4.5.2 The Modified ESDIRK23 Step Function

The modified ESDIRK23 step function is shown in Algorithm 19. It is worth noting that the modified ESDIRK23 method does not use Newton iterations any more than the unmodified ESDIRK23 method does. It has an additional evaluation of $\frac{\partial}{\partial x} g(t_n, x_n)$. Note also that both the Residual and the ψ_i are different from the same in the unmodified method.

Data: $f(t, x(t))$, $\frac{\partial}{\partial x} f(t, x(t))$, $g(x(t))$, $\frac{\partial}{\partial x} g(x(t))$, $f(t_n, x_n)$, $g(x_n)$, t_n, x_n, h, $AbsTol$, $RelTol$, b, c, d, γ, $AbsTol$, $RelTol$
Result: x_{n+1}, e_{n+1}, f_{n+1}, g_{n+1}, α
set $X_1 = x_n$;
set $f_1 = f(t_n, x_n)$;
set $\frac{\partial}{\partial x} R(x_n) = \frac{\partial}{\partial x} g(t_n, x_n) - h\gamma \frac{\partial}{\partial x} f(t_n, x_n)$;
for $i = 2 \ldots 3$ **do**
    calculate $\psi_i = g_n + h \sum_{j=1}^{i-1} Ai, jf_j$;
    calculate $x^0 = x_n$;
    set $R(x(t)) = g(x(t)) - h\gamma f(t_n + c_i h, x(t)) - \psi_i$;
    use Algorithm 1 to obtain X_i;
end
set $x_{n+1} = X_3$;
set $f_{n+1}, x_{n+1}) = f_3$;
set $g_{n+1} = G_3$;
calculate $e_{n+1} = h \cdot \sum_{i=1}^{3} d_i f_i$;

Algorithm 19: The modified ESDIRK23 step function.

4.6 Summary

Every method consist of a main algorithm and a step function. The main algorithm is in large parts the same for both the modified and unmodified versions of each method. The main algorithm and both the modified and unmodified step functions have been discussed and illustrated using pseudo-code.

Chapter 5

Implementation of Parallel Simulations

This Chapter describes the implementation of parallel simulations in `Matlab` and `C` and provides pseudo-code. MPI may be used for implementing parallel simulations in `C` and the relevant functions from this interface are described.

5.1 Introduction

This Chapter describes how the simulations needed to obtain the results in Chapter 6 may be parallelized in order to decrease the runtime. Simulating in parallel may improve runtime since computations are done simultaneously. This improvement is attainable because the simulations are completely independent of each other, which means that there is no need for communication between the processes during simulation.

The goal when running the simulations is to obtain the approximations, e.g. to write them onto a file. In a serial implementation, all simulations can easily be written onto a single file, however this is more cumbersome when multiple processes are to write to the same file. Alternatively, each process may write to its own file, or each simulation may be saved in a separate file. This project is not concerned with the possible increase in loading time, when data is stored across several files, and it is expected that this is negligible.

An example of a parallel simulation which would not be effective is to do a single simulation of the system in equation 6.15 with piecewise approximations to the analytically optimal inlet rates as they are described in Section 6.5. The simulation of this system is in practice a number of subsimulations with constant inlet rates, where each uses the approximation from the previous as initial condition.

This would mean that only one process could run at a time, since each process would need the results from the previous as initial conditions. Hence, the only change when using parallel instead of sequential simulations would be the increase of unnecessary communication between the processes and therefore increased runtime.

5.2 Parallel Simulations In Matlab

In `Matlab`, simulating in parallel is quite easy to implement compared to `C`. The structure is like a `for-loop`, however the order of execution is not deterministic. The command `parfor` is used instead of `for` and then `Matlab` delegates the body of the loop to the processes. This may be used as seen in Algorithm 20.

There are several things to be aware of when using `Matlab`s parfor. Since the body of the loop is executed in any order, any assignment like `t = f(A(i));` is left unchanged outside the loop, i.e. `t` is unchanged. Assigning to elements in a vector or array do affect the variable outside the loop. For example, `C(i) = h(t,u)` does change `C`. Another use which is important to be aware of is that if an assignment is deterministic, it may be affected outside the loop. This could be `if p(i), s = s+1 end`, where `s` is then affected. [Mathworks, 2013].

```
1 parfor each set of parameters do
2 |   simulation;
3 end
```

Algorithm 20: Implementation of parallel simulations in Matlab.

5.3 Simple Parallel Simulations In C

As mentioned in Section 5.1, the simulations are assumed to be completely independent of each other, which means that the parallel simulations may be done without any communication between the nodes. The simple implementation of parallel simulations is shown in Algorithm 21.

As will be shown in Chapter 8, this implementation has the disadvantage that some simulations may take longer for some sets of parameters than others if using adaptive step size. Consider a case with two processors available, and 10 simulations to be done. If half take 1 second and the other half takes 2 seconds, then the total runtime is 15 seconds when using serial simulations. If then one process does all the *slow* simulations, the runtime is 10 seconds, even though it may had been expected that the runtime would be near halved, i.e. 7.

```
1 for each process do
2 |   acquire rank;
3 |   acquire number of processes;
4 |   acquire sets of parameters, based on rank and number of processes;
5 |   for each set of parameters do
6 |   |   simulation;
7 |   end
8 end
```

Algorithm 21: Simple implementation of parallel simulations.

5.4 Advanced Parallel Simulations In C

To account for the drawbacks of the simple parallel simulations, one may sacrifice a processor to take care of keeping the remaining processors busy. If only two processors are available, this would only increase the runtime since just one process is simulating, but if a large number of processors are available, e.g. 16, then the decrease in runtime is expected to be near 15, which is not much less than 16.

The implementation of such a program is shown in Algorithm 22. As with the simple implementation, all slave nodes may initialize themselves. The master node is not needed until the first slave node is done with the simulations for the set of parameters. This may be implemented such that the master node is also doing simulations, but is somewhat more cumbersome, and without the scope of this project.

```
1  for master node do
2  |  acquire number of processes;
3  |  acquire chunks of parameters based on number of processes;
4  |  while simulations remain to be done do
5  |  |  wait for a message from any slave node;
6  |  |  at receival of message send out new set of parameters;
7  |  end
8  |  for each process do
9  |  |  send stop signal;
10 |  end
11 end
12 for each slave node do
13 |  acquire rank;
14 |  acquire number of processes;
15 |  acquire a set of parameters, based on rank and number of processes;
16 |  while stop signal not received do
17 |  |  for each set of parameters do
18 |  |  |  simulation;
19 |  |  end
20 |  |  send message requesting new set of parameters from master
   |  |  node;
21 |  |  receive message with new set of parameters from master node;
22 |  end
23 end
```

Algorithm 22: Advanced implementation of parallel simulations.

5.5 Message-Passing Interface

In practice, implementing parallel simulations in `C` is done with the Message-Passing Interface, MPI. This interface provides functions for sending and receiving messages between processes and much more. [Dongarra et al., 1996].

Any program using multiple processes should only call MPI functions between the two function calls

```
MPI_Init(&argc,&argv);
<MPI function calls>
MPI_Finalize();
```

`argc` and `argv` are from the command line. Acquiring the rank is done with

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

which stores the rank of the process in the variable `rank`. The total number of processes is stored in the variable `size` by the call

```
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

A message may be send by the call

```
MPI_Send(send,n,MPI_INT,dest,tag,MPI_COMM_WORLD);
```

where `send` is a pointer to a contigous memory of size `n` times the size of `MPI_INT` which is what may be received. The type may also be `MPI_DOUBLE` or others. `dest` indicates which process the message should be sent to. `tag` is used to identify the message. The last argument indicates which processes are valid for the send. In the context of this project, all functions are always called with `MPI_COMM_WORLD`. The receive function is not much different

```
MPI_Recv(rec,n,MPI_INT,source,tag,MPI_COMM_WORLD,&status);
```

Like the send function, `rec` is a pointer to memory of size `n` times the size of `MPI_INT` or whatever type is passed. `source` indicates what process to receive

a message from and may be set to `MPI_ANY_SOURCE` to indicate that messages from any process should be received. This is used in the advanced parallel simulations. `status` is pointer to a structure `MPI_Status` which contain the source and tag of the message.

5.6 Summary

This Chapter has described the implementation of parallel simulations in `Matlab` by the use of `parfor`-loops.

The two implementations of parallel simulations in `C` has also been described and the functions from the MPI interface which are used to acquire rank and number of processes as well as sending and receiving messages from one process to another, have been explained.

Chapter 6

Fed Batch Fermenter Problem

In this Chapter, it is illustrated how the numerical methods for solving IVPs, introduced in the Chapter 4, may be used to test different control strategies for increasing the total production of biomass in a fed batch fermenter by controlling the rates of water and substrate inlet. The model of the fed batch fermenter is described in Section 6.1.

The control strategies should be robust towards pertubations in the parameters and the strategies are tested on a large set of pertubated parameters. The different control strategies are shown below, where the optimal inlet rates are analytically derived in [Jørgensen, 2013]

1. Constant inlet rates
2. Optimal inlet rates
3. Piecewise approximation of 2.
4. Optimal inlet rates using substrate feedback
5. Piecewise approximation of 4.
6. Optimal inlet rates using biomass/substrate feedback
7. Piecewise approximation of 6.

6.1 Model of Fed Batch Fermenter

This Section describes the model of the Fed Batch Fermenter. In the fermenter is a biomass and substrate solution in water and the biomass transforms substrate into more biomass. Meanwhile, substrate and water are supplied to the tank, and it is the rate of these inlets that are sought to be controlled as to optimize the total production. The substrate to biomass reaction is satisfies the following,

$$\gamma_s S \longrightarrow X,$$

where S is substrate and X is biomass and with the reaction rate

$$r(C_X, C_S) = \mu(C_S)C_X,$$

where

$$\mu(C_S) = \mu_{max}\frac{C_S}{K_S + C_S + \frac{C_S^2}{K_I}}.$$

Hence the production rates are

$$\begin{aligned} R_X(C_X, C_S) &= r(C_X, C_S) \\ R_S(C_X, C_S) &= -\gamma_s r(C_X, C_S). \end{aligned}$$

The model is based on conservation of mass, and assumes that the inlets and fermenter content has identical density. Hence the change in total mass, ρV, equals the mass added from the substrate, and water inlets, ρF_s and ρF_w minus whatever may be harvested from the fermenter during production, ρF. This is equation (6.1a).

The change in biomass, VC_X, is equal to what is produced from substrate, $R_X V$, minus what is harvested, FC_X, which is expressed in equation (6.1b).

The change in mass of substrate, VC_S equals what is added from the substrate inlet, $F_s C_{S,in}$ and whatever is produced, $R_S V$, minus what is harvested FC_S, expressed in equation (6.1c).

$$\frac{d}{dt}(\rho V) = \rho F_s + \rho F_w - \rho F, \qquad V(t_0) = V_0 \tag{6.1a}$$

$$\frac{d}{dt}(VC_X) = -FC_X + R_X V, \qquad C_X(t_0) = C_{X,0} \tag{6.1b}$$

$$\frac{d}{dt}(VC_S) = F_s C_{S,in} - FC_S + R_S V, \quad C_S(t_0) = C_{S,0} \tag{6.1c}$$

The model is now in the form (1.1). In the next Section, it will be transformed into the form (1.2) as both forms are used in Chapter 8, where simulations are timed for both the modified and unmodified methods.

Symbol	Value	Unit
ρ	1	$\mathrm{kg/m^3}$
F	0	$\mathrm{m^3/hr}$
γ_s	1.777	kg substrate/kg biomass
μ_{max}	0.37	1/hr
K_S	0.021	$\mathrm{kg/m^3}$
K_I	0.38	$\mathrm{kg/m^3}$

Table 6.1: Parameters in the Fed Batch Fermenter model.

6.2 Simulation of Fed Batch Fermenter

This Section describes the choice of certain parameter values and initial conditions. In Table 6.1, can be seen the parameter values used in the simulations.

The initial conditions are chosen to be

$$V_0 = 100\,\mathrm{m^3} \tag{6.2}$$

$$C_{X,0} = 20\,\frac{\mathrm{kg}}{\mathrm{m^3}} \tag{6.3}$$

$$C_{S,0} = 0.0893\,\frac{\mathrm{kg}}{\mathrm{m^3}} \tag{6.4}$$

$$P_0 = 0\,\mathrm{kg} \tag{6.5}$$

V_0 is chosen rather arbitrarily and so is $C_{X,0}$. The choice of $C_{S,0}$ is explained in a following Subsection. P_0 is chosen as is because the production is nil at the start of the production.

In the simulations F is chosen to be zero, meaning that there is no harvesting during the production of biomass. All biomass is simply harvested when the the tank is full, and the process is repeated. It is assumed that the tank has a capacity of $V_{max} = 1200\,\mathrm{m^3}$.

The time span of simulation is not fixed since, it is generally not known in advance how long it takes for the tank to fill up, for any given inlet rates and parameters, e.g. piecewise constant parameters. For the optimal inlet rates with no feedback, described in Section 6.4, the time span is

$$t_{final} = \frac{1}{\mu^* \log\left(\frac{V_{max}}{V_0}\right)}. \tag{6.6}$$

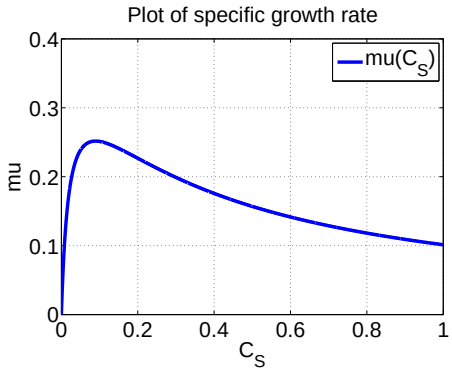


Figure 6.1: Plot of the function $\mu(C_S)$, with the parameters from Table 6.1

6.2.1 Specific Growth Rate

We investigate the function $\mu(C_S)$ as we want an initial substrate concentration, $C_{S,0}$, which gives the maximum reaction rate. The derivative of $\mu(C_S)$ is

$$\frac{\mathrm{d}}{\mathrm{d}C_S}\mu(C_S) = -\mu_{max}\frac{K_I(-K_SK_I + C_S^2)}{(K_SK_I + C_SK_I + C_S^2)^2},$$

which is zero at 0.0893, where $\mu(0.0893) = 0.25$. As can be seen from Figure 6.1 this is clearly a local maximum. In this plot the parameters from Table 6.1 are used. These optimal values will be used in the simulation of the model in later sections.

$$C_{S,optimal} = 0.0893 \tag{6.7}$$
$$\mu(C_{S,optimal}) = 0.25 \tag{6.8}$$

6.2.2 Transformation of the IVP

In this Subsection the differential equations (6.1) are rewritten into the form $\frac{d}{dt}\mathbf{x} = f(\mathbf{x})$ instead of the current form $\frac{d}{dt}g(\mathbf{x}) = f(\mathbf{x})$ and a differential equation which describes the change in production at time t is added.$\mathbf{x} = (V, C_X, C_S, P)$ and $g(\mathbf{x}) = (\rho V, VC_X, VC_S, P)$.

It is assumed that ρ is a constant, and we set $F = 0$. Hence from equation 6.1a we get

$$\frac{d}{dt}V = F_s + F_w. \tag{6.9}$$

Using the chain rule on the left hand side in (6.1b), we obtain

$$\frac{d}{dt}(VC_X) = V\frac{d}{dt}C_X + C_X\frac{d}{dt}V.$$

(6.9) is substituted

$$V\frac{d}{dt}C_X + C_X\frac{d}{dt}V = V\frac{d}{dt}C_X + C_X(F_s + F_w).$$

From (6.1b) we know that

$$V\frac{d}{dt}C_X + C_X(F_s + F_w) = R_X V.$$

From this we obtain the expression for the derivative of C_X.

$$\frac{d}{dt}C_X = R_X - C_X\frac{F_s + F_w}{V}$$

The expression for $\frac{\mathrm{d}}{\mathrm{d}t}C_S$ can be derived in a similar manner using (6.1c) and (6.9).

$$\begin{aligned}
\frac{d}{dt}(VC_S) &= V\frac{d}{dt}C_S + C_S\frac{d}{dt}V && (6.10)\\
&= V\frac{d}{dt}C_S + C_S(F_s + F_w) && (6.11)\\
&= F_s C_{S,in} + R_S V && (6.12)\\
\Rightarrow \quad \frac{d}{dt}C_S &= R_S + \frac{F_s C_{S,in} - C_S(F_s + F_w)}{V}. && (6.13)
\end{aligned}$$

The total production at a given time, t is calculated as follows

$$P(t) = \int_{t_0}^{t} R_X(t)V(t)\mathrm{d}t, \tag{6.14}$$

and is implemented as a part of the system of differential equations by differentiating equation (6.14), which gives

$$\frac{\mathrm{d}}{\mathrm{d}t}P(t) = R_X V.$$

The IVP (6.1) is put into the form (1.2) as

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}t}V &= F_s + F_w, & V(t_0) &= V_0 && (6.15a)\\
\frac{\mathrm{d}}{\mathrm{d}t}C_X &= R_X - C_X\frac{F_s + F_w}{V}, & C_X(t_0) &= C_{X,0} && (6.15b)\\
\frac{\mathrm{d}}{\mathrm{d}t}C_S &= R_S + \frac{F_s C_{S,in} - C_S(F_s + F_w)}{V}, & C_S(t_0) &= C_{S,0} && (6.15c)\\
\frac{\mathrm{d}}{\mathrm{d}t}P &= R_X V, & P(t_0) &= P_0. && (6.15d)
\end{aligned}$$

6.3 Constant Inlet Rates

In this section we use

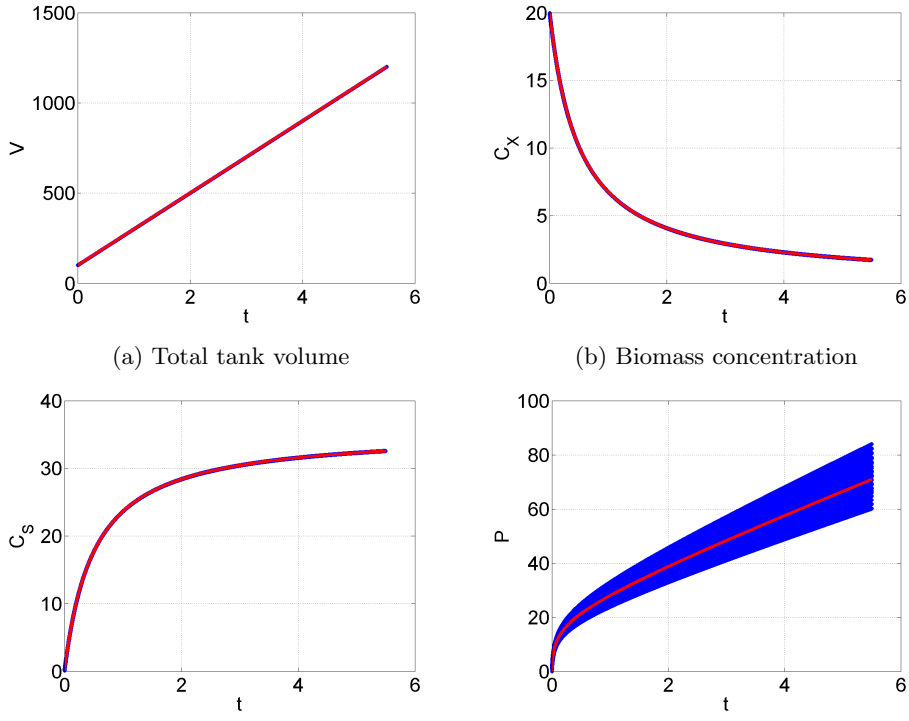
$$F_s = 100\,\mathrm{m}^3/\mathrm{hr} \tag{6.16}$$

$$F_w = 100\,\mathrm{m}^3/\mathrm{hr}, \tag{6.17}$$

as this value is somewhat reasonable, compared to the analytically optimal rates used in the following section. In Figure 6.2 can be seen the simulation for the exact parameter values. We see that the volume continues up to a total of 1200. The biomass concentration decays in a somewhat exponential manner towards zero and the substrate concentration grows exponentially towards around 35. The production grows explosively at first and then assumes a somewhat constant increase and ends at around 65. The production time is just above 5.

We will later see that this production is extremely low. The explanation is that the substrate concentration becomes far larger than its optimal value and the biomass concentration decays. Both of these contribute to a decrease in the reaction rate of the substrate to biomass reaction.

In Figure 6.3 can be seen the histogram of productions when the parameter values are varied. All productions are between 50 and 100 and the shape is reminiscent of a normal distribution.



(a) Total tank volume

(b) Biomass concentration

(c) Substrate concentration

(d) Production

Figure 6.2: Trajectories of the approximate solution to the fed batch problem (6.15) for different sets of parameters. Constant inlet rates have been used.

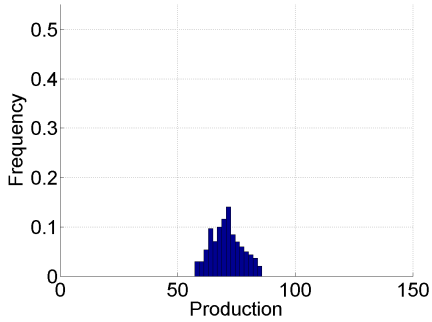


Figure 6.3: Fed batch biomass production for different sets of parameters. Constant inlet rates have been used.

6.4 Analytically Optimal Inlet Rates

In this Section we use the analytically optimal inlet rates, derived in [Jørgensen, 2013].

$$F_s^* = \alpha_s V_0 \exp(\alpha t)$$
$$F_w^* = \alpha_w V_0 \exp(\alpha t),$$

where

$$\begin{aligned} \alpha_s &= \frac{\gamma_s + C_S^*/C_X^*}{C_{S,in}} r^* \\ \alpha_w &= -\frac{\gamma_s - (C_{S,in} - C_S^*)/C_X^*}{C_{S,in}} r^* \\ \alpha &= \alpha_s + \alpha_w. \end{aligned}$$

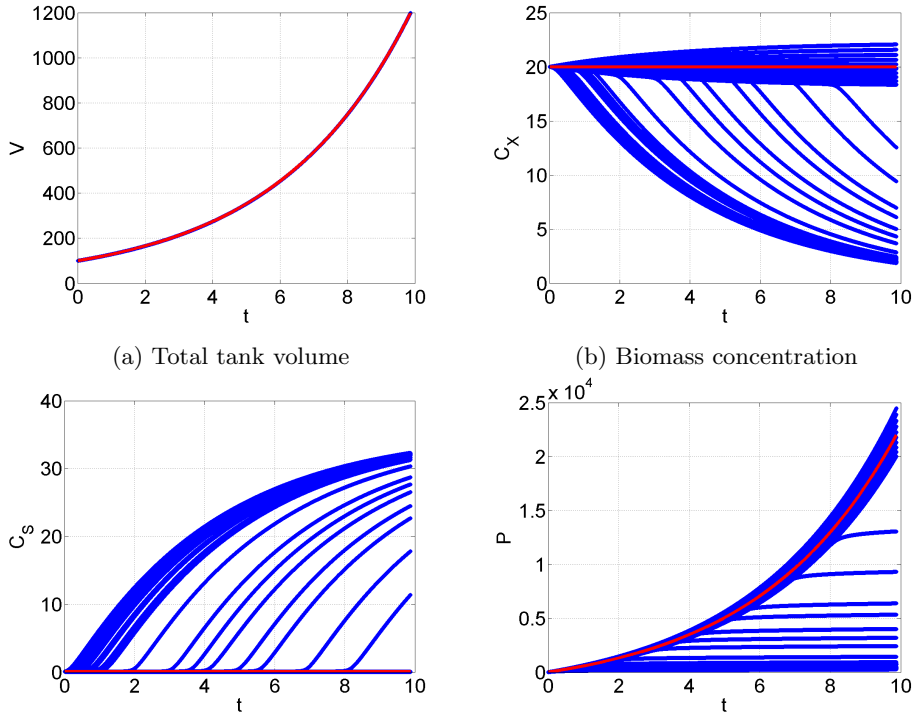
As both inlet rates should be positive at all times, we must require that $\alpha_s \geq 0$ and $\alpha_w \geq 0$. We use $C_{S,in} = 2(C_S^* + \gamma_S C_X^*)$, which is twice the minimum value for which $\alpha_w \geq 0$. This value also has the particular property that $\alpha_s = \alpha_w$ as shown below.

$$\begin{aligned} C_{S,in} &= 2(C_S^* + \gamma_S C_X^*) \\ C_X\gamma_S + C_S^* &= -C_X\gamma_S + C_{S,in} - C_S^* \\ \gamma_S + C_S^*/C_X^* &= -(\gamma_S - (C_{S,in} - C_S^*)/C_X^*) \\ \frac{\gamma_S + C_S^*/C_X^*}{C_{S,in}} r^* &= -\frac{\gamma_S - (C_{S,in} - C_S^*)/C_X^*}{C_{S,in}} r^* \\ \alpha_s &= \alpha_w \\ F_s &= F_w \end{aligned}$$

In Figure 6.4 are shown the simulation for these optimal inlet rates for different sets of parameters, and the behavior is quite different from when using constant inlet rates. Both volume and production increases in an exponential manner while biomass and substrate concentrations are steady at their optimal values.

The production is in this case more than 20000, which is enormous compared to the productions in the previous section which were all in the range of 50 to 100. However, for some sets of parameters the system experiences the same behavior as with constant inlet rates. The production time is just around 9, which is a tad more than in the previous section where it was around 5.5.

The histogram for the optimal inlet rates shown in Figure 6.5 is also very different from the one we saw for constant inlet rates. More than half the values are located in the vincinity of zero, but a large part is clustered in area around



(a) Total tank volume

(b) Biomass concentration

(c) Substrate concentration

(d) Production

Figure 6.4: Trajectories of the approximate solution to the fed batch problem (6.15) for different sets of parameters. The analytically optimal inlet rates have been used.

20000 to 25000. In conclusion these optimal inlet rates are far superior to the constant inlet rates used in the previous section, as a there is a large chance that the production might be very big. However, there is still a big risk that the production will be on the scale of hundreds.

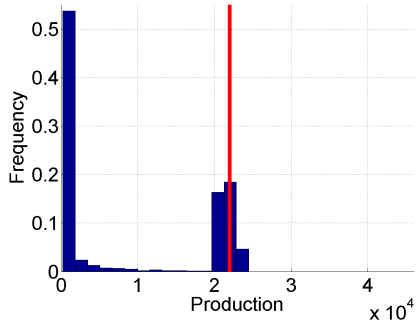


Figure 6.5: Fed batch biomass production for different sets of parameters. The analytically optimal inlet rates have been used.

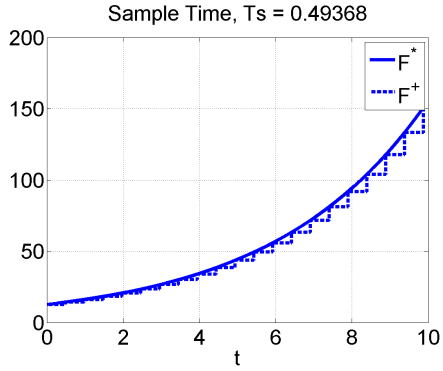


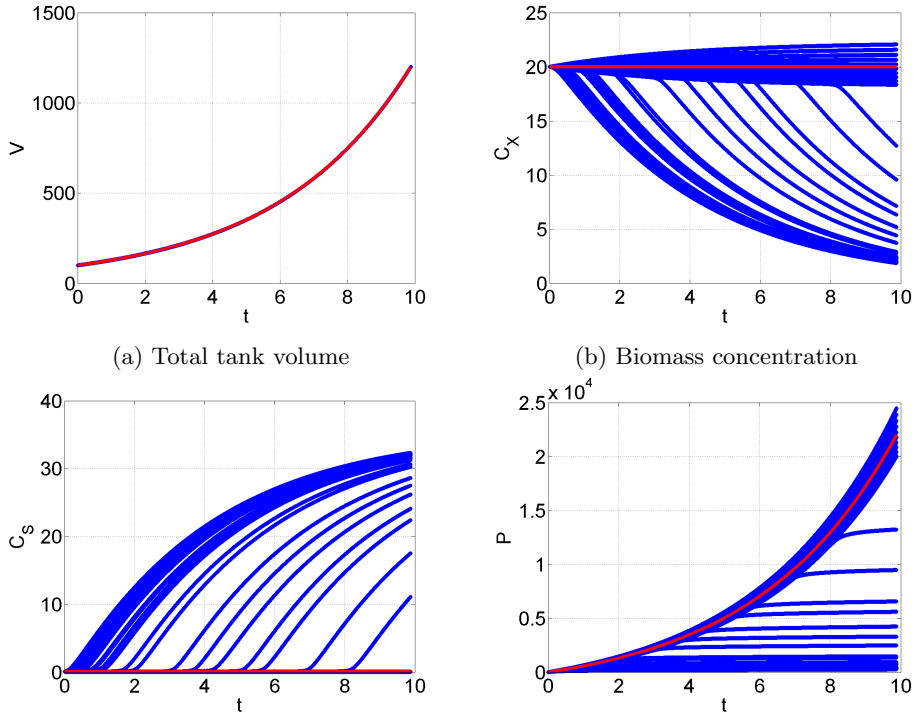
Figure 6.6: The optimal inlet rates together with piecewise constant approximations of these. $N_k = 20$.

6.5 Piecewise Constant Approximations

In this section we use piecewise constant approximations of the optimal inlet rates described in the previous section. The idea is to keep the inlet rates constant in a number of time intervals, N_k. We use the left optimal inlet rate which means we will get a stair function which is below the optimal inlet rate, see Figure 6.6. In mathematical notation, the approximations are

$$F^{+}_{s/w}(t) = F^{*}_{s/w}(t_i), \quad \text{for } t \in [t_i, t_{i+1}]. \tag{6.18}$$

In Figure 6.7, it is seen that, for $N_k = 2000$, the trajectories are very much similar to those of purely optimal inlet rates. Even though it seems that the two concentrations are still constant, they do vary very little as a consequence



(a) Total tank volume

(b) Biomass concentration

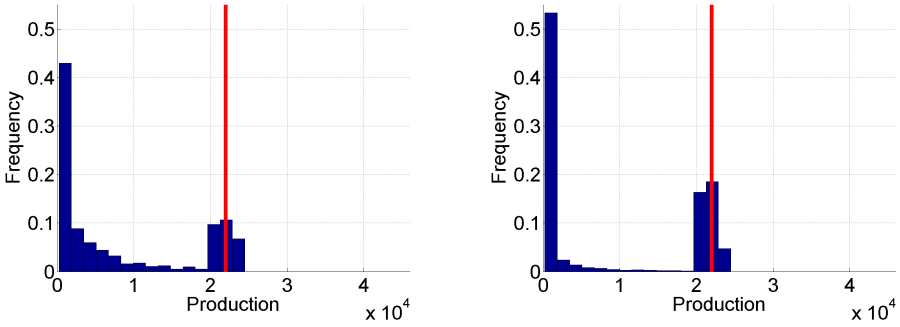
(c) Substrate concentration

(d) Production

Figure 6.7: Trajectories of the solution to the fed batch problem (6.15) for different sets of parameters. Piecewise constant approximations to the analytically optimal inlet rates have been used. $N_k = 2000$.

of the inlet rates not being completely optimal.

In Figure 6.8 we see histograms for $N_k = 20$, and $N_k = 2000$. Clearly it is beneficial to use a higher number of samples as the probability of a large production is increased when N_k increases. However in both cases there is more than 50% chance that the production will be very small, as for the two previous strategies.



(a) $N_k = 20$ (b) $N_k = 2000$

Figure 6.8: Fed batch biomass production for different sets of parameters. Piecewise constant approximations to the analytically optimal inlet rates have been used.

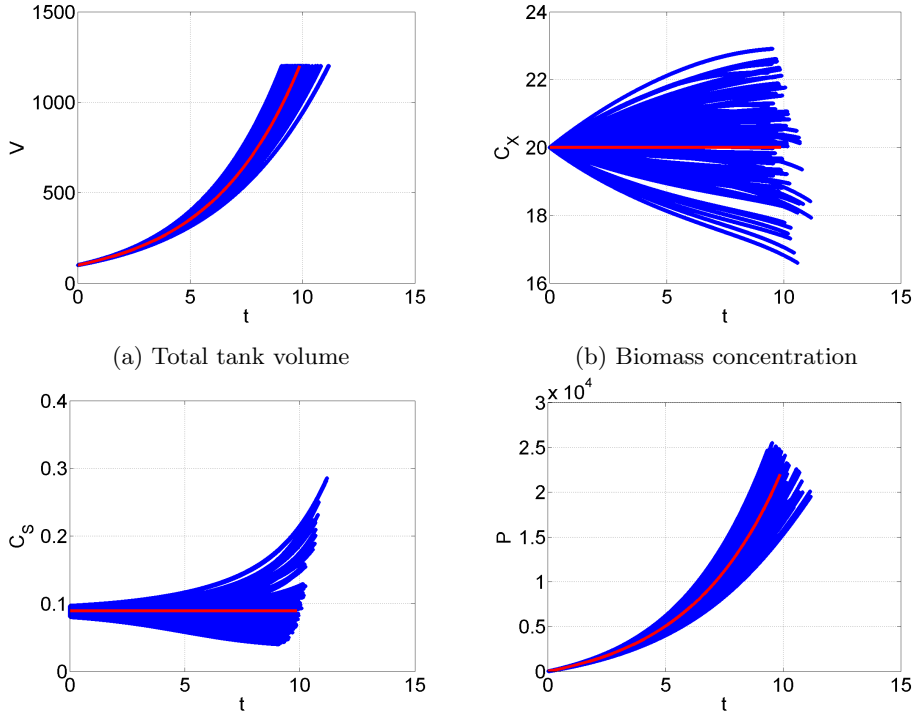
6.6 Substrate Feedback for Optimal Inlet Rates

In this Section we will see the effect of adding a feedback term to the expression for the substrate concentration. The substrate feedback inlet rates are as follows

$$F_s = F_s^* + K_s(C_{S,0} - C_S) \tag{6.19}$$
$$F_w = F_w^*. \tag{6.20}$$

The effect of this feedback term should counteract the tendency that the substrate concentration becomes too large, and the biomass concentration too small, since the optimal inlet rates are based on keeping both concentrations at their optimum values.

The histograms for $K_s = 10, 50, 100$ and 300 are shown in Figure 6.10. The effect is not at all clear for $K_s = 10$, but for $K_s = 50$ we begin to see the lower part of the production distribution shift to the right. For this value of K_s there is no risk of a production lower than 7000, a great improvement over the previous strategies. The effect is even greater for $K_s = 100$, where the lowest production is around 12000, and for $K_s = 300$, all productions lie in the range from around 18000 to 28000.



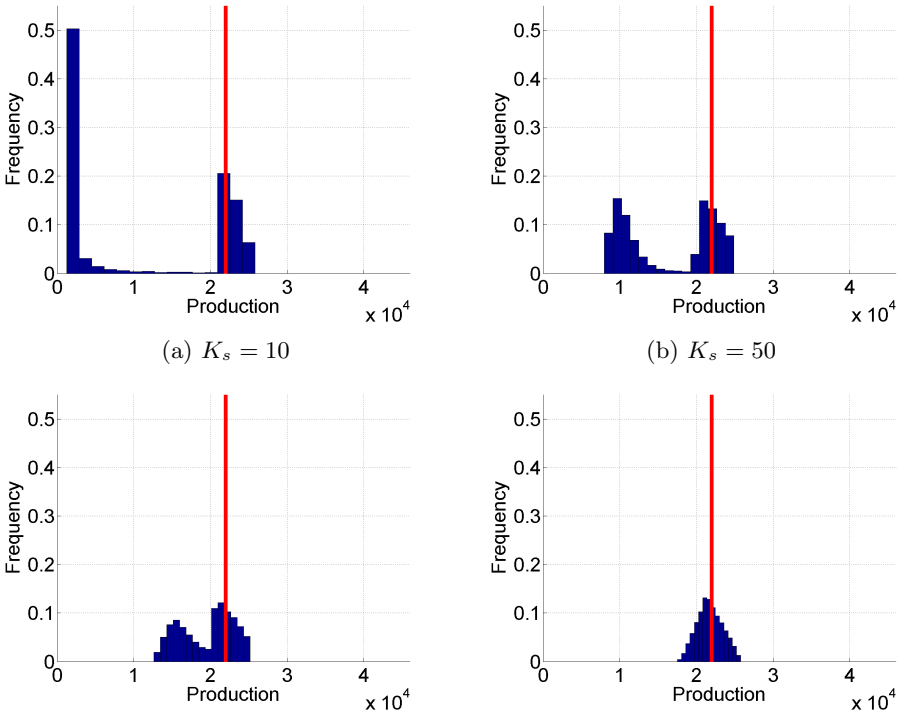
(a) Total tank volume

(b) Biomass concentration

(c) Substrate concentration

(d) Production

Figure 6.9: Trajectories of the approximate solution to the fed batch problem (6.15) for different sets of parameters. The analytically optimal inlet rates with substrate feedback have been used. $K_s = 300$.



(a) $K_s = 10$ (b) $K_s = 50$

(c) $K_s = 100$ (d) $K_s = 300$

Figure 6.10: Fed batch biomass production for different sets of parameters. The analytically optimal inlet rates with substrate feedback have been used.

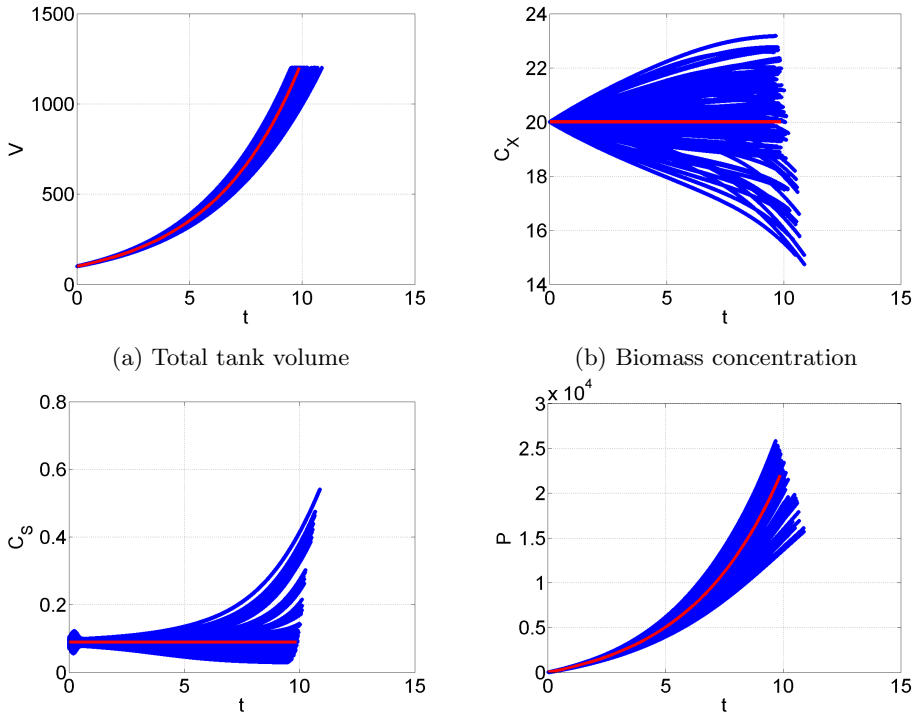
6.7 Substrate Feedback for Piecewise Constant Inlet Rates

In this Section, piecewise constant approximations to the inlet rates described in the previous Section, are used. We consequently use $N_k = 2000$ as this approximation is very close to the actual optimal rates, of course depending on the time interval.

In Figure 6.11 is shown the simulation for $K_s = 300$, and we see that both the volume and production simulations behaves much like when using the optimal inlet rates and hence also when using the piecewise approximations of these.

The histograms for this strategy is shown in Figure 6.12. The histograms are much like the ones seen in the previous Section. However the distribution is somewhat stretched as the lowest value of one of the distributions is lower than when using the substrate feedback for the optimal inlet rates with the same value of K_s. The highest values seem unchanged.

In other words, the distribution seem to have two parts, one of high production and one of low productions. In these terms it is the low production which is shifted to the left compared to the histograms in the previous Section. This means that there is a slightly higher risk of lower productions and a lower chance of higher productions.



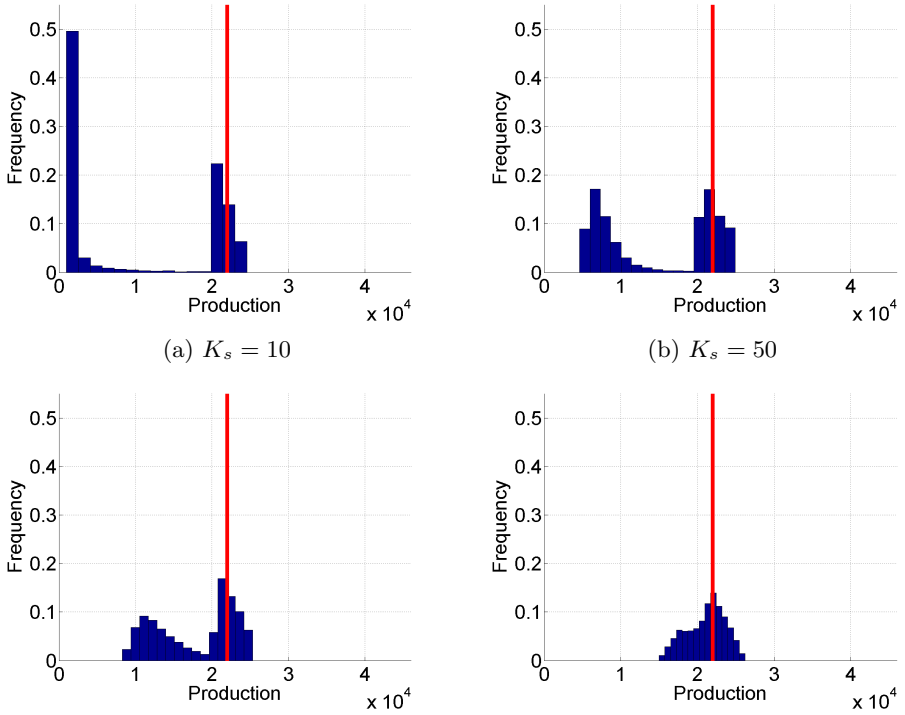
(a) Total tank volume

(b) Biomass concentration

(c) Substrate concentration

(d) Production

Figure 6.11: Trajectories of the approximate solution to the fed batch problem (6.15) for different sets of parameters. Piecewise constant approximations to the analytically optimal inlet rates with substrate feedback have been used. $N_k = 2000$ and $K_s = 300$.



(a) $K_s = 10$ (b) $K_s = 50$

(c) $K_s = 100$ (d) $K_s = 300$

Figure 6.12: Fed batch biomass production for different sets of parameters. Piecewise constant approximations to the analytically optimal inlet rates with substrate feedback have been used. $N_k = 2000$.

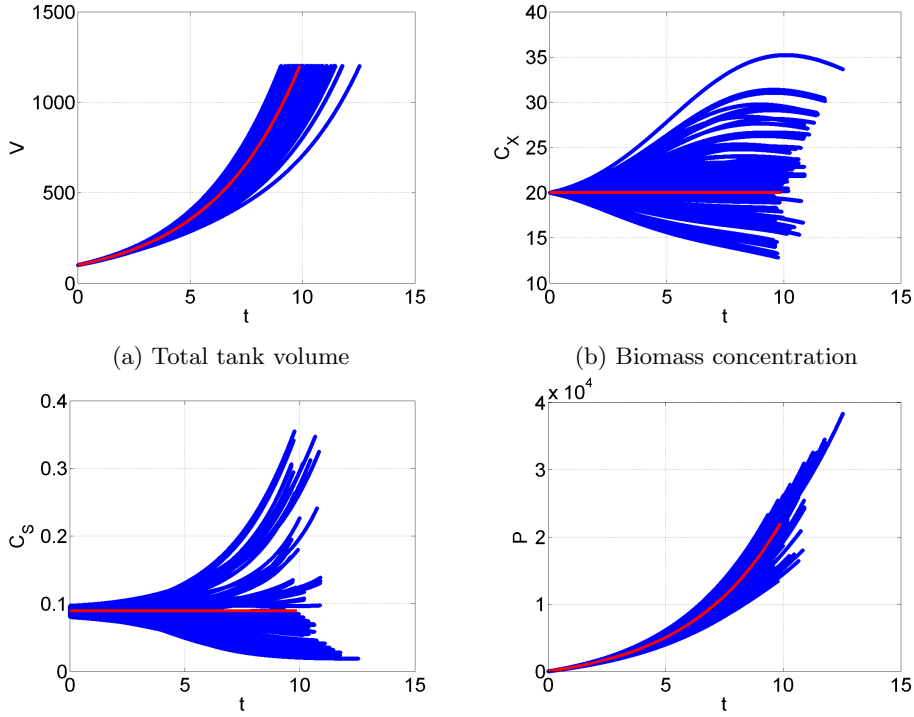
6.8 Biomass/Substrate Feedback for Optimal Inlet Rates

The substrate feedback did in fact improve the robustness towards pertubations in the parameters, so using the same approach for the biomass concentration may have an even greater effect. The deviation of the biomass concentration from its optimal value will control the water inlet rate. Hence the inlet rates are as follows.

$$F_s = F_s^* + K_s(C_{S,0} - C_S) \tag{6.21}$$
$$F_w = F_w^* + K_w(C_{X,0} - C_X). \tag{6.22}$$

In Figure 6.14 are shown the histograms for this strategy. Here we really see some results. Even for $K_s = 10$ and $K_w = 4$ the high part of the distribution has shifted, though not significantly. We see that using feedback for both substrate and biomass gives far larger productions than we have seen before.

We see, however, that the distributions are more spread than earlier, e.g. for $K_s = 300$ the minimum value is around 12000 where it was around 18000 when using only substrate feedback. All in all this strategy seems successful.



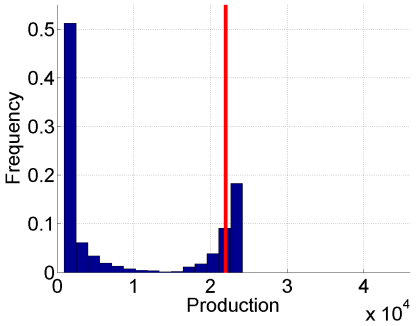
(a) Total tank volume

(b) Biomass concentration

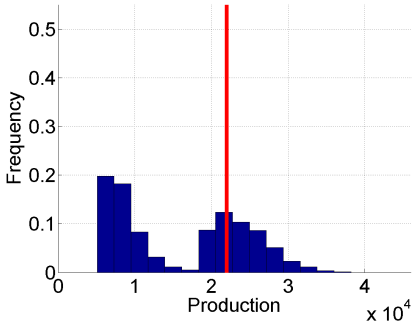
(c) Substrate concentration

(d) Production

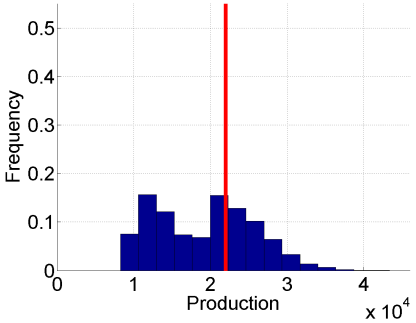
Figure 6.13: Trajectories of the solution to the fed batch problem (6.15) for different sets of parameters. The analytically optimal inlet rates with biomass and substrate feedback have been used. $K_s = 300$.



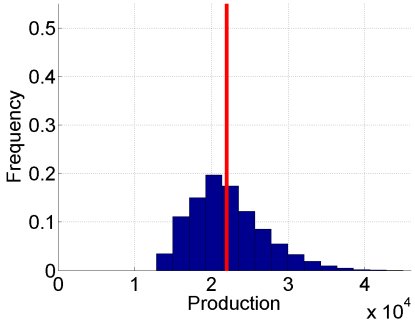
(a) $K_s = 10$, $K_w = 4$



(b) $K_s = 50$, $K_w = 4$



(c) $K_s = 100$, $K_w = 4$



(d) $K_s = 300$, $K_w = 4$

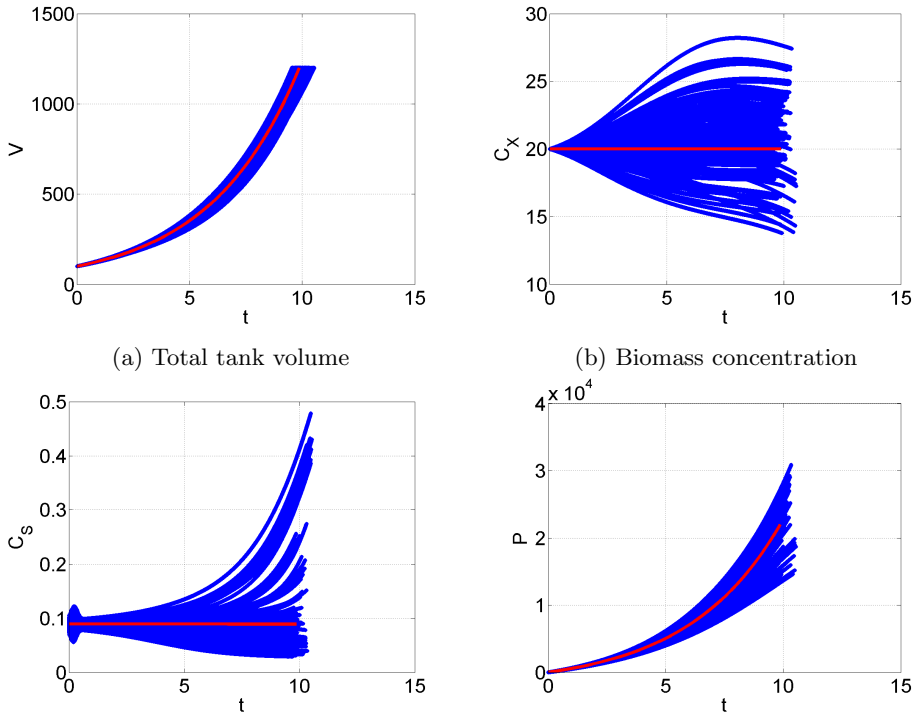
Figure 6.14: Fed batch biomass production for different sets of parameters. The analytically optimal inlet rates with biomass and substrate feedback have been used.

6.9 Biomass/Substrate Feedback for Piecewise Constant Inlet Rates

We now use piecewise constant approximations of the inlet rates used in the previous section. We use $N_k = 2000$ as we have done earlier.

The simulations for $K_s = 300$ and $K_w = 4$ is shown in Figure 6.15. These simulations resemble the simulations in the previous Section quite well, aside from some rapid variation in the substrate concentration during the first half hour or so.

In Figure 6.14 can be seen the histograms for this method. We see that the chance of having very large productions is far less compared to last section. However we still see an improvement in the way that the distribution has shifted to the right compared to the histograms in Section 6.7.



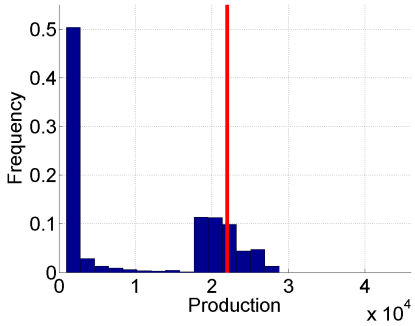
(a) Total tank volume

(b) Biomass concentration

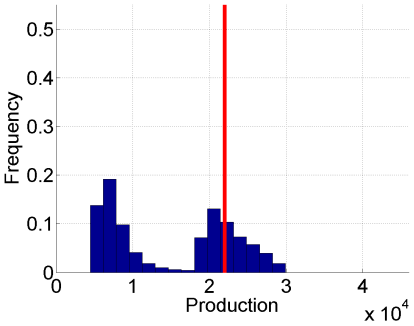
(c) Substrate concentration

(d) Production

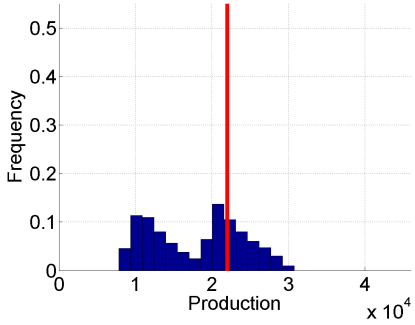
Figure 6.15: Trajectories of the solution to the fed batch problem (6.15) for different sets of parameters. Piecewise constant approximations to the analytically optimal inlet rates with biomass and substrate feedback have been used. $N_k = 2000$, $K_s = 300$ and $K_w = 4$.



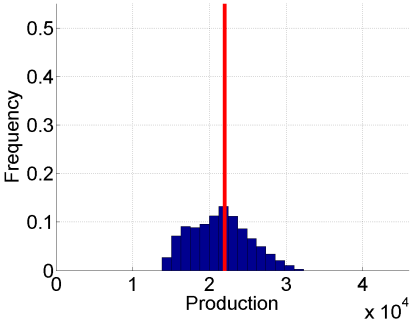
(a) $K_s = 10$, $K_w = 4$



(b) $K_s = 50$, $K_w = 4$



(c) $K_s = 100$, $K_w = 4$



(d) $K_s = 300$, $K_w = 4$

Figure 6.16: Fed batch biomass production for different sets of parameters. Piecewise constant approximations to the analytically optimal inlet rates with biomass and substrate feedback have been used.

6.10 Summary

Using constant inlet rates was not effective for the choice $F_s = F_w = 100$, the production was in the range from 50 to 100, which is far lower than what may be achieved using variable inlet rates.

Using the optimal inlet rates derived by Jørgensen [2013], the total production is on the scale of 20000 for certain sets of parameters, however there is more than 50 percent chance that it's on the scale of 1000. The simulations of the problem using the exact parameters from Table 6.1, gives a total production of around 22000.

The inlet rates may be set to be piecewise constant over a number of intervals as illustrated in Figure 6.6. Doing this with 20 intervals decreases the chance of having a total production on the scale of 20000 to around 25 percent. There is still a slight chance of having a production between around 2000 and 20000, however, there is more than 40 percent chance of having a production on the scale of 1000. If the number of intervals is instead 2000, the situation is much like for the optimal inlet rates.

The inlet rate of substrate may be modified according to the varying substrate concentration, as described in Section 6.6. If $K_s = 10$ the situation is not much different from not using feedback. Using $K_s = 50$ eliminates the risk of having a total production lower than 7500. Increasing K_s to 100 and 300 increases the minimum total production, such that it is around 11000 and around 18000, respectively.

Using piecewise approximations to these inlet rates, decreases the minimum total production, however the tendency for increasing K_s is still the same. Here 2000 intervals are used. The minimum production for $K_s = 50$ is around 5000, for $K_s = 100$ it is around 8000 and for $K_s = 300$ it's around 15000.

The inlet rate of water may also be controlled by the varying concentration of biomass, as described in Section 6.8. A suitable value for K_w is 4, found by numerical experiments. For the same values of K_s just mentioned, the tendency is that the maximum production is increased, while the minimum production is decreased slightly. Production may be as high as more than 40000 for $K_s = 300$ and $K_w = 4$.

Using piecewise approximations to these inlet rates decreases the maximum production considerably, however the chance of these high productions were already low. Except for this decrease, there is not much difference from the productions when using piecewise approximations to the optimal inlet rates

with biomass and substrate feedback.

In conclusion, the feedback strategy is effective, whether using only substrate or both substrate and biomass feedback. The values of K_s and K_w which gives the most productive distribution, are $K_s = 300$ and $K_w = 4$.

Chapter 7

Test of Numerical Methods

This Chapter describes two simple test problems. The first is used to demonstrate convergence of the methods. The second will illustrate the effectivity of using adaptive step size, and this problem may be put in both of the forms (1.1) and (1.2). The use of the Runge-Kutta Toolbox in both `C` and `Matlab` will be illustrated by providing code for solving this problem.

The approximate solutions to this test problem will illustrate some differences between the methods. The global error of the approximation and the number of steps used to obtain these approximations are discussed.

7.1 Test Problems

This Section discusses the two simple test problem which will demonstrate order of convergence and illustrate some differences between the methods when using adaptive step size in the following Sections. The first simple test problem is

$$\frac{\mathrm{d}}{\mathrm{d}t}x(t) = \cos(t), \quad x(0) = 0, \tag{7.1}$$

which has the solution

$$x(t) = \sin(t). \tag{7.2}$$

This problem is formally in the form of (1.2), but may also be considered to be in the form of (1.1), where $g(x(t)) = x(t)$ and $\frac{\partial}{\partial x}g(x(t)) = 1$, such that the modified methods may also be tested. This is used to obtain the convergence tables seen in Figure 7.1. The second problem is

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\begin{bmatrix} x_1(t)\, x_2(t) \\ x_2(t) \end{bmatrix}\right) = \begin{bmatrix} \cos(t) \\ \sin(t) \end{bmatrix}, \quad x(0) = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \tag{7.3}$$

which has the solution

$$x(t) = \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} = \begin{bmatrix} \sin(t) + 2 \\ -\cos(t) + 2 \\ -\cos(t) + 2 \end{bmatrix}. \tag{7.4}$$

This may be put into the standard form (1.2). As the second differential equation is already in this form, only the first equation requires any modification. The chain rule is applied.

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\begin{bmatrix} x_1(t)\, x_2(t) \\ x_2(t) \end{bmatrix}\right) = \begin{bmatrix} x_2(t) & x_1(t) \\ 0 & 1 \end{bmatrix} \frac{\mathrm{d}}{\mathrm{d}t}\begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix}. \tag{7.5}$$

This is easily solved and the IVP (7.3) is transformed into

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix}\right) = \begin{bmatrix} \dfrac{\cos(t) - \sin(t)\, x_1(t)}{x_2(t)} \\ \sin(t) \end{bmatrix}, \quad x(0) = \begin{bmatrix} 2 \\ 1 \end{bmatrix}. \tag{7.6}$$

This is not a model based on a any conservation laws. However, using one of the unmodified methods still introduce additional error in the discretization since the chain rule is only valid in the limit. Hence, the modified methods may be expected to solve this problem more accurately. As can be seen from Figure 7.2 the solution (7.4) is oscillatory, and clearly non-stiff.

7.2 Test of Runge-Kutta Toolbox In Matlab

This Section gives example code for approximating the solution to the IVP (7.3) using adaptive step size in `Matlab`.

The right-hand-side function, $g(x(t))$ and the Jacobi matrices of these is expected to be implemented as follows. `fun` and `Jac` is the right-hand-side and Jacobi matrix corresponding to the form (7.6) and `fun2` and `Jac2` corresponds to the form (7.3).

```
function f = fun(t,x)
f = [(cos(t) - sin(t).*x(1))./x(2) ; sin(t)];
end

function J = Jac(t,x)
J = [-sin(t)./x(2) , (-cos(t) + sin(t).*x(1))./x(2).^2 ; 0 , 1];
end

function f = fun2(t,x)
f = [cos(t);sin(t)];
end

function J = Jac2(t,x)
J = zeros(2);
end

function g = gfun(x)
g = [x(1).*x(2);x(2)];
end

function dgdx = gJac(x)
dgdx = [x(2) , x(1) ; 0 , 1];
end
```

Below is shown a `Matlab`-script which computes the simulations which are plotted in Figure 7.2. The use of RK4, RKF45, DOPRI54 and the modified versions of these are similar to that of the unmodified and modifed Euler. The script should define, the time span, the initial conditions and absolute and relative tolerances.

```
% time span
ts = [0,10];

% initial conditions
x0 = xtrue(0);

% Tolerances
Tol = 1e-3;
AbsTol = ones(size(x0'))*Tol; RelTol = Tol;

% Simulations
[t,x] = Euler      (@fun,                    ts,x0,AbsTol,RelTol);
[t,x] = ESDIRK23   (@fun, @Jac,              ts,x0,AbsTol,RelTol);
[t,x] = EulerMod   (@fun2,      @gfun,@gJac,ts,x0,AbsTol,RelTol);
[t,x] = ESDIRK23Mod(@fun2,@Jac2,@gfun,@gJac,ts,x0,AbsTol,RelTol);
```

This illustrates the difference between the methods quite well. The unmodified explicit methods only take the right-hand-side function, $f(t, x(t))$, as input where the unmodified implicit method, ESDIRK23, also requires the Jacobian matrix of this function. All the modified methods require the $g(x(t))$ and $\frac{\partial}{\partial x} g(x(t))$ together with the right-hand-side function, $f(t, x(t))$, and the implicit modified method also requires $\frac{\partial}{\partial x} f(t, x(t))$. The remaining inputs are the same.

The methods expect $f(t, x(t))$ and $g(x(t))$ to return column vectors. The initial condition should also be a column vector, however, the `AbsTol` vector should be a row vector.

This script uses the tolerances $AbsTol_i = RelTol = 10^{-3} \, \forall i$, however the absolute and relative tolerances need not be the same, and the absolute tolerance need not contain identical elements. A problem may be solved best by using one relative tolerance and different absolute tolerances for each variable. The impact of these tolerances lie in the norm described in Section 3.3 used for the error estimate and the residual in the Newton iterations.

7.3 Test of Runge-Kutta Toolbox In C

This Section gives example code for approximating the solution to the IVP (7.3) using adaptive step size in `C`. The code is is the equivalent of the `Matlab` code shown in Section 7.2.

In `C`, the right-hand-side function, $g(x(t))$ and the Jacobi matrices of these are implemented as shown below, where, like in the `Matlab` code, `fun` and `Jac` corresponds to (7.6) and `fun2` and `Jac2` corresponds to (7.3).

```
void fun (const double t, const double *x,
          const void *params, double *f){
          f[0] = (cos(t)-sin(t)*x[0])/x[1]; f[1] = sin(t); }

void Jac (const double t, const double *x,
          const void *params, double *J){
          J[0*2+0] = -sin(t)/x[1];
          J[0*2+1] = (-cost(t)+sin(t)*x[0])/(x[1]*x[1]);
          J[1*2+0] = 0; J[1*2+1] = 1; }

void fun2(const double t, const double *x,
          const void *params, double *f){
          f[0] = cos(t); f[1] = sin(t); }

void Jac2(const double t, const double *x,
          const void *params, double *J){
          J[0*2+0] = 0; J[0*2+1] = 0;
          J[1*2+0] = 0; J[1*2+1] = 0; }

void gfun(const double t, const double *x,
          const void *params, double *g){
          g[0] = x[0]*x[1]; g[1] = x[1]; }

void gJac(const double t, const double *x,
          const void *params, double *dgdx){
          dgdx[0*2+0] = x[1]; dgdx[0*2+1] = x[0];
          dgdx[1*2+0] = 0;    dgdx[1*2+1] = 1; }
```

Below is shown a `C`-program which computes the simulations plotted in Figure 7.2. Like the `Matlab` version, the use of RK4, RKF45, DOPRI54 and the modified versions of these are similar to that of Euler. The program should define, number of variables, the number of time values, parameters, tolerances, initial

conditions, time span and function pointers. The number of time values is set to 2, which the methods interpret as adaptive step size, where more than 2 time values are used. The program should also allocate memory for storage of the approximation.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include "RungeKuttaToolbox.h"
int main(){
// Number of variables and timesteps
int    nx = 2, nt = 2, size = nt+200, N;
// Initial conditions, time span and parameters
double x0[] = {2,1}, tspan[] = {0,10}, params[] = {0};
// Tolerances
double Tol = 0.001, AbsTol = {Tol,Tol,Tol,Tol}, RelTol = Tol;
// Approximation vectors
double *t = malloc(   size*sizeof(double))
double *x = malloc(nx*size*sizeof(double));
ODEModel_t *pfun = fun, *pfun2 = fun2, *pgfun = gfun;
ODEModel_t *pJac = Jac, *pJac2 = Jac2, *pgJac = gJac;

// Simulations
N = Euler      (pfun,
                nx,nt,tspan,x0,AbsTol,RelTol,params,t,x);
N = ESDIRK23   (pfun, pJac,
                nx,nt,tspan,x0,AbsTol,RelTol,params,t,x);
N = EulerMod   (pfun2,      pgfun,pgJac,
                nx,nt,tspan,x0,AbsTol,RelTol,params,t,x);
N = ESDIRK23Mod(pfun2,pJac2,pgfun,pgJac,
                nx,nt,tspan,x0,AbsTol,RelTol,params,t,x);
}
```

The `C` versions need `nx` and `nt` as input where `Matlab` reads these from the length of the `tspan` and `x0` vectors. `size` is set to 202, which is then the maximum number of steps the method may take. The `ODEModel_t` is defined in the Runge-Kutta Toolbox header.

In `C`, the arrays don't have orientation like vectors do in `Matlab`. The matrices, however, are implemented as arrays. This project uses row major, which means that for a matrix of dimension n, the first n elements of the array is the first

row, the next n elements, the second row and so forth. The time span vector `tspan` has only two elements, regardless of the number of time steps, `nt`. If the method should used fixed step size, only `nt` should be changed, where in `Matlab` the `tspan` vector should be changed.

The methods take a parameter vector, `params`, and so do the functions described above, even though the function evaluations require no parameters. The functions get a const void pointer as input and if the parameters are to be used it should be cast to a pointer type, such as `double`.

It is the users responsibility to allocate a sufficient amount of memory for the approximations. If the output `N` is larger than or equal to `size` in the above example, it is recommended that the approximation is discarded, and the method is rerun with more memory allocated.

Note also that the functions `gfun` and `gJac` take `t` as input, even though $g(x(t))$ and $\frac{\partial}{\partial x}g(x(t))$ do not depend explicitly on t. This input is simply ignored by these functions and is only passed such that it matches the `ODEModel_t` type.

7.4 Test Results

In Figure 7.1 is shown the maximum element of the global error plotted against step size, when each method approximates the solution to (7.1) using fixed step size.

It is easily inspected that Euler's method is of first order, whether modified or not, the classic Runge-Kutta of fourth order, the Runge-Kutta-Fehlberg and Dormand-Prince methods of fifth order and ESDIRK23 of second order as was expected.

The global error of the modified RK4 and DOPRI54 does not continue to decrease when the step size, h, is decreased. This is because the accuracy of the modified methods is influenced by the user supplied $AbsTol$ and $RelTol$. These tolerances are set to $AbsTol_i = RelTol = 10^{-3}, \forall i$. The RKF45 is not affected by this for the step sizes tested and nor is Euler or ESDIRK23. The two latter methods, however, have very large errors compared to RKF45. For these three methods, the error when approximating this simple test problem, is the same whether modified or not

A peculiar detail is that the max-norm of the global error for Euler's method is exactly equal to the step size used. For the remaining methods the global error is far smaller than that of Euler's method, even for large step sizes. Even though order of convergence is not directly related to the actual size of the error, it is clearly seen that given a step size, the higher order methods have smaller errors.

The methods are tested on approximating the solution to 7.3 using adaptive step size, and their approximations are shown in Figure 7.2. The number of steps used and the global error of the approximation is shown in Table 7.1.

For the most part, the modified methods produce more accurate solutions than the unmodified methods. For the Euler method, the error is halved, where For RK4 it is increased slightly. For RKF45 it decreases by more than a factor 30. For DOPRI54 it is a little more than halved and for ESDIRK23 it is almost halved.

For the Euler method, the modified version uses four more steps, which is negligible compared to how many steps are used. The RK4 method uses more steps in the modified case, even though the approximation is less accurate. The RKF45 and DOPRI54 use one and two steps less in the modified case and the ESDIRK23 uses six more, which is negligible.

Method	Euler	RK4	RKF45	DOPRI54	ESDIRK23
Unmodified Methods					
Error	0.139	$0.183{\cdot}10^{-3}$	$5.656{\cdot}10^{-3}$	$0.545{\cdot}10^{-3}$	0.0192
Steps	104	21	16	16	55
Modified Methods					
Error	0.050967	$0.25826{\cdot}10^{-3}$	$0.16089{\cdot}10^{-3}$	$0.19964{\cdot}10^{-3}$	0.010263
Steps	108	27	15	14	61

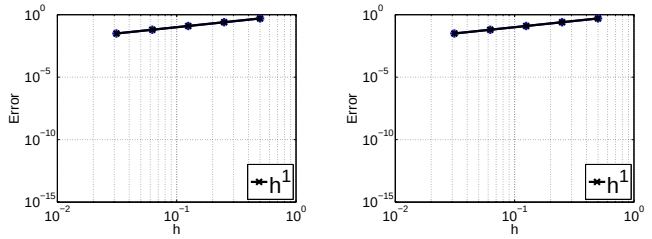
Table 7.1: Global error and number of steps for the approximations by the Runge-Kutta Toolbox methods approximating the solution of (7.3) (modified methods) and (7.6) (unmodified methods). The absolute and relative tolerances are both $AbsTol_i = RelTol = 10^{-3}$, $\forall i$.

In Figure 7.2a it can be seen that the approximation produced by the modified Euler method is more accurate around the local maximum around $t = 2\pi$ than that of the unmodified Euler. In Figure 7.2b it can be seen that the modified ESDIRK23 method uses more steps around the peaks at $t = \pi$ and $t = 3\pi$.

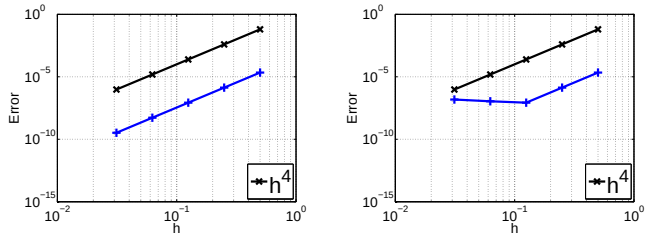
For the unmodified methods RK4 produces the best approximation in the sense that it has the lowest global error. The goal, however, when using adaptive step size is to use as large steps as possible and stay within the given tolerances. RK4 uses 21 steps, as opposed to RKF45 and DOPRI54 which both use 16. the unmodified RKF45 does not stay within the tolerance whereas the unmodified DOPRI54 does.

For the modified methods, both RKF45 and DOPRI54 stay within the limit and use far fewer steps than the other methods.

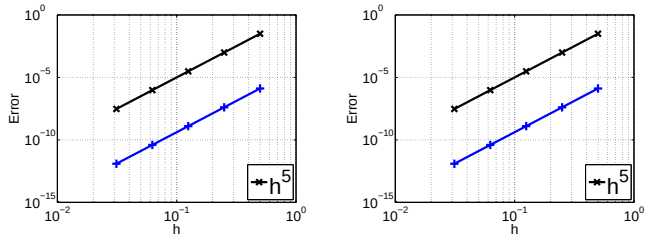
For both the modified and unmodified methods, Euler and ESDIRK23 does not stay within the tolerance, and use far more steps than the rest of the methods. However, for the Euler method, each step is cheaper with respect to computational cost, and if a low accuracy is required this method may be very fast. As for ESDIRK23, it may be superior for stiff problems, where larger steps are admitted. For this problem, however, the two methods perform considerably worse than the other three.



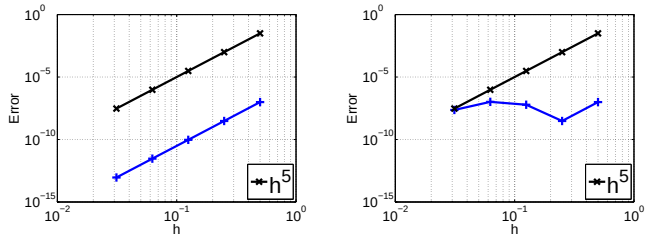
(a) Euler



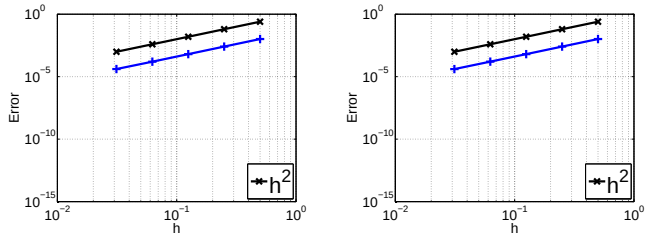
(b) Classic Runge-Kutta



(c) Runge-Kutta-Fehlberg

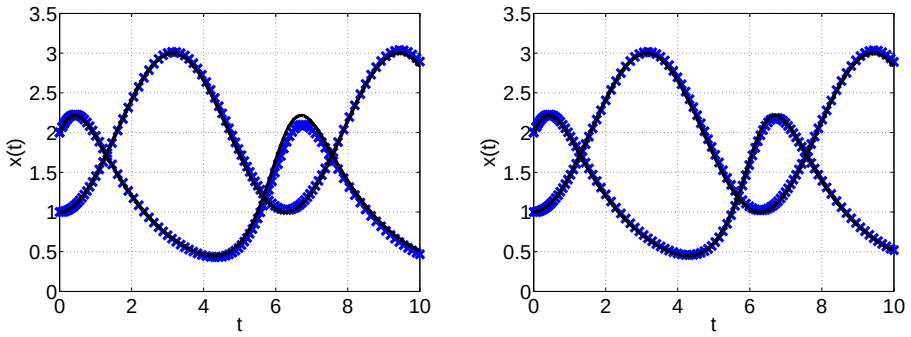


(d) Dormand-Prince

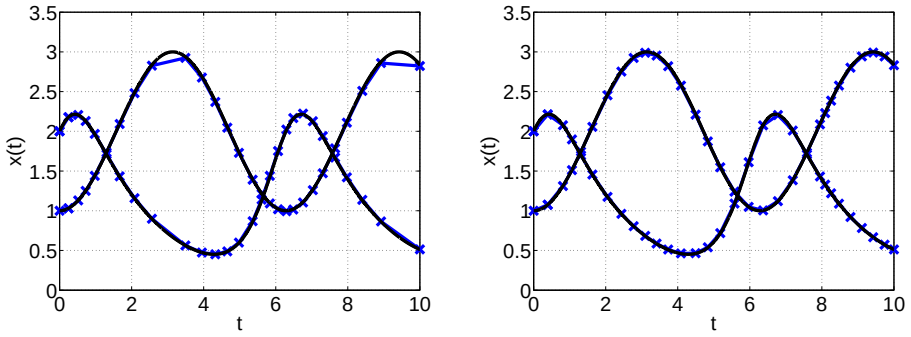


(e) ESDIRK23

Figure 7.1: Convergence tests for the Runge-Kutta Toolbox methods approximating the solution of (7.1) (blue +). Left: Unmodified methods. Right: Modified methods.



(a) Euler's method



(b) ESDIRK23

Figure 7.2: Plots of numerical approximations to the solution of (7.6) (left) and (7.3) (right) by the Runge-Kutta Toolbox methods using adaptive step size (blue x), together with the true solution (black line). The absolute and relative tolerances are both $AbsTol_i = RelTol = 10^{-3}$, $\forall i$.

7.5 Summary

This Chapter has presented example code for solving an IVP with the modified and unmodified versions of Euler and ESDIRK23. The remaining methods use the same syntax as Euler. Code is presented for both `Matlab` and `C`. All the methods were tested on the problems (7.1) and (7.3). Testing the methods on the former for different fixed step sizes showed that all the unmodified methods have the expected order of convergence. It also showed that the global error might behave differently for the modified methods since it then depends on the accuracy of Newton iterations as well, however for Euler, RKF45 and ESDIRK23 the modified and unmodified methods were equally accurate for this problem..

Testing the methods on (7.3) showed that Euler and ESDIRK23 are considerably less precise than the RK4, RKF45 and DOPRI54. In general the modified methods produced more accurate approximations, however, for RK4 the error was slightly larger for the modified method. The modified RKF45 produced a far more accurate approximation than the unmodified RKF45.

For Euler and ESDIRK23, the modified methods used a few more steps. The modified RK4 used 27 steps whereas the unmodified RK4 used 21, which is a relatively large difference, when the approximation is not more accurate. For RKF45 the number of steps is one less for the modified method, which is very good, considering that the error is more than 30 times smaller. For DOPRI54 the modified method uses two fewer steps and the error was more than halved.

Chapter 8

Comparison of Runtimes

In this Chapter, the methods of the Runge-Kutta Toolbox will be tested and compared on runtimes. The methods will be tested on the time it takes to do the computations needed to obtain the results, discussed in Section 6.4, where the optimal inlet rates are used but without any feedback.

The methods will be compared on using fixed step size, with low and high precision. For the explicit methods, precision plays no rule, however, for the modified methods and ESDIRK23 it does. Likewise, the methods will be tested on using adaptive step size, for both high and low precision. When using adaptive step size, all implemented methods are affected by the tolerances.

The two platforms, `Matlab` and `C` are also compared on the runtimes of the methods when using adaptive step size with low precision.

The simulations that are used to time the methods may be carried out either sequentially or in parallel. The sequential and parallel simulations are also timed for adaptive step size with low precision. The comparison is between sequential and parallel simulations in `Matlab`, parallel in `Matlab` and `C`, and sequential and parallel in `C`. The latter uses two implementations of parallel simulations. These two implementations are also tested against each other using adaptive step size with low precision and a varying number of processes.

8.1 Introduction

The simulation of the system in equation (6.15) will be done for a $\pm\%10$ interval of the values of the parameters γ_s, μ_{max}, K_S and K_I listed in Table 6.1. The simulations will be done for ten equally spaced values in the interval which results in 10^4 simulations. This procedure is exactly the same as the one used to obtain the results in Section 6.4. This is a realistic number of simulations and will show the great improvement in performance when using `C` over `Matlab` and when using parallel computing over serial computing.

Each method will be run with adaptive step size with high precision defined as $AbsTol_i = RelTol = 10^{-6}, \forall i$ and with low precision defined as $AbsTol_i = RelTol = 10^{-3}, \forall i$. The methods will also be run for the step sizes $h = 0.01, 0.005, 0.0025$, with low and high precision. This only affects the modified methods and ESDIRK23, which use the precision in the Newton iterations. The remaining methods produce the same simulations regardless of the supplied precision when using fixed step size.

The grave improvement, which is observed in the runtimes, is attainable because the simulations are completely independent of each other, which means that there is no need for communication between the processes.

A completely different example would be using parallel computing to do a single simulation of the system in equation 6.15 with piecewise approximations to the analytically optimal inlet rates as described in Section 6.5, which would only increase the runtime. In practice this is implemented as a series of simulations over each interval where the inlet rates are constant, i.e. a series of simulations similar to those needed to obtain the results in Section 6.3.

This would mean that only one process could run at a time, since each process would need the results from the previous as initial conditions. Hence, the only change when using parallel instead of serial computing would be the increase of communication between the processes and therefore increased runtime.

8.2 Comparison of Methods

This Section compares runtimes for the implemented methods using fixed and adaptive step size with both low and high precision, as defined in Section 8.1. The runtimes are obtained for the `C` implementations.

The runtimes are shown in Table 8.1. When the methods use fixed step size, the runtime decreases roughly linearly. For several of the modified methods, the runtime is a little less than doubled when the step size is halved, which may be explained by the Newton iterations requiring less iterations since the initial guess, which is the previous step, is better.

The unmodified explicit methods all have roughly the same runtime whether they are used with high or low precision as expected. The modified methods are slower when using fixed step size and high precision rather than low precision, however, only by between 10% and 20%.

For low precision and adaptive step size, the unmodified explicit methods take just about a second or less, whereas ESDIRK23 uses more than 5 seconds. The modified Euler, RKF45 and DOPRI54 use about 11 to 15 seconds, whereas RK4 uses more than 20. RK4 uses both step doubling and has many stages, whereas Euler uses step doubling but only has a single stage and RKF45 and DOPRI54 have many stages but use embedded error estimation. The modified ESDIRK23 uses 40 seconds, which is twice as much as any of the other methods when using low precision.

The modified ESDIRK23 is very competitive for fixed step sizes, compared to the unmodified ESDIRK23 which is only slightly faster, regardless of precision. This is not the case when using adaptive step size, where the unmodified ESDIRK23 is about 14 times faster.

The case is very different when using high precision and adaptive step size. Here Eulers method uses almost 13 seconds, more than twenty times as long as for low precision. RK4 uses 1.6 compared to 1 seconds for low precision, and RKF45 and DOPRI54 has the same runtime regardless of precision. ESDIRK23 uses almost 70 seconds. For the unmodified methods the tendency is that the high order methods, RK4, RKF45 and DOPRI54, do not suffer much from increasing the precision whereas Euler and ESDIRK23 spend a lot more time when using high precision and adaptive step size. This tendency carries over to the modified methods, where Euler uses almost 270 seconds and ESDIRK23, 525. However, the unmodified RKF45 and DOPRI54 didn't have higher runtimes when using high precision where the modified versions of these two, more than double in runtime.

In conclusion, the higher order methods may be used with advantage when using high precision, whereas the lower order methods may be effective for low precision. The modified methods are generally slower, except for ESDIRK23 for fixed step sizes where the modified and unmodified versions have runtimes close to each other.

Step size	Adap.	0.01	0.005	0.0025
High Precision				
Unmodified Methods				
Euler	12.90	0.65	1.30	2.61
RK4	1.61	5.39	10.73	21.31
RKF45	0.99	8.83	17.68	36.03
DOPRI54	1.10	10.12	20.13	40.64
ESDIRK23	69.57	44.58	85.40	171.72
Modified Methods				
Euler	268.18	34.50	66.49	128.51
RK4	35.25	121.96	240.03	475.84
RKF45	32.46	198.04	397.41	757.30
DOPRI54	30.10	185.58	362.95	720.34
ESDIRK23	525.81	49.73	96.75	183.90
Low Precision				
Unmodified Methods				
Euler	0.60	0.66	1.31	2.59
RK4	0.98	5.33	10.67	21.25
RKF45	0.99	8.77	17.60	35.01
DOPRI54	1.10	10.19	20.22	40.35
ESDIRK23	5.43	46.46	88.32	175.39
Modified Methods				
Euler	11.54	28.33	56.62	112.33
RK4	20.27	104.11	206.99	412.67
RKF45	12.84	171.07	344.21	668.91
DOPRI54	14.20	172.57	344.54	665.73
ESDIRK23	40.62	41.78	80.57	173.12

Table 8.1: Runtimes for the C implementations of the Runge-Kutta Toolbox methods when approximating the solution to the IVP (6.1) with the optimal inlet rates described in Section 6.4, for ten thousand sets of parameters. The methods use fixed and adaptive step size with both high and low precision.

8.3 Comparison of C and Matlab

This Section compares runtimes of sequential and parallel simulations in `Matlab` and `C`. In `C`, two implementations of parallel simulations are tested. In these tests the methods use adaptive step size and low precision.

`Matlab` can use a maximum of 12 processes and the runtimes for parallel simulations in `C`, shown in Table 8.2, all use 12 processes as well.

It is easily inspected that the simulations are much faster in `C`. For sequential Euler simulations, `C` is faster by a factor of 300. For the other unmodified methods, the speed-up from `C` to `Matlab` is between 100 and 250. For the modified methods, the speed-up is around 80 to 100, except for the modified ESDIRK23, whose runtime is about 230 longer in `Matlab`.

In `Matlab`, all the runtimes of the parallel simulations are between 11 and 12 times lower than that of sequential simulations, where 12 is maximum, when using 12 processes.

In `C`, the parallel simulations from Algorithm 21, are between 5 and 10 times faster, which is very low considering that the maximum is 12. This low speed-up may be due to some simulations being more time-consuming than others and hence some processes finish early and stand idle while the rest finish.

For most of the advanced parallel simulations, the speed-up is between 10 and 11, where 11 is now maximum since one process, the master node, is not doing any simulations. In any of the cases, the advanced parallel simulations are faster than the simple ones, however it may be expected that the simple is faster if every simulation takes the same time, at which all processes may terminate simultaneously.

The sequential simulations in `C` are faster than the parallel simulations in `Matlab` for the maximum number of processes.

In conclusion, `C` may be used with great advantage when it comes to runtimes, regardless of the number of processes used. The advanced parallel simulations are faster than the simple ones, even though the former sacrifices a processor to be a master node. These two implementations will be investigated further in Section 8.4.

Platform	Matlab		C		
Method	seq.	par.	seq.	par.	adv. par.
Unmodified Methods					
Euler	185.02	16.31	0.60	0.07	0.06
RK4	248.03	21.41	0.98	0.15	0.10
RKF45	217.48	18.41	0.99	0.17	0.10
DOPRI54	232.31	19.76	1.10	0.20	0.12
ESDIRK23	495.17	42.99	5.43	0.53	0.50
Modified Methods					
Euler	991.45	86.59	11.54	1.21	1.08
RK4	1594.69	137.85	20.27	2.71	1.92
RKF45	1019.51	88.08	12.84	1.86	1.22
DOPRI54	1083.01	93.52	14.20	2.07	1.42
ESDIRK23	9210.04	801.52	40.62	4.24	4.05

Table 8.2: Runtimes for the methods implemented in the Runge-Kutta Toolbox, run on C when approximating the solution to the IVP (6.1) for ten thousand sets of parameters. The methods use fixed step size with both high and low precision.

8.4 Comparison of Parallel Simulations in C

This Section compares the two implementations of parallel simulations, shown in Algorithms 21 and 22. The methods are run with adaptive step size and low precision and using a varying number of processes.

It can be seen from Table 8.3, that for the simple parallel simulations using several processes is faster than sequential simulations. However, doubling the number of processes does not always halve the runtime. For Euler, increasing the number of processes from 8 to 16 hardly decreases the runtime at all. However, the unmodified Euler and ESDIRK23 do experience a speed-up of almost a factor 2 when increasing the number of processes from 4 to 8. In many of the cases, the speed-up is as low as 3/2, when doubling the number of processes. The reason for this is that the runtime of a single simulation may be very much dependent on the set of parameters, when using adaptive step size, and hence some processes finish earlier than others and stand idle.

For the advanced parallel simulations, the case is somewhat different. This implementation cannot be used with a single process, since the master node does not do any simulations. However, the runtimes for 2 processes may be compared to those of sequential simulations, i.e. the simple parallel simulations using 1 process. Here we see that the runtimes are somewhat identical, actually

with a slight advantage for the advanced implementation, which may be due to some imprecision in the time measuring.

It should be kept in mind that for the advanced parallel simulations, the number of processes working is one less than the number of available processes. Hence, the speed-up from using 2 to 4 processes is close to 3, which is very reasonable. For 4 or more processes, the speed-up is close to 2 when doubling the number of processes, which indicate that the advanced parallel simulation is closer to optimal than the simple simulations.

The runtimes of the simple parallel simulations are faster for 2 and for some of the methods, also for 4 processes. For most of the methods, using the advanced implementation is faster or as fast as the simple, when using 8 processes, and when using 16, the advanced is faster for all the methods.

In conclusion, if a large number of processes is available, e.g. 16, the advanced implementation performs faster, but if only a few, e.g. 2 or 4 processes are available, the simple implementation may be the fastest.

Processors	1	2	4	8	16
Simple Parallel Simulations					
Unmodified Methods					
Euler	0.61	0.33	0.19	0.10	0.09
RK4	1.13	0.70	0.43	0.26	0.15
RKF45	1.15	0.75	0.46	0.27	0.16
DOPRI54	1.31	0.86	0.54	0.31	0.20
ESDIRK23	5.36	2.83	1.50	0.77	0.51
Modified Methods					
Euler	12.15	6.35	3.42	1.88	1.10
RK4	21.20	12.32	7.34	4.17	2.98
RKF45	13.34	8.04	4.89	2.79	2.01
DOPRI54	14.69	9.20	5.61	3.20	2.22
ESDIRK23	45.39	22.91	11.97	6.49	4.27
Advanced Parallel Simulations					
Unmodified Methods					
Euler	N/A	0.64	0.23	0.10	0.05
RK4	N/A	1.15	0.41	0.17	0.09
RKF45	N/A	1.13	0.39	0.17	0.09
DOPRI54	N/A	1.30	0.46	0.20	0.10
ESDIRK23	N/A	5.46	1.81	0.79	0.41
Modified Methods					
Euler	N/A	11.55	3.93	1.70	0.86
RK4	N/A	20.08	6.92	3.00	1.57
RKF45	N/A	12.49	4.43	1.87	0.95
DOPRI54	N/A	14.25	4.89	2.13	1.14
ESDIRK23	N/A	44.03	14.97	6.47	3.53

Table 8.3: Runtimes for the methods implemented in the Runge-Kutta Toolbox, run on C when approximating the solution to the problem stated in equation (6.15) for ten thousand different set of parameters. The methods use adaptive step size and low precision and are timed when using different number of processors.

8.5 Summary

In this Chapter, the methods in the Runge-Kutta Toolbox have been tested against each other on runtime. They have been compared when using both fixed and adaptive step size, and both high and low precision.

In Section 8.2 the implemented methods were tested against each other. It was observed that the higher order methods are faster when using adaptive step size, when high precision is required, whereas the lower order unmodified Euler is ultimately the fastest when using low precision, both for fixed and adaptive step size.

It was seen that when using fixed step size, the runtime roughly doubles when the step size is halved. For fixed step size, Euler is by far the fastest, regardless of precision and modification. For the unmodified methods, Euler is fastest, RK4 second, RKF45 third, DOPRI54 fourth and ESDIRK23 is the slowest. The same tendency is seen for the modified methods except that ESDIRK23 is faster than RK4.

The modified methods are generally slower than the unmodified, except for the modified ESDIRK23 which has nearly the same runtime as the unmodified for fixed step size. RKF45 and DOPRI54 has nearly the same runtimes, regardless of precision and modifications, with a slight advantage to RKF45.

In Section 8.3 it was shown that simulations were obtained far faster in `C` than in `Matlab`. The parallel simulations were between 11 and 12 times faster when using 12 processes in `Matlab`. In `C` the speed-up was between 5 and 10 when using the simple parallel simulations, where 5 is a very little speed-up. For the advanced parallel simulations, the speed-up was between 10 and 11, where 11 is the maximum, when one process is the master node.

In Section 8.4 it was seen that, the simple parallel simulations were fastest, when a small number of processes is used, e.g. 2 or 4. When using 4 or 8 processes some simulations were faster with the simple implementation and for some it was faster with the advanced. However, for 16 processes the advanced parallel simulations were faster for all methods.

Chapter 9

Conclusion

In this Chapter, the conclusions from the project are summarized. Which methods were used, after what principle were they implemented, how do the implementations perform in different settings. How may these methods be used to solve a problem in a test case. How fast are the implementations in `Matlab` and `C` and what speed-up may be achieved when doing parallel simulations.

9.1 Conclusion

In this project, the five methods, the explicit Euler, the Classical Runge-Kutta, Runge-Kutta-Fehlberg, Dormand-Prince and ESDIRK23 have been implemented. These methods are described in Sections 2.3 through 2.7 and have been implemented in an unmodified version, which approximates the solution to initial value problems of the kind

$$\frac{\mathrm{d}}{\mathrm{d}t}x(t) = f(t, x(t)), \quad x(t_0) = x_0,$$

and a modified version which approximates the solution to initial value problems of the kind

$$\frac{\mathrm{d}}{\mathrm{d}t}g(x(t)) = f(t, x(t)), \quad x(t_0) = x_0.$$

The implementation of both versions are described in Chapter 4 and both versions of all five methods have been implemented in both `Matlab` and `C`. The implementations are split up into a main algorithm which calls a step function algorithm. This may lower the number of function evaluations, e.g. when implementing step doubling for error estimation. The implementations may use fixed or adaptive step size where the latter requires an error estimate.

The Euler method and RK4 uses step doubling for error estimation whereas RKF45, DOPRI54 and ESDIRK23 are embedded methods, which use embedded error estimation.

Once the approximation and error estimate have been calculated when using adaptive step size, the step is either failed or accepted based on the error estimate and the step size is adjusted using either asymptotic or PI step size controller.

The modified methods and ESDIRK23 use Newton iterations and if these iterations do not converge sufficiently fast, the step is not updated and the step size is restricted according to the convergence rate of the Newton iterations, depending on whether they converged slowly or diverged.

The use of the implementations are described with an example for both `Matlab` and `C` in Sections 7.2 and 7.3, respectively. The methods were tested on the problem (7.1) using fixed step size to demonstrate convergence. For the unmodified methods, the convergence rates were as expected, Euler was first order, RK4 was fourth order, RKF45 and DOPRI54 were fifth order and ESDIRK23 was second order.

However, for the modified RK4 and DOPRI54, the error did not become arbitrarily low, which may be due to the Newton iterations whose approximate solution depends on the user supplied tolerances. Low tolerance was used as it is defined in Section 8.1.

The methods were also tested on the problem (7.3) using adaptive step size, all with low tolerance as it is defined in Section 8.1. This showed that Euler and ESDIRK23, whether modified or not, produced less accurate approximations than the other methods. The modified methods were generally more accurate, except that the modified RK4 had a larger error than the unmodified version of this.

The Euler method and ESDIRK23 produced less accurate approximations than RK4, RKF45 and DOPRI54, and they used far more steps. RK4 produced an adequately accurate approximation, however using more steps than RKF45 and DOPRI54. The unmodified RKF45 did not stay within the supplied tolerances whereas the modified version did. Both the modified and unmodified DOPRI54 was sufficiently accurate and used the least number of steps.

In Chapter 6, the unmodified RK4 was used with fixed step size to simulate a fed batch fermenter in operation for 10^4 sets of parameters, using different strategies for controlling inlet rates of substrate and water. It was shown that using feedback from both the substrate and biomass resulted in the distribution with the highest productions.

In Chapter 8 the implementations were tested against each other. It was observed that when using fixed step size, the runtime roughly doubles when the step size is halved. Of the unmodified methods, Euler was the fastest, followed by RK4, RKF45, DOPRI54 and last ESDIRK23. The same tendency is seen for the modified methods except that in this case, ESDIRK23 is faster than RK4.

Whether modified or not, the higher order methods, RK4, RKF45 and DOPRI54 were faster when using adaptive step size and high precision whereas the low order unmodified Euler method was the fastest when using low precision. ESDIRK23 was in any case very slow when using adaptive step size.

Generally the modified methods were slower than the unmodified, except for ESDIRK23 which had nearly the same runtime when using fixed step size, whether modified or not and regardless of precision.

In Section 8.3, it was shown that both the sequential and parallel simulations were obtained between 80 and 300 times faster in `C` compared to `Matlab`. Using parallel simulations was between 11 and 12 times faster than sequential simulations in `Matlab`, when using the maximum number of processes available in

`Matlab`, which is 12.

Using simple parallel simulations in `C` only gave a speed-up of between 5 and 10, compared to 12 which is maximum. Using the advanced parallel simulations, the speed-up was between 10 and 11, where 11 is maximum.

These two implementations of parallel simulations were tested further in Section 8.4, where it was concluded that for a low number of processes, e.g. 2 and 4, the simple parallel simulations were faster, however, the advanced was the fastest when using 16 processes.

Appendix A

Solving Linear Systems of Equations

This Chapter describes Gauss elimination which may be used in Newton iterations, to solve (2.7) for Δx^k. This method is equivalent to another method called LU-factorization, which is also described briefly. Both these methods require back substitution and solving a system of linear equations using LU-factorization also requires forward substitution, which both are also described.

A.1 Gaussian Elimination

Gaussian Elimination may be used for solving linear systems of equations such as the ones in Newton iterations. Say the system is

$$Ax = b, \tag{A.1}$$

where $A \in \mathbb{R}^{n \times n}$ and $x, b \in \mathbb{R}^n$. The Gaussian elimination algorithm transforms the system A.1 into an upper triangular form

$$Ux = c,$$

which may then be solved using back substitution. In `Matlab` solving a linear system of equations can be done by using the backslash operator \. A procedure for Gaussian elimination is shown in Algorithm 23. This algorithm assumes that the pivot elements are non-zero. It is also assumed that A and b are arrays which start in 0, e.g. the first elements are $A_{0,0}$ and b_0 as this is the case with `C` arrays. The transformation into the upper triangular form requires $O(n^3)$ flops. [Elden, 2010].

Data: A, b
Result: A is transformed to upper triangular form. b is changed accordingly.

for $k = 0 \ldots n-2$ **do**
  for $i = k+1 \ldots n-1$ **do**
    $m = \frac{A_{i,k}}{A_{k,k}}$;
    for $j = k+1 \ldots n-1$ **do**
      $A_{i,j} = A_{i,j} - m \cdot A_{k,j}$;
    end
    $b_i = b_i - m \cdot b_k$;
  end
end
assign $U_{i,j} = A_{i,j}$, for $j \geq i$ and $U_{i,j} = 0$ otherwise ;
assign $c = b$;

Algorithm 23: Gaussian elimination.

A.2 LU-Factorization

An alternative to Gaussian elimination is using LU-factorization on A. If A is nonsingular it may be put into the form

$$PA = LU,$$

where P is a permutation matrix, L is unit lower triangular matrix and U is an upper triangular matrix. Using this factorization, the linear system of equations (A.1), may be transformed into

$$PAx = LUx = Pb.$$

The last equation may be solved by

- Solving $Ly = Pb$ for y using forward substitution
- Solving $Ux = y$ for x using back substitution

Procedures for LU-factorization can be found in Chapter 8 of [Elden, 2010].

A.3 Back and Forward Substitution

Once a linear system of equations is on either upper triangular or lower triangular form it may be solved using either forward or back substitution. If the matrix is in upper triangular form it may be solved using backward substitution as shown in Algorithm 24. If it is in lower triangular form it may be solved using forward substitution as shown in Algorithm 25.

Data: U, c
Result: x which solves $Ux = c$.
$x_{n-1} = \frac{c_{n-1}}{U_{n-1,n-1}}$;
for $i = dim - 2 \ldots 0$ **do**
  $x_i = c_i$;
  for $j = i + 1 \ldots n - 1$ **do**
    $x_i = x_i - U_{i,j} x_j$;
  end
  $x_i = \frac{x_i}{U_{i,i}}$;
end

Algorithm 24: Back substitution.

Data: L, d
Result: y which solves $Ly = d$.
$y_0 = \frac{d_0}{L_{0,0}}$;
$y_{n-1} = \frac{d_{n-1}}{U_{n-1,n-1}}$ **for** $i = 1 \ldots dim - 1$ **do**
  $y_i = d_i$;
  for $j = 0 \ldots i - 1$ **do**
    $y_i = y_i - L_{i,j} y_j$;
  end
  $y_i = \frac{y_i}{L_{i,i}}$;
end

Algorithm 25: Forward substitution.

A.4 LAPACK In C

This Section describes the functions used to solve the linear system (2.13) in `C`. The library LAPACK provides several functions for linear algebra. In this project, it has only been used to solve linear systems of equations but may also be used to add vectors and matrices and much more. Netlib [2013].

The method `dgesv` may be used for solving a linear system of equations $Ax = B$, where $A \in \mathbb{R}^{N \times N}$ and $B \in \mathbb{R}^{N \times NRHS}$. `CBLAS_ORDER` indicates whether to use row major or column major and is set to `CblasRowMajor`. Since the matrix A is represented by a one-dimensional array, it is important to indicate whether the method should assume that each row is stored contigously or each column is stored contigously. This project stores each row contigously, as is mentioned in Section 7.3. The function call would be

```
clapack_dgesv(const enum CBLAS_ORDER Order,
              const int N,
              const int NRHS,
              double *A, const int lda, int *ipiv,
              double *b, const int ldb);
```

where `N` is the dimension of `A`, `NRHS` is the number of right-hand-sides which in this project is always 1. `lda` and `ldb` is in this project the same as `N`. `ipiv` is an output array which gives information about the row interchanges made during the solve.

Effectively `dgesv` calls the functions `dgetrf` to compute an LU-factorization, and then solves the factorized system using `dgetrs`. Since the Jacobi ma-

trix $\frac{\partial}{\partial x}R(x^k)$ is approximated with $\frac{\partial}{\partial x}R(x^0)$ in the Newton iterations, the LU-factorization may be reused in each Newton iteration, thus saving a lot of LU-factorizations. The factorization is called as

```
clapack_dgetrf(const enum CBLAS_ORDER Order,
               const int M,
               const int N,
               double *A, const int lda, int *ipiv);
```

where `M` is equal to `N`. This may then be solved using

```
clapack_dgetrs(const enum CBLAS_ORDER Order,
               const enum CBLAS_TRANSPOSE Trans,
               const int N,
               const int NRHS,
               double *A, const int lda, int *ipiv,
               double *B, const int ldb);
```

where `CBLAS_TRANSPOSE` should be set to `CblasNoTrans`.

Appendix B

Implementations In C

This Chapter includes the source code for the methods in `C`. Each implementation is split up into the main function and the step function. The implementations are split up into two files. One containing the unmodified methods, and one containing the modified methods. The former file also contains The Newton function which is used in the explicit modified methods together with help functions, which adds vectors, finds minimum or maximum element of an array etc. Both files contain a type definition of `ODEModel_t` which the methods expect as input for the functions $f(t, x(t))$, $\frac{\partial}{\partial x} f(t, x(t))$, $g(x(t))$ and $\frac{\partial}{\partial x} g(x(t))$.

B.1 Unmodified Methods

Listing B.1: Implementation of the unmodified methods in C

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <string.h>
5 #include "clapack.h"
6 #include "RungeKuttaToolbox.h"
7
8 int Euler(
```

```
9  ODEModel_t *fun,
10 int nx, int nt,
11 double *tspan,
12 double *x0,
13 double *AbsTol, double RelTol,
14 void *params,
15 double *t,
16 double *x){
17
18 int i, firststep, fixedstepsize = nt > 2;
19 double h = 0.001, epsilon = 0.8, p = 1, phat = p+1, kp =
       ...0.4/phat, kI = 0.3/phat, E, Eold;
20 double *fn    = malloc(nx*sizeof(double));
21 double *xfull = malloc(nx*sizeof(double)), *xhalf = malloc(
       ...nx*sizeof(double)), *xdouble = malloc(nx*sizeof(
       ...double));
22 double *e     = malloc(nx*sizeof(double)), *absx  = malloc(
       ...nx*sizeof(double)), *Tol     = malloc(nx*sizeof(
       ...double));
23
24 // Initial time and initial conditions
25 t[0] = tspan[0];
26 Vadd1((x+0*nx),1.0,x0,nx);
27 // If using fixed step size
28 if(fixedstepsize){
29  h = (tspan[1] - tspan[0])/nt;
30  // For every step
31  for(i = 0; i < nt; i++){
32   t[i+1] = t[i] + h;
33   fun(t[i],(x+i*nx),params,fn);
34   EulerStep(fun,nx,t[i],(x+i*nx),fn,h,params,(x+(i+1)*nx));
35  }
36 }else{
37 // If using adaptive step size
38  firststep = 1;
39  i = 0;
40  // While end time has not been reached
41  while(t[i] < tspan[1]){
42   // Make sure end time is not passed
43   if(h > tspan[1] - t[i]){
44    h = tspan[1] - t[i];
45   }
46   // Full step
47   fun(t[i],(x+i*nx),params,fn);
48   EulerStep(fun,nx,t[i],(x+i*nx),fn,h,params,xfull);
49   // Double step
50   EulerStep(fun,nx,t[i],(x+i*nx),fn,h/2,params,xhalf);
```

```
51      fun(t[i]+h/2,xhalf,params,fn);
52      EulerStep(fun,nx,t[i]+h/2,xhalf,fn,h/2,params,xdouble);
53      // Error estimate
54      Vadd2(e,1.0,xfull,-1.0,xdouble,nx);
55      Vabs(e,e,nx);
56      Vabs(absx,xfull,nx);
57      Vadd2(Tol,1.0,AbsTol,RelTol,absx,nx);
58      Vdiv(e,e,Tol,nx);
59      E = max(e,nx);
60      // Avoid division by zero in PI step size
61      if(E < pow(10,-10)){ E = pow(10,-10); }
62      // Fail or accept step
63      if(E <= 1){
64       // Update step
65       t[i+1] = t[i] + h;
66       Vadd1((x+(i+1)*nx),1.0,xdouble,nx);
67       if(firststep){
68        // New asymptotic step size
69        h = h*pow(epsilon/E,1.0/phat);
70        firststep = 0;
71       }else{
72        // New PI step size
73        h = h*pow(epsilon/E,kI)*pow(Eold/E,kp);
74       }
75       // Save error for use in the PI step size controller
76       Eold = E;
77       i++;
78
79      }else{
80       // New asymptotic step size
81       h = h*pow(epsilon/E,1.0/phat);
82      }
83     }
84
85    }
86    free(fn); free(xfull); free(xhalf); free(xdouble); free(e);
          ... free(absx); free(Tol);
87    // Return number of steps
88    return i;
89  }
90
91  void EulerStep(
92    ODEModel_t *fun,
93    int nx,
94    double tn,
95    double *xn,
96    double *fn,
```

```
97  double h,
98  void *params,
99  double *xnp1){
100
101 // Next step
102 Vadd2(xnp1,1.0,xn,h,fn,nx);
103 }
104
105 int RK4(
106  ODEModel_t *fun,
107  int nx, int nt,
108  double *tspan,
109  double *x0,
110  double *AbsTol, double RelTol,
111  void *params,
112  double *t,
113  double *x){
114  int i, firststep, fixedstepsize = nt > 2, s = 4;
115  double h = 0.001, epsilon = 0.8, p = 4, phat = p+1, kp =
         ...0.4/(p+1), kI = 0.3/(p+1), E, Eold;
116  double *fn    = malloc(nx*sizeof(double));
117  double *xfull = malloc(nx*sizeof(double)), *xhalf = malloc(
         ...nx*sizeof(double)), *xdouble = malloc(nx*sizeof(
         ...double));
118  double *e     = malloc(nx*sizeof(double)), *absx  = malloc(
         ...nx*sizeof(double)), *Tol     = malloc(nx*sizeof(
         ...double));
119  double A[s][s], b[] = {1/6.0,1/3.0,1/3.0,1/6.0}, c[] =
         ...{0.0,1/2.0,1/2.0,1.0};
120  A[1][0] = 1/2.0, A[2][1] = 1/2.0, A[3][2] = 1.0;
121  // Initial time and initial conditions
122  t[0] = tspan[0];
123  Vadd1((x+0*nx),1.0,x0,nx);
124  // If using fixed step size
125  if(fixedstepsize){
126   h = (tspan[1] - tspan[0])/nt;
127   // For every step
128   for(i = 0; i < nt; i++){
129    t[i+1] = t[i] + h;
130    fun(t[i],(x+i*nx),params,fn);
131    RK4Step(fun,nx,t[i],(x+i*nx),fn,h,params,(x+(i+1)*nx),A,b
           ...,c);
132   }
133  }else{
134  // If using adaptive step size
135   firststep = 1;
136   i = 0;
```

```
137   // While end time has not been reached
138   while(t[i] < tspan[1]){
139    // Make sure the end time is not passed
140    if(h > tspan[1] - t[i]){
141     h = tspan[1] - t[i];
142    }
143    // Full step
144    fun(t[i],(x+i*nx),params,fn);
145    RK4Step(fun,nx,t[i],(x+i*nx),fn,h,params,xfull,A,b,c);
146    // Double step
147    RK4Step(fun,nx,t[i],(x+i*nx),fn,h/2,params,xhalf,A,b,c);
148    fun(t[i]+h/2,xhalf,params,fn);
149    RK4Step(fun,nx,t[i]+h/2,xhalf,fn,h/2,params,xdouble,A,b,c
           ...);
150    // Error estimate
151    Vadd2(e,1.0,xfull,-1.0,xdouble,nx);
152    Vabs(e,e,nx);
153    Vabs(absx,xfull,nx);
154    Vadd2(Tol,1.0,AbsTol,RelTol,absx,nx);
155    Vdiv(e,e,Tol,nx);
156    E = max(e,nx);
157    // Avoid division by zero in PI step size
158    if(E < pow(10,-10)){ E = pow(10,-10); }
159    // Accept or fail step
160    if(E <= 1){
161     // Update step
162     t[i+1] = t[i] + h;
163     Vadd1((x+(i+1)*nx),1.0,xdouble,nx);
164     if(firststep){
165      // New asymptotic step size
166      h = h*pow(epsilon/E,1.0/phat);
167      firststep = 0;
168     }else{
169      // New PI step size
170      h = h*pow(epsilon/E,kI)*pow(Eold/E,kp);
171     }
172     // Save error for PI step size controller
173     Eold = E;
174     i++;
175
176    }else{
177     // New asymptotic step size
178     h = h*pow(epsilon/E,1.0/phat);
179    }
180   }
181
182  }
```

```
183  free(fn); free(xfull); free(xhalf); free(xdouble); free(e);
         ... free(absx); free(Tol);
184  // Return number of steps taken
185  return i;
186 }
187
188 void RK4Step(
189  ODEModel_t *fun,
190  int nx,
191  double tn,
192  double *xn,
193  double *fn,
194  double h,
195  void *params,
196  double *xnp1,
197  double A[][4], double b[], double c[]){
198  double *X2 = malloc(nx*sizeof(double)), *X3 = malloc(nx*
         ...sizeof(double)), *X4 = malloc(nx*sizeof(double));
199  double *f2 = malloc(nx*sizeof(double)), *f3 = malloc(nx*
         ...sizeof(double)), *f4 = malloc(nx*sizeof(double));
200  // Stage 2
201  Vadd2(X2,1,xn,A[1][0]*h,fn,nx);
202  fun(tn+c[1]*h,X2,params,f2);
203  // Stage 3
204  Vadd2(X3,1,xn,A[2][1]*h,f2,nx);
205  fun(tn+c[2]*h,X3,params,f3);
206  // Stage 4
207  Vadd2(X4,1,xn,A[3][2]*h,f3,nx);
208  fun(tn+c[3]*h,X4,params,f4);
209  // Next step
210  Vadd5(xnp1,1,xn,b[0]*h,fn,b[1]*h,f2,b[2]*h,f3,b[3]*h,f4,nx)
         ...;
211  free(X2); free(X3); free(X4); free(f2); free(f3); free(f4);
212 }
213
214 int RKF45(
215  ODEModel_t *fun,
216  int nx, int nt,
217  double *tspan,
218  double *x0,
219  double *AbsTol, double RelTol,
220  void *params,
221  double *t,
222  double *x){
223
224  int i, firststep, fixedstepsize = nt > 2, s = 6;
```

```
225 double h = 0.001, epsilon = 0.8, p = 4, phat = p+1, kp =
        ...0.4/(p+1), kI = 0.3/(p+1), E, Eold;
226 double *fn    = malloc(nx*sizeof(double));
227 double *xfull = malloc(nx*sizeof(double));
228 double *e     = malloc(nx*sizeof(double)), *absx  = malloc(
        ...nx*sizeof(double)), *Tol = malloc(nx*sizeof(double))
        ...;
229 double A[s][s], b[] =
        ...{25/216.0,0.0,1408/2565.0,2197/4104.0,-1/5.0,0.0},
        ...bhat[] =
        ...{16/135.0,0.0,6656/12825.0,28561/56430.0,-9/50.0,2/55.0};
        ...
230 double c[] = {0.0,1/4.0,3/8.0,12/13.0,1.0,1/2.0}, d[s];
231 A[1][0] =  1/4.0;
232 A[2][0] =      3/32.0, A[2][1] =       9/32.0;
233 A[3][0] = 1932/2197.0, A[3][1] = -7200/2197.0, A[3][2] =
        ...7296/2197.0;
234 A[4][0] =   439/216.0, A[4][1] =   -8.0, A[4][2] =
        ...3680/513.0, A[4][3] = -845/4104.0;
235 A[5][0] =     -8/27.0, A[5][1] =          2.0, A[5][2] =
        ...-3544/2565.0, A[5][3] = 1859/4104.0, A[5][4] =
        ...-11/40.0;
236 // Calculate d = b-bhat
237 Vadd2(d,1.0,b,-1.0,bhat,s);
238 // Initial time and initial conditions
239 t[0] = tspan[0];
240 Vadd1((x+0*nx),1.0,x0,nx);
241 // If using adaptive step size
242 if(fixedstepsize){
243  h = (tspan[1] - tspan[0])/nt;
244  // For every step
245  for(i = 0; i < nt; i++){
246   t[i+1] = t[i] + h;
247   fun(t[i],(x+i*nx),params,fn);
248   RKF45Step(fun,nx,t[i],(x+i*nx),fn,h,params,(x+(i+1)*nx),e
        ...,A,bhat,c,d);
249  }
250 }else{
251 // If using adaptive step size
252  firststep = 1;
253  i = 0;
254  // While end time has not been reached
255  while(t[i] < tspan[1]){
256   // Make sure sure end time is not passed
257   if(h > tspan[1] - t[i]){
258    h = tspan[1] - t[i];
259   }
```

```
260     // Full step
261     fun(t[i],(x+i*nx),params,fn);
262     RKF45Step(fun,nx,t[i],(x+i*nx),fn,h,params,xfull,e,A,bhat
            ...,c,d);
263     // Error estimate
264     Vabs(e,e,nx);
265     Vabs(absx,xfull,nx);
266     Vadd2(Tol,1.0,AbsTol,RelTol,absx,nx);
267     Vdiv(e,e,Tol,nx);
268     E = max(e,nx);
269     // Avoid division by zero in PI step size
270     if(E < pow(10,-10)){ E = pow(10,-10); }
271     // Accept or fail step
272     if(E <= 1){
273      // Update step
274      t[i+1] = t[i] + h;
275      Vadd1((x+(i+1)*nx),1.0,xfull,nx);
276      if(firststep){
277       // New asymptotic step size
278       h = h*pow(epsilon/E,1.0/phat);
279       firststep = 0;
280      }else{
281       // New PI step size
282       h = h*pow(epsilon/E,kI)*pow(Eold/E,kp);
283      }
284      // Save error for PI step size controller
285      Eold = E;
286      i++;
287
288     }else{
289      // New asymptotic step size
290      h = h*pow(epsilon/E,1.0/phat);
291     }
292    }
293
294  }
295  free(fn); free(xfull); free(e); free(absx); free(Tol);
296  // Return number of steps
297  return i;
298 }
299
300 void RKF45Step(
301  ODEModel_t *fun,
302  int nx,
303  double tn,
304  double *xn,
305  double *fn,
```

```
306  double h,
307  void *params,
308  double *xnp1,
309  double *e,
310  double A[][6], double bhat[], double c[], double d[]){
311
312  double *X2 = malloc(nx*sizeof(double)), *X3 = malloc(nx*
         ...sizeof(double)), *X4 = malloc(nx*sizeof(double));
313  double *X5 = malloc(nx*sizeof(double)), *X6 = malloc(nx*
         ...sizeof(double));
314  double *f2 = malloc(nx*sizeof(double)), *f3 = malloc(nx*
         ...sizeof(double)), *f4 = malloc(nx*sizeof(double));
315  double *f5 = malloc(nx*sizeof(double)), *f6 = malloc(nx*
         ...sizeof(double));
316  // Stage 2
317  Vadd2(X2,1,xn,A[1][0]*h,fn,nx);
318  fun(tn+c[1]*h,X2,params,f2);
319  // Stage 3
320  Vadd3(X3,1,xn,A[2][0]*h,fn,A[2][1]*h,f2,nx);
321  fun(tn+c[2]*h,X3,params,f3);
322  // Stage 4
323  Vadd4(X4,1,xn,A[3][0]*h,fn,A[3][1]*h,f2,A[3][2]*h,f3,nx);
324  fun(tn+c[3]*h,X4,params,f4);
325  // Stage 5
326  Vadd5(X5,1,xn,A[4][0]*h,fn,A[4][1]*h,f2,A[4][2]*h,f3,A
         ...[4][3]*h,f4,nx);
327  fun(tn+c[4]*h,X5,params,f5);
328  // Stage 6
329  Vadd6(X6,1,xn,A[5][0]*h,fn,A[5][1]*h,f2,A[5][2]*h,f3,A
         ...[5][3]*h,f4,A[5][4]*h,f5,nx);
330  fun(tn+c[5]*h,X6,params,f6);
331  // Next step
332  Vadd6(xnp1,1,xn,bhat[0]*h,fn,bhat[2]*h,f3,bhat[3]*h,f4,bhat
         ...[4]*h,f5,bhat[5]*h,f6,nx);
333  // Embedded error estimate
334  Vadd5(e,d[0]*h,fn,d[2]*h,f3,d[3]*h,f4,d[4]*h,f5,d[5]*h,f6,
         ...nx);
335
336  free(X2); free(X3); free(X4); free(X5); free(X6);
337  free(f2); free(f3); free(f4); free(f5); free(f6);
338 }
339
340
341
342 int DOPRI54(
343  ODEModel_t *fun,
344  int nx, int nt,
```

```
345 double *tspan,
346 double *x0,
347 double *AbsTol, double RelTol,
348 void *params,
349 double *t,
350 double *x){
351 int i, firststep, fixedstepsize = nt > 2, s = 7;
352 double h = 0.001, epsilon = 0.8, p = 4, phat = p+1, kp =
      ...0.4/(p+1), kI = 0.3/(p+1), E, Eold;
353 double *fn    = malloc(nx*sizeof(double)), *fnp1  = malloc(
      ...nx*sizeof(double));
354 double *xfull = malloc(nx*sizeof(double));
355 double *e     = malloc(nx*sizeof(double)), *absx  = malloc(
      ...nx*sizeof(double)), *Tol = malloc(nx*sizeof(double))
      ...;
356 double A[s][s], b[] =
      ...{5179/57600.0,0.0,7571/16695.0,393/640.0,-92097/339200.0,187/2100.0,1/
      ...
357 double bhat[] =
      ...{35/384.0,0.0,500/1113.0,125/192.0,-2187/6784.0,11/84.0,0.0},
      ... c[] = {0.0,1/5.0,3/10.0,4/5.0,8/9.0,1.0,1.0};
358 double d[s];
359 A[1][0] =        1/5.0;
360 A[2][0] =       3/40.0, A[2][1] =        9/40.0;
361 A[3][0] =      44/45.0, A[3][1] =      -56/15.0, A[3][2] =
      ...      32/9.0;
362 A[4][0] = 19372/6561.0, A[4][1] = -25360/2187.0, A[4][2] =
      ...64448/6561.0, A[4][3] =   -212/729.0;
363 A[5][0] =  9017/3168.0, A[5][1] =     -355/33.0, A[5][2] =
      ...46732/5247.0, A[5][3] =     49/176.0, A[5][4] =
      ...-5103/18656.0;
364 A[6][0] =     35/384.0,                           A[6][2] =
      ...  500/1113.0, A[6][3] =    125/192.0, A[6][4] =
      ...-2187/6784.0, A[6][5] = 11/84.0;
365 // Calculate d = b-bhat
366 Vadd2(d,1.0,b,-1.0,bhat,s);
367 // Initial time and initial conditions
368 t[0] = tspan[0];
369 Vadd1((x+0*nx),1.0,x0,nx);
370 fun(t[0],(x+0*nx),params,fn);
371 // If using fixed step size
372 if(fixedstepsize){
373  h = (tspan[1] - tspan[0])/nt;
374  // For every step
375  for(i = 0; i < nt; i++){
376   t[i+1] = t[i] + h;
```

```
377     DOPRI54Step(fun,nx,t[i],(x+i*nx),fn,h,params,(x+(i+1)*nx)
            ...,e,fn,A,c,d);
378   }
379  }else{
380  // If using adaptive step size
381   firststep = 1;
382   i = 0;
383   // While end time has not been reached
384   while(t[i] < tspan[1]){
385    // Make sure end time is not passed
386    if(h > tspan[1] - t[i]){
387     h = tspan[1] - t[i];
388    }
389    // Full step
390    DOPRI54Step(fun,nx,t[i],(x+i*nx),fn,h,params,xfull,e,fnp1
            ...,A,c,d);
391    // Error estimate
392    Vabs(e,e,nx);
393    Vabs(absx,xfull,nx);
394    Vadd2(Tol,1.0,AbsTol,RelTol,absx,nx);
395    Vdiv(e,e,Tol,nx);
396    E = max(e,nx);
397    // Avoid division by zero in PI step size
398    if(E < pow(10,-10)){ E = pow(10,-10); }
399    // Accept or fail step
400    if(E <= 1){
401     // Update step
402     t[i+1] = t[i] + h;
403     Vadd1((x+(i+1)*nx),1.0,xfull,nx);
404     Vadd1(fn,1.0,fnp1,nx);
405     if(firststep){
406      // New asymptotic step size
407      h = h*pow(epsilon/E,1.0/phat);
408      firststep = 0;
409     }else{
410      // New PI step size
411      h = h*pow(epsilon/E,kI)*pow(Eold/E,kp);
412     }
413     // Save error for PI step size controller
414     Eold = E;
415     i++;
416
417    }else{
418     // New asymptotic step size
419     h = h*pow(epsilon/E,1.0/phat);
420    }
421   }
```

```
422
423  }
424  free(fn); free(fnp1); free(xfull); free(e); free(absx);
         ...free(Tol);
425  // Return number of steps
426  return i;
427 }
428
429 void DOPRI54Step(
430  ODEModel_t *fun,
431  int nx,
432  double tn,
433  double *xn,
434  double *fn,
435  double h,
436  void *params,
437  double *xnp1,
438  double *e,
439  double *fnp1,
440  double A[][7], double c[], double d[]){
441
442  double *X2 = malloc(nx*sizeof(double)), *X3 = malloc(nx*
         ...sizeof(double)), *X4 = malloc(nx*sizeof(double));
443  double *X5 = malloc(nx*sizeof(double)), *X6 = malloc(nx*
         ...sizeof(double)), *X7 = malloc(nx*sizeof(double));
444  double *f2 = malloc(nx*sizeof(double)), *f3 = malloc(nx*
         ...sizeof(double)), *f4 = malloc(nx*sizeof(double));
445  double *f5 = malloc(nx*sizeof(double)), *f6 = malloc(nx*
         ...sizeof(double)), *f7 = malloc(nx*sizeof(double));
446  // Stage 2
447  Vadd2(X2,1,xn,A[1][0]*h,fn,nx);
448  fun(tn+h/5.0,X2,params,f2);
449  // Stage 3
450  Vadd3(X3,1,xn,A[2][0]*h,fn,A[2][1]*h,f2,nx);
451  fun(tn+h*3/10.0,X3,params,f3);
452  // Stage 4
453  Vadd4(X4,1,xn,A[3][0]*h,fn,A[3][1]*h,f2,A[3][2]*h,f3,nx);
454  fun(tn+h*4/5.0,X4,params,f4);
455  // Stage 5
456  Vadd5(X5,1,xn,A[4][0]*h,fn,A[4][1]*h,f2,A[4][2]*h,f3,A
         ...[4][3]*h,f4,nx);
457  fun(tn+h*8/9.0,X5,params,f5);
458  // Stage 6
459  Vadd6(X6,1,xn,A[5][0]*h,fn,A[5][1]*h,f2,A[5][2]*h,f3,A
         ...[5][3]*h,f4,A[5][4]*h,f5,nx);
460  fun(tn+h      ,X6,params,f6);
461  // Stage 7 is also the next step
```

```
462  Vadd6(xnp1,1,xn,A[6][0]*h,fn,A[6][2]*h,f3,A[6][3]*h,f4,A
         ...[6][4]*h,f5,A[6][5]*h,f6,nx);
463  fun(tn+h,    xnp1,params,f7);
464  // Embedded error estimation
465  Vadd6(e,d[0]*h,fn,d[2]*h,f3,d[3]*h,f4,d[4]*h,f5,d[5]*h,f6,d
         ...[6]*h,f7,nx);
466  // The function evaluation is saved and reused
467  Vadd1(fnp1,1.0,f7,nx);
468
469  free(X2); free(X3); free(X4); free(X5); free(X6); free(X7);
470  free(f2); free(f3); free(f4); free(f5); free(f6); free(f7);
471 }
472
473 int ESDIRK23(
474  ODEModel_t *fun,
475  ODEModel_t *Jac,
476  int nx, int nt,
477  double *tspan,
478  double *x0,
479  double *AbsTol, double RelTol,
480  void *params,
481  double *t,
482  double *x){
483
484  int i, firststep, fixedstepsize = nt > 2, s = 3;
485  double h = 0.001, halpha, hold, hmax = 10, hmin = 0.000001,
         ... epsilon = 0.8, p = 2, phat = p+1, kp = 1/phat, kI =
         ... 1/phat, E, Eold;
486  double *fn    = malloc(nx*sizeof(double)), *fnp1  = malloc(
         ...nx*sizeof(double));
487  double *xfull = malloc(nx*sizeof(double));
488  double *e     = malloc(nx*sizeof(double)), *absx  = malloc(
         ...nx*sizeof(double)), *Tol = malloc(nx*sizeof(double))
         ...;
489  double gamma = 1-1.0/sqrt(2.0), a31 = (1.0-gamma)/2, b[] =
         ...{a31,a31,gamma};
490  double bhat[] = {(6*gamma-1)/(12*gamma),1/(12*gamma*(1-2*
         ...gamma)),(1-3*gamma)/(3*(1-2*gamma))}, c[] = {0,2*
         ...gamma,1}, d[s];
491  double Convergence, Divergence, alpharatio, *stepoutput;
492  // Calculate d = b-bhat
493  Vadd2(d,1.0,b,-1.0,bhat,s);
494  // Initial time and initial conditions
495  t[0] = tspan[0];
496  Vadd1((x+0*nx),1.0,x0,nx);
497  fun(t[0],(x+0*nx),params,fn);
498  // If using fixed step size
```

```
499  if(fixedstepsize){
500   h = (tspan[1] - tspan[0])/nt;
501   // For every step
502   for(i = 0; i < nt; i++){
503    t[i+1] = t[i] + h;
504    stepoutput = ESDIRK23Step(fun,Jac,nx,t[i],(x+i*nx),fn,h,
           ...AbsTol,RelTol,params,(x+(i+1)*nx),e,fn,b,c,d,gamma
           ...);
505   }
506  }else{
507  // If using adaptive step size
508   firststep = 1;
509   i = 0;
510   // While end time has not been reached
511   while(t[i] < tspan[1]){
512    // Make sure end time is not passed
513    if(h > tspan[1] - t[i]){
514     h = tspan[1] - t[i];
515    }
516    // Full step
517    stepoutput = ESDIRK23Step(fun,Jac,nx,t[i],(x+i*nx),fn,h,
           ...AbsTol,RelTol,params,xfull,e,fnp1,b,c,d,gamma);
518    Convergence = stepoutput[0], Divergence = stepoutput[1],
           ...alpharatio = stepoutput[2]; free(stepoutput);
519    // Error estimate
520    Vabs(e,e,nx);
521    Vabs(absx,xfull,nx);
522    Vadd2(Tol,1.0,AbsTol,RelTol,absx,nx);
523    Vdiv(e,e,Tol,nx);
524    E = max(e,nx);
525    // Avoid division by zero in PI step size
526    if(E < pow(10,-10)){ E = pow(10,-10); }
527    if(Convergence){
528     // Fail or accept step
529     if(E <= 1){
530      // Update step
531      t[i+1] = t[i] + h;
532      Vadd1((x+(i+1)*nx),1.0,xfull,nx);
533      Vadd1(fn,1.0,fnp1,nx);
534      if(firststep){
535       // New asymptotic step size
536       h = h*pow(epsilon/E,1.0/phat);
537       // Make sure step size is not too small or too large
538       //if(h > hmax){ h = hmax; }else if(h < hmin){ h = hmin
               ...; }
539       firststep = 0;
540      }else{
```

```
541       // New PI step size
542       h = h*(h/hold)*pow(epsilon/E,kI)*pow(Eold/E,kp);
543       // Make sure step size is not too small or too large
544       if(h > hmax){ h = hmax; }else if(h < hmin){ h = hmin;
              ...}
545      }
546      // Save error and step size for use in PI step size
             ...controller
547      Eold = E;
548      hold = h;
549      i++;
550     }else{
551      // New asymptotic step size
552      h = h*pow(epsilon/E,1.0/phat);
553      // Make sure step size is not too small or too large
554      if(h > hmax){ h = hmax; }else if(h < hmin){ h = hmin; }
555     }
556
557     if(alpharatio < 1){
558      h = h*alpharatio;
559     }
560    }else if(Divergence){
561     halpha = h*alpharatio;
562     if(halpha > 0.5*h){ h = halpha; }else{ h = 0.5*h; }
563    }else{
564     if(alpharatio < 1){
565      halpha = h*alpharatio;
566      if(halpha > 0.5*h){ h = halpha; }else{ h = 0.5*h; }
567     }else{
568      h = 0.5*h;
569     }
570    }
571   }
572  }
573  free(fn); free(fnp1); free(xfull); free(e); free(absx);
         ...free(Tol);
574  // Return number of steps
575  return i;
576 }
577
578 double *ESDIRK23Step(
579  ODEModel_t *fun,
580  ODEModel_t *Jac,
581  int nx,
582  double tn,
583  double *xn,
584  double *fn,
```

```
585 double h,
586 double *AbsTol, double RelTol,
587 void *params,
588 double *xnp1,
589 double *e,
590 double *fnp1,
591 double b[], double c[], double d[], double gamma){
592
593 int i, Convergence, SlowConvergence = 0, Divergence = 0,
        ...iter, itermax = 10;
594 double epsilon = 0.8, tau = 0.1*epsilon, alpha = 0.0,
        ...alpharef = 0.4;
595
596 double *X2   = malloc(nx*sizeof(double)), *X3   = malloc(nx
        ...*sizeof(double));
597 double *f2   = malloc(nx*sizeof(double)), *f3   = malloc(nx
        ...*sizeof(double));
598 double *phi2 = malloc(nx*sizeof(double)), *phi3 = malloc(nx
        ...*sizeof(double)),   *R  = malloc(nx*sizeof(double));
599 double *absX = malloc(nx*sizeof(double)), *Tol  = malloc(nx
        ...*sizeof(double)),   *dX = malloc(nx*sizeof(double));
600 double *J = malloc(nx*nx*sizeof(double)), *dRdx = malloc(nx
        ...*nx*sizeof(double)), *I = calloc(nx*nx,sizeof(double
        ...));
601 double T, rNewton, rNewtonOld, a21 = gamma, *output =
        ...malloc(sizeof(double)*3);
602 // Parameters used for LAPACK functions dgetrf and dgetrs
603 const enum CBLAS_ORDER Order = CblasRowMajor;
604 const enum CBLAS_TRANSPOSE Trans = CblasNoTrans;
605 int N = nx, M = N, NRHS = 1, LDA = nx, LDB = N, *IPIV =
        ...malloc(nx*sizeof(int));
606 // Jacobian Update
607 Jac(tn,xn,params,J);
608 for(i = 0; i < nx; i++){ *(I+i*(nx+1)) = 1.0; }
609 Madd2(nx,dRdx,1.0,I,-h*gamma,J);
610 // LU Factorization of dRdX
611 clapack_dgetrf(Order,M,N,dRdx,LDA,IPIV);
612 // Stage 2 of the ESDIRK23 method
613 Vadd2(phi2,1.0,xn,h*a21,fn,nx);
614 // Initial guess for the state by Euler step
615 T = tn + c[1]*h;
616 Vadd2(X2,1.0,xn,c[1]*h,fn,nx);
617 // Newton iterations
618 fun(T,X2,params,f2);
619 // R = X2 - h*gamma*f2 - phi2
620 Vadd3(R,1.0,X2,-h*gamma,f2,-1.0,phi2,nx);
621 // rNewton = || |R|/(AbsTol + |X2|*RelTol) ||_inf
```

```
622  Vabs(e,R,nx);
623  Vabs(absX,X2,nx);
624  Vadd2(Tol,1.0,AbsTol,RelTol,absX,nx);
625  Vdiv(e,e,Tol,nx);
626  rNewton = max(e,nx);
627  rNewtonOld = rNewton;
628  iter = 0;
629  Convergence = 0;
630  while(!Convergence && !SlowConvergence && !Divergence){
631   // Solve (I - h*gamma*J) dX = R for dX (notice the lacking
         ... minus). dX is stored in R
632   clapack_dgetrs(Order,Trans,N,NRHS,dRdx,LDA,IPIV,R,LDB);
633   //Backslash(nx,dRdx,dX,R);
634   // Update X2 by adding -dX (notice the minus)
635   Vadd2(X2,1.0,X2,-1.0,R,nx);
636   fun(T,X2,params,f2);
637   // R = X2 - h*gamma*f2 - phi2
638   Vadd3(R,1.0,X2,-h*gamma,f2,-1.0,phi2,nx);
639   // rNewton = || |R|/(AbsTol + |X2|*RelTol) ||_inf
640   Vabs(e,R,nx);
641   Vabs(absX,X2,nx);
642   Vadd2(Tol,1.0,AbsTol,RelTol,absX,nx);
643   Vdiv(e,e,Tol,nx);
644   rNewton = max(e,nx);
645   if(alpha < rNewton/rNewtonOld){ alpha = rNewton/rNewtonOld
         ...; }
646   Convergence = rNewton < tau;
647   SlowConvergence = iter > itermax;
648   Divergence  = alpha > 1;
649   rNewtonOld = rNewton;
650   iter++;
651  }
652  // Stage 3 of the ESDIRK23 method
653  Vadd3(phi3,1.0,xn,b[0]*h,fn,b[1]*h,f2,nx);
654  // Initial guess for the state
655  T = tn + h;
656  Vadd2(X3,1.0,xn,h,fn,nx);
657  // Newton iterations
658  fun(T,X3,params,f3);
659  // R = X3 - h*gamma*f3 - phi3
660  Vadd3(R,1.0,X3,-h*gamma,f3,-1.0,phi3,nx);
661  // rNewton = || |R|/(AbsTol + |X3|*RelTol) ||_inf
662  Vabs(e,R,nx);
663  Vabs(absX,X3,nx);
664  Vadd2(Tol,1.0,AbsTol,RelTol,absX,nx);
665  Vdiv(e,e,Tol,nx);
666  rNewton = max(e,nx);
```

```
667  rNewtonOld = rNewton;
668  iter = 0;
669  Convergence = 0;
670  while(!Convergence && !SlowConvergence && !Divergence){
671   // Solve (I - h*gamma*J) dX = R for dX (notice the lacking
         ... minus)
672   clapack_dgetrs(Order,Trans,N,NRHS,dRdx,LDA,IPIV,R,LDB);
673   //Backslash(nx,dRdx,dX,R);
674   // Update X3 by adding -dX (notice the minus)
675   Vadd2(X3,1.0,X3,-1.0,R,nx);
676   fun(T,X3,params,f3);
677   // R = X3 - h*gamma*f3 - phi3
678   Vadd3(R,1.0,X3,-h*gamma,f3,-1.0,phi3,nx);
679   // rNewton = || |R|/(AbsTol + |X3|*RelTol) ||_inf
680   Vabs(e,R,nx);
681   Vabs(absX,X3,nx);
682   Vadd2(Tol,1.0,AbsTol,RelTol,absX,nx);
683   Vdiv(e,e,Tol,nx);
684   rNewton = max(e,nx);
685   if(alpha < rNewton/rNewtonOld){ alpha = rNewton/rNewtonOld
         ...; }
686   Convergence = rNewton < tau;
687   SlowConvergence = iter > itermax;
688   Divergence  = alpha > 1;
689   rNewtonOld = rNewton;
690   iter++;
691  }
692  // Update step
693  Vadd1(xnp1,1.0,X3,nx);
694  // Embedded error estimate
695  Vadd3(e,d[0]*h,fn,d[1]*h,f2,d[2]*h,f3,nx);
696  // The function evaluation is saved and reused
697  Vadd1(fnp1,1.0,f3,nx);
698  // Return Convergence booleans and alpha ratio
699  output[0] = Convergence, output[1] = Divergence, output[2]
         ...= alpharef/alpha;
700
701  free(X2); free(X3);   free(f2);  free(f3); free(phi2); free
         ...(phi3);
702  free(R);  free(absX); free(Tol); free(dX);
703  free(J);  free(dRdx); free(I);   free(IPIV);
704
705  return output;
706 }
707
708 void NewtonSolve(
709  ODEModel_t *gfun,
```

```
710  ODEModel_t *gJac,
711  int nx,
712  double *gnp1,
713  double *x0,
714  double *AbsTol, double RelTol,
715  void *params,
716  double *xnp1,
717  double *output){
718  int Convergence, SlowConvergence = 0, Divergence = 0,
         ...itermax = 10, iter;
719  double epsilon = 0.8, tau = 0.1*epsilon, alpha = 0.0,
         ...alpharef = 0.4;
720  double *g = malloc(nx*sizeof(double));
721  double *R = malloc(nx*sizeof(double)), *absgnp1 = malloc(nx
         ...*sizeof(double)), *Tol = malloc(nx*sizeof(double));
722  double *e = malloc(nx*sizeof(double)), *dgdx = malloc(nx*nx
         ...*sizeof(double)), *dxn = malloc(nx*sizeof(double));
723  double rNewton, rNewtonOld;
724  // Parameters used for LAPACK functions dgetrf and dgetrs
725  const enum CBLAS_ORDER Order = CblasRowMajor;
726  const enum CBLAS_TRANSPOSE Trans = CblasNoTrans;
727  int N = nx, M = N, NRHS = 1, LDA = nx, LDB = N, *IPIV =
         ...malloc(nx*sizeof(int));
728  // Initial guess for xnp1 is the previous step
729  Vadd1(xnp1,1.0,x0,nx);
730  // Calculate residual
731  gfun(0,xnp1,params,g);
732  gJac(0,xnp1,params,dgdx);
733  // LU Factorization of dgdx
734  clapack_dgetrf(Order,M,N,dgdx,LDA,IPIV);
735  // Residual
736  Vadd2(R,1.0,g,-1.0,gnp1,nx);
737  Vabs(e,R,nx);
738  Vabs(absgnp1,gnp1,nx);
739  Vadd2(Tol,1.0,AbsTol,RelTol,absgnp1,nx);
740  Vdiv(e,e,Tol,nx);
741  // rNewton = || |R|/(AbsTol + |gnp1|*RelTol) ||_inf
742  rNewton = max(e,nx);
743  rNewtonOld = rNewton;
744  iter = 0;
745  // Check for convergence
746  Convergence = rNewton < tau;
747  while(!Convergence && !Divergence && !SlowConvergence){
748   // Solve Dg*dxn = R = (g-gnp1). dxn is stored in R
749   clapack_dgetrs(Order,Trans,N,NRHS,dgdx,LDA,IPIV,R,LDB);
750   //Backslash(nx,dgdx,dxn,R);
751   // Update xnp1 by adding -dxn (notice the minus)
```

```
752   Vadd2(xnp1,1.0,xnp1,-1.0,R,nx);
753   gfun(0,xnp1,params,g);
754   // Calculate residual
755   Vadd2(R,1.0,g,-1.0,gnp1,nx);
756   Vabs(e,R,nx);
757   Vadd2(Tol,1.0,AbsTol,RelTol,absgnp1,nx);
758   Vdiv(e,e,Tol,nx);
759   // rNewton = || |R|/(AbsTol + |gnp1|*RelTol) ||_inf
760   rNewton = max(e,nx);
761   if(alpha < rNewton/rNewtonOld){ alpha = rNewton/rNewtonOld
          ...; }
762   Convergence = rNewton < tau;
763   SlowConvergence = iter > itermax;
764   Divergence  = alpha > 1;
765   rNewtonOld = rNewton;
766   iter++;
767  }
768  free(g); free(R); free(absgnp1); free(Tol); free(e); free(
          ...dgdx); free(dxn); free(IPIV);
769  // Return Convergence booleans and alpha ratio
770  output[0] = Convergence, output[1] = Divergence, output[2]
          ...= alpharef/alpha;
771 }
772
773 void Backslash(int nx, double *A, double *x, double *b){
774  double *U = malloc(nx*nx*sizeof(double)), *c = malloc(nx*
          ...sizeof(double));
775  memcpy(U,A,nx*nx*sizeof(double));
776  memcpy(c,b,nx*sizeof(double));
777  // Solves Ax = b
778  // GaussianElimination puts A in an upper triangular form,
          ...U
779  GaussianElimination(nx,U,c);
780  // BackSubstituion solves Ux = c
781  BackSubstitution(nx,U,x,c);
782  free(U); free(c);
783 }
784
785 void GaussianElimination(int nx, double *A, double *b){
786  int i,j,k;
787  double m;
788  for(k = 0; k < nx-1; k++){
789   for(i = k+1; i < nx; i++){
790    m = *(A+i*nx+k)/(*(A+k*nx+k));
791    for(j = k+1; j < nx; j++){
792     *(A+i*nx+j) -= *(A+k*nx+j)*m;
793    }
```

```
794     b[i] = b[i] - m*b[k];
795    }
796   }
797  }
798
799  void BackSubstitution(int nx, double *U, double *x, double *
         ...c){
800   int i,j;
801   x[nx-1] = c[nx-1]/(*(U + (nx-1)*nx + (nx-1)));
802   for(i = nx-2; i >= 0; i--){
803    x[i] = c[i];
804    for(j = i+1; j < nx; j++){
805     x[i] -= *(U+i*nx+j)*(x[j]);
806    }
807    x[i] = x[i]/(*(U+i*nx+i));
808   }
809  }
810
811  void Vdiv(double *A, double *B, double *C, int nx){
812   // Divide two vectors elemenwise
813   // Answer is stored in A
814   int j; for(j = 0; j < nx; j++){ A[j] = B[j]/C[j]; }
815  }
816
817  void Madd2(int nx, double *A, double b, double *B, double c,
         ... double *C){
818   int i;
819   for(i = 0; i < nx; i++){
820    Vadd2((A+i*nx),b,(B+i*nx),c,(C+i*nx),nx);
821   }
822  }
823
824  void Vadd7(double *A, double b, double *B, double c, double
         ...*C, double d, double *D, double e, double *E, double
         ...f, double *F, double g, double *G, double h, double *
         ...H, int nx){
825   // Add six vectors, each with a coefficient
826   // Answer is stored in A
827   int j; for(j = 0; j < nx; j++){ A[j] = b*B[j] + c*C[j] + d*
          ...D[j] + e*E[j] + f*F[j] + g*G[j] + h*H[j]; }
828  }
829
830  void Vadd6(double *A, double b, double *B, double c, double
         ...*C, double d, double *D, double e, double *E, double
         ...f, double *F, double g, double *G, int nx){
831   // Add six vectors, each with a coefficient
832   // Answer is stored in A
```

```
833  int j; for(j = 0; j < nx; j++){ A[j] = b*B[j] + c*C[j] + d*
         ...D[j] + e*E[j] + f*F[j] + g*G[j]; }
834 }
835
836 void Vadd5(double *A, double b, double *B, double c, double
         ...*C, double d, double *D, double e, double *E, double
         ...f, double *F, int nx){
837  // Add five vectors, each with a coefficient
838  // Answer is stored in A
839  int j; for(j = 0; j < nx; j++){ A[j] = b*B[j] + c*C[j] + d*
         ...D[j] + e*E[j] + f*F[j]; }
840 }
841
842 void Vadd4(double *A, double b, double *B, double c, double
         ...*C, double d, double *D, double e, double *E, int nx)
         ...{
843  // Add four vectors, each with a coefficient
844  // Answer is stored in A
845  int j; for(j = 0; j < nx; j++){ A[j] = b*B[j] + c*C[j] + d*
         ...D[j] + e*E[j]; }
846 }
847
848 void Vadd3(double *A, double b, double *B, double c, double
         ...*C, double d, double *D, int nx){
849  // Add three vectors, each with a coefficient
850  // Answer is stored in A
851  int j; for(j = 0; j < nx; j++){ A[j] = b*B[j] + c*C[j] + d*
         ...D[j]; }
852 }
853
854 void Vadd2(double *A, double b, double *B, double c, double
         ...*C, int nx){
855  // Add two vectors, each with a coefficient
856  // Answer is stored in A
857  int j; for(j = 0; j < nx; j++){ A[j] = b*B[j] + c*C[j]; }
858 }
859
860 void Vadd1(double *A, double b, double *B, int nx){
861  // Assign one vector with a coefficient
862  // Answer is stored in A
863  int j; for(j = 0; j < nx; j++){ A[j] = b*B[j]; }
864 }
865
866 void Vprint(double *A, int nx){
867  // Print vector
868  int j; for(j = 0; j < nx; j++){ printf(" %2.4f ",A[j]); }
         ...printf("\n");
```

```
869 }
870
871 void Vabs(double *A, double *B, int nx){
872  // Take absolute value of a vector
873  // Answer is stored in A
874  int j; for(j = 0; j < nx; j++){ A[j] = fabs(B[j]); }
875 }
876
877 double max(double *A, int nx){
878  // Return maximum element in A
879  int j; double maxelem = A[0]; for(j = 1; j < nx; j++){ if(A
         ...[j] > maxelem){ maxelem = A[j]; }} return maxelem;
880 }
881
882 double min(double *A, int nx){
883  // Return minimum element in A
884  int j; double minelem = A[0]; for(j = 1; j < nx; j++){ if(A
         ...[j] < minelem){ minelem = A[j]; }} return minelem;
885 }
```

B.2 Modified Methods

Listing B.2: Implementation of the modified methods in C

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include <string.h>
5  #include "clapack.h"
6  #include "RungeKuttaToolbox.h"
7
8  int EulerMod(
9   ODEModel_t *fun,
10  ODEModel_t *gfun,
11  ODEModel_t *gJac,
12  int nx, int nt,
13  double *tspan,
14  double *x0,
15  double *AbsTol, double RelTol,
16  void *params,
17  double *t,
18  double *x){
19
20  int i,j, firststep, fixedstepsize = nt > 2;
```

```
21 double h = 0.001, halpha, hmin = 0.000001, hmax = 10,
       ...epsilon = 0.8;
22 double p = 1, phat = p+1, kp = 0.4/phat, kI = 0.3/phat, E,
       ...Eold;
23 double *fn      = malloc(nx*sizeof(double));
24 double *xfull   = malloc(nx*sizeof(double)), *xhalf =
       ...malloc(nx*sizeof(double)), *xdouble = malloc(nx*
       ...sizeof(double));
25 double *e       = malloc(nx*sizeof(double)), *absx  =
       ...malloc(nx*sizeof(double)), *Tol     = malloc(nx*
       ...sizeof(double));
26 double *gnp1    = malloc(nx*sizeof(double)), *gfull =
       ...malloc(nx*sizeof(double)), *ghalf   = malloc(nx*
       ...sizeof(double));
27 double *gdouble = malloc(nx*sizeof(double));
28 double Convergence, Con, Divergence, Div, alpharatio, *
       ...stepoutput;
29 // Initial time and initial conditions
30 t[0] = tspan[0];
31 Vadd1((x+0*nx),1.0,x0,nx);
32 gfun(t[0],x0,params,gnp1);
33 // If using fixed step size
34 if(fixedstepsize){
35  h = (tspan[1] - tspan[0])/nt;
36  // For every step
37  for(i = 0; i < nt; i++){
38   t[i+1] = t[i] + h;
39   fun(t[i],(x+i*nx),params,fn);
40   stepoutput = EulerModStep(fun,gfun,gJac,nx,t[i],(x+i*nx),
         ...fn,gnp1,h,AbsTol,RelTol,params,(x+(i+1)*nx),gnp1);
         ... free(stepoutput);
41  }
42 }else{
43 // If using adaptive step size
44  firststep = 1;
45  i = 0;
46  // While end time has not been reached
47  while(t[i] < tspan[1]){
48   // Make sure end time is not passed
49   if(h > tspan[1] - t[i]){
50    h = tspan[1] - t[i];
51   }
52   // Full step
53   fun(t[i],(x+i*nx),params,fn);
54   stepoutput = EulerModStep(fun,gfun,gJac,nx,t[i],(x+i*nx),
         ...fn,gnp1,h,AbsTol,RelTol,params,xfull,gfull);
```

```
55    Convergence = stepoutput[0], Divergence = stepoutput[1];
          ...free(stepoutput);
56    // Double step
57    stepoutput = EulerModStep(fun,gfun,gJac,nx,t[i],(x+i*nx),
          ...fn,gnp1,h/2,AbsTol,RelTol,params,xhalf,ghalf);
58    Con = stepoutput[0], Div = stepoutput[1]; free(stepoutput
          ...);
59    Convergence = Convergence && Con;
60    Divergence  = Divergence  || Div;
61    fun(t[i]+h/2,xhalf,params,fn);
62    stepoutput = EulerModStep(fun,gfun,gJac,nx,t[i]+h/2,xhalf
          ...,fn,ghalf,h/2,AbsTol,RelTol,params,xdouble,gdouble
          ...);
63    Con = stepoutput[0], Div = stepoutput[1], alpharatio =
          ...stepoutput[2]; free(stepoutput);
64
65    Convergence = Convergence && Con;
66    Divergence  = Divergence  || Div;
67    // Error estimate
68    Vadd2(e,1.0,xfull,-1.0,xdouble,nx);
69    Vabs(e,e,nx);
70    Vabs(absx,xfull,nx);
71    Vadd2(Tol,1.0,AbsTol,RelTol,absx,nx);
72    Vdiv(e,e,Tol,nx);
73    E = max(e,nx);
74    // Avoid division by zero in PI step size
75    if(E < pow(10.0,-10.0)){ E = pow(10.0,-10.0); }
76    if(Convergence){
77     // Fail or accept the step
78     if(E <= 1){
79      // Update step
80      t[i+1] = t[i] + h;
81      Vadd1((x+(i+1)*nx),1.0,xdouble,nx);
82      Vadd1(gnp1,1.0,gdouble,nx);
83      if(firststep){
84       // New asymptotic step size
85       h = h*pow(epsilon/E,1.0/phat);
86       // Make sure step size is not too small or too large
87       if(h > hmax){ h = hmax; }else if(h < hmin){ h = hmin;
             ...}
88       firststep = 0;
89      }else{
90       // New PI step size
91       h = h*pow(epsilon/E,kI)*pow(Eold/E,kp);
92       // Make sure step size is not too small or too large
93       if(h > hmax){ h = hmax; }else if(h < hmin){ h = hmin;
             ...}
```

```
94        }
95        // Save error for use in the PI step size controller
96        Eold = E;
97        i++;
98
99       }else{
100       // New asymptotic step size
101       h = h*pow(epsilon/E,1.0/phat);
102       // Make sure step size is not too small or too large
103       if(h > hmax){ h = hmax; }else if(h < hmin){ h = hmin; }
104      }
105
106      if(alpharatio < 1){ h = h*alpharatio; }
107     }else if(Divergence){
108      halpha = h*alpharatio;
109      if(halpha > 0.5*h){ h = halpha; }else{ h = 0.5*h; }
110     }else{
111      if(alpharatio < 1){
112       halpha = h*alpharatio;
113       if(halpha > 0.5*h){ h = halpha; }else{ h = 0.5*h; }
114      }else{
115       h = 0.5*h;
116      }
117     }
118    }
119   }
120   free(fn);   free(xfull); free(xhalf); free(xdouble); free(e
          ...);  free(absx); free(Tol);
121   free(gnp1); free(gfull); free(ghalf); free(gdouble);
122   // Return number of steps
123   return i;
124  }
125
126  double *EulerModStep(
127   ODEModel_t *fun,
128   ODEModel_t *gfun,
129   ODEModel_t *gJac,
130   int nx,
131   double tn,
132   double *xn,
133   double *fn, double *gn,
134   double h,
135   double *AbsTol, double RelTol,
136   void *params,
137   double *xnp1, double *gnp1){
138
139   // Next step
```

```
140  double *output = malloc(3*sizeof(double));
141  double *Conv = malloc(4*sizeof(double)), *Div = malloc(4*
         ...sizeof(double)), *alpharat = malloc(4*sizeof(double)
         ...);
142
143  Vadd2(gnp1,1,gn,h,fn,nx);
144  NewtonSolve(gfun,gJac,nx,gnp1,xn,AbsTol,RelTol,params,xnp1,
         ...output);
145
146  free(Conv); free(Div); free(alpharat);
147  return output;
148 }
149
150 int RK4Mod(
151  ODEModel_t *fun,
152  ODEModel_t *gfun,
153  ODEModel_t *gJac,
154  int nx, int nt,
155  double *tspan,
156  double *x0,
157  double *AbsTol, double RelTol,
158  void *params,
159  double *t,
160  double *x){
161
162  int i,j, firststep, fixedstepsize = nt > 2, s = 4;;
163  double h = 0.001, halpha, hmin = 0.000000001, hmax = 10,
         ...epsilon = 0.8;
164  double p = 4, phat = p+1, kp = 0.4/phat, kI = 0.3/phat, E,
         ...Eold;
165  double *fn      = malloc(nx*sizeof(double));
166  double *xfull   = malloc(nx*sizeof(double)), *xhalf =
         ...malloc(nx*sizeof(double)), *xdouble = malloc(nx*
         ...sizeof(double));
167  double *e       = malloc(nx*sizeof(double)), *absx  =
         ...malloc(nx*sizeof(double)), *Tol     = malloc(nx*
         ...sizeof(double));
168  double *gnp1    = malloc(nx*sizeof(double)), *gfull =
         ...malloc(nx*sizeof(double)), *ghalf   = malloc(nx*
         ...sizeof(double));
169  double *gdouble = malloc(nx*sizeof(double));
170  double Convergence, Con, Divergence, Div, alpharatio, *
         ...stepoutput;
171  double A[s][s], b[] = {1/6.0,1/3.0,1/3.0,1/6.0}, c[] =
         ...{0.0,1/2.0,1/2.0,1.0};
172  A[1][0] = 1/2.0, A[2][1] = 1/2.0, A[3][2] = 1.0;
173  // Initial time and initial conditions
```

```
174  t[0] = tspan[0];
175  Vadd1((x+0*nx),1.0,x0,nx);
176  gfun(t[0],x0,params,gnp1);
177  // If using fixed step size
178  if(fixedstepsize){
179   h = (tspan[1] - tspan[0])/nt;
180   // For every step
181   for(i = 0; i < nt; i++){
182    t[i+1] = t[i] + h;
183    fun(t[i],(x+i*nx),params,fn);
184    stepoutput = RK4ModStep(fun,gfun,gJac,nx,t[i],(x+i*nx),fn
          ...,gnp1,h,AbsTol,RelTol,params,(x+(i+1)*nx),gnp1,A,b
          ...,c); free(stepoutput);
185   }
186  }else{
187  // If using adaptive step size
188   firststep = 1;
189   i = 0;
190   // While end time has not been reached
191   while(t[i] < tspan[1]){
192    // Make sure end time is not passed
193    if(h > tspan[1] - t[i]){
194     h = tspan[1] - t[i];
195    }
196    // Full step
197    fun(t[i],(x+i*nx),params,fn);
198    stepoutput = RK4ModStep(fun,gfun,gJac,nx,t[i],(x+i*nx),fn
          ...,gnp1,h,AbsTol,RelTol,params,xfull,gfull,A,b,c);
199    Convergence = stepoutput[0], Divergence = stepoutput[1];
          ...free(stepoutput);
200    // Double step
201    stepoutput = RK4ModStep(fun,gfun,gJac,nx,t[i],(x+i*nx),fn
          ...,gnp1,h/2,AbsTol,RelTol,params,xhalf,ghalf,A,b,c);
202    Con = stepoutput[0], Div = stepoutput[1]; free(stepoutput
          ...);
203    Convergence = Convergence && Con;
204    Divergence  = Divergence  || Div;
205    fun(t[i]+h/2,xhalf,params,fn);
206    stepoutput = RK4ModStep(fun,gfun,gJac,nx,t[i]+h/2,xhalf,
          ...fn,ghalf,h/2,AbsTol,RelTol,params,xdouble,gdouble,
          ...A,b,c);
207    Con = stepoutput[0], Div = stepoutput[1], alpharatio =
          ...stepoutput[2]; free(stepoutput);
208
209    Convergence = Convergence && Con;
210    Divergence  = Divergence  || Div;
211    // Error estimate
```

```
212     Vadd2(e,1.0,xfull,-1.0,xdouble,nx);
213     Vabs(e,e,nx);
214     Vabs(absx,xfull,nx);
215     Vadd2(Tol,1.0,AbsTol,RelTol,absx,nx);
216     Vdiv(e,e,Tol,nx);
217     E = max(e,nx);
218     // Avoid division by zero in PI step size
219     if(E < pow(10.0,-10.0)){ E = pow(10.0,-10.0); }
220     if(Convergence){
221      // Fail or accept the step
222      if(E <= 1){
223       // Update step
224       t[i+1] = t[i] + h;
225       Vadd1((x+(i+1)*nx),1.0,xdouble,nx);
226       Vadd1(gnp1,1.0,gdouble,nx);
227       if(firststep){
228        // New asymptotic step size
229        h = h*pow(epsilon/E,1.0/phat);
230        // Make sure step size is not too small or too large
231        if(h > hmax){ h = hmax; }else if(h < hmin){ h = hmin;
               ...}
232        firststep = 0;
233       }else{
234        // New PI step size
235        h = h*pow(epsilon/E,kI)*pow(Eold/E,kp);
236        // Make sure step size is not too small or too large
237        if(h > hmax){ h = hmax; }else if(h < hmin){ h = hmin;
               ...}
238       }
239       // Save error for use in the PI step size controller
240       Eold = E;
241       i++;
242      }else{
243       // New asymptotic step size
244       h = h*pow(epsilon/E,1.0/phat);
245       // Make sure step size is not too small or too large
246       if(h > hmax){ h = hmax; }else if(h < hmin){ h = hmin; }
247      }
248
249      if(alpharatio < 1){ h = h*alpharatio; }
250     }else if(Divergence){
251      halpha = h*alpharatio;
252      if(halpha > 0.5*h){ h = halpha; }else{ h = 0.5*h; }
253     }else{
254      if(alpharatio < 1){
255       halpha = h*alpharatio;
256       if(halpha > 0.5*h){ h = halpha; }else{ h = 0.5*h; }
```

```
257     }else{
258      h = 0.5*h;
259     }
260    }
261   }
262  }
263  free(fn);   free(xfull); free(xhalf); free(xdouble); free(e
         ...);  free(absx); free(Tol);
264  free(gnp1); free(gfull); free(ghalf); free(gdouble);
265  // Return number of steps
266  return i;
267 }
268
269 double *RK4ModStep(
270  ODEModel_t *fun,
271  ODEModel_t *gfun,
272  ODEModel_t *gJac,
273  int nx,
274  double tn,
275  double *xn,
276  double *fn, double *gn,
277  double h,
278  double *AbsTol, double RelTol,
279  void *params,
280  double *xnp1, double *gnp1,
281  double A[][4], double b[], double c[]){
282
283  // Next step
284  double *G2 = malloc(nx*sizeof(double)), *G3 = malloc(nx*
         ...sizeof(double)), *G4 = malloc(nx*sizeof(double));
285  double *X2 = malloc(nx*sizeof(double)), *X3 = malloc(nx*
         ...sizeof(double)), *X4 = malloc(nx*sizeof(double));
286  double *f2 = malloc(nx*sizeof(double)), *f3 = malloc(nx*
         ...sizeof(double)), *f4 = malloc(nx*sizeof(double));
287  double *output = malloc(3*sizeof(double));
288  double *Conv = malloc(4*sizeof(double)), *Div = malloc(4*
         ...sizeof(double)), *alpharat = malloc(4*sizeof(double)
         ...);
289  // Stage 2
290  Vadd2(G2,1,gn,A[1][0]*h,fn,nx);
291  NewtonSolve(gfun,gJac,nx,G2,xn,AbsTol,RelTol,params,X2,
         ...output);
292  Conv[0] = output[0], Div[0] = output[1], alpharat[0] =
         ...output[2];
293  fun(tn+h*c[1],X2,params,f2);
294  // Stage 3
295  Vadd2(G3,1,gn,A[2][1]*h,f2,nx);
```

```
296  NewtonSolve(gfun,gJac,nx,G3,X2,AbsTol,RelTol,params,X3,
         ...output);
297  Conv[1] = output[0], Div[1] = output[1], alpharat[1] =
         ...output[2];
298  fun(tn+h*c[2],X3,params,f3);
299  // Stage 4
300  Vadd2(G4,1,gn,A[3][2]*h,f3,nx);
301  NewtonSolve(gfun,gJac,nx,G4,X3,AbsTol,RelTol,params,X4,
         ...output);
302  Conv[2] = output[0], Div[2] = output[1], alpharat[2] =
         ...output[2];
303  fun(tn+h*c[3],X4,params,f4);
304
305  Vadd5(gnp1,1,gn,b[0]*h,fn,b[1]*h,f2,b[2]*h,f3,b[3]*h,f4,nx)
         ...;
306  NewtonSolve(gfun,gJac,nx,gnp1,X4,AbsTol,RelTol,params,xnp1,
         ...output);
307  Conv[3] = output[0], Div[3] = output[1], alpharat[3] =
         ...output[2];
308
309  // Return Convergence booleans and alpha ratio
310  output[0] = min(Conv,4), output[1] = max(Div,4), output[2]
         ...= min(alpharat,4);
311
312  free(G2);   free(G3);  free(G4);
313  free(X2);   free(X3);  free(X4);
314  free(f2);   free(f3);  free(f4);
315  free(Conv); free(Div); free(alpharat);
316  return output;
317 }
318
319 int RKF45Mod(
320  ODEModel_t *fun,
321  ODEModel_t *gfun,
322  ODEModel_t *gJac,
323  int nx, int nt,
324  double *tspan,
325  double *x0,
326  double *AbsTol, double RelTol,
327  void *params,
328  double *t,
329  double *x){
330
331  int i,j, firststep, fixedstepsize = nt > 2, s = 6;;
332  double h = 0.001, halpha, hmin = 0.000001, hmax = 10,
         ...epsilon = 0.8;
```

```
333 double p = 4, phat = p+1, kp = 0.4/phat, kI = 0.3/phat, E,
        ...Eold;
334 double *fn      = malloc(nx*sizeof(double));
335 double *xfull   = malloc(nx*sizeof(double));
336 double *e       = malloc(nx*sizeof(double)), *absx  =
        ...malloc(nx*sizeof(double)), *Tol = malloc(nx*sizeof(
        ...double));
337 double *gnp1    = malloc(nx*sizeof(double)), *gfull =
        ...malloc(nx*sizeof(double));
338 double Convergence, Con, Divergence, Div, alpharatio, *
        ...stepoutput;
339 double A[6][6], b[] =
        ...{25/216.0,0.0,1408/2565.0,2197/4104.0,-1/5.0,0.0},
        ...bhat[] =
        ...{16/135.0,0.0,6656/12825.0,28561/56430.0,-9/50.0,2/55.0};
        ...
340 double c[] = {0.0,1/4.0,3/8.0,12/13.0,1.0,1/2.0}, d[6];
341 A[1][0] =  1/4.0;
342 A[2][0] =      3/32.0, A[2][1] =       9/32.0;
343 A[3][0] = 1932/2197.0, A[3][1] = -7200/2197.0, A[3][2] =
        ...7296/2197.0;
344 A[4][0] =   439/216.0, A[4][1] =   -8.0, A[4][2] =
        ...3680/513.0, A[4][3] = -845/4104.0;
345 A[5][0] =     -8/27.0, A[5][1] =          2.0, A[5][2] =
        ...-3544/2565.0, A[5][3] = 1859/4104.0, A[5][4] =
        ...-11/40.0;
346 // Calculate d = b-bhat
347 Vadd2(d,1.0,b,-1.0,bhat,s);
348 // Initial time and initial conditions
349 t[0] = tspan[0];
350 Vadd1((x+0*nx),1.0,x0,nx);
351 gfun(t[0],(x+0*nx),params,gnp1);
352 // If using fixed step size
353 if(fixedstepsize){
354  h = (tspan[1] - tspan[0])/nt;
355  // For every step
356  for(i = 0; i < nt; i++){
357   t[i+1] = t[i] + h;
358   fun(t[i],(x+i*nx),params,fn);
359   stepoutput = RKF45ModStep(fun,gfun,gJac,nx,t[i],(x+i*nx),
          ...fn,gnp1,h,AbsTol,RelTol,params,(x+(i+1)*nx),e,gnp1
          ...,A,bhat,c,d);
360   free(stepoutput);
361  }
362 }else{
363 // If using adaptive step size
364  firststep = 1;
```

```
365   i = 0;
366   // While end time has not been reached
367   while(t[i] < tspan[1]){
368    // Make sure end time is not passed
369    if(h > tspan[1] - t[i]){
370     h = tspan[1] - t[i];
371    }
372    // Full step
373    fun(t[i],(x+i*nx),params,fn);
374    stepoutput = RKF45ModStep(fun,gfun,gJac,nx,t[i],(x+i*nx),
           ...fn,gnp1,h,AbsTol,RelTol,params,xfull,e,gfull,A,
           ...bhat,c,d);
375    Convergence = stepoutput[0], Divergence = stepoutput[1],
           ...alpharatio = stepoutput[2]; free(stepoutput);
376    // Error estimate
377    Vabs(e,e,nx);
378    Vabs(absx,xfull,nx);
379    Vadd2(Tol,1.0,AbsTol,RelTol,absx,nx);
380    Vdiv(e,e,Tol,nx);
381    E = max(e,nx);
382    // Avoid division by zero in PI step size
383    if(E < pow(10.0,-10.0)){ E = pow(10.0,-10.0); }
384    if(Convergence){
385     // Fail or accept the step
386     if(E <= 1){
387      // Update step
388      t[i+1] = t[i] + h;
389      Vadd1((x+(i+1)*nx),1.0,xfull,nx);
390      Vadd1(gnp1,1.0,gfull,nx);
391      if(firststep){
392       // New asymptotic step size
393       h = h*pow(epsilon/E,1.0/phat);
394       // Make sure step size is not too small or too large
395       if(h > hmax){ h = hmax; }else if(h < hmin){ h = hmin;
              ...}
396       firststep = 0;
397      }else{
398       // New PI step size
399       h = h*pow(epsilon/E,kI)*pow(Eold/E,kp);
400       // Make sure step size is not too small or too large
401       if(h > hmax){ h = hmax; }else if(h < hmin){ h = hmin;
              ...}
402      }
403      // Save error for use in the PI step size controller
404      Eold = E;
405      i++;
406
```

```
407     }else{
408      // New asymptotic step size
409      h = h*pow(epsilon/E,1.0/phat);
410      // Make sure step size is not too small or too large
411      if(h > hmax){ h = hmax; }else if(h < hmin){ h = hmin; }
412     }
413
414     if(alpharatio < 1){ h = h*alpharatio; }
415    }else if(Divergence){
416     halpha = h*alpharatio;
417     if(halpha > 0.5*h){ h = halpha; }else{ h = 0.5*h; }
418    }else{
419     if(alpharatio < 1){
420      halpha = h*alpharatio;
421      if(halpha > 0.5*h){ h = halpha; }else{ h = 0.5*h; }
422     }else{
423      h = 0.5*h;
424     }
425    }
426   }
427  }
428  free(fn);   free(xfull); free(e); free(absx); free(Tol);
429  free(gnp1); free(gfull);
430  // Return number of steps
431  return i;
432 }
433
434 double *RKF45ModStep(
435  ODEModel_t *fun,
436  ODEModel_t *gfun,
437  ODEModel_t *gJac,
438  int nx,
439  double tn,
440  double *xn,
441  double *fn, double *gn,
442  double h,
443  double *AbsTol, double RelTol,
444  void *params,
445  double *xnp1, double *e, double *gnp1,
446  double A[][6], double bhat[], double c[], double d[]){
447
448  // Next step
449  double *G2 = malloc(nx*sizeof(double)), *G3 = malloc(nx*
         ...sizeof(double)), *G4 = malloc(nx*sizeof(double));
450  double *G5 = malloc(nx*sizeof(double)), *G6 = malloc(nx*
         ...sizeof(double));
```

```
451 double *X2 = malloc(nx*sizeof(double)), *X3 = malloc(nx*
        ...sizeof(double)), *X4 = malloc(nx*sizeof(double));
452 double *X5 = malloc(nx*sizeof(double)), *X6 = malloc(nx*
        ...sizeof(double));
453 double *f2 = malloc(nx*sizeof(double)), *f3 = malloc(nx*
        ...sizeof(double)), *f4 = malloc(nx*sizeof(double));
454 double *f5 = malloc(nx*sizeof(double)), *f6 = malloc(nx*
        ...sizeof(double));
455 double *gnp1lo = malloc(nx*sizeof(double)), *xnp1lo =
        ...malloc(nx*sizeof(double)), *output = malloc(3*sizeof
        ...(double));
456 double *Conv = malloc(6*sizeof(double)), *Div = malloc(6*
        ...sizeof(double)), *alpharat = malloc(6*sizeof(double)
        ...);
457 // Stage 2
458 Vadd2(G2,1,gn,A[1][0]*h,fn,nx);
459 NewtonSolve(gfun,gJac,nx,G2,xn,AbsTol,RelTol,params,X2,
        ...output);
460 Conv[0] = output[0], Div[0] = output[1], alpharat[0] =
        ...output[2];
461 fun(tn+h*c[1],X2,params,f2);
462 // Stage 3
463 Vadd3(G3,1,gn,A[2][0]*h,fn,A[2][1]*h,f2,nx);
464 NewtonSolve(gfun,gJac,nx,G3,X2,AbsTol,RelTol,params,X3,
        ...output);
465 Conv[1] = output[0], Div[1] = output[1], alpharat[1] =
        ...output[2];
466 fun(tn+h*c[2],X3,params,f3);
467 // Stage 4
468 Vadd4(G4,1,gn,A[3][0]*h,fn,A[3][1]*h,f2,A[3][2]*h,f3,nx);
469 NewtonSolve(gfun,gJac,nx,G4,X3,AbsTol,RelTol,params,X4,
        ...output);
470 Conv[2] = output[0], Div[2] = output[1], alpharat[2] =
        ...output[2];
471 fun(tn+h*c[3],X4,params,f4);
472 // Stage 5
473 Vadd5(G5,1,gn,A[4][0]*h,fn,A[4][1]*h,f2,A[4][2]*h,f3,A
        ...[4][3]*h,f4,nx);
474 NewtonSolve(gfun,gJac,nx,G5,X4,AbsTol,RelTol,params,X5,
        ...output);
475 Conv[3] = output[0], Div[3] = output[1], alpharat[3] =
        ...output[2];
476 fun(tn+h*c[4],X5,params,f5);
477 // Stage 6
478 Vadd6(G6,1,gn,A[5][0]*h,fn,A[5][1]*h,f2,A[5][2]*h,f3,A
        ...[5][3]*h,f4,A[5][4]*h,f5,nx);
```

```
479  NewtonSolve(gfun,gJac,nx,G6,X5,AbsTol,RelTol,params,X6,
         ...output);
480  Conv[4] = output[0], Div[4] = output[1], alpharat[4] =
         ...output[2];
481  fun(tn+h*c[5],X6,params,f6);
482
483  Vadd6(gnp1,1,gn,bhat[0]*h,fn,bhat[2]*h,f3,bhat[3]*h,f4,bhat
         ...[4]*h,f5,bhat[5]*h,f6,nx);
484  NewtonSolve(gfun,gJac,nx,gnp1,X6,AbsTol,RelTol,params,xnp1,
         ...output);
485  Conv[5] = output[0], Div[5] = output[1], alpharat[5] =
         ...output[2];
486
487  // Embedded error estimate
488  Vadd5(e,d[0]*h,fn,d[2]*h,f3,d[3]*h,f4,d[4]*h,f5,d[5]*h,f6,
         ...nx);
489
490  // Return Convergence booleans and alpha ratio
491  output[0] = min(Conv,6), output[1] = max(Div,6), output[2]
         ...= min(alpharat,6);
492
493  free(G2); free(G3); free(G4); free(G5); free(G6);
494  free(X2); free(X3); free(X4); free(X5); free(X6);
495  free(f2); free(f3); free(f4); free(f5); free(f6);
496  free(gnp1lo); free(xnp1lo); free(Conv); free(Div); free(
         ...alpharat);
497  return output;
498 }
499
500 int DOPRI54Mod(
501  ODEModel_t *fun,
502  ODEModel_t *gfun,
503  ODEModel_t *gJac,
504  int nx, int nt,
505  double *tspan,
506  double *x0,
507  double *AbsTol, double RelTol,
508  void *params,
509  double *t,
510  double *x){
511
512  int i,j, firststep, fixedstepsize = nt > 2, s = 7;;
513  double h = 0.001, halpha, hmin = 0.000001, hmax = 10,
         ...epsilon = 0.8;
514  double p = 4, phat = p+1, kp = 0.4/phat, kI = 0.3/phat, E,
         ...Eold;
```

```
515 double *fn      = malloc(nx*sizeof(double)), *fnp1  =
        ...malloc(nx*sizeof(double));
516 double *xfull   = malloc(nx*sizeof(double));
517 double *e       = malloc(nx*sizeof(double)), *absx  =
        ...malloc(nx*sizeof(double)), *Tol = malloc(nx*sizeof(
        ...double));
518 double *gnp1    = malloc(nx*sizeof(double)), *gfull =
        ...malloc(nx*sizeof(double));
519 double Convergence, Con, Divergence, Div, alpharatio, *
        ...stepoutput;
520 double A[s][s], b[] =
        ...{5179/57600.0,0.0,7571/16695.0,393/640.0,-92097/339200.0,187/2100.0
        ...
521 double bhat[] =
        ...{35/384.0,0.0,500/1113.0,125/192.0,-2187/6784.0,11/84.0,0.0},
        ... c[] = {0.0,1/5.0,3/10.0,4/5.0,8/9.0,1.0,1.0};
522 double d[s];
523 A[1][0] =         1/5.0;
524 A[2][0] =       3/40.0, A[2][1] =        9/40.0;
525 A[3][0] =      44/45.0, A[3][1] =      -56/15.0, A[3][2] =
        ...       32/9.0;
526 A[4][0] = 19372/6561.0, A[4][1] = -25360/2187.0, A[4][2] =
        ...64448/6561.0, A[4][3] =   -212/729.0;
527 A[5][0] =  9017/3168.0, A[5][1] =     -355/33.0, A[5][2] =
        ...46732/5247.0, A[5][3] =     49/176.0, A[5][4] =
        ...-5103/18656.0;
528 A[6][0] =     35/384.0,                             A[6][2] =
        ...  500/1113.0, A[6][3] =    125/192.0, A[6][4] =
        ...-2187/6784.0, A[6][5] = 11/84.0;
529 // Calculate d = b-bhat
530 Vadd2(d,1.0,b,-1.0,bhat,s);
531 // Initial time and initial conditions
532 t[0] = tspan[0];
533 Vadd1((x+0*nx),1.0,x0,nx);
534 fun(t[0],(x+0*nx),params,fn);
535 gfun(t[0],(x+0*nx),params,gnp1);
536 // If using fixed step size
537 if(fixedstepsize){
538  h = (tspan[1] - tspan[0])/nt;
539  // For every step
540  for(i = 0; i < nt; i++){
541   t[i+1] = t[i] + h;
542   stepoutput = DOPRI54ModStep(fun,gfun,gJac,nx,t[i],(x+i*nx
          ...),fn,gnp1,h,AbsTol,RelTol,params,(x+(i+1)*nx),e,fn
          ...,gnp1,A,c,d);
543   free(stepoutput);
544  }
```

```
545  }else{
546  // If using adaptive step size
547   firststep = 1;
548   i = 0;
549   // While end time has not been reached
550   while(t[i] < tspan[1]){
551    // Make sure end time is not passed
552    if(h > tspan[1] - t[i]){
553     h = tspan[1] - t[i];
554    }
555    // Full step
556    stepoutput = DOPRI54ModStep(fun,gfun,gJac,nx,t[i],(x+i*nx
           ...),fn,gnp1,h,AbsTol,RelTol,params,xfull,e,fnp1,
           ...gfull,A,c,d);
557    Convergence = stepoutput[0], Divergence = stepoutput[1],
           ...alpharatio = stepoutput[2]; free(stepoutput);
558    // Error estimate
559    Vabs(e,e,nx);
560    Vabs(absx,xfull,nx);
561    Vadd2(Tol,1.0,AbsTol,RelTol,absx,nx);
562    Vdiv(e,e,Tol,nx);
563    E = max(e,nx);
564    // Avoid division by zero in PI step size
565    if(E < pow(10.0,-10.0)){ E = pow(10.0,-10.0); }
566    if(Convergence){
567     // Fail or accept the step
568     if(E <= 1){
569      // Update step
570      t[i+1] = t[i] + h;
571      Vadd1((x+(i+1)*nx),1.0,xfull,nx);
572      Vadd1(fn,1.0,fnp1,nx);
573      Vadd1(gnp1,1.0,gfull,nx);
574      if(firststep){
575       // New asymptotic step size
576       h = h*pow(epsilon/E,1.0/phat);
577       // Make sure step size is not too small or too large
578       if(h > hmax){ h = hmax; }else if(h < hmin){ h = hmin;
              ...}
579       firststep = 0;
580      }else{
581       // New PI step size
582       h = h*pow(epsilon/E,kI)*pow(Eold/E,kp);
583       // Make sure step size is not too small or too large
584       if(h > hmax){ h = hmax; }else if(h < hmin){ h = hmin;
              ...}
585      }
586      // Save error for use in the PI step size controller
```

```
587       Eold = E;
588       i++;
589
590      }else{
591       // New asymptotic step size
592       h = h*pow(epsilon/E,1.0/phat);
593       // Make sure step size is not too small or too large
594       if(h > hmax){ h = hmax; }else if(h < hmin){ h = hmin; }
595      }
596
597      if(alpharatio < 1){ h = h*alpharatio; }
598     }else if(Divergence){
599      halpha = h*alpharatio;
600      if(halpha > 0.5*h){ h = halpha; }else{ h = 0.5*h; }
601     }else{
602      if(alpharatio < 1){
603       halpha = h*alpharatio;
604       if(halpha > 0.5*h){ h = halpha; }else{ h = 0.5*h; }
605      }else{
606       h = 0.5*h;
607      }
608     }
609    }
610   }
611   free(fn);   free(fnp1);  free(xfull); free(e);  free(absx);
          ... free(Tol);
612   free(gnp1); free(gfull);
613   // Return number of steps
614   return i;
615  }
616
617  double *DOPRI54ModStep(
618   ODEModel_t *fun,
619   ODEModel_t *gfun,
620   ODEModel_t *gJac,
621   int nx,
622   double tn,
623   double *xn,
624   double *fn, double *gn,
625   double h,
626   double *AbsTol, double RelTol,
627   void *params,
628   double *xnp1, double *e, double* fnp1, double *gnp1,
629   double A[][7], double c[], double d[]){
630   // Next step
631   double *G2 = malloc(nx*sizeof(double)), *G3 = malloc(nx*
          ...sizeof(double)), *G4 = malloc(nx*sizeof(double));
```

```
632 double *G5 = malloc(nx*sizeof(double)), *G6 = malloc(nx*
        ...sizeof(double));
633 double *X2 = malloc(nx*sizeof(double)), *X3 = malloc(nx*
        ...sizeof(double)), *X4 = malloc(nx*sizeof(double));
634 double *X5 = malloc(nx*sizeof(double)), *X6 = malloc(nx*
        ...sizeof(double));
635 double *f2 = malloc(nx*sizeof(double)), *f3 = malloc(nx*
        ...sizeof(double)), *f4 = malloc(nx*sizeof(double));
636 double *f5 = malloc(nx*sizeof(double)), *f6 = malloc(nx*
        ...sizeof(double));
637 double *gnp1lo = malloc(nx*sizeof(double)), *xnp1lo =
        ...malloc(nx*sizeof(double)), *output = malloc(3*sizeof
        ...(double));
638 double *Conv = malloc(6*sizeof(double)), *Div = malloc(6*
        ...sizeof(double)), *alpharat = malloc(6*sizeof(double)
        ...);
639 // Stage 2
640 Vadd2(G2,1,gn,A[1][0]*h,fn,nx);
641 NewtonSolve(gfun,gJac,nx,G2,xn,AbsTol,RelTol,params,X2,
        ...output);
642 Conv[0] = output[0], Div[0] = output[1], alpharat[0] =
        ...output[2];
643 fun(tn+h*c[1],X2,params,f2);
644 // Stage 3
645 Vadd3(G3,1,gn,A[2][0]*h,fn,A[2][1]*h,f2,nx);
646 NewtonSolve(gfun,gJac,nx,G3,X2,AbsTol,RelTol,params,X3,
        ...output);
647 Conv[1] = output[0], Div[1] = output[1], alpharat[1] =
        ...output[2];
648 fun(tn+h*c[2],X3,params,f3);
649 // Stage 4
650 Vadd4(G4,1,gn,A[3][0]*h,fn,A[3][1]*h,f2,A[3][2]*h,f3,nx);
651 NewtonSolve(gfun,gJac,nx,G4,X3,AbsTol,RelTol,params,X4,
        ...output);
652 Conv[2] = output[0], Div[2] = output[1], alpharat[2] =
        ...output[2];
653 fun(tn+h*c[3],X4,params,f4);
654 // Stage 5
655 Vadd5(G5,1,gn,A[4][0]*h,fn,A[4][1]*h,f2,A[4][2]*h,f3,A
        ...[4][3]*h,f4,nx);
656 NewtonSolve(gfun,gJac,nx,G5,X4,AbsTol,RelTol,params,X5,
        ...output);
657 Conv[3] = output[0], Div[3] = output[1], alpharat[3] =
        ...output[2];
658 fun(tn+h*c[4],X5,params,f5);
659 // Stage 6
```

```
660  Vadd6(G6,1,gn,A[5][0]*h,fn,A[5][1]*h,f2,A[5][2]*h,f3,A
         ...[5][3]*h,f4,A[5][4]*h,f5,nx);
661  NewtonSolve(gfun,gJac,nx,G6,X5,AbsTol,RelTol,params,X6,
         ...output);
662  Conv[4] = output[0], Div[4] = output[1], alpharat[4] =
         ...output[2];
663  fun(tn+h*c[5],X6,params,f6);
664  // Stage 7 is also the next step
665  Vadd6(gnp1,1,gn,A[6][0]*h,fn,A[6][2]*h,f3,A[6][3]*h,f4,A
         ...[6][4]*h,f5,A[6][5]*h,f6,nx);
666  NewtonSolve(gfun,gJac,nx,gnp1,X6,AbsTol,RelTol,params,xnp1,
         ...output);
667  Conv[5] = output[0], Div[5] = output[1], alpharat[5] =
         ...output[2];
668  fun(tn+h*c[6],xnp1,params,fnp1);
669
670  // Embedded error estimation
671  Vadd6(e,d[0]*h,fn,d[2]*h,f3,d[3]*h,f4,d[4]*h,f5,d[5]*h,f6,d
         ...[6]*h,fnp1,nx);
672
673  // Return Convergence booleans and alpha ratio
674  output[0] = min(Conv,6), output[1] = max(Div,6), output[2]
         ...= min(alpharat,6);
675
676  free(G2); free(G3); free(G4); free(G5); free(G6);
677  free(X2); free(X3); free(X4); free(X5); free(X6);
678  free(f2); free(f3); free(f4); free(f5); free(f6);
679  free(Conv); free(Div); free(alpharat);
680  return output;
681 }
682
683 int ESDIRK23Mod(
684  ODEModel_t *fun,
685  ODEModel_t *Jac,
686  ODEModel_t *gfun,
687  ODEModel_t *gJac,
688  int nx, int nt,
689  double *tspan,
690  double *x0,
691  double *AbsTol, double RelTol,
692  void *params,
693  double *t,
694  double *x){
695
696  int i, firststep, fixedstepsize = nt > 2, s = 3;
697  double h = 0.001, halpha, hold, hmax = 10, hmin = 0.000001,
         ... epsilon = 0.8, p = 2, phat = p+1, kp = 1/phat, kI =
```

```
      ... 1/phat, E, Eold;
698 double *fn    = malloc(nx*sizeof(double)), *fnp1  = malloc(
      ...nx*sizeof(double));
699 double *xfull = malloc(nx*sizeof(double));
700 double *e     = malloc(nx*sizeof(double)), *absx  = malloc(
      ...nx*sizeof(double)), *Tol = malloc(nx*sizeof(double))
      ...;
701 double *gfull = malloc(nx*sizeof(double)), *gnp1  = malloc(
      ...nx*sizeof(double));
702 double gamma = 1-1.0/sqrt(2.0), a31 = (1.0-gamma)/2, b[] =
      ...{a31,a31,gamma};
703 double bhat[] = {(6*gamma-1)/(12*gamma),1/(12*gamma*(1-2*
      ...gamma)),(1-3*gamma)/(3*(1-2*gamma))}, c[] = {0,2*
      ...gamma,1}, d[s];
704 double Convergence, Divergence, alpharatio, *stepoutput;
705 // Calculate d = b-bhat
706 Vadd2(d,1.0,b,-1.0,bhat,s);
707 // Initial time and initial conditions
708 t[0] = tspan[0];
709 Vadd1((x+0*nx),1.0,x0,nx);
710 fun(t[0],(x+0*nx),params,fn);
711 gfun(t[0],(x+0*nx),params,gnp1);
712 // If using fixed step size
713 if(fixedstepsize){
714  h = (tspan[1] - tspan[0])/nt;
715  // For every step
716  for(i = 0; i < nt; i++){
717   t[i+1] = t[i] + h;
718   stepoutput = ESDIRK23ModStep(fun,Jac,gfun,gJac,nx,t[i],(x
        ...+i*nx),fn,gnp1,h,AbsTol,RelTol,params,(x+(i+1)*nx)
        ...,e,fn,gnp1,b,c,d,gamma);
719  }
720 }else{
721 // If using adaptive step size
722  firststep = 1;
723  i = 0;
724  // While end time has not been reached
725  while(t[i] < tspan[1]){
726   // Make sure end time is not passed
727   if(h > tspan[1] - t[i]){
728    h = tspan[1] - t[i];
729   }
730   // Full step
731   stepoutput = ESDIRK23ModStep(fun,Jac,gfun,gJac,nx,t[i],(x
        ...+i*nx),fn,gnp1,h,AbsTol,RelTol,params,xfull,e,fnp1
        ...,gfull,b,c,d,gamma);
```

```
732     Convergence = stepoutput[0], Divergence = stepoutput[1],
            ...alpharatio = stepoutput[2]; free(stepoutput);
733     // Error estimate
734     Vabs(e,e,nx);
735     Vabs(absx,xfull,nx);
736     Vadd2(Tol,1.0,AbsTol,RelTol,absx,nx);
737     Vdiv(e,e,Tol,nx);
738     E = max(e,nx);
739     // Avoid division by zero in PI step size
740     if(E < pow(10,-10)){ E = pow(10,-10); }
741     if(Convergence){
742      // Fail or accept step
743      if(E <= 1){
744       // Update step
745       t[i+1] = t[i] + h;
746       Vadd1((x+(i+1)*nx),1.0,xfull,nx);
747       Vadd1(fn,1.0,fnp1,nx);
748       Vadd1(gnp1,1.0,gfull,nx);
749       if(firststep){
750        // New asymptotic step size
751        h = h*pow(epsilon/E,1.0/phat);
752        // Make sure step size is not too small or too large
753        //if(h > hmax){ h = hmax; }else if(h < hmin){ h = hmin
               ...; }
754        firststep = 0;
755       }else{
756        // New PI step size
757        h = h*(h/hold)*pow(epsilon/E,kI)*pow(Eold/E,kp);
758        // Make sure step size is not too small or too large
759        if(h > hmax){ h = hmax; }else if(h < hmin){ h = hmin;
               ...}
760       }
761       // Save error and step size for use in PI step size
              ...controller
762       Eold = E;
763       hold = h;
764       i++;
765      }else{
766       // New asymptotic step size
767       h = h*pow(epsilon/E,1.0/phat);
768       // Make sure step size is not too small or too large
769       if(h > hmax){ h = hmax; }else if(h < hmin){ h = hmin; }
770      }
771
772      if(alpharatio < 1){
773       h = h*alpharatio;
774      }
```

```
775     }else if(Divergence){
776      halpha = h*alpharatio;
777      if(halpha > 0.5*h){ h = halpha; }else{ h = 0.5*h; }
778     }else{
779      if(alpharatio < 1){
780       halpha = h*alpharatio;
781       if(halpha > 0.5*h){ h = halpha; }else{ h = 0.5*h; }
782      }else{
783       h = 0.5*h;
784      }
785     }
786    }
787   }
788   free(fn); free(fnp1); free(xfull); free(e); free(absx);
          ...free(Tol);
789   free(gfull); free(gnp1);
790   // Return number of steps
791   return i;
792  }
793
794  double *ESDIRK23ModStep(
795   ODEModel_t *fun,
796   ODEModel_t *Jac,
797   ODEModel_t *gfun,
798   ODEModel_t *gJac,
799   int nx,
800   double tn,
801   double *xn,
802   double *fn, double *gn,
803   double h,
804   double *AbsTol, double RelTol,
805   void *params,
806   double *xnp1, double *e, double *fnp1, double *gnp1,
807   double b[], double c[], double d[], double gamma){
808
809   int i, Convergence, SlowConvergence = 0, Divergence = 0,
          ...iter, itermax = 10;
810   double epsilon = 0.8, tau = 0.1*epsilon, alpha = 0.0,
          ...alpharef = 0.4;
811
812   double *X2   = malloc(nx*sizeof(double)), *X3   = malloc(nx
          ...*sizeof(double));
813   double *f2   = malloc(nx*sizeof(double)), *f3   = malloc(nx
          ...*sizeof(double));
814   double *phi2 = malloc(nx*sizeof(double)), *phi3 = malloc(nx
          ...*sizeof(double)),   *R  = malloc(nx*sizeof(double));
```

```
815 double *absG = malloc(nx*sizeof(double)), *Tol  = malloc(nx
        ...*sizeof(double)),   *dX = malloc(nx*sizeof(double));
816 double *J    = malloc(nx*nx*sizeof(double)), *dRdx = malloc
        ...(nx*nx*sizeof(double)), *dgdx = malloc(nx*nx*sizeof(
        ...double));
817 double *G2   = malloc(nx*sizeof(double)), *G3 = malloc(nx*
        ...sizeof(double));
818 double T, rNewton, rNewtonOld, a21 = gamma, *output =
        ...malloc(sizeof(double)*3);
819 // Parameters used for LAPACK functions dgetrf and dgetrs
820 const enum CBLAS_ORDER Order = CblasRowMajor;
821 const enum CBLAS_TRANSPOSE Trans = CblasNoTrans;
822 int N = nx, M = N, NRHS = 1, LDA = nx, LDB = N, *IPIV =
        ...malloc(nx*sizeof(int));
823 // Jacobian Update
824 Jac(tn,xn,params,J);
825 gJac(tn,xn,params,dgdx);
826 Madd2(nx,dRdx,1.0,dgdx,-h*gamma,J);
827 // LU Factorization of dRdX
828 clapack_dgetrf(Order,M,N,dRdx,LDA,IPIV);
829 // Stage 2 of the ESDIRK23 method
830 Vadd2(phi2,1.0,gn,h*a21,fn,nx);
831 // Initial guess for the state by Euler step
832 T = tn + c[1]*h;
833 Vadd1(X2,1.0,xn,nx);
834 // Newton iterations
835 fun(T,X2,params,f2);
836 gfun(T,X2,params,G2);
837 // R = X2 - h*gamma*f2 - phi2
838 Vadd3(R,1.0,G2,-h*gamma,f2,-1.0,phi2,nx);
839 // rNewton = || |R|/(AbsTol + |X2|*RelTol) ||_inf
840 Vabs(e,R,nx);
841 Vabs(absG,G2,nx);
842 Vadd2(Tol,1.0,AbsTol,RelTol,absG,nx);
843 Vdiv(e,e,Tol,nx);
844 rNewton = max(e,nx);
845 rNewtonOld = rNewton;
846 iter = 0;
847 Convergence = 0;
848 while(!Convergence && !SlowConvergence && !Divergence){
849  // Solve (I - h*gamma*J) dX = R for dX (notice the lacking
         ... minus). dX is stored in R
850  clapack_dgetrs(Order,Trans,N,NRHS,dRdx,LDA,IPIV,R,LDB);
851  //Backslash(nx,dRdx,dX,R);
852  // Update X2 by adding -dX (notice the minus)
853  Vadd2(X2,1.0,X2,-1.0,R,nx);
854  fun(T,X2,params,f2);
```

```
855   gfun(T,X2,params,G2);
856   // R = X2 - h*gamma*f2 - phi2
857   Vadd3(R,1.0,G2,-h*gamma,f2,-1.0,phi2,nx);
858   // rNewton = || |R|/(AbsTol + |X2|*RelTol) ||_inf
859   Vabs(e,R,nx);
860   Vabs(absG,G2,nx);
861   Vadd2(Tol,1.0,AbsTol,RelTol,absG,nx);
862   Vdiv(e,e,Tol,nx);
863   rNewton = max(e,nx);
864   if(alpha < rNewton/rNewtonOld){ alpha = rNewton/rNewtonOld
          ...; }
865   Convergence = rNewton < tau;
866   SlowConvergence = iter > itermax;
867   Divergence  = alpha > 1;
868   rNewtonOld = rNewton;
869   iter++;
870 }
871 // Stage 3 of the ESDIRK23 method
872 Vadd3(phi3,1.0,gn,b[0]*h,fn,b[1]*h,f2,nx);
873 // Initial guess for the state
874 T = tn + h;
875 Vadd1(X3,1.0,xn,nx);
876 // Newton iterations
877 fun(T,X3,params,f3);
878 gfun(T,X3,params,G3);
879 // R = X3 - h*gamma*f3 - phi3
880 Vadd3(R,1.0,G3,-h*gamma,f3,-1.0,phi3,nx);
881 // rNewton = || |R|/(AbsTol + |X3|*RelTol) ||_inf
882 Vabs(e,R,nx);
883 Vabs(absG,G3,nx);
884 Vadd2(Tol,1.0,AbsTol,RelTol,absG,nx);
885 Vdiv(e,e,Tol,nx);
886 rNewton = max(e,nx);
887 rNewtonOld = rNewton;
888 iter = 0;
889 Convergence = 0;
890 while(!Convergence && !SlowConvergence && !Divergence){
891   // Solve (I - h*gamma*J) dX = R for dX (notice the lacking
          ... minus). dX is stored in R
892   clapack_dgetrs(Order,Trans,N,NRHS,dRdx,LDA,IPIV,R,LDB);
893   //Backslash(nx,dRdx,dX,R);
894   // Update X3 by adding -dX (notice the minus)
895   Vadd2(X3,1.0,X3,-1.0,R,nx);
896   fun(T,X3,params,f3);
897   gfun(T,X3,params,G3);
898   // R = X3 - h*gamma*f3 - phi3
899   Vadd3(R,1.0,G3,-h*gamma,f3,-1.0,phi3,nx);
```

```
900   // rNewton = || |R|/(AbsTol + |X3|*RelTol) ||_inf
901   Vabs(e,R,nx);
902   Vabs(absG,G3,nx);
903   Vadd2(Tol,1.0,AbsTol,RelTol,absG,nx);
904   Vdiv(e,e,Tol,nx);
905   rNewton = max(e,nx);
906   if(alpha < rNewton/rNewtonOld){ alpha = rNewton/rNewtonOld
          ...; }
907   Convergence = rNewton < tau;
908   SlowConvergence = iter > itermax;
909   Divergence  = alpha > 1;
910   rNewtonOld = rNewton;
911   iter++;
912  }
913  // Update step
914  Vadd1(xnp1,1.0,X3,nx);
915  // The function evaluation is saved and reused
916  Vadd1(fnp1,1.0,f3,nx);
917  Vadd1(gnp1,1.0,G3,nx);
918  // Embedded error estimate
919  Vadd3(e,d[0]*h,fn,d[1]*h,f2,d[2]*h,f3,nx);
920  // Return Convergence booleans and alpha ratio
921  output[0] = Convergence, output[1] = Divergence, output[2]
          ...= alpharef/alpha;
922
923  free(X2); free(X3);   free(f2);  free(f3); free(phi2); free
          ...(phi3);
924  free(R);  free(absG); free(Tol); free(dX);
925  free(J);  free(dRdx);
926  free(G2); free(G3);   free(dgdx); free(IPIV);
927
928  return output;
929 }
```

B.3 Functions Used for Timing Simulations

Listing B.3: Wrap function which calls the requested solver in C.

```
1 int Solver(
2  ODEModel_t *fun,
3  ODEModel_t *Jac,
4  ODEModel_t *gfun,
5  ODEModel_t *gJac,
6  int nx, int nt,
7  double *tspan,
```

```
 8  double *x0,
 9  double *AbsTol, double RelTol,
10  void *params,
11  double *t,
12  double *x,
13  int method){
14  int N;
15  if( method == 1){  N = Euler      (fun,nx,nt,tspan,x0,
        ...AbsTol,RelTol,params,t,x); }
16  else if(method == 2){  N = RK4        (fun,nx,nt,tspan,x0,
        ...AbsTol,RelTol,params,t,x); }
17  else if(method == 3){  N = RKF45      (fun,nx,nt,tspan,x0,
        ...AbsTol,RelTol,params,t,x); }
18  else if(method == 4){  N = DOPRI54    (fun,nx,nt,tspan,x0,
        ...AbsTol,RelTol,params,t,x); }
19  else if(method == 5){  N = ESDIRK23   (fun,Jac,nx,nt,tspan,
        ...x0,AbsTol,RelTol,params,t,x); }
20  else if(method == 6){  N = EulerMod   (fun,gfun,gJac,nx,nt,
        ...tspan,x0,AbsTol,RelTol,params,t,x); }
21  else if(method == 7){  N = RK4Mod     (fun,gfun,gJac,nx,nt,
        ...tspan,x0,AbsTol,RelTol,params,t,x); }
22  else if(method == 8){  N = RKF45Mod   (fun,gfun,gJac,nx,nt,
        ...tspan,x0,AbsTol,RelTol,params,t,x); }
23  else if(method == 9){  N = DOPRI54Mod (fun,gfun,gJac,nx,nt,
        ...tspan,x0,AbsTol,RelTol,params,t,x); }
24  else if(method == 10){ N = ESDIRK23Mod(fun,Jac,gfun,gJac,nx
        ...,nt,tspan,x0,AbsTol,RelTol,params,t,x); }
25  return N;
26 }
```

B.4 Sequential Simulations

Listing B.4: Function which does simulations for requested parameters, method, tolerances and step size.

```
1 #include <string.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <math.h>
5 #include <time.h>
6 #include "FedBatchProblem.h"
7 #include "RungeKuttaToolbox.h"
8
9 double FedBatchVaryParameters(int method, double Tol, double
      ... h, int nstart, int nend){
```

```
10  int k,i,j,m,n,p,q,r;
11   // Help variables
12  int nx = 4;
13  // Parameter arrays
14  double *params     = malloc(14*sizeof(double));
15  double *parameters = malloc(14*sizeof(double));
16  // Initial conditions
17  double V0 = 100.0, Vmax = 1200.0, CX0 = 20, CS0 = 0.0893,
        ...P0 = 0.0;
18  // Solver variables (time and approximation vector are at
        ...least 202 big, for adaptive step size)
19  int nt = 2, Nend, size;
20  // Initial conditions and time span
21  double x0[] = {V0,CX0,CS0,P0}, tspan[2], mustar;
22  // Feedback parameters
23  double Ks = 0, Kw = 0;
24  // Tolerances
25  double AbsTol[] = {Tol,Tol,Tol,Tol}, RelTol = Tol;
26  // Time vector, Approximation vector and Production vector
27  double *t = malloc((size+1)*sizeof(double)), *x = malloc((
        ...size+1)*nx*sizeof(double)), prod[10][10][10][10];
28  // Variables used for measuring the total run-time
29  clock_t start,end;
30  double diff;
31  // Function pointers
32  ODEModel_t *pfun  = FedBatchOptimalInletMod;
33  ODEModel_t *pJac  = FedBatchOptimalInletModJac;
34  ODEModel_t *pgfun = gfun;
35  ODEModel_t *pgJac = gJac;
36
37  // Assign function pointers according to method
38  if(method < 6){ pfun  = FedBatchOptimalInlet,   pJac  =
        ...FedBatchOptimalInletJac; }
39  else{   pfun  = FedBatchOptimalInletMod, pJac  =
        ...FedBatchOptimalInletModJac;
40          pgfun = gfun,           pgJac = gJac; }
41
42  // Get parameters
43  FedBatchParameters(CX0,CS0,Ks,Kw,CX0,CS0,params);
44  FedBatchParameters(CX0,CS0,Ks,Kw,CX0,CS0,parameters);
45  // Define time span
46  mustar = params[13];
47  tspan[0] = 0.0, tspan[1] = 1.0/mustar*log(Vmax/V0);
48  // Initialize N according to step size
49  if(h == 0){ nt = 2; }
50  else{ nt = (int)((tspan[1] - tspan[0])/h); }
51  size = 2000+nt;
```

```
52  t = malloc((size+1)*sizeof(double)), x = malloc((size+1)*nx
        ...*sizeof(double));
53
54  // The simulations are done for nend-nstart combinations of
        ... parameters
55  for(k = nstart; k < nend; k++){
56   i = k/1000; j = (k%1000)/100, m = (k%100)/10, n = k%10;
57   // The varied parameters are updated
58   parameters[5] = params[5]*(0.9 + 2/90.0*i);
59   parameters[6] = params[6]*(0.9 + 2/90.0*j);
60   parameters[7] = params[7]*(0.9 + 2/90.0*m);
61   parameters[8] = params[8]*(0.9 + 2/90.0*n);
62
63   // The simulation is done for the current set of
         ...parameters
64   Nend = Solver(pfun,pJac,pgfun,pgJac,nx,nt,tspan,x0,AbsTol,
         ...RelTol,parameters,t,x,method);
65  }
66
67  // Pointers are freed
68  free(t); free(x); free(params); free(parameters);
69
70  return diff;
71 }
```

B.5 Simple Parallel Simulations

Listing B.5: Simple parallel simulations using MPI.

```
1  #include <mpi.h>
2  #include <string.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <math.h>
6  #include <time.h>
7  #include "FedBatchProblem.h"
8  #include "RungeKuttaToolbox.h"
9
10 // Method for measuring runtime
11 double FedBatchVaryParameters(
12  int method,
13  double h,
14  int nstart, int nend);
15
16 // Solver wrap
```

```
17 int Solver(
18  ODEModel_t *fun,
19  ODEModel_t *Jac,
20  ODEModel_t *gfun,
21  ODEModel_t *gJac,
22  int nx, int nt,
23  double *tspan,
24  double *x0,
25  double *AbsTol, double RelTol,
26  void *params,
27  double *t,
28  double *x,
29  int method);
30
31 int main(int argc, char **argv){
32  // CONTROL
33  int method;
34  double h = 0.01;
35  // Variables used for measuring the total run-time
36  double start,end;
37  double diff;
38  // Variables used in MPI
39  int rank, comm_size;
40  int npp, nstart, nend;
41  int jobs = 10000;
42
43  MPI_Init(&argc,&argv);
44
45         MPI_Comm_rank (MPI_COMM_WORLD, &rank);
46         MPI_Comm_size (MPI_COMM_WORLD, &comm_size);
47  for(method = 1; method < 11; method++){
48  // The simulations are done for 10000 combinations of
       ...parameters and the time measuring is commenced
49  MPI_Barrier(MPI_COMM_WORLD);
50  start = MPI_Wtime();
51
52  npp = jobs/comm_size;
53
54  nstart =  rank   *npp;
55  nend   = (rank+1)*npp;
56  if(rank == comm_size-1){ nend = jobs; }
57
58  FedBatchVaryParameters(method,h,nstart,nend);
59
60  // The time measurement is stopped and the run time is
       ...calculated
61  MPI_Barrier(MPI_COMM_WORLD);
```

```
62  end = MPI_Wtime();
63  //diff = ((double)(end-start))/CLOCKS_PER_SEC;
64  diff = end-start;
65  if(rank==0){ printf("Method %2d ran in %.2lf seconds, using
        ... %d processes\n",method,diff,comm_size); }
66  }
67  MPI_Finalize();
68
69  return 0;
70 }
71
72 double FedBatchVaryParameters(int method, double h, int
       ...nstart, int nend){
73  int k,i,j,m,n,p,q,r;
74   // Help variables
75  int nx = 4;
76  // Parameter arrays
77  double *params     = malloc(14*sizeof(double));
78  double *parameters = malloc(14*sizeof(double));
79  // Initial conditions
80  double V0 = 100.0, Vmax = 1200.0, CX0 = 20, CS0 = 0.0893,
       ...P0 = 0.0;
81  // Solver variables (time and approximation vector are at
       ...least 202 big, for adaptive step size)
82  int nt = 2, Nend, size;
83  // MARK Initial conditions and time span
84  double x0[] = {V0,CX0,CS0,P0}, tspan[2], mustar;
85  // Feedback parameters
86  double Ks = 0, Kw = 0;
87  // Tolerances
88  double Tol = 0.001, AbsTol[] = {Tol,Tol,Tol,Tol}, RelTol =
       ...Tol;
89  // Time vector, Approximation vector and Production vector
90  double *t = malloc((size+1)*sizeof(double)), *x = malloc((
       ...size+1)*nx*sizeof(double)), prod[10][10][10][10];
91  // Variables used for measuring the total run-time
92  clock_t start,end;
93  double diff;
94  // Function pointers
95  ODEModel_t *pfun  = FedBatchOptimalInletMod;
96  ODEModel_t *pJac  = FedBatchOptimalInletModJac;
97  ODEModel_t *pgfun = gfun;
98  ODEModel_t *pgJac = gJac;
99
100 // Assign function pointers according to method
101 if(method < 6){ pfun  = FedBatchOptimalInlet,   pJac  =
       ...FedBatchOptimalInletJac; }
```

```
102  else{   pfun  = FedBatchOptimalInletMod, pJac  =
         ...FedBatchOptimalInletModJac;
103          pgfun = gfun,           pgJac = gJac; }
104
105  // Get parameters
106  FedBatchParameters(CX0,CS0,Ks,Kw,CX0,CS0,params);
107  FedBatchParameters(CX0,CS0,Ks,Kw,CX0,CS0,parameters);
108  // Define time span
109  mustar = params[13];
110  tspan[0] = 0.0, tspan[1] = 1.0/mustar*log(Vmax/V0);
111  // Initialize N according to step size
112  if(h == 0){ nt = 2; }
113  else{ nt = (int)((tspan[1] - tspan[0])/h); }
114  size = 2000+nt;
115  t = malloc((size+1)*sizeof(double)), x = malloc((size+1)*nx
         ...*sizeof(double));
116
117  // The simulations are done for nend-nstart combinations of
         ... parameters
118  for(k = nstart; k < nend; k++){
119   i = k/1000; j = (k%1000)/100, m = (k%100)/10, n = k%10;
120   // The varied parameters are updated
121   parameters[5] = params[5]*(0.9 + 2/90.0*i);
122   parameters[6] = params[6]*(0.9 + 2/90.0*j);
123   parameters[7] = params[7]*(0.9 + 2/90.0*m);
124   parameters[8] = params[8]*(0.9 + 2/90.0*n);
125
126   // The simulation is done for the current set of
          ...parameters
127   Nend = Solver(pfun,pJac,pgfun,pgJac,nx,nt,tspan,x0,AbsTol,
          ...RelTol,parameters,t,x,method);
128  }
129
130  // Pointers are freed
131  free(t); free(x); free(params); free(parameters);
132
133  return diff;
134 }
135
136 int Solver(
137  ODEModel_t *fun,
138  ODEModel_t *Jac,
139  ODEModel_t *gfun,
140  ODEModel_t *gJac,
141  int nx, int nt,
142  double *tspan,
143  double *x0,
```

```
144  double *AbsTol, double RelTol,
145  void *params,
146  double *t,
147  double *x,
148  int method){
149  int N;
150  if( method == 1){  N = Euler      (fun,nx,nt,tspan,x0,
         ...AbsTol,RelTol,params,t,x); }
151  else if(method == 2){  N = RK4        (fun,nx,nt,tspan,x0,
         ...AbsTol,RelTol,params,t,x); }
152  else if(method == 3){  N = RKF45      (fun,nx,nt,tspan,x0,
         ...AbsTol,RelTol,params,t,x); }
153  else if(method == 4){  N = DOPRI54    (fun,nx,nt,tspan,x0,
         ...AbsTol,RelTol,params,t,x); }
154  else if(method == 5){  N = ESDIRK23   (fun,Jac,nx,nt,tspan,
         ...x0,AbsTol,RelTol,params,t,x); }
155  else if(method == 6){  N = EulerMod   (fun,gfun,gJac,nx,nt,
         ...tspan,x0,AbsTol,RelTol,params,t,x); }
156  else if(method == 7){  N = RK4Mod     (fun,gfun,gJac,nx,nt,
         ...tspan,x0,AbsTol,RelTol,params,t,x); }
157  else if(method == 8){  N = RKF45Mod   (fun,gfun,gJac,nx,nt,
         ...tspan,x0,AbsTol,RelTol,params,t,x); }
158  else if(method == 9){  N = DOPRI54Mod (fun,gfun,gJac,nx,nt,
         ...tspan,x0,AbsTol,RelTol,params,t,x); }
159  else if(method == 10){ N = ESDIRK23Mod(fun,Jac,gfun,gJac,nx
         ...,nt,tspan,x0,AbsTol,RelTol,params,t,x); }
160  return N;
161 }
```

B.6 Advanced Parallel Simulations

Listing B.6: Advanced parallel simulations using MPI.

```
1  #include <mpi.h>
2  #include <string.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <math.h>
6  #include <time.h>
7  #include "FedBatchProblem.h"
8  #include "RungeKuttaToolbox.h"
9
10 // Method for measuring runtime
11 double FedBatchVaryParameters(
12  int method,
```

```
13  double h,
14  int nstart, int nend);
15
16 // Solver wrap
17 int Solver(
18  ODEModel_t *fun,
19  ODEModel_t *Jac,
20  ODEModel_t *gfun,
21  ODEModel_t *gJac,
22  int nx, int nt,
23  double *tspan,
24  double *x0,
25  double *AbsTol, double RelTol,
26  void *params,
27  double *t,
28  double *x,
29  int method);
30
31 int main(int argc, char **argv){
32  // CONTROL
33  int k, method;
34  double h = 0.01;
35  // Variables used for measuring the total run-time
36  double start,end;
37  double diff;
38  // Variables used in MPI
39  int rank, comm_size;
40  int chunk, nstart, nend;
41  int resolution;
42  int *send = malloc(sizeof(int)*2), *rec = malloc(sizeof(int
        ...)*2);
43  int jobs = 10000;
44  int progress;
45  short tag=0;
46  MPI_Status status;
47
48  MPI_Init(&argc,&argv);
49
50         MPI_Comm_rank (MPI_COMM_WORLD, &rank);
51         MPI_Comm_size (MPI_COMM_WORLD, &comm_size);
52  for(method = 1; method < 11; method++){
53  // The simulations are done for 10000 combinations of
        ...parameters and the time measuring is commenced
54  MPI_Barrier(MPI_COMM_WORLD);
55  start = MPI_Wtime();
56
57  resolution = 40;
```

```
58 //chunk = jobs/(resolution*(comm_size-1))+1;
59 chunk = 10;
60
61 if(rank==0){
62  progress = (comm_size-1)*chunk;
63  k = comm_size;
64  // Keep assigning parameters until all simulations have
        ...been done
65  while(progress < jobs){
66   MPI_Recv(rec,2,MPI_INT,MPI_ANY_SOURCE,tag,MPI_COMM_WORLD
        ...,&status);
67   *(send)   = (k-1)*chunk;
68   *(send+1) = k*chunk;
69   if(*(send+1) > jobs){ *(send+1) = jobs; }
70   progress = k*chunk;
71   k++;
72   MPI_Send(send,2,MPI_INT,status.MPI_SOURCE,tag,
        ...MPI_COMM_WORLD);
73  }
74  // Send stop signal
75  for(k=1;k<comm_size;k++){
76   *send   = -1;
77   MPI_Send(send,2,MPI_INT,k,tag,MPI_COMM_WORLD);
78  }
79 }else{
80  // self-initialization
81  nstart  = (rank-1)*chunk;
82  nend    = rank*chunk;
83  *rec    = 0;
84  // While content of received message is not stop signal
85  while(*rec != -1){
86   // Do simulations for received set of parameters
87   FedBatchVaryParameters(method,h,nstart,nend);
88   *send = 0, *(send+1) = 0;
89   MPI_Send(send,2,MPI_INT,0,tag,MPI_COMM_WORLD);
90   MPI_Recv(rec, 2, MPI_INT, 0, tag, MPI_COMM_WORLD,&status)
        ...;
91   nstart = *rec; nend = *(rec+1);
92  }
93 }
94
95 // The time measurement is stopped and the run time is
      ...calculated
96 MPI_Barrier(MPI_COMM_WORLD);
97 end = MPI_Wtime();
98 diff = end-start;
```

```
99  if(rank==0){ printf("Method %2d ran in %5.2lf seconds,
        ...using %d processes with chunk size: %d\n",method,
        ...diff,comm_size,chunk); }
100 }
101 MPI_Finalize();
102
103 return 0;
104 }
105
106 double FedBatchVaryParameters(int method, double h, int
        ...nstart, int nend){
107 int k,i,j,m,n,p,q,r;
108  // Help variables
109 int nx = 4;
110 // Parameter arrays
111 double *params     = malloc(14*sizeof(double));
112 double *parameters = malloc(14*sizeof(double));
113 // Initial conditions
114 double V0 = 100.0, Vmax = 1200.0, CX0 = 20, CS0 = 0.0893,
        ...P0 = 0.0;
115 // Solver variables (time and approximation vector are at
        ...least 202 big, for adaptive step size)
116 int nt = 2, Nend, size;
117 // MARK Initial conditions and time span
118 double x0[] = {V0,CX0,CS0,P0}, tspan[2], mustar;
119 // Feedback parameters
120 double Ks = 0, Kw = 0;
121 // Tolerances
122 double Tol = 0.001, AbsTol[] = {Tol,Tol,Tol,Tol}, RelTol =
        ...Tol;
123 // Time vector, Approximation vector and Production vector
124 double *t = malloc((size+1)*sizeof(double)), *x = malloc((
        ...size+1)*nx*sizeof(double)), prod[10][10][10][10];
125 // Variables used for measuring the total run-time
126 clock_t start,end;
127 double diff;
128 // Function pointers
129 ODEModel_t *pfun  = FedBatchOptimalInletMod;
130 ODEModel_t *pJac  = FedBatchOptimalInletModJac;
131 ODEModel_t *pgfun = gfun;
132 ODEModel_t *pgJac = gJac;
133
134 // Assign function pointers according to method
135 if(method < 6){ pfun  = FedBatchOptimalInlet,   pJac  =
        ...FedBatchOptimalInletJac; }
136 else{   pfun  = FedBatchOptimalInletMod, pJac  =
        ...FedBatchOptimalInletModJac;
```

```
137           pgfun = gfun,            pgJac = gJac; }
138
139  // Get parameters
140  FedBatchParameters(CX0,CS0,Ks,Kw,CX0,CS0,params);
141  FedBatchParameters(CX0,CS0,Ks,Kw,CX0,CS0,parameters);
142  // Define time span
143  mustar = params[13];
144  tspan[0] = 0.0, tspan[1] = 1.0/mustar*log(Vmax/V0);
145  // Initialize N according to step size
146  if(h == 0){ nt = 2; }
147  else{ nt = (int)((tspan[1] - tspan[0])/h); }
148  size = 2000+nt;
149  t = malloc((size+1)*sizeof(double)), x = malloc((size+1)*nx
         ...*sizeof(double));
150
151  // The simulations are done for nend-nstart combinations of
         ... parameters
152  for(k = nstart; k < nend; k++){
153   i = k/1000; j = (k%1000)/100, m = (k%100)/10, n = k%10;
154   // The varied parameters are updated
155   parameters[5] = params[5]*(0.9 + 2/90.0*i);
156   parameters[6] = params[6]*(0.9 + 2/90.0*j);
157   parameters[7] = params[7]*(0.9 + 2/90.0*m);
158   parameters[8] = params[8]*(0.9 + 2/90.0*n);
159
160   // The simulation is done for the current set of
          ...parameters
161   Nend = Solver(pfun,pJac,pgfun,pgJac,nx,nt,tspan,x0,AbsTol,
          ...RelTol,parameters,t,x,method);
162  }
163
164  // Pointers are freed
165  free(t); free(x); free(params); free(parameters);
166
167  return diff;
168 }
169
170 int Solver(
171  ODEModel_t *fun,
172  ODEModel_t *Jac,
173  ODEModel_t *gfun,
174  ODEModel_t *gJac,
175  int nx, int nt,
176  double *tspan,
177  double *x0,
178  double *AbsTol, double RelTol,
179  void *params,
```

```
180  double *t,
181  double *x,
182  int method){
183  int N;
184  if( method == 1){  N = Euler       (fun,nx,nt,tspan,x0,
         ...AbsTol,RelTol,params,t,x); }
185  else if(method == 2){  N = RK4         (fun,nx,nt,tspan,x0,
         ...AbsTol,RelTol,params,t,x); }
186  else if(method == 3){  N = RKF45       (fun,nx,nt,tspan,x0,
         ...AbsTol,RelTol,params,t,x); }
187  else if(method == 4){  N = DOPRI54     (fun,nx,nt,tspan,x0,
         ...AbsTol,RelTol,params,t,x); }
188  else if(method == 5){  N = ESDIRK23    (fun,Jac,nx,nt,tspan,
         ...x0,AbsTol,RelTol,params,t,x); }
189  else if(method == 6){  N = EulerMod    (fun,gfun,gJac,nx,nt,
         ...tspan,x0,AbsTol,RelTol,params,t,x); }
190  else if(method == 7){  N = RK4Mod      (fun,gfun,gJac,nx,nt,
         ...tspan,x0,AbsTol,RelTol,params,t,x); }
191  else if(method == 8){  N = RKF45Mod    (fun,gfun,gJac,nx,nt,
         ...tspan,x0,AbsTol,RelTol,params,t,x); }
192  else if(method == 9){  N = DOPRI54Mod (fun,gfun,gJac,nx,nt,
         ...tspan,x0,AbsTol,RelTol,params,t,x); }
193  else if(method == 10){ N = ESDIRK23Mod(fun,Jac,gfun,gJac,nx
         ...,nt,tspan,x0,AbsTol,RelTol,params,t,x); }
194  return N;
195 }
```

Appendix C

Implementations In Matlab

This Chapter includes the implementation code for the methods in `Matlab`. Each implementation has its own file and is split up into the main function and the step function. The implementation of Newton iterations also have its own file.

C.1 Unmodified Methods

Listing C.1: Implementation of the unmodified Euler in Matlab.

```
function [t,x] = Euler(fun,tspan,x0,AbsTol,RelTol,varargin)
% Solves x' = f(t,x) using Euler's method
%
% INPUTS
% fun    : function handle
% tspan  : span of time in which to approximate solution
%          or time vector in which to approximate solution
% x0     : initial conditions
% AbsTol: Absolute tolerance
% RelTol: Relative tolerance
%
% OUTPUTS
```

```
13 % t : time vector
14 % x : approximation vector
15
16 %% INITIALIZATION
17 % Various parameters
18 epsilon = 0.8;
19 p       = 1;
20 phat    = p+1;
21 kp      = 0.4/phat;
22 kI      = 0.3/phat;
23 h       = 1e-3; % Initial step size if using step size
       ...controller
24 fixedstepsize = length(tspan) > 2;
25 x       = zeros(length(tspan),length(x0));
26 x(1,:)  = x0;
27 %% Fixed step size
28 if(fixedstepsize)
29 h = (tspan(end) - tspan(1))/(length(tspan)-1);
30 t = tspan;
31 for i = 1:length(t)-1
32     % Euler steps
33     fn = feval(fun,t(i),x(i,:),varargin{:})';
34     x(i+1,:) = EulerStep(fun,fn,t(i),x(i,:),h,varargin{:});
35 end
36 else
37 %% Step size control
38 t    = zeros(1,1);
39 t(1) = tspan(1);
40 firststep = true;
41
42 i = 1;
43 while(t(i) < tspan(end))
44 % Make sure endpoint is included
45 if(h > tspan(end) - t(i))
46     h = tspan(end) - t(i);
47 end
48 % Full step
49 fn = feval(fun,t(i),x(i,:),varargin{:})';
50 xfull = EulerStep(fun,fn,t(i),x(i,:),h,varargin{:});
51
52 % Double step
53 xhalf = EulerStep(fun,fn,t(i),x(i,:),h/2,varargin{:});
54 fn = feval(fun,t(i)+h/2,xhalf,varargin{:})';
55 xdouble = EulerStep(fun,fn,t(i) + h/2,xhalf,h/2,varargin{:})
       ...;
56
57 % Error estimate
```

```
58 e = xfull - xdouble;
59 E = max(1e-10,max(abs(e)./(AbsTol + abs(xfull)*RelTol)));
60 % Accept or dismiss step
61 if(E<=1)
62     % Next step
63     t(i+1)   = t(i)+h;
64     x(i+1,:) = xdouble;
65     if(firststep)
66         % If it is the first step use asymptotic step size
               ...controller
67         h = h*(epsilon/E)^(1/phat);
68         firststep = false;
69     else
70         % New PI step size
71         h = h*(epsilon/E)^kI*(Eold/E)^kp;
72     end
73     % Save the error for use in the PI step size controller
74     Eold = E;
75     i = i+1;
76 else
77     % New asymptotic step size
78     h = h*(epsilon/E)^(1/phat);
79 end
80 end
81 end
82 end
83
84 function xnp1 = EulerStep(fun,fn,tn,xn,h,varargin)
85 xnp1 = xn + h*fn;
86 end
```

Listing C.2: Implementation of the unmodified RK4 in Matlab.

```
1
2 function [t,x] = RK4(fun,tspan,x0,AbsTol,RelTol,varargin)
3 % Solves x' = f(t,x) using the Classic Runge-Kutta method
4 %
5 % INPUTS
6 % fun    : function handle
7 % tspan  : span of time in which to approximate solution
8 %          or time vector in which to approximate solution
9 % x0     : initial conditions
10 % AbsTol: Absolute tolerance
11 % RelTol: Relative tolerance
12 %
13 % OUTPUTS
14 % t : time vector
15 % x : approximation vector
```

```
16
17 %% INITIALIZATION
18 % Butcher Tableau
19 A = [0,0,0,0 ;
20       1/2,0,0,0;
21       0,1/2,0,0;
22       0,0,1,0];
23 b = [1/6,1/3,1/3,1/6];
24 c = [0,1/2,1/2,1];
25 % Various parameters
26 epsilon = 0.8;
27 p = 4;
28 phat = p+1;
29 kp = 0.4/phat;
30 kI = 0.3/phat;
31 h = 1e-3; % Initial step size if using step size controller
32 fixedstepsize = length(tspan) > 2;
33 x = zeros(length(tspan),length(x0));
34 x(1,:) = x0;
35
36 if(fixedstepsize)
37 h = (tspan(end) - tspan(1))/(length(tspan)-1);
38 t = tspan;
39 for i = 1:length(t)-1
40     % RK4 steps
41     fn = feval(fun,t(i),x(i,:),varargin{:})';
42     x(i+1,:) = RK4Step(fun,fn,t(i),x(i,:),h,A,b,c,varargin
           ...{:});
43 end
44 else
45 t    = zeros(1,1);
46 t(1) = tspan(1);
47 firststep = true;
48
49 %% RK4 steps
50 i = 1;
51 while(t(i) < tspan(end))
52 % Make sure endpoint is included
53 if(h > tspan(end) - t(i))
54     h = tspan(end) - t(i);
55 end
56 % Full step
57 fn = feval(fun,t(i),x(i,:),varargin{:})';
58 xfull = RK4Step(fun,fn,t(i),x(i,:),h,A,b,c,varargin{:});
59
60 % Double step
61 xhalf = RK4Step(fun,fn,t(i),x(i,:),h/2,A,b,c,varargin{:});
```

```
62 fn = feval(fun,t(i)+h/2,xhalf,varargin{:})';
63 xdouble = RK4Step(fun,fn,t(i)+h/2,xhalf,h/2,A,b,c,varargin
       ...{:});
64
65 % Error estimate
66 e = xfull - xdouble;
67 E = max(1e-10,max(abs(e)./(AbsTol + abs(xfull)*RelTol)));
68
69 if(E<=1)
70     % Next step
71     t(i+1)   = t(i)+h;
72     x(i+1,:) = xdouble;
73     if(firststep)
74         % New asymptotic step size
75         h = h*(epsilon/E)^(1/phat);
76         firststep = false;
77     else
78         % New PI step size
79         h = h*(epsilon/E)^kI*(Eold/E)^kp;
80     end
81     % Save the error for use in the PI step size controller
82     Eold = E;
83     i = i+1;
84 else
85     % New asymptotic step size
86     h = h*(epsilon/E)^(1/phat);
87 end
88 end
89 end
90 end
91
92 function xnp1 = RK4Step(fun,fn,tn,xn,h,A,b,c,varargin)
93 X2 = xn + A(2,1)*h*fn;
94 f2 = feval(fun,tn + c(2)*h,X2,varargin{:})';
95
96 X3 = xn + A(3,2)*h*f2;
97 f3 = feval(fun,tn + c(3)*h,X3,varargin{:})';
98
99 X4 = xn + A(4,3)*h*f3;
100 f4 = feval(fun,tn + h,X4,varargin{:})';
101 xnp1 = xn + h*(b(1)*fn + b(2)*f2 + b(3)*f3 + b(4)*f4);
102 end
```

Listing C.3: Implementation of the unmodified RKF45 in Matlab.

```
1 function [t,x,E] = RKF45(fun,tspan,x0,AbsTol,RelTol,varargin
      ...)
2 % Solves x' = f(t,x) using the Runge-Kutta-Fehlberg method
```

```
3  %
4  % INPUTS
5  % fun    : function handle
6  % tspan : span of time in which to approximate solution
7  %          or time vector in which to approximate solution
8  % x0     : initial conditions
9  % AbsTol: Absolute tolerance
10 % RelTol: Relative tolerance
11 %
12 % OUTPUTS
13 % t : time vector
14 % x : approximation vector
15
16 %% INITIALIZATION
17 % RKF45 Butcher Tableau
18 A     = [0,0,0,0,0,0;
19          1/4,0,0,0,0,0;
20          3/32,9/32,0,0,0,0;
21          1932/2197,-7200/2197,7296/2197,0,0,0;
22          439/216,-8,3680/513,-845/4104,0,0;
23          -8/27 ,2,-3544/2565,1859/4104,-11/40,0];
24 b     = [25/216,0,1408/2565,2197/4104,-1/5,0];
25 bhat  = [16/135,0,6656/12825,28561/56430,-9/50,2/55];
26 c     = [0,1/4,3/8,12/13,1,1/2];
27 d     = b-bhat;
28 % Various parameters
29 epsilon = 0.8;
30 p = 4;
31 phat = p+1;
32 kp = 0.4/phat;
33 kI = 0.3/phat;
34 h = 1e-3; % Initial step size if using step size controller
35 fixedstepsize = length(tspan) > 2;
36 x = zeros(length(tspan),length(x0));
37 x(1,:) = x0;
38 if(fixedstepsize)
39 h = (tspan(end) - tspan(1))/(length(tspan)-1);
40 t = tspan;
41 for i = 1:length(t)-1
42     % RKF45 steps
43     fn = feval(fun,t(i),x(i,:),varargin{:})';
44     x(i+1,:) = RKF45Step(fun,fn,t(i),x(i,:),h,A,bhat,c,d,
           ...varargin{:});
45 end
46 else
47 t     = zeros(1,1);
48 t(1) = tspan(1);
```

```
49 firststep = true;
50
51 %% RK4 steps
52 i = 1;
53 while(t(i) < tspan(end))
54 % Make sure endpoint is included
55 if(h > tspan(end) - t(i))
56     h = tspan(end) - t(i);
57 end
58 % Full step
59 fn = feval(fun,t(i),x(i,:),varargin{:})';
60 [xfull,e] = RKF45Step(fun,fn,t(i),x(i,:),h,A,bhat,c,d,
       ...varargin{:});
61
62 % Error estimate
63 E = max(1e-10,max(abs(e)./(AbsTol + abs(xfull)*RelTol)));
64
65 if(E<=1)
66     % Next step
67     t(i+1)   = t(i)+h;
68     x(i+1,:) = xfull;
69     if(firststep)
70         % New asymptotic step size
71         h = h*(epsilon/E)^(1/phat);
72         firststep = false;
73     else
74         % New PI step size
75         h = h*(epsilon/E)^kI*(Eold/E)^kp;
76     end
77     % Save the error for use in the PI step size controller
78     Eold = E;
79     i = i+1;
80 else
81     % New asymptotic step size
82     h = h*(epsilon/E)^(1/phat);
83 end
84 end
85 end
86 end
87
88 function [xnp1,e] = RKF45Step(fun,fn,tn,xn,h,A,bhat,c,d,
       ...varargin)
89 X2 = xn + h*A(2,1)*fn;
90 f2 = feval(fun,tn + c(2)*h,X2,varargin{:})';
91
92 X3 = xn + h*(A(3,1)*fn + A(3,2)*f2);
93 f3 = feval(fun,tn + c(3)*h,X3,varargin{:})';
```

```
94
95  X4 = xn + h*(A(4,1)*fn + A(4,2)*f2 + A(4,3)*f3);
96  f4 = feval(fun,tn + c(4)*h,X4,varargin{:})';
97
98  X5 = xn + h*(A(5,1)*fn + A(5,2)*f2 + A(5,3)*f3 + A(5,4)*f4);
99  f5 = feval(fun,tn + c(5)*h,X5,varargin{:})';
100
101 X6 = xn + h*(A(6,1)*fn + A(6,2)*f2 + A(6,3)*f3 + A(6,4)*f4 +
        ... A(6,5)*f5);
102 f6 = feval(fun,tn + c(6)*h,X6,varargin{:})';
103
104 % xnp1 = xn + h*(b(1)*fn + b(3)*f3 + b(4)*f4 + b(5)*f5 + b
        ...(6)*f6);
105 xnp1 = xn + h*(bhat(1)*fn + bhat(3)*f3 + bhat(4)*f4 + bhat
        ...(5)*f5 + bhat(6)*f6);
106
107 e = h*(d(1)*fn + d(3)*f3 + d(4)*f4 + d(5)*f5 + d(6)*f6);
108 end
```

Listing C.4: Implementation of the unmodified DOPRI54 in Matlab.

```
1  function [t,x] = DOPRI54(fun,tspan,x0,AbsTol,RelTol,varargin
       ...)
2  % Solves x' = f(t,x) using the Dormand-Prince method
3  %
4  % INPUTS
5  % fun    : function handle
6  % tspan  : span of time in which to approximate solution
7  %          or time vector in which to approximate solution
8  % x0     : initial conditions
9  % AbsTol: Absolute tolerance
10 % RelTol: Relative tolerance
11 %
12 % OUTPUTS
13 % t : time vector
14 % x : approximation vector
15
16 %% INITIALIZATION
17 % DOPRI54 Butcher Tableau
18 A     = [0,0,0,0,0,0,0;
19          1/5,0,0,0,0,0,0;
20          3/40,9/40,0,0,0,0,0;
21          44/45,-56/15,32/9,0,0,0,0;
22          19372/6561,-25360/2187,64448/6561,-212/729,0,0,0;
23          9017/3168,-355/33,46732/5247,49/176,-5103/18656,0,0;
24          35/384,0,500/1113,125/192,-2187/6784,11/84,0];
25
```

```
b     =
      ...[5179/57600,0,7571/16695,393/640,-92097/339200,187/2100,1/40];
      ...
bhat  = [35/384,0,500/1113,125/192,-2187/6784,11/84,0];
c     = [0,1/5,3/10,4/5,8/9,1,1];
d     =  b-bhat;
% Various parameters
epsilon = 0.8;
p = 4;
kp = 0.4/(p+1);
kI = 0.3/(p+1);
h = 1e-3; % Initial step size if using step size controller
fixedstepsize = length(tspan) > 2;
x = zeros(length(tspan),length(x0));
x(1,:) = x0;
fn = feval(fun,tspan(1),x(1,:),varargin{:})';
if(fixedstepsize)
h = (tspan(end) - tspan(1))/(length(tspan)-1);
t = tspan;
for i = 1:length(t)-1
    % DOPRI54 steps
    [x(i+1,:),~,fn] = DOPRI54Step(fun,fn,t(i),x(i,:),h,A,c,d
        ...,varargin{:});
end
else
t     = zeros(1,1);
t(1) = tspan(1);
firststep = true;

%% DOPRI54 steps
i = 1;
while(t(i) < tspan(end))
% Make sure endpoint is included
if(h > tspan(end) - t(i))
    h = tspan(end) - t(i);
end
% Full step
[xfull,e,fnp1] = DOPRI54Step(fun,fn,t(i),x(i,:),h,A,c,d,
     ...varargin{:});

% Error estimate
E = max(1e-10,max(abs(e)./(AbsTol + abs(xfull)*RelTol)));
if(E<=1)
    % Next step
    t(i+1)   = t(i)+h;
    x(i+1,:) = xfull;
    fn = fnp1;
```

```
69      if(firststep)
70          % New asymptotic step size
71          h = h*(epsilon/E)^(1/(p+1));
72          firststep = false;
73      else
74          % New PI step size
75          h = h*(epsilon/E)^kI*(Eold/E)^kp;
76      end
77      % Save the error for use in the PI step size controller
78      Eold = E;
79      i = i+1;
80  else
81      % New asymptotic step size
82      h = h*(epsilon/E)^(1/(p+1));
83  end
84  end
85  end
86  end
87
88  function [xnp1,e,fnp1] = DOPRI54Step(fun,fn,tn,xn,h,A,c,d,
        ...varargin)
89  X2 = xn + A(2,1)*h*fn;
90  f2 = feval(fun,tn + c(2)*h,X2,varargin{:})';
91
92  X3 = xn + h*(A(3,1)*fn + A(3,2)*f2);
93  f3 = feval(fun,tn + c(3)*h,X3,varargin{:})';
94
95  X4 = xn + h*(A(4,1)*fn + A(4,2)*f2 + A(4,3)*f3);
96  f4 = feval(fun,tn + c(4)*h,X4,varargin{:})';
97
98  X5 = xn + h*(A(5,1)*fn + A(5,2)*f2 ...
99       + A(5,3)*f3 + A(5,4)*f4);
100 f5 = feval(fun,tn + c(5)*h,X5,varargin{:})';
101
102 X6 = xn + h*(A(6,1)*fn + A(6,2)*f2 + A(6,3)*f3 ...
103      + A(6,4)*f4 + A(6,5)*f5);
104 f6 = feval(fun,tn + c(6)*h,X6,varargin{:})';
105
106 X7 = xn + h*(A(7,1)*fn + A(7,3)*f3 + A(7,4)*f4 ...
107      + A(7,5)*f5 + A(7,6)*f6);
108 f7 = feval(fun,tn + c(7)*h,X7,varargin{:})';
109
110 xnp1 = X7;
111
112 e = h*(d(1)*fn + d(3)*f3 + d(4)*f4 + d(5)*f5 ...
113     + d(6)*f6 + d(7)*f7);
114 fnp1 = f7;
```

```
115 end
```

Listing C.5: Implementation of the unmodified ESDIRK23 in Matlab.

```
1  function [t,x] = ESDIRK23(fun,Jac,tspan,x0,AbsTol,RelTol,
       ...varargin)
2  % Solves x' = f(t,x) using the ESDIRK23 method
3  %
4  % INPUTS
5  % fun    : function handle
6  % tspan  : span of time in which to approximate solution
7  %          or time vector in which to approximate solution
8  % x0     : initial conditions
9  % AbsTol: Absolute tolerance
10 % RelTol: Relative tolerance
11 %
12 % OUTPUTS
13 % t : time vector
14 % x : approximation vector
15
16 %% INITIALIZATION
17 % ESDIRK23 Parameters
18 gamma = 1-1/sqrt(2);
19 a31 = (1-gamma)/2;
20 c  = [0; 2*gamma; 1];
21 b  = [a31;a31;gamma];
22 bhat = [    (6*gamma-1)/(12*gamma); ...
23     1/(12*gamma*(1-2*gamma)); ...
24     (1-3*gamma)/(3*(1-2*gamma))    ];
25 d  = b-bhat;
26 % Various parameters
27 epsilon = 0.8;
28 p = 2;
29 phat = p+1;
30 kp = 1/phat;
31 kI = 1/phat;
32 h = 1e-3; % Initial step size if using step size controller
33 hmin = 1e-6;
34 hmax = 1e1;
35 fixedstepsize = length(tspan) > 2;
36 x = zeros(length(tspan),length(x0));
37 x(1,:) = x0;
38 fn = feval(fun,tspan(1),x(1,:),varargin{:})';
39 if(fixedstepsize)
40 h = (tspan(end) - tspan(1))/(length(tspan)-1);
41 t = tspan;
42 for i = 1:length(t)-1
43     % ESDIRK23 steps
```

```
44      [x(i+1,:),~,fn] = ESDIRK23Step(fun,Jac,fn,t(i),x(i,:),h,
            ...AbsTol,RelTol,b,c,d,gamma,varargin{:});
45  end
46  else
47  t    = zeros(1,1);
48  t(1) = tspan(1);
49  firststep = true;
50
51  %% Main ESDIRK Integrator
52  i = 1;
53  while(t(i) < tspan(end))
54  % Make sure endpoint is included
55  if(h > tspan(end) - t(i))
56      h = tspan(end) - t(i);
57  end
58  % Full step
59  [xfull,e,fnp1,Convergence,Divergence,alpharatio] =
       ...ESDIRK23Step(fun,Jac,fn,t(i),x(i,:),h,AbsTol,RelTol,b
       ...,c,d,gamma,varargin{:});
60
61  % Error estimate
62  E = max(1e-10,max(abs(e)./(AbsTol + abs(xfull)*RelTol)));
63  % Step size control
64  if(Convergence)
65      if(E<=1)
66          % Next step
67          t(i+1)   = t(i)+h;
68          x(i+1,:) = xfull;
69          fn       = fnp1;
70          if(firststep)
71              % New asymptotic step size
72              h = h*(epsilon/E)^(1/phat);
73              firststep = false;
74          else
75              % New PI step size
76              hrat  = h/h_old;
77              Erat1 = (epsilon/E)^kI;
78              Erat2 = (Eold/E)^kp;
79              h = max(hmin,min(hmax,h*hrat*Erat1*Erat2));
80          end
81          % Save the error for use in PI step size controller
82          Eold = E;
83          h_old = h;
84          i = i+1;
85      else
86      % New asymptotic step size
87      h = max(hmin,min(hmax,h*(epsilon/E)^(1/phat)));
```

```
88      end
89
90      if(alpharatio < 1)
91          h = h*alpharatio;
92      end
93  elseif(Divergence)
94      halpha = h*alpharatio;
95      h = max(0.5*h,halpha);
96  else
97      if(alpharatio < 1)
98          halpha = h*alpharatio;
99          h = max(0.5*h,halpha);
100     else
101         h = h/2;
102     end
103 end
104 end
105 end
106 end
107
108 function [xnp1,e,fnp1,Convergence,Divergence,alpharatio] =
        ...ESDIRK23Step(fun,Jac,fn,tn,xn,h,AbsTol,RelTol,b,c,d,
        ...gamma,varargin)
109     % Various parameters
110     SlowConvergence = false;
111     Divergence      = false;
112     a21             = gamma;
113     itermax         = 1e1;
114     epsilon         = 0.8;
115     tau             = 0.1*epsilon;
116     alpha           = 0;
117     alpharef        = 0.4;
118     I               = eye(length(xn));
119
120     % Jacobian Update
121     J = feval(Jac,tn,xn,varargin{:});
122     dRdx = I - h*gamma*J;
123     [L,U,pivot] = lu(dRdx,'vector');
124
125     % Stage 2 of the ESDIRK23 Method
126     psi2 = xn + h*a21*fn;
127
128     % Initial guess for the state
129     T2 = tn + c(2)*h;
130     X2 = xn + c(2)*h*fn;
131
132     f2  = feval(fun,T2,X2,varargin{:})';
```

```
133     R2 = (X2 - h*gamma*f2 - psi2)';
134     rNewton = norm(R2'./(AbsTol + abs(X2).*RelTol),inf);
135     rNewtonOld = rNewton;
136     iter = 0;
137     Convergence = false;
138     while(~Convergence && ~SlowConvergence && ~Divergence)
139         dX = U\(L\(-R2(pivot,1)));
140         X2 = X2 + dX';
141         f2 = feval(fun,T2,X2,varargin{:})';
142         R2 = (X2 - h*gamma*f2 - psi2)';
143         rNewton = norm(R2'./(AbsTol + abs(X2).*RelTol),inf);
144         alpha = max(alpha,rNewton/rNewtonOld);
145         Convergence = rNewton < tau;
146         SlowConvergence = iter > itermax;
147         Divergence = alpha > 1;
148         rNewtonOld = rNewton;
149         iter = iter+1;
150     end
151
152     % Stage 3
153     psi3 = xn + h*(b(1)*fn+b(2)*f2);
154
155     % Initial guess for the state
156     T3 = tn + c(3)*h;
157     X3 = xn + c(3)*h*fn;
158
159     f3 = feval(fun,T3,X3,varargin{:})';
160     R3 = (X3 - h*gamma*f3 - psi3)';
161     rNewton = norm(R3'./(AbsTol + abs(X3).*RelTol),inf);
162     rNewtonOld = rNewton;
163     iter = 0;
164     Convergence = false;
165     while(~Convergence && ~SlowConvergence && ~Divergence)
166         dX = U\(L\(-R3(pivot,1)));
167         X3 = X3 + dX';
168         f3 = feval(fun,T3,X3,varargin{:})';
169         R3 = (X3 - h*gamma*f3 - psi3)';
170         rNewton = norm(R3'./(AbsTol + abs(X3).*RelTol),inf);
171         alpha = max(alpha,rNewton/rNewtonOld);
172         Convergence = rNewton < tau;
173         SlowConvergence = iter > itermax;
174         Divergence = alpha > 1;
175         rNewtonOld = rNewton;
176         iter = iter+1;
177     end
178
179     xnp1 = X3;
```

```
180     fnp1 = f3;
181     e = h*(d(1)*fn + d(2)*f2 + d(3)*f3);
182     alpharatio =  alpharef/alpha;
183 end
```

C.2 Modified Methods

Listing C.6: Implementation of the modified Euler in Matlab.

```
1  function [t,x] = EulerMod(fun,gfun,gJac,tspan,x0,AbsTol,
       ...RelTol,varargin)
2  % Solves g(x)' = f(t,x) using Euler's method
3  %
4  % fun   : function handle
5  % gfun  : function handle
6  % tspan : span of time in which to approximate solution
7  %         or time vector in which to approximate solution
8  % x0    : initial conditions
9  % AbsTol: Absolute tolerance
10 % RelTol: Relative tolerance
11 %
12 % OUTPUTS
13 % t : time vector
14 % x : approximation vector
15
16 %% INITIALIZATION
17 epsilon = 0.8;
18 p       = 1;
19 phat    = p+1;
20 kp      = 0.4/phat;
21 kI      = 0.3/phat;
22 h       = 1e-3; % Initial step size if using step size
       ...controller
23 hmin    = 1e-6;
24 hmax    = 1e1;
25 fixedstepsize = length(tspan) > 2;
26 x       = zeros(length(tspan),length(x0));
27 x(1,:)  = x0;
28 gnp1    = feval(gfun,x0,varargin{:})';
29 if(fixedstepsize)
30 h = (tspan(end) - tspan(1))/(length(tspan)-1);
31 t = tspan;
32 for i = 1:length(t)-1
33     % Euler steps
34     fn = feval(fun,t(i),x(i,:),varargin{:})';
```

```
35      [x(i+1,:),gnp1] = EulerModStep(fun,gfun,gJac,fn,gnp1,t(i
            ...),x(i,:),h,AbsTol,RelTol,varargin{:});
36  end
37  else
38  t     = zeros(1,1);
39  t(1)  = tspan(1);
40  firststep = true;
41
42  %% Euler steps
43  i = 1;
44  while(t(i) < tspan(end))
45  % Make sure endpoint is included
46  if(h > tspan(end) - t(i))
47      h = tspan(end) - t(i);
48  end
49  % Full step
50  fn = feval(fun,t(i),x(i,:),varargin{:})';
51  [xfull,~,Convergence,Divergence] = EulerModStep(fun,gfun,
        ...gJac,fn,gnp1,t(i),x(i,:),h,AbsTol,RelTol,varargin{:})
        ...;
52
53  % Double step
54  [xhalf,ghalf,Con,Div] = EulerModStep(fun,gfun,gJac,fn,gnp1,t
        ...(i),x(i,:),h/2,AbsTol,RelTol,varargin{:});
55  Convergence = Convergence && Con;
56  Divergence  = Divergence  || Div;
57  fn = feval(fun,t(i)+h/2,xhalf,varargin{:})';
58  [xdouble,gdouble,Con,Div,alpharatio] = EulerModStep(fun,gfun
        ...,gJac,fn,ghalf,t(i) + h/2,xhalf,h/2,AbsTol,RelTol,
        ...varargin{:});
59  Convergence = Convergence && Con;
60  Divergence  = Divergence  || Div;
61
62  % Error estimate
63  e = xfull - xdouble;
64  E = max(1e-10,max(abs(e)./(AbsTol + abs(xfull)*RelTol)));
65  %fprintf('h = %.4f, E = %.4f\n',h,E);
66  % disp(xfull)
67  % disp(xdouble)
68  if(Convergence)
69      if(E<=1)
70  %disp('Step');
71          % Next step
72          t(i+1)   = t(i)+h;
73          x(i+1,:) = xdouble;
74          gnp1 = gdouble;
75          if(firststep)
```

```
76              % New asymptotic step size
77              h = max(hmin,min(hmax,h*(epsilon/E)^(1/phat)));
78              firststep = false;
79          else
80              % New PI step size
81              h = max(hmin,min(hmax,h*(epsilon/E)^kI*(Eold/E)^
                    ...kp));
82          end
83          % Save the error for use in the PI step size
                ...controller
84          Eold = E;
85          i = i+1;
86      else
87  %disp('Fail');
88          % New asymptotic step size
89          h = max(hmin,min(hmax,h*(epsilon/E)^(1/phat)));
90      end
91
92      if(alpharatio < 1)
93          h = h*alpharatio;
94      end
95  elseif(Divergence)
96  %disp('Div');
97      halpha = h*alpharatio;
98      h      = max(0.5*h,halpha);
99  else
100 %disp('Slow');
101     if(alpharatio < 1)
102         halpha = h*alpharatio;
103         h      = max(0.5*h,halpha);
104     else
105         h = h/2;
106     end
107 end
108 end
109 end
110 end
111
112 function [xnp1,gnp1,Convergence,Divergence,alpharatio] =
        ...EulerModStep(fun,gfun,gJac,fn,gn,tn,xn,h,AbsTol,
        ...RelTol,varargin)
113 % Full Euler step
114 gnp1 = gn + h*fn;
115 [xnp1,Convergence,Divergence,alpharatio] = NewtonSolve(gfun,
        ...gJac,gnp1,xn,AbsTol,RelTol,varargin{:});
116 end
```

Listing C.7: Implementation of the modified RK4 in Matlab.

```
function [t,x] = RK4Mod(fun,gfun,gJac,tspan,x0,AbsTol,RelTol
    ...,varargin)
% Solves g(x)' = f(t,x) using Euler's method
%
% fun    : function handle
% gfun   : function handle
% tspan  : span of time in which to approximate solution
%          or time vector in which to approximate solution
% x0     : initial conditions
% AbsTol: Absolute tolerance
% RelTol: Relative tolerance
%
% OUTPUTS
% t : time vector
% x : approximation vector

%% INITIALIZATION
% Butcher Tableau
A = [0,0,0,0 ;
     1/2,0,0,0;
     0,1/2,0,0;
     0,0,1,0];
b = [1/6,1/3,1/3,1/6];
c = [0,1/2,1/2,1];
% Various Parameters
epsilon = 0.8;
p       = 4;
phat    = p+1;
kp      = 0.4/phat;
kI      = 0.3/phat;
h       = 1e-3; % Initial step size if using step size
    ...controller
hmin    = 1e-6;
hmax    = 1e1;
fixedstepsize = length(tspan) > 2;
x       = zeros(length(tspan),length(x0));
x(1,:)  = x0;
gnp1    = feval(gfun,x0,varargin{:})';
if(fixedstepsize)
h = (tspan(end) - tspan(1))/(length(tspan)-1);
t = tspan;
for i = 1:length(t)-1
    % Euler steps
    fn = feval(fun,t(i),x(i,:),varargin{:})';
    [x(i+1,:),gnp1] = RK4ModStep  (fun,gfun,gJac,fn,gnp1,t(i
        ...),x(i,:),h,AbsTol,RelTol,A,b,c,varargin{:});
```

```
44 end
45 else
46 t    = zeros(1,1);
47 t(1) = tspan(1);
48 firststep = true;
49
50 %% Euler steps
51 i = 1;
52 while(t(i) < tspan(end))
53 % Make sure endpoint is included
54 if(h > tspan(end) - t(i))
55     h = tspan(end) - t(i);
56 end
57 % Full step
58 fn = feval(fun,t(i),x(i,:),varargin{:})';
59 [xfull,~,Convergence,Divergence] = RK4ModStep(fun,gfun,gJac,
       ...fn,gnp1,t(i),x(i,:),h,AbsTol,RelTol,A,b,c,varargin
       ...{:});
60
61 % Double step
62 [xhalf,ghalf,Con,Div] = RK4ModStep(fun,gfun,gJac,fn,gnp1,t(i
       ...),x(i,:),h/2,AbsTol,RelTol,A,b,c,varargin{:});
63 Convergence = Convergence && Con;
64 Divergence  = Divergence  || Div;
65 fn = feval(fun,t(i)+h/2,xhalf,varargin{:})';
66 [xdouble,gdouble,Con,Div,alpharatio] = RK4ModStep(fun,gfun,
       ...gJac,fn,ghalf,t(i) + h/2,xhalf,h/2,AbsTol,RelTol,A,b,
       ...c,varargin{:});
67 Convergence = Convergence && Con;
68 Divergence  = Divergence  || Div;
69
70 % Error estimate
71 e = xfull - xdouble;
72 E = max(1e-10,max(abs(e)./(AbsTol + abs(xfull)*RelTol)));
73 if(Convergence)
74     if(E<=1)
75         % Next step
76         t(i+1)   = t(i)+h;
77         x(i+1,:) = xdouble;
78         gnp1 = gdouble;
79         if(firststep)
80             % New asymptotic step size
81             h = max(hmin,min(hmax,h*(epsilon/E)^(1/phat)));
82             firststep = false;
83         else
84             % New PI step size
```

```
85              h = max(hmin,min(hmax,h*(epsilon/E)^kI*(Eold/E)^
                    ...kp));
86          end
87          % Save the error for use in the PI step size
                ...controller
88          Eold = E;
89          i = i+1;
90      else
91          % New asymptotic step size
92          h = max(hmin,min(hmax,h*(epsilon/E)^(1/phat)));
93      end
94
95      if(alpharatio < 1)
96          h = h*alpharatio;
97      end
98  elseif(Divergence)
99      halpha = h*alpharatio;
100     h      = max(0.5*h,halpha);
101 else
102     if(alpharatio < 1)
103         halpha = h*alpharatio;
104         h      = max(0.5*h,halpha);
105     else
106         h = h/2;
107     end
108 end
109 end
110 end
111 end
112
113 function [xnp1,gnp1,Convergence,Divergence,alpharatio] =
        ...RK4ModStep(fun,gfun,gJac,fn,gn,tn,xn,h,AbsTol,RelTol,
        ...A,b,c,varargin)
114 % Full RK4 step
115 G2 = gn + A(2,1)*h*fn;
116 [X2,Conv1,Div1,alpharat1] = NewtonSolve(gfun,gJac,G2,xn,
        ...AbsTol,RelTol,varargin{:});
117 f2 = feval(fun,tn + c(2)*h,X2,varargin{:})';
118
119 G3 = gn + A(3,2)*h*f2;
120 [X3,Conv2,Div2,alpharat2] = NewtonSolve(gfun,gJac,G3,X2,
        ...AbsTol,RelTol,varargin{:});
121 f3 = feval(fun,tn + c(3)*h,X3,varargin{:})';
122
123 G4 = gn + A(4,3)*h*f3;
124 [X4,Conv3,Div3,alpharat3] = NewtonSolve(gfun,gJac,G4,X3,
        ...AbsTol,RelTol,varargin{:});
```

```
125 f4 = feval(fun,tn + h,X4,varargin{:})';
126
127 gnp1 = gn + h*(b(1)*fn + b(2)*f2 + b(3)*f3 + b(4)*f4);
128 [xnp1,Conv4,Div4,alpharat4] = NewtonSolve(gfun,gJac,gnp1,X4,
        ...AbsTol,RelTol,varargin{:});
129
130 Convergence = Conv1 && Conv2 && Conv3 && Conv4;
131 Divergence  = Div1  || Div2  || Div3  || Div4;
132 alpharatio  = min([alpharat1,alpharat2,alpharat3,alpharat4])
        ...;
133 end
```

Listing C.8: Implementation of the modified RKF45 in Matlab.

```
1  function [t,x] = RKF45Mod(fun,gfun,gJac,tspan,x0,AbsTol,
       ...RelTol,varargin)
2  % Solves g(x)' = f(t,x) using Euler's method
3  %
4  % fun   : function handle
5  % gfun  : function handle
6  % tspan : span of time in which to approximate solution
7  %         or time vector in which to approximate solution
8  % x0    : initial conditions
9  % AbsTol: Absolute tolerance
10 % RelTol: Relative tolerance
11 %
12 % OUTPUTS
13 % t : time vector
14 % x : approximation vector
15
16 %% INITIALIZATION
17 % RKF45 Butcher Tableau
18 A    = [0,0,0,0,0,0;
19         1/4,0,0,0,0,0;
20         3/32,9/32,0,0,0,0;
21         1932/2197,-7200/2197,7296/2197,0,0,0;
22         439/216,-8,3680/513,-845/4104,0,0;
23         -8/27 ,2,-3544/2565,1859/4104,-11/40,0];
24 b    = [25/216,0,1408/2565,2197/4104,-1/5,0];
25 bhat = [16/135,0,6656/12825,28561/56430,-9/50,2/55];
26 c    = [0,1/4,3/8,12/13,1,1/2];
27 d    = b-bhat;
28 % Various Parameters
29 epsilon = 0.8;
30 p       = 4;
31 phat    = p+1;
32 kp      = 0.4/phat;
33 kI      = 0.3/phat;
```

```
34 h        = 1e-3; % Initial step size if using step size
       ...controller
35 hmin     = 1e-6;
36 hmax     = 1e1;
37 fixedstepsize = length(tspan) > 2;
38 x        = zeros(length(tspan),length(x0));
39 x(1,:)   = x0;
40 gnp1     = feval(gfun,x0,varargin{:})';
41 if(fixedstepsize)
42 h = (tspan(end) - tspan(1))/(length(tspan)-1);
43 t = tspan;
44 for i = 1:length(t)-1
45     % Euler steps
46     fn = feval(fun,t(i),x(i,:),varargin{:})';
47     [x(i+1,:),~,gnp1] = RKF45ModStep(fun,gfun,gJac,fn,gnp1,t
           ...(i),x(i,:),h,AbsTol,RelTol,A,bhat,c,d,varargin
           ...{:});
48 end
49 else
50 t     = zeros(1,1);
51 t(1) = tspan(1);
52 firststep = true;
53
54 %% Euler steps
55 i = 1;
56 while(t(i) < tspan(end))
57 % Make sure endpoint is included
58 if(h > tspan(end) - t(i))
59     h = tspan(end) - t(i);
60 end
61 % Full step
62 fn = feval(fun,t(i),x(i,:),varargin{:})';
63 [xfull,e,gfull,Convergence,Divergence,alpharatio] =
       ...RKF45ModStep(fun,gfun,gJac,fn,gnp1,t(i),x(i,:),h,
       ...AbsTol,RelTol,A,bhat,c,d,varargin{:});
64
65 % Error estimate
66 E = max(1e-10,max(abs(e)./(AbsTol + abs(xfull)*RelTol)));
67 %fprintf('h = %.4f, E = %.4f\n',h,E);
68 % disp(xfull)
69 % disp(xdouble)
70 if(Convergence)
71     if(E<=1)
72 %disp('Step');
73         % Next step
74         t(i+1)    = t(i)+h;
75         x(i+1,:) = xfull;
```

```
76          gnp1      = gfull;
77          if(firststep)
78              % New asymptotic step size
79              h = max(hmin,min(hmax,h*(epsilon/E)^(1/phat)));
80              firststep = false;
81          else
82              % New PI step size
83              h = max(hmin,min(hmax,h*(epsilon/E)^kI*(Eold/E)^
                    ...kp));
84          end
85          % Save the error for use in the PI step size
                ...controller
86          Eold = E;
87          i = i+1;
88      else
89  %disp('Fail');
90          % New asymptotic step size
91          h = max(hmin,min(hmax,h*(epsilon/E)^(1/phat)));
92      end
93
94      if(alpharatio < 1)
95          h = h*alpharatio;
96      end
97  elseif(Divergence)
98  %disp('Div');
99      halpha = h*alpharatio;
100     h      = max(0.5*h,halpha);
101 else
102 %disp('Slow');
103     if(alpharatio < 1)
104         halpha = h*alpharatio;
105         h      = max(0.5*h,halpha);
106     else
107         h = h/2;
108     end
109 end
110 end
111 end
112 end
113
114 function [xnp1,e,gnp1,Convergence,Divergence,alpharatio] =
        ...RKF45ModStep(fun,gfun,gJac,fn,gn,tn,xn,h,AbsTol,
        ...RelTol,A,bhat,c,d,varargin)
115 %% Full RKF45 step
116 G2 = gn + A(2,1)*h*fn;
117 [X2,Conv1,Div1,alpharat1] = NewtonSolve(gfun,gJac,G2,xn,
        ...AbsTol,RelTol,varargin{:});
```

```
118 f2 = feval(fun,tn + c(2)*h,X2,varargin{:})';
119
120 G3 = gn + h*(A(3,1)*fn + A(3,2)*f2);
121 [X3,Conv2,Div2,alpharat2] = NewtonSolve(gfun,gJac,G3,X2,
        ...AbsTol,RelTol,varargin{:});
122 f3 = feval(fun,tn + c(3)*h,X3,varargin{:})';
123
124 G4 = gn + h*(A(4,1)*fn + A(4,2)*f2 + A(4,3)*f3);
125 [X4,Conv3,Div3,alpharat3] = NewtonSolve(gfun,gJac,G4,X3,
        ...AbsTol,RelTol,varargin{:});
126 f4 = feval(fun,tn + c(4)*h,X4,varargin{:})';
127
128 G5 = gn + h*(A(5,1)*fn + A(5,2)*f2 + A(5,3)*f3 + A(5,4)*f4);
129 [X5,Conv4,Div4,alpharat4] = NewtonSolve(gfun,gJac,G5,X4,
        ...AbsTol,RelTol,varargin{:});
130 f5 = feval(fun,tn + c(5)*h,X5,varargin{:})';
131
132 G6 = gn + h*(A(6,1)*fn + A(6,2)*f2 + A(6,3)*f3 + A(6,4)*f4 +
        ... A(6,5)*f5);
133 [X6,Conv5,Div5,alpharat5] = NewtonSolve(gfun,gJac,G6,X5,
        ...AbsTol,RelTol,varargin{:});
134 f6 = feval(fun,tn + c(6)*h,X6,varargin{:})';
135
136 gnp1 = gn + h*(bhat(1)*fn + bhat(3)*f3 + bhat(4)*f4 + bhat
        ...(5)*f5 + bhat(6)*f6);
137 [xnp1,Conv6,Div6,alpharat6] = NewtonSolve(gfun,gJac,gnp1,X6,
        ...AbsTol,RelTol,varargin{:});
138
139 e = h*(d(1)*fn + d(3)*f3 + d(4)*f4 + d(5)*f5 + d(6)*f6);
140
141 Convergence = Conv1 && Conv2 && Conv3 && Conv4 && Conv5 &&
        ...Conv6;
142 Divergence  = Div1  || Div2  || Div3  || Div4  || Div5  ||
        ...Div6;
143 alpharatio  = min([alpharat1,alpharat2,alpharat3,alpharat4,
        ...alpharat5,alpharat6]);
144 end
```

Listing C.9: Implementation of the modified DOPRI54 in Matlab.

```
1 function [t,x] = DOPRI54Mod(fun,gfun,gJac,tspan,x0,AbsTol,
        ...RelTol,varargin)
2 % Solves g(x)' = f(t,x) using Euler's method
3 %
4 % fun   : function handle
5 % gfun  : function handle
6 % tspan : span of time in which to approximate solution
7 %         or time vector in which to approximate solution
```

```
% x0     : initial conditions
% AbsTol: Absolute tolerance
% RelTol: Relative tolerance
%
% OUTPUTS
% t : time vector
% x : approximation vector

%% INITIALIZATION
% DOPRI54 Butcher Tableau
A     = [0,0,0,0,0,0,0;
         1/5,0,0,0,0,0,0;
         3/40,9/40,0,0,0,0,0;
         44/45,-56/15,32/9,0,0,0,0;
         19372/6561,-25360/2187,64448/6561,-212/729,0,0,0;
         9017/3168,-355/33,46732/5247,49/176,-5103/18656,0,0;
         35/384,0,500/1113,125/192,-2187/6784,11/84,0];

b     =
     ...[5179/57600,0,7571/16695,393/640,-92097/339200,187/2100,1/40];
     ...
bhat  = [35/384,0,500/1113,125/192,-2187/6784,11/84,0];
c     = [0,1/5,3/10,4/5,8/9,1,1];
d     =  b-bhat;
% Various Parameters
epsilon = 0.8;
p       = 4;
phat    = p+1;
kp      = 0.4/phat;
kI      = 0.3/phat;
h       = 1e-3; % Initial step size if using step size
     ...controller
hmin    = 1e-6;
hmax    = 1e1;
fixedstepsize = length(tspan) > 2;
x       = zeros(length(tspan),length(x0));
x(1,:)  = x0;
gnp1    = feval(gfun,x0,varargin{:})';
fn = feval(fun,tspan(1),x(1,:),varargin{:})';
if(fixedstepsize)
h = (tspan(end) - tspan(1))/(length(tspan)-1);
t = tspan;
for i = 1:length(t)-1
    % Euler steps
    [x(i+1,:),~,fn,gnp1] = DOPRI54ModStep(fun,gfun,gJac,fn,
         ...gnp1,t(i),x(i,:),h,AbsTol,RelTol,A,c,d,varargin
         ...{:});
```

```
50 end
51 else
52 t    = zeros(1,1);
53 t(1) = tspan(1);
54 firststep = true;
55
56 %% DOPRI54 steps
57 i = 1;
58 while(t(i) < tspan(end))
59 % Make sure endpoint is included
60 if(h > tspan(end) - t(i))
61     h = tspan(end) - t(i);
62 end
63 % Full step
64 [xfull,e,fnp1,gfull,Convergence,Divergence,alpharatio] =
       ...DOPRI54ModStep(fun,gfun,gJac,fn,gnp1,t(i),x(i,:),h,
       ...AbsTol,RelTol,A,c,d,varargin{:});
65
66 % Error estimate
67 E = max(1e-10,max(abs(e)./(AbsTol + abs(xfull)*RelTol)));
68 %fprintf('h = %.4f, E = %.4f\n',h,E);
69 % disp(xfull)
70 % disp(xdouble)
71 if(Convergence)
72     if(E<=1)
73 %disp('Step');
74         % Next step
75         t(i+1)   = t(i)+h;
76         x(i+1,:) = xfull;
77         fn = fnp1;
78         gnp1 = gfull;
79         if(firststep)
80             % New asymptotic step size
81             h = max(hmin,min(hmax,h*(epsilon/E)^(1/phat)));
82             firststep = false;
83         else
84             % New PI step size
85             h = max(hmin,min(hmax,h*(epsilon/E)^kI*(Eold/E)^
                   ...kp));
86         end
87         % Save the error for use in the PI step size
               ...controller
88         Eold = E;
89         i = i+1;
90     else
91 %disp('Fail');
92         % New asymptotic step size
```

```
93          h = max(hmin,min(hmax,h*(epsilon/E)^(1/phat)));
94      end
95
96      if(alpharatio < 1)
97          h = h*alpharatio;
98      end
99  elseif(Divergence)
100 %disp('Div');
101     halpha = h*alpharatio;
102     h      = max(0.5*h,halpha);
103 else
104 %disp('Slow');
105     if(alpharatio < 1)
106         halpha = h*alpharatio;
107         h      = max(0.5*h,halpha);
108     else
109         h = h/2;
110     end
111 end
112 end
113 end
114 end
115
116 function [xnp1,e,fnp1,gnp1,Convergence,Divergence,alpharatio
        ...] = DOPRI54ModStep(fun,gfun,gJac,fn,gn,tn,xn,h,AbsTol
        ...,RelTol,A,c,d,varargin)
117 %% Full DOPRI54 step
118 G2 = gn + h*A(2,1)*fn;
119 [X2,Conv1,Div1,alpharat1] = NewtonSolve(gfun,gJac,G2,xn,
        ...AbsTol,RelTol,varargin{:});
120 f2 = feval(fun,tn + c(2)*h,X2,varargin{:})';
121
122 G3 = gn + h*(A(3,1)*fn + A(3,2)*f2);
123 [X3,Conv2,Div2,alpharat2] = NewtonSolve(gfun,gJac,G3,X2,
        ...AbsTol,RelTol,varargin{:});
124 f3 = feval(fun,tn + c(3)*h,X3,varargin{:})';
125
126 G4 = gn + h*(A(4,1)*fn + A(4,2)*f2 + A(4,3)*f3);
127 [X4,Conv3,Div3,alpharat3] = NewtonSolve(gfun,gJac,G4,X3,
        ...AbsTol,RelTol,varargin{:});
128 f4 = feval(fun,tn + c(4)*h,X4,varargin{:})';
129
130 G5 = gn + h*(A(5,1)*fn + A(5,2)*f2 + A(5,3)*f3 + A(5,4)*f4);
131 [X5,Conv4,Div4,alpharat4] = NewtonSolve(gfun,gJac,G5,X4,
        ...AbsTol,RelTol,varargin{:});
132 f5 = feval(fun,tn + c(5)*h,X5,varargin{:})';
133
```

```
134 G6 = gn + h*(A(6,1)*fn + A(6,2)*f2 + A(6,3)*f3 + A(6,4)*f4 +
        ... A(6,5)*f5);
135 [X6,Conv5,Div5,alpharat5] = NewtonSolve(gfun,gJac,G6,X5,
        ...AbsTol,RelTol,varargin{:});
136 f6 = feval(fun,tn + c(6)*h,X6,varargin{:})';
137
138 G7 = gn + h*(A(7,1)*fn + A(7,3)*f3 + A(7,4)*f4 + A(7,5)*f5 +
        ... A(7,6)*f6);
139 [X7,Conv6,Div6,alpharat6] = NewtonSolve(gfun,gJac,G7,X6,
        ...AbsTol,RelTol,varargin{:});
140 f7 = feval(fun,tn + c(7)*h,X7,varargin{:})';
141
142 xnp1 = X7;
143 fnp1 = f7;
144 gnp1 = G7;
145
146 e = h*(d(1)*fn + d(3)*f3 + d(4)*f4 + d(5)*f5 + d(6)*f6 + d
        ...(7)*f7);
147
148 Convergence = Conv1 && Conv2 && Conv3 && Conv4 && Conv5 &&
        ...Conv6;
149 Divergence  = Div1  || Div2  || Div3  || Div4  || Div5  ||
        ...Div6;
150 alpharatio  = min([alpharat1,alpharat2,alpharat3,alpharat4,
        ...alpharat5,alpharat6]);
151 end
```

Listing C.10: Implementation of the modified ESDIRK23 in Matlab.

```
1  function [t,x] = ESDIRK23Mod(fun,Jac,gfun,gJac,tspan,x0,
       ...AbsTol,RelTol,varargin)
2  % Solves x' = f(t,x) using the ESDIRK23 method
3  %
4  % INPUTS
5  % fun   : function handle
6  % tspan : span of time in which to approximate solution
7  %         or time vector in which to approximate solution
8  % x0    : initial conditions
9  % AbsTol: Absolute tolerance
10 % RelTol: Relative tolerance
11 %
12 % OUTPUTS
13 % t : time vector
14 % x : approximation vector
15
16 %% INITIALIZATION
17 % ESDIRK23 Parameters
18 gamma = 1-1/sqrt(2);
```

```
a31 = (1-gamma)/2;
c   = [0; 2*gamma; 1];
b   = [a31;a31;gamma];
bhat = [     (6*gamma-1)/(12*gamma); ...
     1/(12*gamma*(1-2*gamma)); ...
     (1-3*gamma)/(3*(1-2*gamma))     ];
d   = b-bhat;
% Various parameters
epsilon = 0.8;
p = 2;
phat = p+1;
kp = 1/phat;
kI = 1/phat;
h = 1e-3; % Initial step size if using step size controller
hmin = 1e-6;
hmax = 1e1;
fixedstepsize = length(tspan) > 2;
x = zeros(length(tspan),length(x0));
x(1,:) = x0;
fn = feval(fun,tspan(1),x(1,:),varargin{:})';
gnp1 = feval(gfun,x0,varargin{:})';
if(fixedstepsize)
h = (tspan(end) - tspan(1))/(length(tspan)-1);
t = tspan;
for i = 1:length(t)-1
    % ESDIRK23 steps
    [x(i+1,:),~,fn,gnp1] = ESDIRK23Step(fun,Jac,gfun,gJac,fn
        ...,gnp1,t(i),x(i,:),h,AbsTol,RelTol,b,c,d,gamma,
        ...varargin{:});
end
else
t    = zeros(1,1);
t(1) = tspan(1);
firststep = true;

%% Main ESDIRK Integrator
i = 1;
while(t(i) < tspan(end))
% Make sure endpoint is included
if(h > tspan(end) - t(i))
    h = tspan(end) - t(i);
end
% Full step
[xfull,e,fnp1,gfull,Convergence,Divergence,alpharatio] =
    ...ESDIRK23Step(fun,Jac,gfun,gJac,fn,gnp1,t(i),x(i,:),h,
    ...AbsTol,RelTol,b,c,d,gamma,varargin{:});

```

```
% Error estimate
E = max(1e-10,max(abs(e)./(AbsTol + abs(xfull)*RelTol)));
% Step size control
if(Convergence)
    if(E<=1)
        % Next step
        t(i+1)   = t(i)+h;
        x(i+1,:) = xfull;
        fn = fnp1;
        gnp1 = gfull;
        if(firststep)
            % New asymptotic step size
            h = h*(epsilon/E)^(1/phat);
            firststep = false;
        else
            % New PI step size
            hrat  = h/h_old;
            Erat1 = (epsilon/E)^kI;
            Erat2 = (Eold/E)^kp;
            h = max(hmin,min(hmax,h*hrat*Erat1*Erat2));
        end
        % Save the error for use in PI step size controller
        Eold = E;
        h_old = h;
        i = i+1;
    else
    % New asymptotic step size
    h = max(hmin,min(hmax,h*(epsilon/E)^(1/phat)));
    end

    if(alpharatio < 1)
        h = h*alpharatio;
    end
elseif(Divergence)
    halpha = h*alpharatio;
    h = max(0.5*h,halpha);
else
    if(alpharatio < 1)
        halpha = h*alpharatio;
        h = max(0.5*h,halpha);
    else
        h = h/2;
    end
end
end
end
end
```

```
109
110 function [xnp1,e,fnp1,gnp1,Convergence,Divergence,alpharatio
        ...] = ESDIRK23Step(fun,Jac,gfun,gJac,fn,gn,tn,xn,h,
        ...AbsTol,RelTol,b,c,d,gamma,varargin)
111     % Various parameters
112     SlowConvergence = false;
113     Divergence      = false;
114     a21             = gamma;
115     itermax         = 1e1;
116     epsilon         = 0.8;
117     tau             = 0.1*epsilon;
118     alpha           = 0;
119     alpharef        = 0.4;
120
121     % Jacobian Update
122     J    = feval(Jac,tn,xn,varargin{:});
123     dgdx = feval(gJac,  xn,varargin{:});
124     dRdx = dgdx - h*gamma*J;
125     [L,U,pivot] = lu(dRdx,'vector');
126
127     % Stage 2 of the ESDIRK23 Method
128     psi2 = gn + h*a21*fn;
129
130     % Initial guess for the state
131     T2 = tn + c(2)*h;
132     X2 = xn;
133
134     f2  = feval(fun,T2,X2,varargin{:})';
135     G2 = feval(gfun,X2,varargin{:})';
136     R2 = (G2 - h*gamma*f2 - psi2)';
137     rNewton = norm(R2'./(AbsTol + abs(G2).*RelTol),inf);
138     rNewtonOld = rNewton;
139     iter = 0;
140     Convergence = false;
141     while(~Convergence && ~SlowConvergence && ~Divergence)
142         dX = U\(L\(-R2(pivot,1)));
143         X2 = X2 + dX';
144         f2 = feval(fun,T2,X2,varargin{:})';
145         G2 = feval(gfun,X2,varargin{:})';
146         R2 = (G2 - h*gamma*f2 - psi2)';
147         rNewton = norm(R2'./(AbsTol + abs(G2).*RelTol),inf);
148         alpha = max(alpha,rNewton/rNewtonOld);
149         Convergence = rNewton < tau;
150         SlowConvergence = iter > itermax;
151         Divergence = alpha > 1;
152         rNewtonOld = rNewton;
153         iter = iter+1;
```

```
154     end
155
156     % Stage 3
157     psi3 = gn + h*(b(1)*fn+b(2)*f2);
158
159     % Initial guess for the state
160     T3 = tn + c(3)*h;
161     X3 = xn;
162
163     f3 = feval(fun,T3,X3,varargin{:})';
164     G3 = feval(gfun,X3,varargin{:})';
165     R3 = (G3 - h*gamma*f3 - psi3)';
166     rNewton = norm(R3'./(AbsTol + abs(G3).*RelTol),inf);
167     rNewtonOld = rNewton;
168     iter = 0;
169     Convergence = false;
170     while(~Convergence && ~SlowConvergence && ~Divergence)
171         dX = U\(L\(-R3(pivot,1)));
172         X3 = X3 + dX';
173         f3 = feval(fun,T3,X3,varargin{:})';
174         G3 = feval(gfun,X3,varargin{:})';
175         R3 = (G3 - h*gamma*f3 - psi3)';
176         rNewton = norm(R3'./(AbsTol + abs(G3).*RelTol),inf);
177         alpha = max(alpha,rNewton/rNewtonOld);
178         Convergence = rNewton < tau;
179         SlowConvergence = iter > itermax;
180         Divergence = alpha > 1;
181         rNewtonOld = rNewton;
182         iter = iter+1;
183     end
184     xnp1 = X3;
185     fnp1 = f3;
186     gnp1 = G3;
187
188     e = h*(d(1)*fn + d(2)*f2 + d(3)*f3);
189     alpharatio = alpharef/alpha;
190 end
```

C.3 Newton Iterations

Listing C.11: Implementation of Newton iterations in Matlab.

```
1 function [xnp1,Convergence,Divergence,alpharatio] =
      ...NewtonSolve(gfun,gJac,gnp1,x0,AbsTol,RelTol,varargin)
2 % Various parameters
```

```
SlowConvergence = false;
Divergence      = false;
itermax         = 1e1;
epsilon         = 0.8;
tau             = 0.1*epsilon;
alpha           = 0;
alpharef        = 0.4;
%% Obtain Solution
% Start guess
xnp1 = x0;
g  = feval(gfun,xnp1,varargin{:})';
dg = feval(gJac,xnp1,varargin{:});
[L,U,pivot] = lu(dg,'vector');
% Residual
R = g-gnp1;
rNewton = norm(R./(AbsTol+abs(gnp1)*RelTol),inf);
rNewtonOld = rNewton;
iter = 0;
Convergence = rNewton < tau;
% Newton steps
while(~Convergence && ~SlowConvergence && ~Divergence)
    R = R';
    dxn = U\(L\(R(pivot,1)));
    xnp1 = xnp1 - dxn';
    g = feval(gfun,xnp1,varargin{:})';
    R = g-gnp1;
    rNewton = norm(R./(AbsTol+abs(gnp1)*RelTol),'inf');
    alpha = max(alpha,rNewton/rNewtonOld);
    Convergence = rNewton < tau;
    SlowConvergence = iter > itermax;
    Divergence = alpha > 1;
    rNewtonOld = rNewton;
    iter = iter+1;
end
alpharatio = alpharef/alpha;
end
```

C.4 Sequential Simulations

Listing C.12: Implementation of sequential simulations in Matlab.

```
function runtime = FedBatchVaryParameters(method,h)
% Calculates the production for various parameter
    ...combinations using the
```

```
% numerical method specified by 'method' and measures the
    ...runtime

%% INITIALIZATION
% Initial conditions / Optimal Values
V0  = 100; Vmax = 1200;
CX0 = 20; % Also optimal
CS0 = 0.0893; % Also optimal
P0  = 0;
% parameters
params     = FedBatchParameters(CX0,CS0,0,0,CX0,CS0); % Ks =
    ... Kw = 0
parameters = FedBatchParameters(CX0,CS0,0,0,CX0,CS0);
mustar = params(14);

% Solver inputs
x0      = [V0, CX0, CS0, P0];
tspan   = [0 1/mustar*log(Vmax/V0)];
% Initialize time vector according to step size
if(h==0)
    t = tspan;
else
    t = [tspan(1):h:tspan(end) tspan(end)];
end

% Tolerances
AbsTol = 1e-3;
RelTol = 1e-3;

% Create the pertubated parameter sets
interval = linspace(0.9,1.1,10);
gammas = params(6)*interval;
mumax  = params(7)*interval;
KS     = params(8)*interval;
KI     = params(9)*interval;

%% For every combination of the parameter values above
tic
for i = 1:length(interval);
for j = 1:length(interval);
for k = 1:length(interval);
for l = 1:length(interval);
% Update parameters
parameters(6:9) = [gammas(i),mumax(j),KS(k),KI(l)];

% Simulation
[~,x] = Solver(t,x0,AbsTol,RelTol,method,parameters);
```

```
end
end
end
end
% Save runtime
runtime = toc;
end
```

C.5 Parallel Simulations

Listing C.13: Implementation of parallel simulations in Matlab.

```
function runtime = FedBatchVaryParametersMPI(method,h)
% Calculates the production for various parameter combinations using the
% numerical method specified by 'method' and measures the runtime

%% INITIALIZATION
% Initial conditions / Optimal Values
V0 = 100; Vmax = 1200;
CX0 = 20; % Also optimal
CS0 = 0.0893; % Also optimal
P0 = 0;
% parameters
params = FedBatchParameters(CX0,CS0,0,0,CX0,CS0); % Ks = Kw = 0
mustar = params(14);

% Solver inputs
x0      = [V0, CX0, CS0, P0];
tspan   = [0 1/mustar*log(Vmax/V0)];
% Initialize time vector according to step size
if(h==0)
    t = tspan;
else
    t = [tspan(1):h:tspan(end) tspan(end)];
end

% Tolerances
AbsTol = 1e-3;
RelTol = 1e-3;

% Create the pertubated parameter sets
interval = linspace(0.9,1.1,10);
```

```
31 rho     = params(1);
32 alphas  = params(2);
33 alphaw  = params(3);
34 F       = params(4);
35 CSin    = params(5);
36 gammas  = params(6)*interval;
37 mumax   = params(7)*interval;
38 KS      = params(8)*interval;
39 KI      = params(9)*interval;
40 Ks      = params(10);
41 Kw      = params(11);
42
43 % restructure the parameters, such that they can be utilized
       ... by a parfor
44 [ggammas,gmumax,gKS,gKI] = ndgrid(gammas,mumax,KS,KI);
45 ggammas  = reshape(ggammas,1e4,1);
46 gmumax   = reshape(gmumax ,1e4,1);
47 gKS      = reshape(gKS    ,1e4,1);
48 gKI      = reshape(gKI    ,1e4,1);
49
50 %% For every combination of the parameter values above
51 tic
52 parfor i = 1:length(gKS);
53 % Update parameters
54 parameters = [rho,alphas,alphaw,F,CSin,ggammas(i),gmumax(i),
       ...gKS(i),gKI(i),Ks,Kw,CX0,CS0];
55
56 % Simulation
57 [~,x] = Solver(t,x0,AbsTol,RelTol,method,parameters);
58 end
59 % Save runtime
60 runtime = toc;
61 end
```

Bibliography

Jørgensen, John B. "Optimization of Fermenter Operation". Kgs. Lyngby, Denmark: DTU Compute - Department of Applied Mathematics and Computer Science - Technical University of Denmark, (2013). Print.

LeVeque, Randall J. "Finite Difference Methods for Ordinary and Partial Differential Equations". Seattle, Washington: SIAM, (2007). Book.

Nocedal, Jorge.; Wright, Stephen J. "Numerical Optimization", 2. edition. New York, USA: Springer, (2006). Book.

Völcker, Carsten. "Production Optimization of Oil Reservoirs". Kongens Lyngby, Denmark: Department of Informatics and Mathematical Modelling - Technical University of Denmark, (2011). PhD Thesis.

Eldén, Lars.; Wittmeyer-Koch, Linde.; Nielsen, Hans B. "Introduction to Numerical Computation - analysis and MATLAB®illustrations". Malmö, Sweden: Holmbergs, (2010). Book.

Jørgensen, John B.; Kristensen, Morten R.; Thomsen, Per G. "A Family of ESDIRK Integration Methods". Kgs. Lyngby, Denmark: Informatics and Mathematical Modelling - Technical University of Denmark. Paper.

Engsig-Karup, Allan P.; Thomsen, Per G. "Numerical Solution of Ordinary Differential Equations". Kgs. Lyngby, Denmark: Informatics and Mathematical Modelling - Technical University of Denmark, (2012). Lecture Notes.

Prinz, Peter; Crawford Tony. "C In A Nutshell - A Desktop Quick Reference", 1. edition. United States of America: O'Reilly Media, Inc., (2005). Book.

Netlib. "LAPACK". www.netlib.org, 20 June 2013. Internet: http://www.netlib.org/lapack/. Cached the 21 June 2013.

Dongarra, Jack; Snir, Marc; Otto, Steve; Huss-Lederman, Steven; Walker, David. "MPI: The Complete Reference". United States of America, (1996). Book.

Mathworks. "parfor". www.mathworks.se, 2013. Internet: http://www.mathworks.se/help/distcomp/parfor.html. Cached the 25 June 2013.