NTNU

TKP4555 Process-System Engineering Specialization Module

TKP11 Advanced Process Simulation
Handeling stiff ordinary differential equations (ODEs) using IMEX-methods in JULIA

Kjetil Bohman Sonerud
Department of Chemical Engineering
Norwegian University of Science and Technology

Supervisor:
Heinz A. Preisig
John C. Morud

Abstract

This work gives the reader an introduction to *stiff problems*, and their relevance to the numeric solution of differential equations that are central to science and engineering disciplines in general and chemical engineering in particular. The basic theory of differential equations, numerical methods for solving these and the concept of a stiff problem is introduced, and relevant examples are given.

The main focus is on IMEX methods, where the stiff part of the problem is solved with an *implicit* numerical method and the non-stiff part is solved with an *explicit* method. The test case chosen is the well-known Robertson kinetics problem (Robertson, 1966), which is known to be stiff. Both the original and an expanded version of the problem is solved with an explicit (EX) Euler solver, an implicit (IM) Euler solver and an IMEX Euler solver implemented in the programming language JULIA. The performance and merits of the different methods are contrasted and discussed.

For the current case, the IM Euler method outperforms the IMEX Euler method. This slightly surprising fact is likely due to non-optimized implementation of the latter, along with the fact that the non-stiff part of the Robertson problem is small. Even for the expanded problem – specifically chosen to enhance the relative performance of the IMEX solver – the IM Euler solver appears to be the fastest in terms of both number of Newton-Raphson iterations and CPU-time. Thus, the benefit of solving the non-stiff part with an EX method is apparently not sufficient to enhance the performance of the IMEX method enough to compete with the pure IM method. Both the IM Euler solver and the IMEX Euler solver clearly outperform the EX Euler solver, however – thus, it is clear that the problem is indeed stiff.

Contents

Chapter 1

Introduction

In general, many of the practical problems in science and engineering may be formulated as differential equations. In chemical engineering in particular, these are at the core of many – if not most – of the models employed. Thus, understanding how to correctly solve them is of great importance to the chemical engineer.

Systems of *ordinary differential equations* (ODEs) arise when considering *lumped systems*[1], or if the *partial differential equations* (PDEs) that arise from considering *distributed systems*[2]are approximated using finite difference schemes or similar approximations. As higher-order ODEs may be reduced to a system of first-order ordinary differential equations, the solution of the latter covers a wide range of applications, and is – in practise – often non-trivial. Since very few of the differential equations that are of practical interest may be solved analytically, approximating the solution through the use of *numerical methods* are often the only viable option. One of the central challenges that arises when solving such a system of first-order ODEs numerically are the ones related to *stiffness*. Handling this through the use of so-called IMEX *methods* (further discussed in chapter 3) is the main focus of the current work.

What is a *stiff* differential equation? Simply put, it is (a set of) differential equations that unfold on sufficiently different time scales. Typically – using chemical engineering as an example – this may be reactions with vastly different reaction rates (i.e. both fast and slow reactions occurring at the same time, to be further explored in chapter 4), or phenomena such as simultaneous diffusion and reaction – where one is assumed to be faster than the other. As will be further discussed in section 2.3, the concept of stiffness is largely a practical one – no single theoretical definition applies to all problems normally considered stiff.

> "If a numerical method is forced to use, in a certain interval of integration, a step length which is excessively small in relation to the smoothness of the exact solution in that interval, then the problem is said to be stiff in that interval."
> – (Lambert, 1991)

[1] An example is the assumption of *perfect mixing* in a tank such as an CSTR, so that there are no concentration gradients. The tank may then be regarded as a lumped system.

[2] An example is a PFR, where the concentration gradients in the axial direction seldom may be neglected. If the time evolution of the concentration gradients are of interest, a (set of) partial differential equation(s) must be solved.

It is an observed fact – to be further discussed in chapter 3 – that *implicit methods* normally outperform *explicit methods* when stiff problems arise. Using an explicit method to solve a stiff problem demands a prohibitively small time step as the problems grow sufficiently stiff. That being said, in theory an explicit method can solve all stiff problems as long as the time step chosen is small enough. That is, if time was not an issue, stiffness would not be an issue either. Thus, the latter is very much a practical problem, as noted above.

A natural question arises: if implicit methods can solve systems of equations where explicit methods fail, why are these methods not used all the time? The answer is simply that explicit methods – when they work, i.e. for sufficiently non-stiff systems – are computationally less demanding, thus much faster. As shall be seen in chapter 2, when using an implicit method there is (in general) a non-linear system of equations that must be solved at each time step using e.g. Newton-Raphson method or similar. This is computationally costly, especially for large systems. Thus, if possible, explicit methods are preferred.

In other words, there is always a trade-off between using explicit methods and implicit methods – the first is cheap for non-stiff systems, but due to the fact that very small time-steps must be used for stiff systems, there will come to a point when using the implicit solver is actually *cheaper*[3], even though a system of equations must be solved at each time step. The basic idea of an IMEX method is to exploit this fact and introduce a compromise, as shall be seen in section 2.4.

In the present work, the *explicit Euler scheme* (EX Euler, also called "forward Euler") and the *implicit Euler scheme* (IM Euler, also called "backward Euler") are the numerical solvers implemented. The two are combined into an IMEX Euler solver, as discussed in chapter 3. The implicit Euler and explicit Euler methods are largely chosen for *clearity* and *simplicity*

- They are simple to implement
- They are simple to analyse
- They illustrate the fundamental difference between implicit and explicit methods

The actual implementation of the methods discussed in this work is performed in the fairly recent programming language JULIA[4], which has been an interesting experience. For the purpose of the current project, there is little difference in terms of implementation between JULIA and MATLAB. The main practical challenge is lack of good tools for data visualization – that is, plotting in JULIA is not as convenient as in MATLAB for the time being.

A simple example of a stiff system of differential equations

The following simple example is included in order to introduce the concept of stiffness by comparing the analytical solution of the differential equation system with the numerical solution using EX Euler and

[3] Cheap in terms of computational time needed to solve the problem, measured in CPU-time

[4] The language was–and still is–being developed at MIT. Launched in February 2012, JULIA is a "(...) high-level, high-performance dynamic programming language for technical computing, with syntax that is familiar to users of other technical computing environments." Thus, users from Python or MATLAB environments should feel at home. To the user, JULIA behaves as an interpreted language, using a high-performance just-in-time (JIT) compiler for speed. It is claimed–supported by benchmarking tests–that the speed of JULIA approaches the speed of compiled languages such as Fortran and C for many tasks. See http://www.julialang.org for more information.

IM Euler with different time steps. It will become quite clear that the problem is indeed stiff, and that the EX Euler solver needs very small time steps compared with the IM Euler solver in order to provide a stable solution to the problem of the following second-order linear ordinary differential equation (ODE)

$$y'' + 101y' + 100y = 0 \tag{1.1}$$

Rewritten to a system of first-order ODEs with a set of proposed initial conditions, the following is obtained

$$\begin{aligned} y_1' &= \quad\quad\quad +y_2 \quad\quad\quad \text{Initial condition:} \quad y_1(0) = 1 \\ y_2' &= -100y_1 \quad -101y_2 \quad\quad \text{Initial condition:} \quad y_2(0) = 0 \end{aligned} \tag{1.2}$$

The analytic solution to the above system may be found as

$$y(t) = A\exp(-t) + B\exp(-100t) = \frac{100}{99}A\exp(-t) - \frac{1}{99}\exp(-100t) \tag{1.3}$$

where the values of A and B may be found from the initial conditions. It is apparent from the above that the second term will be a very rapid transient compared to the first term, which is visualized in fig. 1.1.

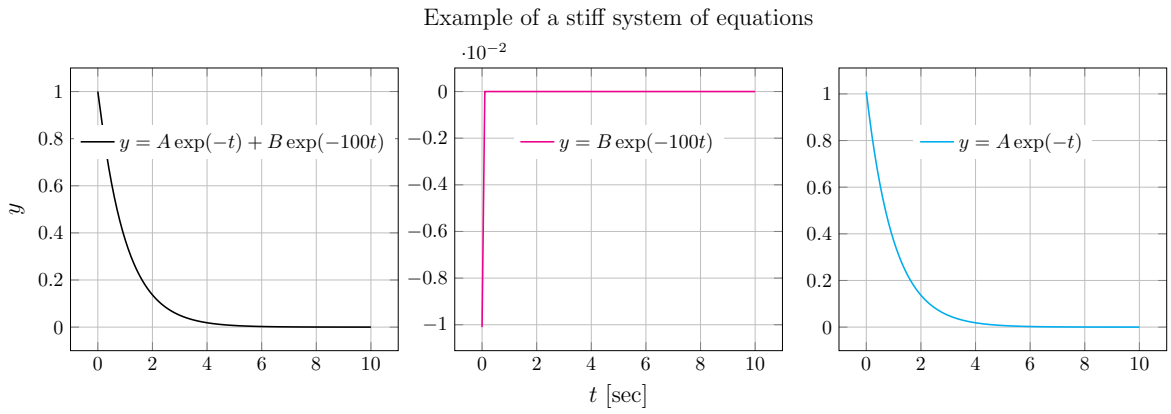

Figure 1.1: Visualization of the stiff ODE-system given in eq. (1.2). The exact solution is plotted, and each of the terms that constitutes this solution is then plotted separately. The rapid transient of the second term in the exact solution given in eq. (1.3) is evident.

The analytic solution to the eq. (1.2) is shown in fig. 1.2 along with the solution by the EX Euler solver (described later in section 3.1) and the IM Euler solver (described later in section 3.2). It is observed that the EX Euler solver fails spectacularly with a step size of $\Delta t = 1\,\text{s}$, and that there are obvious stability issues. The IM Euler solver on the other hand does a fair job of approximating the exact solution, even though there are only 10 time steps in total. When the step size is shortened considerably – by a factor of 10^3 – both methods approximate the exact solution quite well. The latter case is illustrated in fig. 1.3.

This simple example show a few interesting points when it comes to *stiff differential equations*:

- The step size must be very short for explicit methods to cope with sharp transients

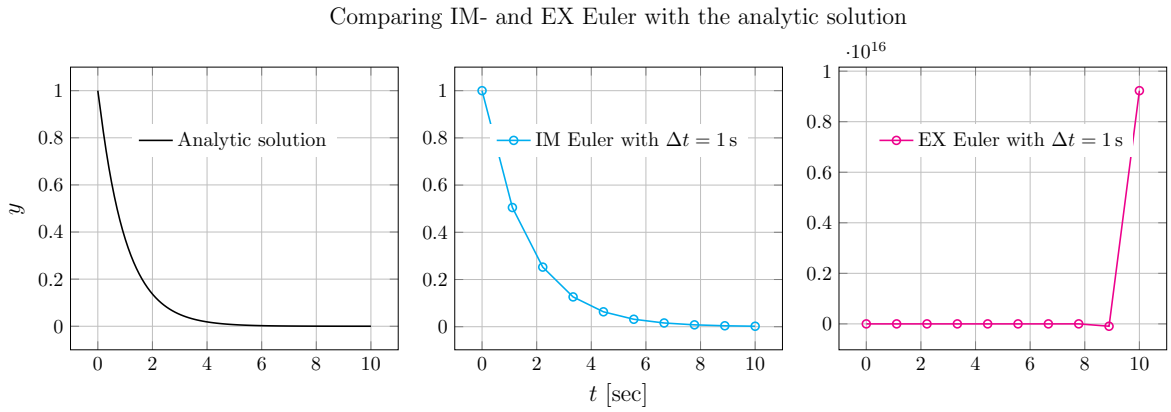

Figure 1.2: Comparison of the analytic solution to those of EX Euler and IM Euler method for the ODE-system described in eq. (1.2). The time step is $\Delta t = 1\,\mathrm{s}$. Note that the IM Euler solution is quite close to the analytical, but that the EX Euler method fails spectacularly.

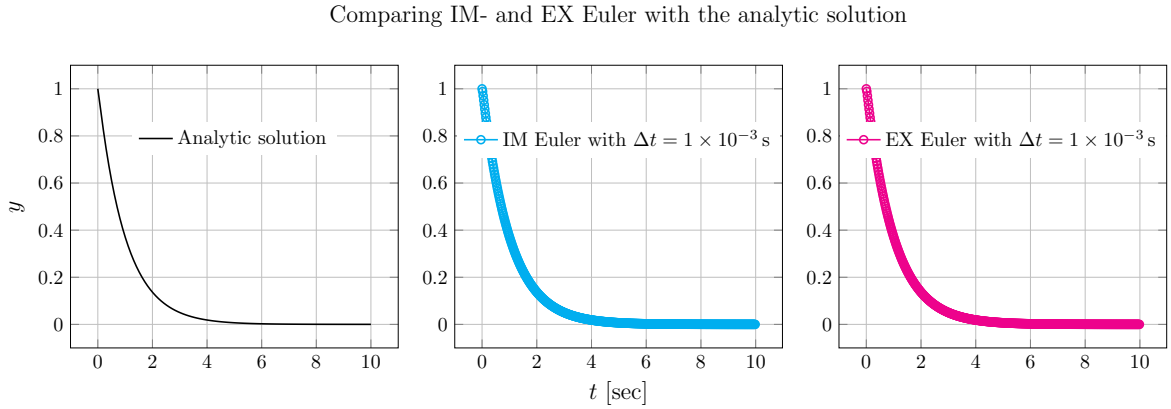

Figure 1.3: Comparison of the analytic solution to those of EX Euler and IM Euler method for the ODE-system described in eq. (1.2). The time step is $\Delta t = 1 \times 10^{-3}\,\mathrm{s}$. Now, both EX Euler and IM Euler are very close to the actual analytic solution.

- The implicit solver show no stability issues even when the step size is large – that is, fewer time steps in total compared to the explicit method
- Apparently, the accuracy of the explicit method is not the issue – rather, the *stability* of the method is the problem when the ODE system at hand is stiff

The latter point may be taken as a definition of stiff problems – stability, rather than accuracy – is governing the step size needed for the explicit method.

Chapter 2

Theory

The purpose of this chapter is to briefly cover the basic theory regarding ordinary differential equations (ODEs), with focus on initial value problems (IVP). The concept of *stiffness* will be further explored, and the strategy of using an IMEX solver to handle such systems efficiently will be introduced. Other strategies for handling stiff systems will be mentioned. For a more in-depth coverage of differential equations and their applications to chemical engineering, (Kreyszig, 2010), (Davis, 1984) and (Beers, 2007) is recommended literature.

2.1 IVP for a system of ODEs

Looking at a system of ODEs describing the behaviour of a set of variables in time, the central problem is calculating the *trajectories* of these variables over time. The solution sought is the n equations $y_j(t)$ describing the trajectories, given by the differential equation system

$$\frac{\mathrm{d}\underline{\boldsymbol{y}}}{\mathrm{d}t} = \dot{\underline{\boldsymbol{y}}}(t) = f(\underline{\boldsymbol{y}}(t); \boldsymbol{\Theta}), \quad \underline{\boldsymbol{y}}(t_0) = \underline{\boldsymbol{y}}_0 \tag{2.1}$$

where $\underline{\boldsymbol{y}}(t) = [y_1(t), y_2(t), \ldots, y_n(t)]^\top$ is the state vector and $\underline{\boldsymbol{y}}_0 = [y_1(0), y_2(0), \ldots, y_n(0)]^\top$ is the vector containing the initial values, i.e. the initial condition given at $t = t_0$. Usually, this is the same as $t = 0$, but this need not be the case in general. $\boldsymbol{\Theta} = [k_1, k_2, \ldots]^\top$ are possible parameters. Given that the integral of the function $f(\underline{\boldsymbol{y}}(t); \boldsymbol{\Theta})$ is defined, the exact solution to the above problem may be found by integration

$$\underline{\boldsymbol{y}}(t) = \underline{\boldsymbol{y}}_0 + \int_{t_0}^{t} f(\underline{\boldsymbol{y}}(\tau); \boldsymbol{\Theta})\, \mathrm{d}\tau \tag{2.2}$$

This approach is possible for some simple examples, but is not possible for a general system. Thus, numerical methods are needed. Note that the problem of calculating the numerical solution to an IVP of ODEs are closely related to the numeric evaluation of a definite integral. The general aim is to develop methods that produce an approximation that resembles the exact solution to within the chosen tolerance.

Note that while the variable of integration above is interpreted to be time, this could also be another

variable – typically a spatial coordinate. It is also true that higher-order ODEs may be reduced to a system of first-order ODEs. The same is true for *partial differential equations* (PDEs) – in many cases, it is possible to reduce a given PDE to a system of first-order ODEs using a finite difference scheme or similar approaches. This is discussed in the literature mentioned earlier, e.g. (Beers, 2007). Thus, the formulation given in eq. (2.1) is quite general, and covers a whole range of problems of interest to the chemical engineer.

2.2 Time-marching ODE-IVP solvers

The solver update $\underline{\boldsymbol{y}}(t)$ in discrete time steps of size Δt to compute $\underline{\boldsymbol{y}}(t_k)$. This length of the time step may in theory vary throughout the integration, but for the simple methods regarded in this work no time-step adjustment is considered. The exact update in each time step is

$$\underline{\boldsymbol{y}}(t_k + \Delta t) - \underline{\boldsymbol{y}}(t_k) = \int_{t_k}^{t_k+\Delta t} \dot{\underline{\boldsymbol{y}}}(\tau)\,\mathrm{d}\tau \tag{2.3}$$

$$= \int_{t_k}^{t_k+\Delta t} f(\underline{\boldsymbol{y}}(\tau); \boldsymbol{\Theta})\,\mathrm{d}\tau \tag{2.4}$$

Following the notation of (Beers, 2007), the approach is to approximate this exact calculation by a method that calculates

$$\underline{\boldsymbol{y}}_{k+1} - \underline{\boldsymbol{y}}_k = (\Delta t) \cdot \underbrace{F[\underline{\boldsymbol{y}}_k, \underline{\boldsymbol{y}}_{k+1}, f(\underline{\boldsymbol{y}}; \boldsymbol{\Theta})]}_{\text{Defines the numerical method}} \tag{2.5}$$

If it is reasonable to assume that $\underline{\boldsymbol{y}}_k = \underline{\boldsymbol{y}}(t_k)$, then the above scheme will allow calculation of $\underline{\boldsymbol{y}}_{k+1} \approx \underline{\boldsymbol{y}}(t_{k+1})$. It is evident that a *local error* is introduced at each time step due to the fact that eq. (2.3) is approximated by eq. (2.5). From the general definition in the latter equation, both the EX and IM Euler schemes may be realized, as discussed in chapter 3.

Note that the family of approximations described in the latter equation are *single-step integration methods*; only the information about the state vector at the beginning and end of the time step Δt – that is $\underline{\boldsymbol{y}}_k$ and $\underline{\boldsymbol{y}}_{k+1}$ – are used. Multi-step methods, as described in (Butcher, 2008), are often more effective and yield better approximations than simple single-step integration methods, but the latter are easier both to implement and to understand. Thus, multi-step methods is beyond the scope of the present work.

2.3 Stiff problems

In general, there is no agreed-upon mathematical definition that applies to all kinds of differential equations. For ODEs with constant coefficients one may look at the eigenvalues of the resulting Jacobian, but this alone is not sufficient for arbitrary non-linear differential equations. It seems that a pragmatic approach is needed, defining a stiff problem on the merits of the symptoms. Such an approach has been taken by many authors, among them Curtiss & Hirschfelder in their classic paper:

> "Stiff equations are equations where certain implicit methods, in particular BDF, perform better, usually tremendously better, than explicit ones."
> – (Curtiss and Hirschfelder, 1952)

It is important to note that it is not the particular differential equation that is stiff, even though the term is commonly used this way. The term stiffness must in the general sense be seen as a practical problem for certain numerical methods, thus the *conditions* that apply to the particular differential equation is important.

> "Although it is common to talk about *stiff differential equations*, an equation *per se* is not stiff. A particular initial value problem for that equation may be stiff, in some regions, but the sizes of these regions depend on the initial values *and* the error tolerance."
> – (Gear, 1982)

2.4 Implicit-explicit (IMEX) methods for ODE systems

In many practical applications, large systems of ODEs that contain both stiff and non-stiff elements must be solved. One approach is to use an explicit method with a short time step, another method is to solve the whole system using an implicit scheme. As shown in chapter 3, the latter ensures stability, but cost of solving large systems of equations at each time step – especially if no analytic Jacobian is available – is computationally costly. Thus, the basic premise of an IMEX solver is to provide a compromise – the stiff parts of the system are solved with an implicit solver to ensure stability, while the non-stiff parts are solved with an explicit solver to reduce computational load (Frank et al., 1997).

For a general ODE system, the idea may be formalized as

$$\dot{\underline{\boldsymbol{y}}}(t) = F(t, \underline{\boldsymbol{y}}) + G(t, \underline{\boldsymbol{y}}) \tag{2.6}$$

where $F(t, \underline{\boldsymbol{y}})$ represent the non-stiff part and $G(t, \underline{\boldsymbol{y}})$ represent the stiff part. Some combination of implicit and explicit numerical methods is then employed to solve the system, referred to as an IMEX method. Note that in general, it is assumed that *a priori* knowledge of the system is exploited to distinguish the stiff and non-stiff part so that the corresponding solver may be employed. In theory, automatic stiffness detection based on e.g. eigenvalue analysis may be implemented, and this approach may have some merit for many applications. However, as stated in section 2.3, a general and robust definition of stiffness is not found in the literature, and there is unlikely to ever be one due to the practical aspects of the problem. It should also be noted that in general, even though the individual explicit and implicit are stable, that does not guarantee the stability of the combined IMEX method (Frank et al., 1997). Thus, care must be taken when the scheme is constructed.

IMEX methods have been applied to a rather large variation of problems in the literature – from solving reaction-diffusion equations with applications in PEM fuel cells (Faragó et al., 2013) to applications in fast animation of physical objects such as cloth using particle systems (Eberhardt et al., 2000).

2.5 Alternative strategies for solving stiff problems

Most stiff problems arise from the large difference in *time scales* of the different trajectories in the solution. An example that will be further explored in chapter 4 is from reaction kinetics, where it is not uncommon to have different reactions present in the same reaction system that are several orders of magnitude apart in terms of reaction rates. Such a problem will naturally be stiff.

In traditional chemical engineering literature, the *pseudo steady state assumption* (PSSA) is often applied to such systems, as described in (Fogler, 2005, p. 379). The main idea is to assume that the reactive intermediates react as soon as they are formed – thus assuming that the net rate of formation is zero. Such an assumption may greatly simplify the equations in question, eliminating the stiff parts of the system. In general, the PSSA is a special case of *time scale analysis*, where the fast time scales may be assumed to be *event dynamic* and the slow time scales may be assumed to be *constant* from the viewpoint of the dynamic system in question. This is illustrated in fig. 2.1. By restricting the *dynamic window* to the scales of interest, the problem of stiffness may be eliminated. Note that this does require *a priori* knowledge of the system, but in many practical cases in chemical engineering, such knowledge is available.

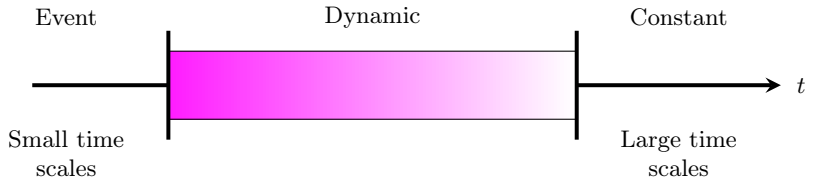

Figure 2.1: Illustration of time scales: the dynamic time scale is flanked to the left by the small time scales, which may be considered like *events*. To the right, the dynamic time scale is flanked by the large time scales, which may be considered *constant*. The illustration is adapted from (Preisig, 2013).

Mathematically, many such time scale problems may be handled within the framework of *singular perturbation*. By scaling the relevant equations, an *inner solution* (for the fast time scales) and an *outer solution* (for the slow time scales) may be found. The two may be combined to form the approximate solution to the original problem. Such an approach – where the solution for the fast trajectories are included in the solution – are particularly useful when the boundary conditions are important. Many flow systems are of this nature (Preisig, 2013).

Chapter 3

Numerical solution schemes

This chapter contain a short overview of the numerical solution schemes used in the current work; the explicit Euler scheme, the implicit Euler scheme and the implicit-explicit Euler scheme. A short section on the Newton-Rapshon method is included, as this is used in the implicit methods to solve the system of equations that arise in each time step.

3.1 Explicit (EX) Euler scheme

From the general formulation in eq. (2.5) given in chapter 2, the *explicit Euler scheme* (EX Euler, also known as "forward Euler") may be found as

$$F_{\text{FE}}[\underline{\boldsymbol{y}}_k, \underline{\boldsymbol{y}}_{k+1}, f(\underline{\boldsymbol{y}}; \boldsymbol{\Theta})] \triangleq f(\underline{\boldsymbol{y}}_k; \boldsymbol{\Theta})) \tag{3.1}$$

where it is evident that only the information about the former time step is used. The iteration scheme follows readily

$$\underline{\boldsymbol{y}}_{k+1} = \underline{\boldsymbol{y}}_k + (\Delta t) f(\underline{\boldsymbol{y}}_k; \boldsymbol{\Theta}) \tag{3.2}$$

It is clear from eq. (3.2) that the new state is given directly from knowledge of the old state and the right-hand side of the ODE system. Thus, the EX Euler method is usually fast, but may be inaccurate. Also, it suffers from *instability* for stiff systems, as shown below. Following (Davis, 1984), a simple single-variable test equation is investigated in order to provide a non-rigorous approach to the concept of stability

$$\frac{\mathrm{d}y}{\mathrm{d}t} = \lambda y, \quad y(0) = y_0 \tag{3.3}$$

When the EX Euler method of eq. (3.2) is applied to eq. (3.3), the following is obtained

$$\begin{aligned} y_{k+1} &= y_k + \Delta t \lambda y_k && (3.4)\\ &= y_k(1 + \Delta t \lambda) && (3.5)\\ &= y_0(1 + \Delta t \lambda)^{k+1} && (3.6) \end{aligned}$$

If the situation where an error in an earlier time step – or in the initial value – grows out of bound as $k \to \infty$ is to be avoided, it must be required that $|1 + \Delta t\lambda| \leq 1$. Assuming λ to be a real (non-complex) value and the fact that $\Delta t > 0$, it is clear that for the EX Euler method to be stable, it must be true that **a)** $\lambda < 0$ and **b)** that Δt is small if λ is large. More precisely, as $\lambda \to \infty$, $\Delta t \to 0$. In relation to the system investigated in chapter 1, it is clear that the (relatively) large value of $\lambda = -100$ results in the EX Euler method must take time steps at least of the order $\Delta t = 10^{-2}$.

3.2 Implicit (IM) Euler scheme

Starting again from the general formulation in eq. (2.5) given in chapter 2, the *implicit Euler scheme* (IM Euler, also known as "backward Euler") may be found as

$$F_{\text{BE}}[\underline{\boldsymbol{y}}_k, \underline{\boldsymbol{y}}_{k+1}, f(\underline{\boldsymbol{y}}; \boldsymbol{\Theta})] \triangleq f(\underline{\boldsymbol{y}}_{k+1}; \boldsymbol{\Theta})) \tag{3.7}$$

where the state of the next time step is used. The iteration scheme of the IM Euler method follows

$$\underline{\boldsymbol{y}}_{k+1} = \underline{\boldsymbol{y}}_k + (\Delta t) f(\underline{\boldsymbol{y}}_{k+1}; \boldsymbol{\Theta}) \tag{3.8}$$

Thus, the iterative scheme necessarily involves the solution of a (non-linear system of) equation(s) at each time step, which adds to the computational cost of the method. While implicit methods in general are more stable – as shown below – and require fewer time steps for equations with stiff parts, they are much more demanding than explicit methods. Thus, if the latter are applicable within a reasonable limit on the time step than must be taken, explicit methods are usually the first try when solving a differential equation numerically. This is true both for the simple EX Euler method and more sophisticated methods such as `ode45` in MATLAB – for non-stiff problems, the explicit method will usually be less costly than the corresponding implicit methods.

In the current work, a Newton-Raphson scheme is implemented to solve the resulting system of differential equations in each time step. The method is described briefly in section 3.3

The stability of the IM Euler method may be investigated similarly to the EX Euler method performed above. Starting from eq. (3.3) and using the IM Euler scheme, the following is obtained

$$y_{k+1} = y_k + \Delta t\lambda y_{k+1} \tag{3.9}$$

$$y_{k+1}(1 - \Delta t\lambda) = y_k \tag{3.10}$$

$$y_{k+1}(1 - \Delta t\lambda)^{k+1} = y_0 \tag{3.11}$$

$$y_{k+1} = \frac{y_0}{(1 - \Delta t\lambda)^{k+1}} \tag{3.12}$$

As long as $\lambda < 0$, it is evident that the IM Euler method is stable for all Δt. Due to this property, it is said to be *unconditionally stable*. It is now evident why the solution of the example system in chapter 1 using the IM Euler method was superior to that of the EX Euler method; using the former, the choice of Δt is only a matter of accuracy, not stability. For the latter, the choice of Δt must be made to ensure stability – disregarding the question of accuracy.

3.3 Newton-Raphson method

The Newton-Raphson method is an iteration method for finding an approximate solution to the root(s) of a (system of) non-linear equations. The method may be derived from a Taylor series expansion[1] for a multi-variable function, cut off after the first term

$$f(\underline{\boldsymbol{x}} + \Delta\underline{\boldsymbol{x}}) \approx f(\underline{\boldsymbol{x}}) + \underline{\underline{\boldsymbol{J}}}(\underline{\boldsymbol{x}})\Delta\underline{\boldsymbol{x}} \tag{3.13}$$

$$0 \approx f(\underline{\boldsymbol{x}}) + \underline{\underline{\boldsymbol{J}}}(\underline{\boldsymbol{x}})\Delta\underline{\boldsymbol{x}} \tag{3.14}$$

$$\implies \quad \Delta\underline{\boldsymbol{x}} \approx -\underline{\underline{\boldsymbol{J}}}(\underline{\boldsymbol{x}})^{-1} f(\underline{\boldsymbol{x}}) \tag{3.15}$$

where $\underline{\underline{\boldsymbol{J}}}(\underline{\boldsymbol{x}})^{-1}$ is the inverse of the *Jacobi matrix* of $f(\underline{\boldsymbol{x}})$, the latter being defined as

$$\underline{\underline{\boldsymbol{J}}}(\underline{\boldsymbol{x}}) = \begin{bmatrix} \left(\frac{\partial f_1}{\partial x_1}\right)_{x_{i\neq 1}} & \left(\frac{\partial f_1}{\partial x_2}\right)_{x_{i\neq 2}} & \cdots & \left(\frac{\partial f}{\partial x_n}\right)_{x_{i\neq n}} \\ \left(\frac{\partial f_2}{\partial x_1}\right)_{x_{i\neq 1}} & \left(\frac{\partial f_2}{\partial x_2}\right)_{x_{i\neq 2}} & \cdots & \left(\frac{\partial f_2}{\partial x_n}\right)_{x_{i\neq n}} \\ \vdots & \vdots & \ddots & \vdots \\ \left(\frac{\partial f_n}{\partial x_1}\right)_{x_{i\neq 1}} & \left(\frac{\partial f_n}{\partial x_2}\right)_{x_{i\neq 2}} & \cdots & \left(\frac{\partial f_n}{\partial x_n}\right)_{x_{i\neq n}} \end{bmatrix} \tag{3.16}$$

Usually, it is preferred to solve the system of linear equations written as

$$\underline{\underline{\boldsymbol{J}}}(\underline{\boldsymbol{x}}_k)(\underline{\boldsymbol{x}}_{k+1} - \underline{\boldsymbol{x}}_k) = -f(\underline{\boldsymbol{x}}_k) \tag{3.17}$$

instead of performing the actual inversion of the Jacobi matrix. The technique is iterative, such that $\underline{\boldsymbol{x}}_{k+1}$ is used as a starting point in the next iteration loop, finding the approximative solution $\underline{\boldsymbol{x}}_{k+2}$. The iteration is continued until the approximative root is sufficiently close to zero, usually governed by a tolerance that is set *a priori* by the user of the scheme based on the practical application at hand.

The Newton-Raphson method is a second-order method, which can be seen by analysis of the neglected terms in the Taylor series (Kreyszig, 2010, p. 804). Due to this fact, the precision of the approximation is doubled at every iteration sufficiently close to the solution. This means that the number of iterations needed for the Newton-Raphson method to converge is often quite low. However, the method is quite sensitive to the initial guess – that is, the first approximation to the solution must be sufficiently close to the actual solution. In practise, this may prove troublesome depending on what is known about the solution before the iteration loop is initiated.

[1] An interesting point is that both the EX Euler method and the IM Euler method may similarly be derived by using the Taylor series as a starting point.

3.4 Implicit-explicit (IMEX) Euler scheme

Assuming that the system of ODEs are readily split into an implicit part and an explicit part as described in chapter 2, the IMEX Euler scheme may be described as follows

$$\begin{bmatrix} \underline{\boldsymbol{y}}^{\mathrm{IM}}_{k+1} \\ \underline{\boldsymbol{y}}^{\mathrm{EX}}_{k+1} \end{bmatrix} = \begin{bmatrix} \underline{\boldsymbol{y}}^{\mathrm{IM}}_{k} \\ \underline{\boldsymbol{y}}^{\mathrm{EX}}_{k} \end{bmatrix} + \Delta t \begin{bmatrix} f(\underline{\boldsymbol{y}}^{\mathrm{IM}}_{k+1}, \underline{\boldsymbol{y}}^{\mathrm{EX}}_{k+1}; \boldsymbol{\Theta}) \\ f(\underline{\boldsymbol{y}}^{\mathrm{EX}}_{k}; \boldsymbol{\Theta}) \end{bmatrix} \tag{3.18}$$

where $\underline{\boldsymbol{y}}^{\mathrm{IM}}$ is taken to be the stiff part of $\underline{\boldsymbol{y}}$ – that is, the part that will be solved implicitly – and $\underline{\boldsymbol{y}}^{\mathrm{EX}}$ is taken to be the non-stiff part of $\underline{\boldsymbol{y}}$ – that is, the part that will be solved explicitly. Note that to solve for $\underline{\boldsymbol{y}}^{\mathrm{IM}}_{k+1}$, it is necessary to first solve for $\underline{\boldsymbol{y}}^{\mathrm{EX}}_{k+1}$, as the latter is used in the solution of the former. Thus, in the Newton-Raphson iteration to solve for $\underline{\boldsymbol{y}}^{\mathrm{IM}}_{k+1}$, it is assumed that $\underline{\boldsymbol{y}}^{\mathrm{EX}}_{k+1}$ is constant during the course of the iterations. This is an approximation, and may explain why the IM method outperforms the IMEX method even though the latter – in theory – should provide a computational advantage. This is further discussed in chapter 5.

The IMEX method described in eq. (3.18) is implemented in JULIA as shown in listing 4. Note that the practical difficulties in terms of implementation of the method – more specifically, the difficulty of finding a general method for passing the implicit and explicit parts from the top-level in order to split the calculations into an implicit and explicit part on lower levels – are evident in the code in chapter A, where a separate Newton-Raphson solver is included for the IMEX problem. A further generalization should be possible, but was not implemented for the current work.

Chapter 4

Model

> "When the equations represent the behaviour of a system containing a number of fast and slow reactions, a forward integration of these equations becomes difficult."
> – (Robertson, 1966)

The following example describing the kinetics of an autocatalytic reaction was proposed by (Robertson, 1966), and has become popular in the numerical studies of stiff equation systems (Hairer and Wanner, 1996; Butcher, 2008). For the short time scales, an investigation of the system may be found in (Gobbert, 1996). The original problem as stated by Robertson is

$$\mathrm{A} \xrightarrow{0.04} \mathrm{B} \qquad \text{(slow)} \tag{4.1a}$$

$$\mathrm{B} + \mathrm{B} \xrightarrow{3\cdot 10^7} \mathrm{C} + \mathrm{B} \qquad \text{(very fast)} \tag{4.1b}$$

$$\mathrm{B} + \mathrm{C} \xrightarrow{1\cdot 10^4} \mathrm{A} + \mathrm{C} \qquad \text{(fast)} \tag{4.1c}$$

where it is noted that eq. (4.1b) is several magnitudes larger than eq. (4.1c), which again is several magnitudes larger than the slow reaction in eq. (4.1a). From this, it is reasonable to predict that the system of equations will be stiff.[1]

Three different versions of the Robertson problem are proposed. The first is a scaled version of the original problem, modified due to the fact that the rather simple methods employed in this work are expected to have serious trouble with the proposed reaction constants. Thus, these are scaled to reduce the stiffness but at the same time maintain the original problem qualitatively.

$$\begin{aligned}
&\text{A:} \quad y_1' = -0.1y_1 + 100y_2y_3 && \text{Initial condition:} \quad y_1(0) = 1 \\
&\text{B:} \quad y_2' = +0.1y_1 - 100y_2y_3 - 10^4y_2^2 && \text{Initial condition:} \quad y_2(0) = 0 \\
&\text{C:} \quad y_3' = +10^4y_2^2 && \text{Initial condition:} \quad y_3(0) = 0
\end{aligned} \tag{4.2}$$

The second version of the problem is the original problem proposed by Robertson, shown in eq. (4.3).

[1] As in interesting side note, many commercial solvers – both explicit and implicit – will fail to integrate this system correctly on the very long time scales (e.g. 1×10^{10} s) (Hairer and Wanner, 1996, p. 144).

The rate constants are the same as in eq. (4.1).

$$\begin{aligned}
&\text{A:} && y_1' = && -0.04y_1 && +10^4 y_2 y_3 && && \text{Initial condition:} && y_1(0) = 1 \\
&\text{B:} && y_2' = && +0.04y_1 && -10^4 y_2 y_3 && -3 \cdot 10^7 y_2^2 && \text{Initial condition:} && y_2(0) = 0 \\
&\text{C:} && y_3' = && && && +3 \cdot 10^7 y_2^2 && \text{Initial condition:} && y_3(0) = 0
\end{aligned} \tag{4.3}$$

The last system is an expanded version of eq. (4.3), made to promote solution by the IMEX method central to this work. The idea behind the reaction scheme shown in eq. (4.4) is to expand the non-stiff parts of the system, thus promoting the effectiveness of the IMEX solver. Here, eq. (4.4d) and eq. (4.4e) represent the two reaction steps that are fast – the rest of the steps are relatively slow in comparison, and it is assumed that they may be solved efficiently with an explicit method.

As it turns out, this approach does not change the fact that the IM method actually outperforms the IMEX method even for this system, which is further discussed in chapter 5. The translation from the reaction scheme into an actual system of equations is similar to the one showed in eq. (4.3), and is omitted. The resulting system of first-order differential equations is 7×7, as is evident from eq. (4.4).

$$\mathrm{A}^{\star\star} \xrightarrow{0.03} \mathrm{A}^{\star} \qquad \text{(slow)} \tag{4.4a}$$

$$\mathrm{A}^{\star} \xrightarrow{0.05} \mathrm{A} \qquad \text{(slow)} \tag{4.4b}$$

$$\mathrm{A} \xrightarrow{0.04} \mathrm{B} \qquad \text{(slow)} \tag{4.4c}$$

$$\mathrm{B} + \mathrm{B} \xrightarrow{3 \cdot 10^7} \mathrm{C} + \mathrm{B} \qquad \text{(very fast)} \tag{4.4d}$$

$$\mathrm{B} + \mathrm{C} \xrightarrow{1 \cdot 10^4} \mathrm{A} + \mathrm{C} \qquad \text{(fast)} \tag{4.4e}$$

$$\mathrm{C} \xrightarrow{0.05} \mathrm{C}^{\star} \qquad \text{(slow)} \tag{4.4f}$$

$$\mathrm{C}^{\star} \xrightarrow{0.04} \mathrm{C}^{\star\star} \qquad \text{(slow)} \tag{4.4g}$$

Chapter 5

Results and discussion

The results using the three different models described in chapter 4 are shown and discussed below. In all cases, it is evident that the proposed problem is indeed stiff, and that the EX Euler solver shows the expected instability unless the time step chosen is very small. For purposed of comparing CPU-time of the different examples, all calculations were performed using a MacBook Pro with a 2,53 GHz Intel Core 2 Duo processor. It should also be noted that to avoid actual profiling of the JULIA JIT-compiler, all functions were compiled once before the performance was measured. Thus, the actual performance is presumed to be the measured variable.

5.1 Scaled Robertson problem

The results from solving the scaled Robertson problem are below. It is evident that for the given step length, the EX Euler solver shown in fig. 5.1 is having severe difficulties with predicting the trajectory of the rapidly changing B component. While the trend predicted is correct, and the instability dies out over time, the illustration is quite striking. The same problem solved with the IM Euler using a Newton-Raphson iteration scheme to solve the system of non-linear equations as discussed in chapter 3 is shown in fig. 5.2.

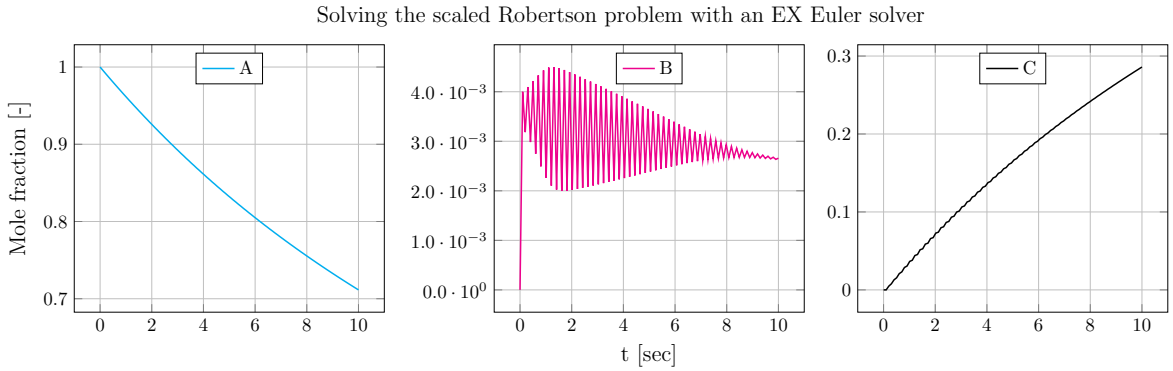

Figure 5.1: Solving the Robertson kinetics problem – scaled version – with EX Euler. The stability problem when calculating the B-trajectory are evident. Is is also observed minor instability when calculating the C-trajectory. The step size in this case is $\Delta t = 0.1\,\mathrm{s}$.

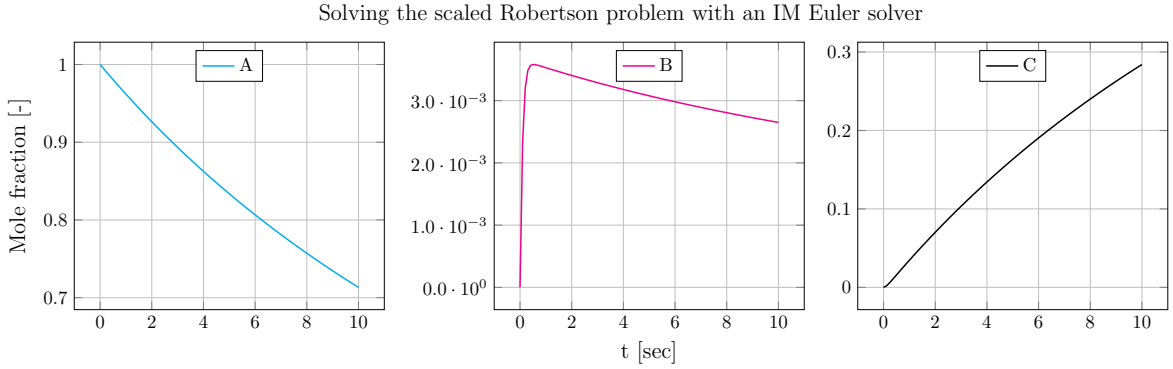

Figure 5.2: Solving the Robertson kinetics problem – scaled version – with IM Euler. The stability problem when calculating the B-trajectory are not present. The step size in this case is $\Delta t = 0.1\,\mathrm{s}$.

5.2 Original Robertson problem

The results from using the same reaction system and the same values for the rate constants as proposed by (Robertson, 1966) and described in chapter 4 are shown in the figures below, and summarized in table 5.1. Note that the time step for the EX Euler solver is chosen so that instability is displayed at the end of the time range, illustrated in fig. 5.4 by zooming in on the instability. The results when solving the problem using IM Euler and IMEX Euler methods are in good comparison with (Gobbert, 1996), and are assumed to be correct. The latter solution is not shown, as it is identical to the one obtained by using the IM Euler solver.

Table 5.1: Original Robertson kinetics problem: key performance data using different numerical methods

Case	Δt	Number of time steps	CPU-time	Total number of N-R iterations
EX Euler (0 s to 40 s)	6×10^{-4} s	66667	0.161 s	–
IM Euler (0 s to 40 s)	1 s	40	0.0239 s	131
IMEX Euler (0 s to 40 s)	1 s	40	0.122 s	131
EX Euler (0 s to 1000 s)	3×10^{-4} s	3333333	7.789 s	–
IM Euler (0 s to 1000 s)	1 s	1000	0.0612 s	2138
IMEX Euler (0 s to 1000 s)	1 s	1000	0.0804 s	2123

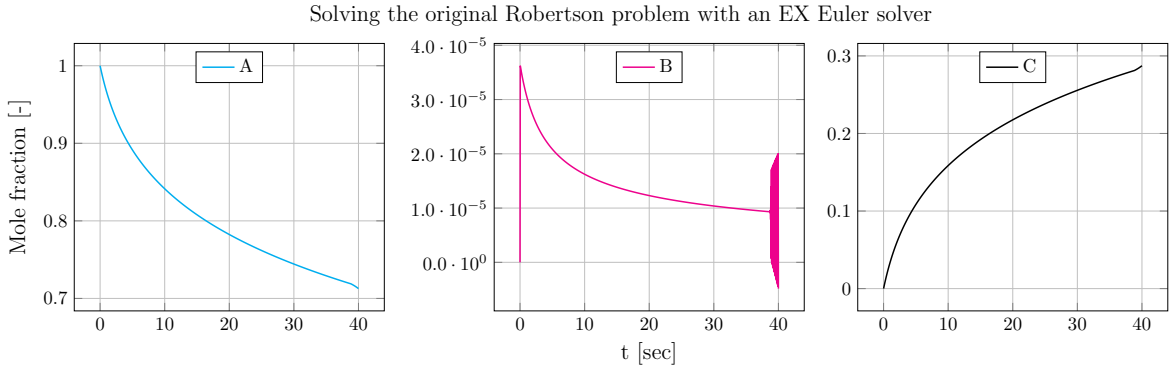

Figure 5.3: Solving the Robertson kinetics problem – original version – with EX Euler. The emerging stability problem when calculating the B-trajectory are evident at the end of the chosen time interval. The step size in this case is $\Delta t = 6 \times 10^{-4}$ s.

To further illustrate the difference in performance between the IM and EX Euler solvers, the simulation is run for 1000 s. The computational results are shown in table 5.1. The step size of the EX Euler solver is still forced to be small in order to avoid stability issues. As the running time of the IM solver is largely governed by the number of N-R iterations necessary – the latter is illustrated in fig. 5.9– the cost of increasing the simulation interval is not as large for the IM solver.

The results from using an IMEX Euler solver to solve the original Robertson problem are shown in

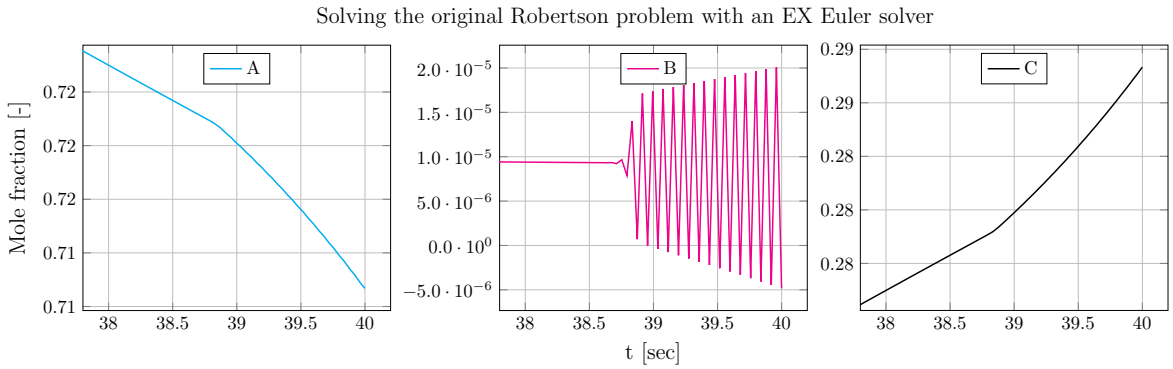

Figure 5.4: Solving the Robertson kinetics problem – original version – with EX Euler. The emerging stability problem when calculating the B-trajectory are evident at the end of the chosen time interval, and are highlighted in this figure. The step size in this case is $\Delta t = 6 \times 10^{-4}\,\mathrm{s}$.

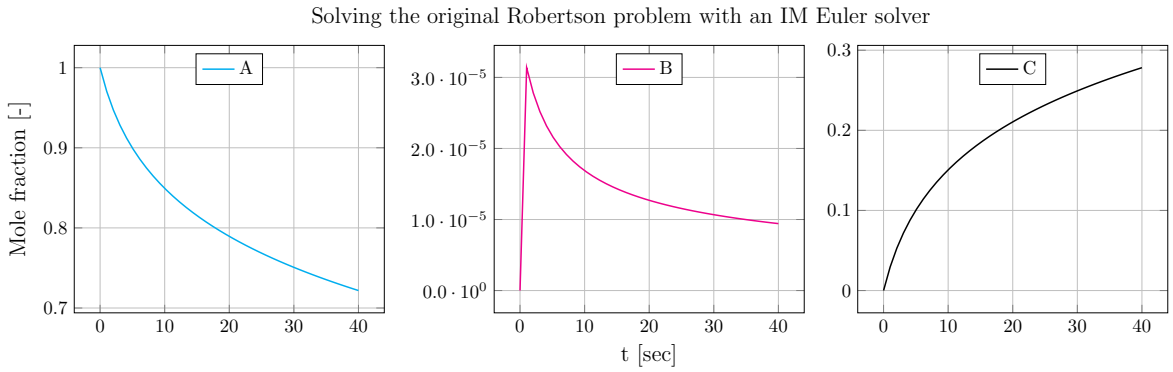

Figure 5.5: Solving the Robertson kinetics problem – original version – with IM Euler. The stability problem when calculating the B-trajectory is not present. The step size in this case is $\Delta t = 1\,\mathrm{s}$.

table 5.1. It is assumed – both by trial-and-error and by time scale assumptions – that the trajectory of B and C should be found using an implicit scheme. In theory, this will save time as a smaller non-linear system of equations (2×2 rather than 3×3) is solved in each time step. The stability issues of the EX Euler solver are no longer present, thus large time steps are permitted. The concentration of A – the relatively slow-changing component – is not included in the N-R iterations.

The plot produced by the IMEX method is identical to the plot produced by the IM Euler method, and is omitted. The performance of this solver seems to be fairly decent. No stability issues are detected, even when using a fairly large time step of $\Delta t = 1\,\mathrm{s}$. As will be further discussed, it is noted from table 5.1 that the computational time needed for the IMEX method is larger than for the IM method applied to the same problem with the same step length.

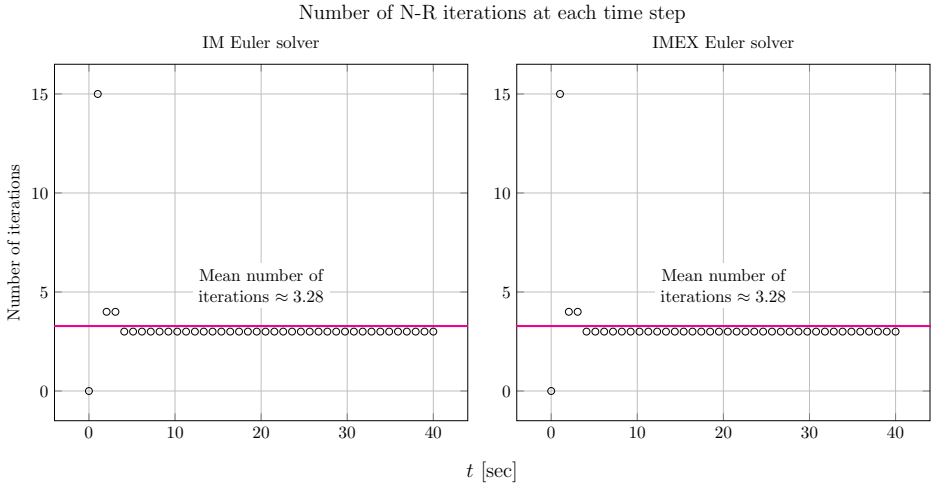

Figure 5.6: The number of iterations needed by the Newton-Raphson solver at each time step when solving the original Robertson kinetic problem using an IM Euler and IMEX Euler solver. Note that while the maximum number of iterations needed is 15, the average number of iterations needed is close to 3. It is evident that the number of iterations needed closely follows the sharp B-transient discussed earlier.

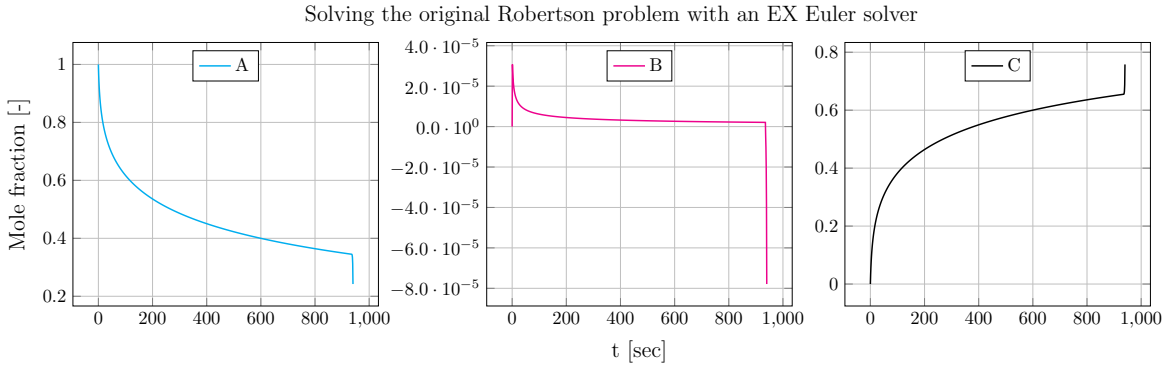

Figure 5.7: Solving the Robertson kinetics problem – original version – with EX Euler for the time interval 0 s to 1000 s. The emerging stability problem when calculating the B-trajectory are evident at the end of the chosen time interval. The step size in this case is $\Delta t = 3 \times 10^{-4}$ s.

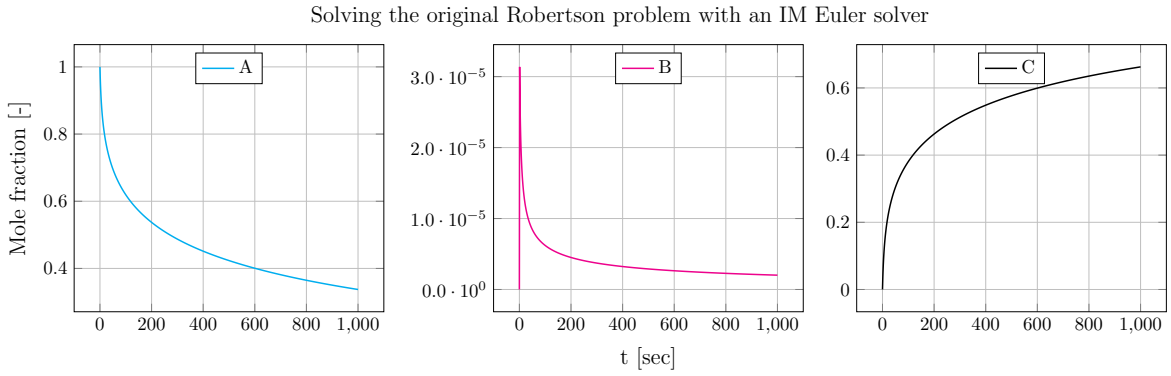

Figure 5.8: Solving the Robertson kinetics problem – original version – with IM Euler for the time interval 0 s to 1000 s. The stability problem when calculating the B-trajectory is not present. The step size in this case is $\Delta t = 1$ s.

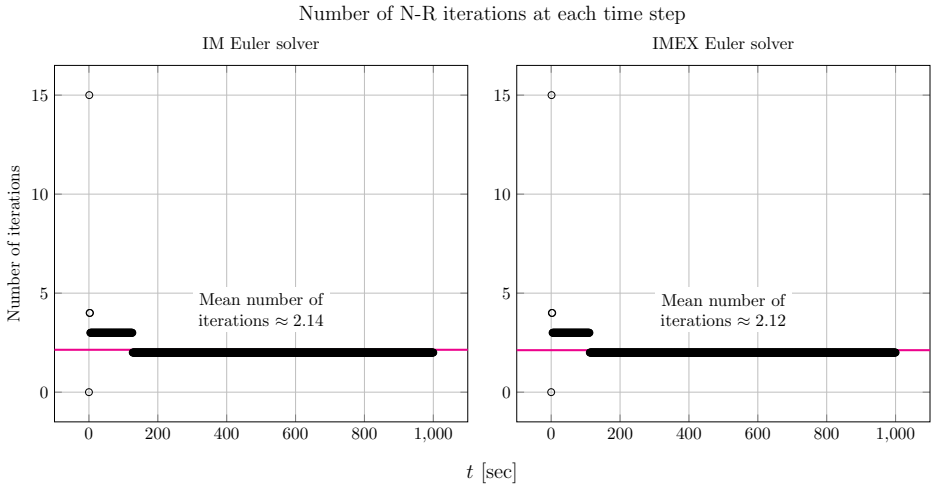

Figure 5.9: The number of iterations needed by the Newton-Raphson solver at each time step when solving the original Robertson kinetic problem using IM Euler and IMEX Euler for the time interval 0 s to 1000 s. Note that while the maximum number of iterations needed is 15, the average number of iterations needed is around 2. It is evident that the number of iterations needed closely follows the sharp B-transient discussed earlier, so that for most of the interval, only a few iterations are needed for convergence.

5.3 Expanded Robertson kinetic problem

The results from solving the expanded Robertson kinetics problem proposed in chapter 4 are shown below. The central idea behind the scheme is to expand the non-stiff part of the equation system, thus making the system more suitable to be solved using an IMEX Euler method. Note the difference in step size of the methods. The EX Euler method requires a small step size of 1×10^{-3} s to solve the problem, resulting in 600 000 steps in total. Note that this step size is found by trial-and-error – it is not evident from the result shown in fig. 5.10, but the step size is just small enough to avoid instability in the result. The IM and IMEX Euler methods, on the other hand, require only 1000 steps in total (thus, a step length of 1 s is used). Both of the latter methods are considerably faster than the EX method for this system. The results are summarized in table 5.2.

Table 5.2: Expanded Robertson kinetics problem: key performance data using different numerical methods

Case	Δt	Number of time steps	CPU-time	Total number of N-R iterations
EX Euler (0 s to 600 s)	1×10^{-3} s	600000	2.160 s	–
IM Euler (0 s to 600 s)	1 s	600	0.0533 s	1471
IMEX Euler (0 s to 600 s)	1 s	600	0.143 s	2506

The interesting – and rather surprising – observation that is made from table 5.2 is that the IM Euler method is actually *faster* than the IMEX Euler method. This is evident from both the table and from fig. 5.13, where the number of Newton-Raphson iterations needed at each time step is plotted. The IM Euler solver consistently need fewer steps than the corresponding IMEX method.

The reason for the latter observation is presumably that for relatively small systems of equations, the cost of actually solving the system of equations is not high due to highly optimized BLAS-routines in the JULIA standard library. Thus, unless the system is large, the added benefit of solving *all* the equations implicitly is more pronounced than the trouble of solving the larger system. It is expected that the added accuracy – thus fewer iterations needed at each time step – when solving the complete system implicitly is the decisive factor.

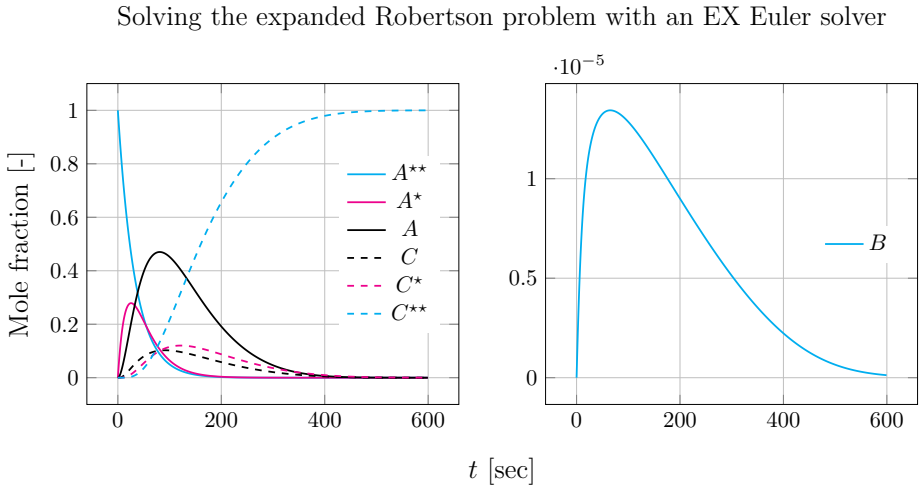

Figure 5.10: Solving the Robertson kinetics problem – expanded version – with an EX Euler for the time interval 0 s to 600 s. The step size in this case is $\Delta t = 1 \times 10^{-3}$ s.

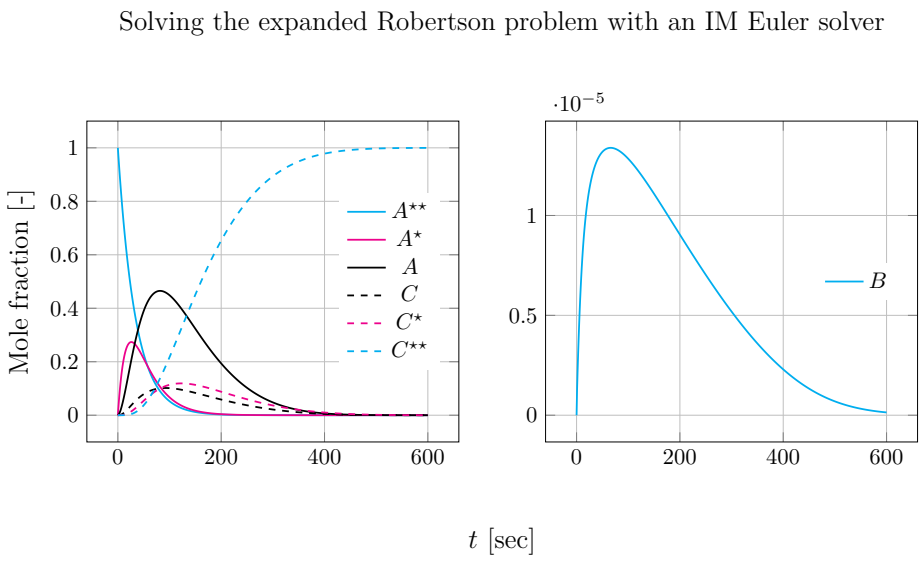

Figure 5.11: Solving the Robertson kinetics problem – expanded version – with an IM Euler for the time interval 0 s to 600 s. The step size in this case is $\Delta t = 1$ s.

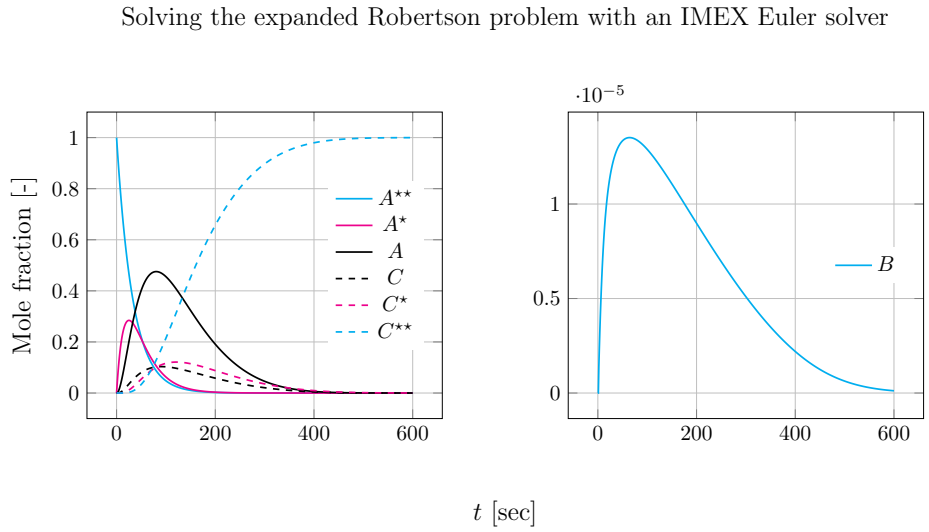

Figure 5.12: Solving the Robertson kinetics problem – expanded version – with an IMEX Euler for the time interval 0 s to 600 s. The step size in this case is $\Delta t = 1$ s.

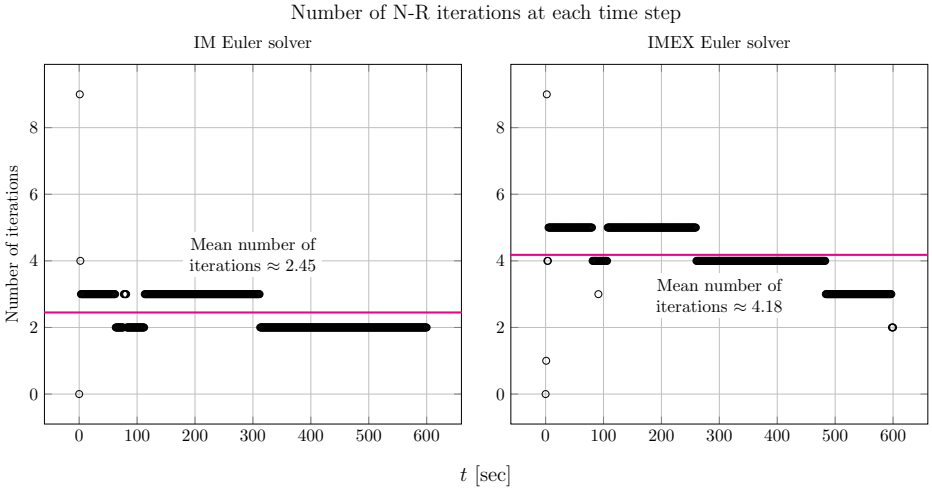

Figure 5.13: The number of iterations needed by the Newton-Raphson solver for an IM and IMEX Euler solver, respectively, at each time step when solving the expanded Robertson kinetic problem on the time interval 0 s to 600 s. Note that the IMEX method in general need more N-R iterations than the IM Euler solver.

Chapter 6

Conclusion and further work

Three numerical methods – the EX Euler method, IM Euler method and the IMEX Euler method for solving IVP for systems of ODEs have been implemented in JULIA. All three methods correctly solve the proposed models described in chapter 4, adapted from (Robertson, 1966), given the the step size is sufficiently small for the EX Euler method. In terms of computational time, both the IM Euler method and the IMEX Euler method outperform the EX Euler method, as is expected due to the step-size restrictions imposed by the stiff problem.

It is more notable that the IM Euler method outperforms the IMEX Euler method, even for an expanded equation system specifically proposed to promote the efficient solution using the IMEX Euler method. The hypothesis is that for a small system of differential equations – such as 7×7 as for the expanded Robertson kinetic model – the added cost of computing the *whole differential equation system implicitly* rather than using an explicit solver for the non-stiff parts is not very high when using the highly-optimized BLAS routines in JULIA. Rather, this saves computational time as a more accurate solution of the resulting system of equations is permitted, thus resulting in a lower number of total Newton-Raphson iterations compared to the IMEX Euler method.

There are many viable options for further work:

- The IMEX implementation should be made completely general, if possible. Currently, some modifications – i.e. separate Newton-Raphson solver for the IMEX solver – are needed
- The IMEX method should be optimized, making the computations more efficient
- The IMEX method should be tested for larger equations systems, preferably ones with a large non-stiff component (i.e. many non-stiff equations) and a small stiff component (i.e. a few stiff equations). It should be investigated whether this promotes the relative performance of the IMEX method.

Bibliography

Beers, K. J. (2007), *Numerical methods for chemical engineering: applications in MATLAB*, Cambridge University Press.

Butcher, J. (2008), *Numerical methods for ordinary differential equations*, Wiley.

Curtiss, C. F. and Hirschfelder, J. O. (1952), 'Integration of stiff equations', *Proceedings of the National Academy of Sciences* **38**, 235–243.

Davis, M. E. (1984), *Numerical methods and modeling for chemical engineers*, New York : Wiley.

Eberhardt, B., Etzmuß, O. and Hauth, M. (2000), Implicit-explicit schemes for fast animation with particle systems, *in* 'In Eurographics Computer Animation and Simulation Workshop', Springer-Verlag, pp. 137–151.

Faragó, I., Izsák, F., Szabó, T. and Kriston, A. (2013), 'An imex scheme for reaction-diffusion equations: application for a pem fuel cell model', *Central European Journal of Mathematics* **11**(4), 746–759.

Fogler, H. S. (2005), *Elements of chemical reaction engineering*, Prentice-Hall.

Frank, J., Hundsdorfer, W. and Verwer, J. G. (1997), 'On the stability of implicit-explicit linear multistep methods', *Applied Numerical Mathematics* **25**, 193 – 205.

Gear, C. (1982), Automatic detection and treatment of oscillatory and/or stiff ordinary differential equations, *in* J. Hinze, ed., 'Numerical Integration of Differential Equations and Large Linear Systems', Vol. 968 of *Lecture Notes in Mathematics*, Springer Berlin Heidelberg, pp. 190–206.

Gobbert, M. K. (1996), Robertson's example for stiff differential equations, Technical report, Arizona State University.

Hairer, E. and Wanner, G. (1996), *Solving Ordinary Differential Equations II : Stiff and Differential-Algebraic Problems*, Springer-Verlag Berlin Heidelberg.

Kreyszig, E. (2010), *Advanced Engineering Mathematics*, John Wiley & Sons.

Lambert, J. D. (1991), *Numerical methods for ordinary differential systems: the initial value problem*, Wiley.

Preisig, H. A. (2013), *Lecture notes in Chemical Process Systems Engineering.* **Updated:** 18.04.2013.

Robertson, H. H. (1966), The solution of a set of reaction rate equations, *in* J. Walsh, ed., 'Numerical Analysis: An Introduction', Academic Press London, p. 178–182.

Appendix A

Julia Code

The main JULIA source code is included for completeness. The IMEX method did not lend itself easily to generalization, so a rather elaborate scheme had to be coded along with a specific Newton-Raphson solver. Thus, the IMEX code for the expanded Robertson kinetics problem from chapter 4 is included, along with its own Newton-Raphson solver.

A.1 EX Euler solver

General explicit (EX) Euler solver implemented in Julia.

Julia code 1: `exEuler.jl`

```
################################################################################
# Explicit Euler solver (EX)
#
# Generalized explicit Euler solver (EX) for systems of equations.
#
# Input:
#   - f0:       [f1(0), f2(0), .... , fn(0)]
#   - time:     [t_start, t_end]
#   - dt:       Size of time step
#   - df:       Function that returns the derivatives of f, given as df(f(t),t)
#   - k_par:    Parameter vector
#
# Output:
#   - f:        [f1(t_start) f1(t_start+dt) f1(t_start+2dt) ...  f1(t_end),
#                f2(t_start) f2(t_start+dt) f2(t_start+2dt) ...  f2(t_end),
#
#                                          ......
#
#                fn(t_start) fn(t_start+dt) fn(t_start+2dt) ...  fn(t_end)]
#
#
#   - t:        [t_start, t_start+dt, t_start+2dt, ... , t_end]
#
```

```
24 # Author:     Kjetil Sonerud
25 # Updated:    2014-12-04 17:51:22
26 ################################################################################
27
28 function exEuler(f0, time, dt, df, k_par)
29     # Calculate number of time steps based on dt
30     numTimeSteps    = int64((time[2] - time[1])/dt)
31     # Initialize time vector
32     tVector         = linspace(time[1], time[2], numTimeSteps)
33
34     # Matrix to collect the data
35     ansMatrix       = zeros(length(f0), length(tVector))
36
37     # Initial conditions
38     ansMatrix[:,1]  = f0
39
40     # Loop
41     for t in 1:length(tVector)-1
42         current_df          = df(ansMatrix[:,t], tVector[t], k_par)
43         ansMatrix[:,t+1]    = ansMatrix[:,t] + current_df*dt
44     end
45
46     return ansMatrix, tVector
47 end
```

A.2 IM Euler solver

General implicit (IM) Euler solver implemented in Julia.

Julia code 2: `imEuler.jl`

```
################################################################################
# Implicit Euler solver (IM)
#
# Generalized implicit Euler solver (IM) for systems of equations, using the
# Newton-Raphson method for solving the system of (non)-linear equations in
# each time step.
#
# Input:
#   - f0:       [f1(0), f2(0), .... , fn(0)]
#   - time:     [t_start, t_end]
#   - dt:       Size of time step
#   - df:       Function that returns the derivatives of f, given as df(f(t),t)
#   - jacobi:   Function that returns the second derivatives of f as a matrix,
#               i.e the Jacobian.
#
# Output:
#   - f:        [f1(t_start) f1(t_start+dt) f1(t_start+2dt) ...  f1(t_end),
#                f2(t_start) f2(t_start+dt) f2(t_start+2dt) ...  f2(t_end),
#
#                                          ......
#
#                fn(t_start) fn(t_start+dt) fn(t_start+2dt) ...  fn(t_end)]
#
#
#   - t:        [t_start, t_start+dt, t_start+2dt, ... , t_end]
#
# Author:     Kjetil Sonerud
# Updated:    2014-12-05 15:13:13
################################################################################

function imEuler(f0, time, dt, df, jacobi, k_par)
    # Include Newton-Raphson
    include("newtonRaphson.jl")

    # Calculate number of time steps based on dt
    numTimeSteps    = int64((time[2] - time[1])/dt)
    # Initialize time vector
    tVector         = linspace(time[1], time[2], numTimeSteps)

    # Matrix to collect the data
    ansMatrix       = zeros((length(f0)+1), length(tVector))

    # Initial condition
    ansMatrix[1:end-1,1]  = f0

    # Loop
    for t in 1:length(tVector)-1
        # Defining the residual function
        residual(y, y_const, time, dt, df, k_par) = [-y + y_const + dt*df(y,time+dt, k_par)]
```

```
50
51          # Calculating the next time step
52          ansMatrix[:,t+1] = newtonRaphson(residual, jacobi, ansMatrix[1:end-1,t], ansMatrix[1:end-1,t], tVector[t], dt, df, k_par)
53      end
54
55      return ansMatrix, tVector
56  end
```

A.3 Newton-Raphson method

Newton-Raphson solver for (non-)linear systems of equations implemented in Julia.

Julia code 3: `newtonRaphson.jl`

```
#############################################################
# Newton-Raphson for solving a system of non-linear equations
#
# Input:
#           residual:       Function to find F(x) = 0
#           jacobi:         Jacobian matrix of F(x)
#           y_nplus1_0:     Initial guess of y_nplus1-vector
#           y_n:            (constant) y_n-vector
#           time:           Current value of t
#
# Output:
#           y_nplus1:       y_nplus1-vector that solve F(x)=0
#
# Author:   Kjetil Sonerud
# Updated:  2014-12-05 11:50:23
#############################################################

function newtonRaphson(residual, jacobi, y_nplus1_0, y_n, time, dt, df, k_par)
    # Tolerances
    delta   = 1e-7;
    epsilon = 1e-7;
    small   = 1e-7;

    # Maximum number of iterations and iteration counter
    maxiter     = 1000;
    itercounter = 0;

    # Flag; condition for loop termination
    flag = 0;

    # Initial function value
    y_nplus1 = y_nplus1_0;

    while flag == 0 && itercounter < maxiter
        # Increment counter
        itercounter += 1

        # Calculating current Jacobi
        jac     = jacobi(y_nplus1, time, k_par)
        res     = residual(y_nplus1, y_n, time, dt, df, k_par)

        # Calculating delta_x, assuming that the Jacobi is non-singular
        delta_y = -jac\res

        # Updating values
        y_nplus1    += delta_y
        res         = residual(y_nplus1, y_n, time, dt, df, k_par)

        # Calculating relative error
```

```
50          rel_error = 2*norm(delta_y)/(norm(y_nplus1) + small)
51
52          # Check for convergence
53          if rel_error < delta && maximum(abs(res)) < epsilon
54              if flag != 1
55                  flag = 2
56              end
57          end
58      end
59      return [y_nplus1, itercounter]
60  end
```

A.4 IMEX method for the expanded Robertson problem

IMEX solver for the expanded Robertson kinetics problem presented in chapter 4 implemented in Julia.

Julia code 4: `imexEuler_expandedRobertson.jl`

```
################################################################################
# Implicit-explicit Euler solver (IMEX) for expanded Robertson problem
#
# Specialized implicit-explicit Euler solver (IMEX) for systems of differential
# equations arising from the expanded Robertson kinetics problem, using the
# Newton-Raphson method for solving the system of (non)-linear equations in each
# time step.
#
# Input:
#   - f0:       [f1(0), f2(0), .... , fn(0)]
#   - time:     [t_start, t_end]
#   - dt:       Size of time step
#   - df:       Function that returns the derivatives of f, given as df(f(t),t)
#   - jacobi:   Function that returns the second derivatives of f as a matrix,
#               i.e the Jacobian.
#
# Output:
#   - f:        [f1(t_start) f1(t_start+dt) f1(t_start+2dt) ...  f1(t_end),
#                f2(t_start) f2(t_start+dt) f2(t_start+2dt) ...  f2(t_end),
#
#                                        ......
#
#                fn(t_start) fn(t_start+dt) fn(t_start+2dt) ...  fn(t_end)]
#
#
#   - t:        [t_start, t_start+dt, t_start+2dt, ... , t_end]
#
# Author:     Kjetil Sonerud
# Updated:    2014-12-09 02:37:23
################################################################################

function imexEuler_expandedRobertson(f0, residual, time, dt, df, jacobi, k_par, isImplicit)
    # Include Newton-Raphson
    include("newtonRaphson_expandedRobertson.jl")

    # Calculate number of time steps based on dt
    numTimeSteps    = int64((time[2] - time[1])/dt)
    # Initialize time vector
    tVector         = linspace(time[1], time[2], numTimeSteps)

     #number of implicit equations
    nrImplicit = countImplicit(isImplicit)

    # Matrix to collect the data
    implicit        = zeros(nrImplicit, length(tVector))
    explicit        = zeros(length(isImplicit)-nrImplicit, length(tVector))
    endVec          = zeros(1, length(tVector))

    # Initial condition
```

```
    i = 1;  e = 1
    for row in 1:length(isImplicit)
        if(isImplicit[row])
            implicit[i,:] = f0[row]
            i+=1;
        else
            explicit[e,:] = f0[row]
            e+=1;
        end
    end

    # Loop
    for t in 1:length(tVector)-1

        # Explicit calculation
        explicit[:,t+1] =   explicit[:,t] + returnExplicit(dt*df(
                            createFullMatrix(explicit[:,t], implicit[:,t], isImplicit), tVector[t], k_par),
                            isImplicit)


        # Implicit calculation
        ImplicitMatrix, endVec[1,t+1] = newtonRaphson_expandedRobertson(
                                        residual, jacobi, createFullMatrix(explicit[:,t+1], implicit[:,t], isImplicit),
                                        createFullMatrix(explicit[:,t+1], implicit[:,t], isImplicit),
                                        tVector[t], dt, df, k_par, isImplicit)

        implicit[:,t+1] = returnImplicit(ImplicitMatrix, isImplicit)
    end
    # Construct result
    result = [createFullMatrixFinal(explicit, implicit, isImplicit, tVector), endVec]
    return result, tVector
end

########################################
# Auxiliary functions
########################################
function createFullMatrix(explicit_t, implicit_t, isImplicit)
     i = 1; e = 1
     tempAnsMatrix  = zeros((length(f0)))
     for row in 1:length(isImplicit)
        if(isImplicit[row])
            tempAnsMatrix[row]=implicit_t[i]
            i+=1;
         else
            tempAnsMatrix[row]=explicit_t[e]
            e+=1;
         end
     end
     return tempAnsMatrix
end

function createFullMatrixFinal(explicit_t, implicit_t, isImplicit, tVector)
     i = 1; e = 1
     tempAnsMatrix  = zeros((length(f0)), length(tVector))
     for row in 1:length(isImplicit)
```

```
        if(isImplicit[row])
            tempAnsMatrix[row, :]=implicit_t[i, :]
            i+=1;
         else
            tempAnsMatrix[row,:]=explicit_t[e,:]
            e+=1;
         end
     end
     return tempAnsMatrix
end

function returnExplicit(fullMatrix, isImplicit)
    e=1; nrImplicit = countImplicit(isImplicit)
    tempExplicit = zeros(length(isImplicit)-nrImplicit, 1)
    for row in 1:length(isImplicit)
        if(!isImplicit[row])
            tempExplicit[e] = fullMatrix[row]
            e+=1;
        end
    end
    return tempExplicit
end

function returnImplicit(fullMatrix, isImplicit)
    i=1; nrImplicit = countImplicit(isImplicit)
    tempImplicit = zeros(nrImplicit)
    for row in 1:length(isImplicit)
        if(isImplicit[row])
            tempImplicit[i] = fullMatrix[row]
            i+=1;
        end
    end
    return tempImplicit
end

function countImplicit(isImplicit)
    nrImplicit = 0;
    for index in 1:length(isImplicit)
        if(isImplicit[index])
            nrImplicit+=1;
        end
    end
    return nrImplicit;
end
```

A.5 Newton-Raphson method for the IMEX method

Specialized Newton-Raphson solver for (non-)linear systems of equations to solved with the IMEX solver for the expanded Robertson problem implemented in Julia.

Julia code 5: `newtonRaphson_expandedRobertson.jl`

```
############################################################
# Newton-Raphson for solving a system of non-linear equations,
# used in the IMEX method for the expanded Robertson problem
#
# Input:
#           residual:       Function to find F(x) = 0
#           jacobi:         Jacobian matrix of F(x)
#           y_nplus1_0:     Initial guess of y_nplus1-vector
#           y_n:            (constant) y_n-vector
#           time:           Current value of t
#
# Output:
#           y_nplus1:       y_nplus1-vector that solve F(x)=0
#
# Author:   Kjetil Sonerud
# Updated:  2014-12-09 02:42:09
############################################################

function newtonRaphson_expandedRobertson(residual, jacobi, y_nplus1_0, y_n, time, dt, df, k_par, isImplicit)
    # Tolerances
    delta   = 1e-7;
    epsilon = 1e-7;
    small   = 1e-7;

    # Maximum number of iterations and iteration counter
    maxiter     = 1000;
    itercounter = 0;

    # Flag; condition for loop termination
    flag = 0;

    # Initial function value
    y_nplus1 = y_nplus1_0;

    function addYnplus1(delta)
        d = 1;
        addOnY = zeros(length(y_nplus1_0))
        for row in 1:length(isImplicit)
            if(isImplicit[row])
                addOnY[row]=delta[d]
                d+=1
            end
        end
        return addOnY
    end

    while flag == 0 && itercounter < maxiter
        # Increment counter
```

```
        itercounter += 1

        # Calculating current Jacobi
        jac     = jacobi(y_nplus1, time, k_par)
        res     = residual(y_nplus1, y_n, time, dt, df, k_par)

        # Calculating delta_y, assuming that the Jacobi is non-singular
        delta_y = -jac\res

        # Updating values
        y_nplus1        += addYnplus1(delta_y)
        res             = residual(y_nplus1, y_n, time, dt, df, k_par)

        # Calculating relative error
        rel_error = 2*norm(delta_y)/(norm(y_nplus1) + small)

        # Check for convergence
        if rel_error < delta && maximum(abs(res)) < epsilon
            if flag != 1
                flag = 2
            end
        end
    end
    return y_nplus1, itercounter
end
```

A.6 Run model cases

Script to run the different cases for the Robertson kinetics problem presented in chapter 4. The relevant data is logged and saved to file for analysis and plotting.

Julia code 6: `runAllCases.jl`

```
################################################################################
# Run all
#
# Run all cases using EX, IM and IMEX methods for the Robertson problem; both
# scaled and original problem
#
# Author:     Kjetil Sonerud
# Updated:    2014-12-07 10:34:35
################################################################################

workspace()

f_handle = open("results/data/log_runAllCases_"*string(strftime("%c", time()))*".txt", "w")
# f_handle = open("results/data/myTestLog.txt", "w")

# Which cases to run?
runCase1 = true
runCase2 = true
runCase3 = true
runCase4 = true


########################################
# Case 1:   scaled Robertson problem
#           0-40 sec
########################################
if runCase1 == true
    # Derivatives, Jacobian and parameters
    k_par               = [0.04, 3e3, 1e1]
    df(f,t,k_par)       = [-k_par[1]*f[1] + k_par[3]*f[2]*f[3], +k_par[1]*f[1]-k_par[3]*f[2]*f[3]-k_par[2]*f[2]*f[2], +k_par[2]*f[
    f0                  = [1,0,0]
    timespan            = [0,10]
    dt                  = 0.1
    jacobi(f,t,k_par)   = -eye(length(f0)) + [-k_par[1] k_par[3]*f[3] k_par[3]*f[2]; k_par[1] (-2*k_par[2]*f[2] - k_par[3]*f[3]) -

    ########################################
    # Run EX Euler
    ########################################
    numPoints   = 200
    gap         = ceil(((timespan[2] - timespan[1])/dt)/numPoints)

    println("EX Euler")
    include("exEuler.jl")

    exEuler(f0, timespan, dt, df, k_par)

    tic();
    x1,t1   = exEuler(f0, timespan, dt, df, k_par)
```

```
    cputime = toc();

    # Not including all data points for plotting
    x1_plot = x1[:,1:gap:end]
    t1_plot = t1[1:gap:end]

    # Save results
    results1 = ["t" "y1" "y2" "y3"; t1_plot x1_plot']
    writedlm("results/data/exEulerData_scaled_robertson.dat", results1, ' ')

    # Write to log file
    write(f_handle, "EX Euler scaled Robertson: \n")
    write(f_handle, "k_par:                "*string(k_par)*"\n")
    write(f_handle, "Total time steps:     "*string(length(t1))*"\n")
    write(f_handle, "dt:                   "*string(dt)*"\n")
    write(f_handle, "CPU-time:             "*string(cputime)*"\n\n")

    ########################################
    # Run IM Euler
    ########################################
    numPoints   = 200
    gap         = ceil(((timespan[2] - timespan[1])/dt)/numPoints)

    println("IM Euler")
    include("imEuler.jl")

    imEuler(f0, timespan, dt, df, jacobi, k_par)

    tic();
    x2,t2   = imEuler(f0, timespan, dt, df, jacobi, k_par)
    cputime = toc();

    # Not including all data points for plotting
    x2_plot = x2[:,1:1e0:end]
    t2_plot = t2[1:1e0:end]

    # Save results
    results2 = ["t" "y1" "y2" "y3" "iterations"; t2_plot x2_plot']
    writedlm("results/data/imEulerData_scaled_robertson.dat", results2, ' ')

    # Write to log file
    write(f_handle, "IM Euler scaled Robertson: \n")
    write(f_handle, "k_par:                "*string(k_par)*"\n")
    write(f_handle, "Total time steps:     "*string(length(t2))*"\n")
    write(f_handle, "dt:                   "*string(dt)*"\n")
    write(f_handle, "CPU-time:             "*string(cputime)*"\n")
    write(f_handle, "Total # iterations:   "*string(sum(x2[end,:]))*"\n")
    write(f_handle, "Mean # iterations:    "*string(sum(x2[end,:])/length(x2[end,:]))*"\n")
    write(f_handle, "Maximum # iterations: "*string(maximum(x2[end,:]))*"\n\n")
end

########################################
# Case 2:   original Robertson problem
#           0-40 sec
########################################
```

```
104 if runCase2 == true
105     # Derivatives, Jacobian and parameters
106     k_par               = [0.04, 3e7, 1e4]
107     df(f,t,k_par)       = [-k_par[1]*f[1] + k_par[3]*f[2]*f[3], +k_par[1]*f[1]-k_par[3]*f[2]*f[3]-k_par[2]*f[2]*f[2], +k_par[2]*f[
108     f0                  = [1,0,0]
109     timespan            = [0,40]
110     jacobi(f,t,k_par)   = -eye(length(f0)) + [-k_par[1] k_par[3]*f[3] k_par[3]*f[2]; k_par[1] (-2*k_par[2]*f[2] - k_par[3]*f[3]) -
111
112     ########################################
113     # Run EX Euler
114     ########################################
115     # Define time step
116     dt          = 6e-4
117
118     numPoints   = 1000
119     gap         = ceil(((timespan[2] - timespan[1])/dt)/numPoints)
120
121     println("EX Euler")
122     include("exEuler.jl")
123
124     exEuler(f0, timespan, dt, df, k_par)
125
126     tic();
127     x1,t1   = exEuler(f0, timespan, dt, df, k_par)
128     cputime = toc();
129
130     # Not including all data points for plotting
131     x1_plot = x1[:,1:gap:end]
132     t1_plot = t1[1:gap:end]
133
134     # Save results
135     results1 = ["t" "y1" "y2" "y3"; t1_plot x1_plot']
136     writedlm("results/data/exEulerData_original_robertson.dat", results1, ' ')
137
138     # Write to log file
139     write(f_handle, "EX Euler original Robertson: \n")
140     write(f_handle, "k_par:                 "*string(k_par)*"\n")
141     write(f_handle, "Total time steps:      "*string(length(t1))*"\n")
142     write(f_handle, "dt:                    "*string(dt)*"\n")
143     write(f_handle, "CPU-time:              "*string(cputime)*"\n\n")
144
145     ########################################
146     # Run IM Euler
147     ########################################
148     # Define time step
149     dt          = 1
150
151     numPoints   = 500
152     gap         = ceil(((timespan[2] - timespan[1])/dt)/numPoints)
153
154     println("IM Euler")
155     include("imEuler.jl")
156
157     imEuler(f0, timespan, dt, df, jacobi, k_par)
158
```

```
159     tic();
160     x2,t2   = imEuler(f0, timespan, dt, df, jacobi, k_par)
161     cputime = toc();
162
163     # Not including all data points for plotting
164     x2_plot = x2[:,1:gap:end]
165     t2_plot = t2[1:gap:end]
166
167     # Save results
168     results2 = ["t" "y1" "y2" "y3" "iterations"; t2_plot x2_plot']
169     writedlm("results/data/imEulerData_original_robertson.dat", results2, ' ')
170
171     # Write to log file
172     write(f_handle, "IM Euler original Robertson: \n")
173     write(f_handle, "k_par:                 "*string(k_par)*"\n")
174     write(f_handle, "Total time steps:      "*string(length(t2))*"\n")
175     write(f_handle, "dt:                    "*string(dt)*"\n")
176     write(f_handle, "CPU-time:              "*string(cputime)*"\n")
177     write(f_handle, "Total # iterations:    "*string(sum(x2[end,:]))*"\n")
178     write(f_handle, "Mean # iterations:     "*string(sum(x2[end,:])/length(x2[end,:]))*"\n")
179     write(f_handle, "Maximum # iterations:  "*string(maximum(x2[end,:]))*"\n\n")
180
181     ########################################
182     # Run IMEX Euler
183     ########################################
184     # Define time step
185     dt          = 1
186
187     # Redefine Jacobian to suit IMEX
188     jacobi(f,t, k_par)  = -eye(length(f0)-1) + [(-2*k_par[2]*f[2] - k_par[3]*f[3]) -k_par[3]*f[2]; 2*k_par[2]*f[2] 0]
189
190     numPoints   = 500
191     gap         = ceil(((timespan[2] - timespan[1])/dt)/numPoints)
192
193     println("IMEX Euler")
194     include("imexEuler_robertson.jl")
195
196     imexEuler_robertson(f0, timespan, dt, df, jacobi, k_par)
197
198     tic();
199     x3,t3   = imexEuler_robertson(f0, timespan, dt, df, jacobi, k_par)
200     cputime = toc();
201
202     # Not including all data points for plotting
203     x3_plot = x3[:,1:gap:end]
204     t3_plot = t3[1:gap:end]
205
206     # Save results
207     results2 = ["t" "y1" "y2" "y3" "iterations"; t3_plot x3_plot']
208     writedlm("results/data/imexEulerData_original_robertson.dat", results2, ' ')
209
210     # Write to log file
211     write(f_handle, "IMEX Euler original Robertson: \n")
212     write(f_handle, "k_par:                 "*string(k_par)*"\n")
213     write(f_handle, "Total time steps:      "*string(length(t3))*"\n")
```

```
    write(f_handle, "dt:                    "*string(dt)*"\n")
    write(f_handle, "CPU-time:              "*string(cputime)*"\n")
    write(f_handle, "Total # iterations:    "*string(sum(x3[end,:]))*"\n")
    write(f_handle, "Mean # iterations:     "*string(sum(x3[end,:])/length(x3[end,:]))*"\n")
    write(f_handle, "Maximum # iterations:  "*string(maximum(x3[end,:]))*"\n\n")
end

#######################################
# Case 3:   original Robertson problem
#           0-1000 sec
#######################################
if runCase3 == true
    # Derivatives, Jacobian and parameters
    k_par               = [0.04, 3e7, 1e4]
    df(f,t,k_par)       = [-k_par[1]*f[1] + k_par[3]*f[2]*f[3], +k_par[1]*f[1]-k_par[3]*f[2]*f[3]-k_par[2]*f[2]*f[2], +k_par[2]*f[
    f0                  = [1,0,0]
    timespan            = [0,1000]
    jacobi(f,t,k_par)   = -eye(length(f0)) + [-k_par[1] k_par[3]*f[3] k_par[3]*f[2]; k_par[1] (-2*k_par[2]*f[2] - k_par[3]*f[3]) -

    #######################################
    # Run EX Euler
    #######################################
    # Define time step
    dt          = 3e-4

    numPoints   = 1000
    gap         = ceil(((timespan[2] - timespan[1])/dt)/numPoints)

    println("EX Euler")
    include("exEuler.jl")

    exEuler(f0, timespan, dt, df, k_par)

    tic();
    x1,t1   = exEuler(f0, timespan, dt, df, k_par)
    cputime = toc();

    # Not including all data points for plotting
    x1_plot = x1[:,1:gap:end]
    t1_plot = t1[1:gap:end]

    # Save results
    results1 = ["t" "y1" "y2" "y3"; t1_plot x1_plot']
    writedlm("results/data/exEulerData_original_1000s_robertson.dat", results1, ' ')

    # Write to log file
    write(f_handle, "EX Euler original 1000s Robertson: \n")
    write(f_handle, "k_par:                 "*string(k_par)*"\n")
    write(f_handle, "Total time steps:      "*string(length(t1))*"\n")
    write(f_handle, "dt:                    "*string(dt)*"\n")
    write(f_handle, "CPU-time:              "*string(cputime)*"\n\n")

    #######################################
    # Run IM Euler
    #######################################
```

```
269     # Define time step
270     dt          = 1
271
272     numPoints   = 1000
273     gap         = ceil(((timespan[2] - timespan[1])/dt)/numPoints)
274
275     println("IM Euler")
276     include("imEuler.jl")
277
278     imEuler(f0, timespan, dt, df, jacobi, k_par)
279
280     tic();
281     x2,t2   = imEuler(f0, timespan, dt, df, jacobi, k_par)
282     cputime = toc();
283
284     # Not including all data points for plotting
285     x2_plot = x2[:,1:gap:end]
286     t2_plot = t2[1:gap:end]
287
288     # Save results
289     results2 = ["t" "y1" "y2" "y3" "iterations"; t2_plot x2_plot']
290     writedlm("results/data/imEulerData_original_1000s_robertson.dat", results2, ' ')
291
292     # Write to log file
293     write(f_handle, "IM Euler original 1000s Robertson: \n")
294     write(f_handle, "k_par:                 "*string(k_par)*"\n")
295     write(f_handle, "Total time steps:      "*string(length(t2))*"\n")
296     write(f_handle, "dt:                    "*string(dt)*"\n")
297     write(f_handle, "CPU-time:              "*string(cputime)*"\n")
298     write(f_handle, "Total # iterations:    "*string(sum(x2[end,:]))*"\n")
299     write(f_handle, "Mean # iterations:     "*string(sum(x2[end,:])/length(x2[end,:]))*"\n")
300     write(f_handle, "Maximum # iterations:  "*string(maximum(x2[end,:]))*"\n\n")
301
302     ########################################
303     # Run IMEX Euler
304     ########################################
305     # Define time step
306     dt          = 1
307
308     # Redefine Jacobian to suit IMEX
309     jacobi(f,t, k_par)  = -eye(length(f0)-1) + [(-2*k_par[2]*f[2] - k_par[3]*f[3]) -k_par[3]*f[2]; 2*k_par[2]*f[2] 0]
310
311     numPoints   = 1000
312     gap         = ceil(((timespan[2] - timespan[1])/dt)/numPoints)
313
314     println("IMEX Euler")
315     include("imexEuler_robertson.jl")
316
317     imexEuler_robertson(f0, timespan, dt, df, jacobi, k_par)
318
319     tic();
320     x3,t3   = imexEuler_robertson(f0, timespan, dt, df, jacobi, k_par)
321     cputime = toc();
322
323     # Not including all data points for plotting
```

```
    x3_plot = x3[:,1:gap:end]
    t3_plot = t3[1:gap:end]

    # Save results
    results2 = ["t" "y1" "y2" "y3" "iterations"; t3_plot x3_plot']
    writedlm("results/data/imexEulerData_original_1000s_robertson.dat", results2, ' ')

    # Write to log file
    write(f_handle, "IMEX Euler original 1000s Robertson: \n")
    write(f_handle, "k_par:                "*string(k_par)*"\n")
    write(f_handle, "Total time steps:     "*string(length(t3))*"\n")
    write(f_handle, "dt:                   "*string(dt)*"\n")
    write(f_handle, "CPU-time:             "*string(cputime)*"\n")
    write(f_handle, "Total # iterations:   "*string(sum(x3[end,:]))*"\n")
    write(f_handle, "Mean # iterations:    "*string(sum(x3[end,:])/length(x3[end,:]))*"\n")
    write(f_handle, "Maximum # iterations: "*string(maximum(x3[end,:]))*"\n\n")
end

########################################
# Case 4:   expanded Robertson problem
#           0-100 sec
########################################
if runCase4 == true
    # Derivatives, Jacobian and parameters
    k_par               = [0.03, 0.05, 0.04, 3e7, 1e4, 0.05, 0.04]
    df(f,t,k_par)       =   [
                                -k_par[1]*f[1],
                                +k_par[1]*f[1] - k_par[2]*f[2],
                                +k_par[2]*f[2] - k_par[3]*f[3] + k_par[5]*f[4]*f[5],
                                +k_par[3]*f[3] - k_par[5]*f[4]*f[5] - k_par[4]*f[4]*f[4],
                                +k_par[4]*f[4]*f[4] - k_par[6]*f[5],
                                +k_par[6]*f[5] - k_par[7]*f[6],
                                +k_par[7]*f[6]
                            ]
    f0                  = [1,0,0,0,0,0,0]
    timespan            = [0,600]
    jacobi(f,t,k_par)   =   (
                                -eye(length(f0))
                                +   [
                                        -k_par[1] 0 0 0 0 0;
                                        +k_par[1]  -k_par[2] 0 0 0 0;
                                        0 +k_par[2] -k_par[3] k_par[5]*f[5] k_par[5]*f[4] 0 0;
                                        0 0 k_par[3] (-2*k_par[4]*f[4] - k_par[5]*f[5]) -k_par[5]*f[4] 0 0;
                                        0 0 0 2*k_par[4]*f[4] -k_par[6] 0 0;
                                        0 0 0 0 k_par[6] -k_par[7] 0;
                                        0 0 0 0 0 k_par[7] 0;
                                    ]
                            )
    ########################################
    # Run EX Euler
    ########################################
    # Define time step
    dt          = 1e-3

    numPoints   = 1000
```

```
379     gap         = ceil(((timespan[2] - timespan[1])/dt)/numPoints)
380
381     println("EX Euler")
382     include("exEuler.jl")
383
384     exEuler(f0, timespan, dt, df, k_par)
385
386     tic();
387     x1,t1   = exEuler(f0, timespan, dt, df, k_par)
388     cputime = toc();
389
390     # Not including all data points for plotting
391     x1_plot = x1[:,1:gap:end]
392     t1_plot = t1[1:gap:end]
393
394     # Save results
395     results1 = ["t" "y1" "y2" "y3" "y4" "y5" "y6" "y7"; t1_plot x1_plot']
396     writedlm("results/data/exEulerData_expanded_robertson.dat", results1, ' ')
397
398     # Write to log file
399     write(f_handle, "EX Euler expanded Robertson: \n")
400     write(f_handle, "k_par:                 "*string(k_par)*"\n")
401     write(f_handle, "Total time steps:      "*string(length(t1))*"\n")
402     write(f_handle, "dt:                    "*string(dt)*"\n")
403     write(f_handle, "CPU-time:              "*string(cputime)*"\n\n")
404
405     ########################################
406     # Run IM Euler
407     ########################################
408     # Define time step
409     dt          = 1
410
411     numPoints   = 1000
412     gap         = ceil(((timespan[2] - timespan[1])/dt)/numPoints)
413
414     println("IM Euler")
415     include("imEuler.jl")
416
417     imEuler(f0, timespan, dt, df, jacobi, k_par)
418
419     tic();
420     x2,t2   = imEuler(f0, timespan, dt, df, jacobi, k_par)
421     cputime = toc();
422
423     # Not including all data points for plotting
424     x2_plot = x2[:,1:gap:end]
425     t2_plot = t2[1:gap:end]
426
427     # Save results
428     results2 = ["t" "y1" "y2" "y3" "y4" "y5" "y6" "y7" "iterations"; t2_plot x2_plot']
429     writedlm("results/data/imEulerData_expanded_robertson.dat", results2, ' ')
430
431     # Write to log file
432     write(f_handle, "IM Euler expanded Robertson: \n")
433     write(f_handle, "k_par:                 "*string(k_par)*"\n")
```

```
    write(f_handle, "Total time steps:      "*string(length(t2))*"\n")
    write(f_handle, "dt:                    "*string(dt)*"\n")
    write(f_handle, "CPU-time:              "*string(cputime)*"\n")
    write(f_handle, "Total # iterations:    "*string(sum(x2[end,:]))*"\n")
    write(f_handle, "Mean # iterations:     "*string(sum(x2[end,:])/length(x2[end,:]))*"\n")
    write(f_handle, "Maximum # iterations:  "*string(maximum(x2[end,:]))*"\n\n")

    ########################################
    # Run IMEX Euler
    ########################################
    # Define time step
    dt          = 1

    # Redefine Jacobian to suit IMEX
    jacobi(f,t, k_par)  = -eye(2) + [(-2*k_par[4]*f[4] - k_par[5]*f[5]) -k_par[5]*f[4]; 2*k_par[4]*f[4] 0]

    numPoints   = 1000
    gap         = ceil(((timespan[2] - timespan[1])/dt)/numPoints)

    println("IMEX Euler for the expanded Robertson problem")
    include("imexEuler_expandedRobertson.jl")

    # Define which equations are to be solved implicit
    isImplicit = [false false false true true false false]

    # Defining the residual function used in the N-R function
    residual(y, y_const, time, dt, df, k_par) = [-y[4:5] + y_const[4:5] + dt*df(y,time+dt, k_par)[4:5]]

    imexEuler_expandedRobertson(f0, residual, timespan, dt, df, jacobi, k_par, isImplicit)

    tic();
    x3,t3   = imexEuler_expandedRobertson(f0, residual, timespan, dt, df, jacobi, k_par, isImplicit)
    cputime = toc();

    # Not including all data points for plotting
    x3_plot = x3[:,1:gap:end]
    t3_plot = t3[1:gap:end]

    # Save results
    results3 = ["t" "y1" "y2" "y3" "y4" "y5" "y6" "y7" "iterations"; t3_plot x3_plot']
    writedlm("results/data/imexEulerData_expanded_robertson.dat", results3, ' ')

    # Write to log file
    write(f_handle, "IMEX Euler expanded Robertson: \n")
    write(f_handle, "k_par:                 "*string(k_par)*"\n")
    write(f_handle, "Total time steps:      "*string(length(t3))*"\n")
    write(f_handle, "dt:                    "*string(dt)*"\n")
    write(f_handle, "CPU-time:              "*string(cputime)*"\n")
    write(f_handle, "Total # iterations:    "*string(sum(x3[end,:]))*"\n")
    write(f_handle, "Mean # iterations:     "*string(sum(x3[end,:])/length(x3[end,:]))*"\n")
    write(f_handle, "Maximum # iterations:  "*string(maximum(x3[end,:]))*"\n\n")
end

########################################
# Close log file
```

```
489 ########################################
490 close(f_handle)
491
492 println("File closed!")
```