UCLouvain

École polytechnique de Louvain

Implicit time integrators for Differential Algebraic Equations in Python

Authors: **Loïc Beuken, Olivier Cheffert**
Supervisor: **Vincent Legat**
Readers: **Pierre-Antoine Absil, Ange Ishimwe**
Academic year 2021–2022
Master [120] in Mathematical Engineering

Acknowledgements

First of all, we would like to express our gratitude to our supervisor Vincent Legat for his support and guidance in this work. His advice helped us to focus our work on such a large subject in order to concentrate on the different essentials. Furthermore, we would like to thank him for supporting us and guiding us in our choices on the directions the work should take as we built up the objectives of our work.

Also, we would like to thank Aleksandra Tutueva and Denis Butusov for their advice and collaboration on the writing and submission of the article. In particular, they shared with us their code, their current research and their valuable advice that led to the co-writing of our first article.

Besides, we thank Pierre-Antoine Absil and Ange Ishimwe for agreeing to be part of the thesis jury as readers.

Contents

Introduction

Systems of ordinary differential equations (ODE) is one of the key techniques for describing dynamical systems. It's a natural way of expressing the evolution of a system whether it's a mechanical [1], electrical [7], chemical [6], biological system [26], etc. From a more contemporary perspective, the solution of ODEs is proving crucial in modeling epidemics [18].

Simulating dynamical systems on discrete computers requires the discretization of ODEs, which is usually performed by numerical integration methods [13]. The requirements for numerical methods increase with the rapid growth of complexity and stiffness of the simulated systems. Therefore, the development of new computationally efficient and stable numerical methods is of certain interest.

One of the main classes of ODE solvers are linear multistep methods which efficiently solve ODE systems using a single evaluation of the right-hand function at each integration step. Within this family, some methods are implicit such as the Backward Differentiation Formula (BDF) or Adams-Moulton method while others are explicit such as the Adams-Bashforth method. Implicit methods possess better numerical stability than the explicit ones but require performing Newton's method iterations on each step and therefore are more computationally expensive [11]. Moreover, explicit methods require smaller step sizes and may lose the order of accuracy in some cases such as stiff equations [17, 22]. On the other hand, there are also the explicit Runge-Kutta methods which require more function evaluations and implicit Runge-Kutta methods which are A-stable but require a greater computational effort than the linear multistep methods due to a bigger system to solve. However, Hairer et al. shown in [13] that an explicit Runge-Kutta method can be faster than a multistep method. Moreover, some Runge-Kutta methods are designed to be efficient such as SIRK and DIRK methods [14], but Runge-Kutta methods have another advantage: they are parallelizable [29, 21].

On top of that, a predictor-corrector method, Adams-Bashforth-Moulton (ABM),

has been proposed as a compromise between explicit and implicit linear multistep methods. The idea is to predict the integration step by Adams-Bashforth and correct it with Adams-Moulton. Thus, computations are entirely explicit and this technique has better numerical stability than Adams-Bashforth [28]. Recently, efficient semi-explicit and semi-implicit modifications of the ABM have been proposed in [33] as a trade-off between the efficiency of explicit methods and the accuracy of the implicit ones.

This work aims to study a more general modelling technique than ODEs: the Differential Algebraic Equations (DAEs). Some mathematical theoretical concepts will be developed to understand and then solve those types of problems. This will come along with how to use them in practice when modelling mechanical, electrical, chemical and control problems. Then, some classical methods (such as BDF or Runge-Kutta) will be turned into DAEs solvers as well as the semi-explicit and semi-implicit modifications of the ABM. Currently, a reference solver is DASSL, a Fortran code based on BDF, through which Linda Petzold has been awarded the J. H. Wilkinson Prize for Numerical Software in 1991. She has described this code in [25] and co-authored a book on solving DAEs [3]. On the other hand, Hairer has studied how Runge-Kutta methods perform compared to BDF in [14] and gave a Fortran implementation of another reference solver based on an implicit Runge-Kutta method called RadauIIa. In this work, semi-explicit and semi-implicit ABM will be compared to these reference solvers. In order to evaluate their performance, six test problems will be proposed and performance will be assessed based on the number of function evaluations. At this point, a numerical stability analysis will have been considered through a 2D Dahlquist test equation. Finally, the implementation of those solvers within the Scipy environment will be explained as well as the different ways of improvements that may lead to boost their performance.

Chapter 1

Differential Algebraic Equation theory

This chapter aims to study the mathematical concepts used along this work. First, the Differential Algebraic Equations (DAEs) will be studied as a generalization of Ordinary Differential Equations (ODEs). Then, some numerical stability analysis concepts will be developed to prepare the discussion about numerical methods applied to DAEs.

1.1 Why DAEs?

Most of the time, modelling physical systems has been realized using ODEs under an explicit form:

$$\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}(t)) \tag{1.1}$$

with $\mathbf{y}$ and $\mathbf{f}$ being vector-valued. On those systems, classical ODEs solvers and numerical stability analysis are applied.

However, when modelling a dynamical system the equations naturally lead to the following general form:

$$\mathbf{F}(t, \mathbf{y}(t), \mathbf{y}'(t)) = \mathbf{0}. \tag{1.2}$$

In theory, this general system can be rewritten under the form of (1.1) but some

models rely on algebraic constraints such that (1.2) can be written as:

$$\mathbf{F}(t, \mathbf{x}, \mathbf{y}, \mathbf{y}') = \mathbf{0} \tag{1.3a}$$

$$\mathbf{G}(t, \mathbf{x}, \mathbf{y}) = \mathbf{0} \tag{1.3b}$$

$\mathbf{y}$ is called the differential variable and $\mathbf{x}$ the algebraic variable. Note that this form may arise if the Jacobian of $\mathbf{F}$ with respect to $\mathbf{y}$' in (1.2) ($\frac{\partial \mathbf{F}}{\partial \mathbf{y}'} = \mathbf{F}_{\mathbf{y}'}$) is singular. This form is a Differential Algebraic Equations system (DAEs).

In the next section, it will be shown that DAEs arise naturally when modelling mechanical, electrical and chemical systems. Later, the link between ODEs and DAEs will be studied but for now, note that some manipulations (sometimes long and complex) are required to go from the form (1.2) to (1.1) which may degrade the readability of the equations and variables. DAEs keep the model intuitive and simple without altering the physical meaning of the variables by various manipulations. Exploring the impact of parameter variation is easier when working directly on the natural physical modelling. Finally, the physical relationships between variables and their derivatives may be obtained automatically by softwares and then solved by a DAEs solver without any reformulation.

1.1.1 Applications

Two main examples will be explained in details to show how DAEs arise naturally when modelling dynamical systems. The first one aims to show that any mechanical system modelled by the Euler-Lagrange equation leads to particular DAEs whose structure will be exploited in later sections. Indeed, when modelling such problems (multibody systems with interconnected bodies or even fixed ones), it's often necessary to use a path constraint. Then, a second example will show through a transformer that electrical network modelling often relies on DAEs due to the Kirchhoff's laws (current and voltage equations).

Moreover, DAEs can be applied on many other scientific fields. For example, in chemistry, conservation laws bring algebraic equations and Stechlinski et al. [31] shown how to use DAEs for process operation in chemical engineering. It's even possible to control chemical system modelled by DAEs (see Kumar et al. [19]) which is a common characteristic with ODEs. It's also possible to obtain DAEs by PDE discretization as Brenan et al. [3] shown for the gas ignition in a closed vessel and for the Navier-Stokes equations. In fact, they shown that using the same idea as the method of lines, i.e. discretizing the spatial derivatives, the modelling process leads naturally to DAEs. Furthermore, Meng Li et al. [20] derived a bioeconomic

differential algebraic predator–prey model with nonlinear prey harvesting. The differential part is based on a classical prey-dependent predator–prey system and the algebraic equation is included due to the consideration of the economic profit of harvesting. Also, A.J. van der Schaft [35] studied how to use the special structures that arise from DAEs when modelling networks and port-Hamiltonian systems.

Mechanical systems

When modelling mechanical systems, the classical method is to use the Euler-Lagrange equation:

$$\frac{d}{dt}\left(\frac{\partial}{\partial \mathbf{q}'}\mathcal{L}(\mathbf{q},\mathbf{q}')\right) = \frac{\partial}{\partial \mathbf{q}}\mathcal{L}(\mathbf{q},\mathbf{q}') \tag{1.4}$$

with

$$\mathcal{L}(\mathbf{q},\mathbf{q}') = T - U - \boldsymbol{\lambda}\cdot\boldsymbol{\phi} \tag{1.5}$$

such that:

- T is the kinetic energy
- U is the potential energy
- $\boldsymbol{\phi}$ is the vector of constraints

Then apply the Euler-Lagrange equation to mechanical constrained systems:

$$\begin{aligned}\frac{d}{dt}\left(\frac{\partial}{\partial \mathbf{q}'}T(\mathbf{q},\mathbf{q}')\right) &= \frac{\partial}{\partial \mathbf{q}}T(\mathbf{q},\mathbf{q}') + f(t,\mathbf{q},\mathbf{q}') + \mathbf{G}^T\lambda \\ \mathbf{0} &= \boldsymbol{\phi}(\mathbf{q})\end{aligned} \tag{1.6}$$

such that

- $\mathbf{q}$ is the vector of state variables
- f are the external forces
- $\boldsymbol{\phi}(\mathbf{q})$ is the constraint such that $\mathbf{G} = d\boldsymbol{\phi}(\mathbf{q})/d\mathbf{q}$

Applying this Euler-Equation to the simple pendulum described below:

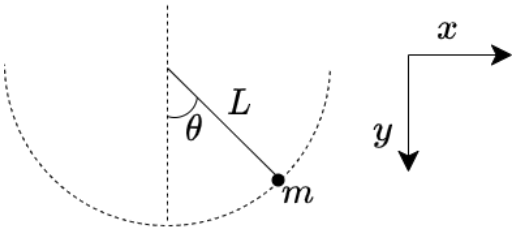

Figure 1.1: The simple pendulum problem

and using:

- $T = \frac{m}{2}(x'^2 + y'^2)$
- The external force is the weight $f = mg$ with $m = 1[kg]$
- $\boldsymbol{\phi}(\mathbf{q}) = 0.5(x^2 + y^2 - L^2)$
- (x, y) are the coordinates of the mass of the pendulum

the simple pendulum problem can be modelled by:

$$\begin{aligned} x'' &= \lambda x \\ y'' &= \lambda y - g \\ 0 &= 0.5(x^2 + y^2 - L^2) \end{aligned} \tag{1.7}$$

where λ is a Lagrange multiplier. This form is a DAE since no derivative of λ appears in the last equation which is the algebraic constraint. Then, λ is the algebraic variable while x, y are the differential ones.

Electrical systems

Connected closed electrical systems can be represented as flow-based graphs with their N electrical components as nodes and their M links as branches. To model those systems, the first step is to write N - 1 Kirchhoff's current equations, then $M - N + 1$ independent Kirchhoff's voltage equations and finally write the laws defining the impedance on the branches. While laws defining impedance bring

derivatives, Kirchhoff's equations naturally give rise to DAEs since they add algebraic sums to the model.

For example, let's take the following electrical transformer:

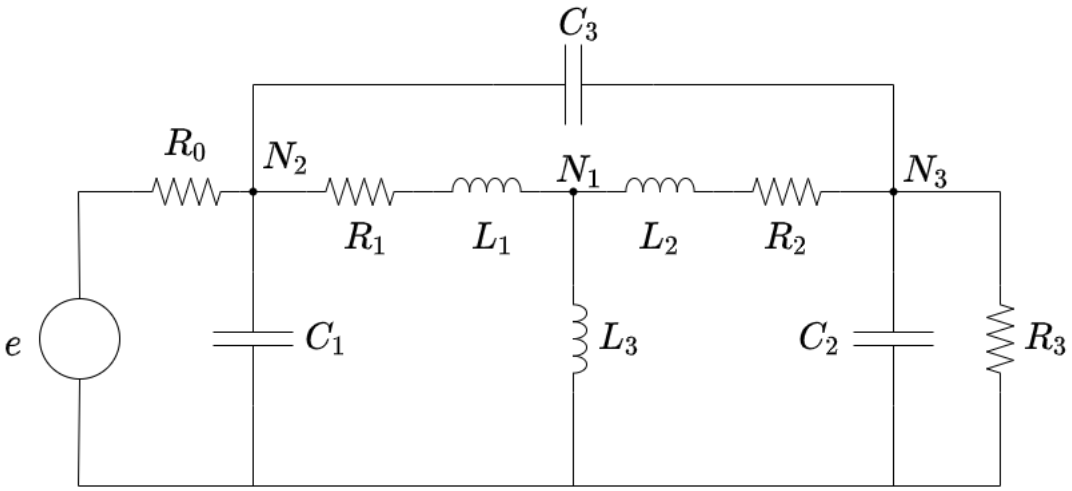

Figure 1.2: Equivalent circuit of a transformer

Taking into account the impedance laws, the Kirchhoff's equations can be formulated as:

$$\begin{aligned}
i_{L_3} &= i_{L_1} - i_{L_2} \\
i_0 &= i_{C_1} + i_{L_1} + i_{C_3} = C_1 v'_{C_1} + i_{L_1} + C_3 v'_{C_3} \\
i_{L_2} &= -i_{C_3} + i_{C_2} + i_3 = -C_3 v'_{C_3} + C_2 v'_{C_2} + i_3 \\
e &= R_0 i_0 + v'_{C_1} \\
v_{C_2} &= R_3 i_3 \\
v_{C_3} &= v_{C_1} - v_{C_2} \\
v_{C_1} &= R_1 i_{L_1} + L_1 i'_{L_1} + L_3 i'_{L_3} \\
L_3 i'_{L_3} &= L_2 i'_{L_2} + R_2 i_{L_2} + v_{C_2}
\end{aligned} \tag{1.8}$$

where i_j is the current going though resistance R_j, i_{L_j} the current going though inductance L_j and i_{C_j} (resp. v_{C_j}) the current (resp. voltage) going though capacitor C_j. Note that the first three equations are the Kirchhoff's current equations at the three nodes (see Figure 1.2) N_1, N_2 and N_3. This problem can be solved for $\mathbf{y} = [i_0, i_3, i_{L_1}, i_{L_2}, i_{L_3}, v_{C_1}, v_{C_2}, v_{C_3}]$ directly by a DAEs solver. Note that a slight manipulation gives a system $\mathbf{M} \cdot \mathbf{y}' = \mathbf{f}(t, \mathbf{y})$ which is another special form of DAEs

that often arises in physical system modelling. Solving this type of DAEs will be studied later with other examples.

This example shows how easy it is to model (automatically) an electrical system by DAEs using directly the Kirchhoff's laws. Modelling this system through ODEs requires manipulations such as finding the right substitutions. However, making the right manipulations/substitutions is not easy for big systems.

1.2 DAEs

Brenan et al. [3] have realized one of the first in-depth studies of the theoretical aspects of DAEs. They covered key concepts such as DAE's index and solvability that must be understood before studying the numerical methods applied to DAEs. This section will consider those two key concepts and classify DAEs depending on their structure. Classifying DAEs is a first step to find an efficient solver for a particular problem.

1.2.1 Index and Solvability

Although there are different types of DAEs, this section will define these concepts in a general way based on the following general nonlinear DAEs form:

$$\mathbf{F}(t, \mathbf{y}, \mathbf{y}') = \mathbf{0}. \tag{1.9}$$

The vector function $\mathbf{y}(t)$ is a solution of this DAE defined on the integration interval $\mathcal{I}$ if it satisfies (1.9) $\forall t \in \mathcal{I}$ and $\mathbf{y} \in \mathcal{C}^1$ on $\mathcal{I}$. In order to solve this DAE, the concept of solvability of DAEs must be defined as in [3].

Definition 1 *Let $\mathcal{I}$ be an open subinterval of $\mathbb{R}$, Ω a connected open subset of $\mathbb{R}^{2m+1}$ and F a differential function from Ω to $\mathbb{R}^m$. Then the DAE is solvable on $\mathcal{I}$ in Ω if there is an r-dimensional family of solutions $\phi(t, c)$ defined on a connected open set $\mathcal{I} \times \tilde{\Omega}$, $\tilde{\Omega} \subset \mathbb{R}^r$, such that:*

1. *$\phi(t, c)$ is defined on all of $\mathcal{I}$ for each $c \in \tilde{\Omega}$*
2. *$(t, \phi(t, c), \phi'(t, c)) \in \Omega$ for $(t, c) \in \mathcal{I} \times \tilde{\Omega}$*

3. *If $\psi(t)$ is any other solution with $(t, \psi(t), \psi'(t)) \in \Omega$, then $\psi(t) = \phi(t, c)$ for some $c \in \tilde{\Omega}$*

4. *The graph of ϕ as a function of (t, c) is an $r + 1$-dimensional manifold.*

More intuitively, the DAE (1.9) is solvable in an open set $\Omega \subset \mathbb{R}^{2m+1}$ if $(t, \mathbf{y}, \mathbf{y}') \in \Omega$ forms a smooth manifold such that each solution is defined by its initial value $\mathbf{y}(t_0) = \mathbf{y}_0$ and satisfies $(t_0, \mathbf{y}_0, \mathbf{y}'_0) \in \Omega$.

Definition 2 *The minimum number of times that (1.9) must be differentiated with respect to t in order to determine $\mathbf{y}'$ as a continuous function of $\mathbf{y}$ and t is the index of the DAE (1.9).*

So if (1.9) has index ν, then its solutions also satisfy the ODE:

$$\mathbf{y}' = \mathbf{G}(t, \mathbf{y}) \tag{1.10}$$

where $\mathbf{G}$ is a function of partial derivatives of $\mathbf{F}$. Note that all solutions of (1.10) are not necessarily solutions of the original DAE. The reduction of the index will be discussed in a later section. Furthermore, an implicit ODE has index zero while DAEs of index greater than one are called higher index DAEs. In fact, those high index DAEs are harder to solve numerically and will be analyzed separately from the index 1 DAEs in the rest of this work.

1.2.2 Types of DAEs

In order to study DAEs, it's useful to classify them and study their solvability, their index and some other properties that will be covered in the next parts. Linear DAEs are the simplest and most understood DAEs since they have been used in control theory. Then, semi-explicit DAEs that arise often in modelling will be covered followed by the general form of DAEs. The goal of this section is to build the theory behind DAEs from its most basic form to its most general one. Moreover, note that some existing solvers require specific forms of DAEs to exploit their characteristics and solve them efficiently. Even if, identifying the type of a DAE is not complicated, it's useful when it comes to solve it numerically.

Linear DAEs with constant coefficients

The linear constant DAE is defined by the following form:

$$\mathbf{A}\mathbf{y}'(t) + \mathbf{B}\mathbf{y}(t) = \mathbf{f}(t) \tag{1.11}$$

with $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{m\times m}$ and $\mathbf{f} \in \mathbb{R}^m$. For a complex λ, $\lambda\mathbf{A} + \mathbf{B}$ is called a matrix pencil.

Theorem 1 *The linear constant coefficient DAE is solvable if and only if* $\lambda\mathbf{A} + \mathbf{B}$ *is a regular pencil.*

This matrix pencil is called singular if $\lambda\mathbf{A}+\mathbf{B}$ is singular $\forall\lambda$, otherwise it's called regular. However, solving DAEs is not the same as solving the ODEs obtained by differentiating the constraints. Indeed, let's take a look at the following example:

Example 1

$$x(t) + y'(t) = f(t) \tag{1.12a}$$
$$y(t) = g(t). \tag{1.12b}$$

Differentiating the constraint (1.12b) and by a substitution into (1.12a), the system becomes:

$$x(t) = f(t) - g'(t) \tag{1.13a}$$
$$y'(t) = g'(t) \tag{1.13b}$$

so the index of this DAE is $\nu = 2$*, since two derivatives are required to get the ODE:*

$$x'(t) = f'(t) - g''(t) \tag{1.14a}$$
$$y'(t) = g'(t) \tag{1.14b}$$

and one solution is given by:

$$x(t) = f(t) - g'(t) \tag{1.15a}$$
$$y(t) = g(t). \tag{1.15b}$$

With this example, it can be observed that:

1. The solution can involve derivative of order $\nu - 1$ of the right hand side function $\mathbf{f}$ if the DAE is higher index.

2. Not all initial conditions of (1.11) admit a smooth solution if $\nu \geq 1$. In other words, all solutions of the ODE are not solutions of the DAE. So the initial condition must be consistent with the algebraic constraints. In this example, many solutions exist for the ODEs system (1.14) while the DAEs equations (1.12) has an unique solution described by the equations (1.15).

3. The previous point suggests that higher index DAEs can have hidden algebraic constraints. In this example, there is a hidden algebraic constraint on $x(t)$ given by (1.13a) while $y(t)$ must satisfy (1.12b).

A similar theory can be applied for non constant coefficients with a matrix pencil $\lambda\mathbf{A}(t) + \mathbf{B}(t)$. More details are available in [3].

Semi-explicit DAEs

Semi-explicit DAEs have a simple structure and are easily recognized since they can be viewed as an ODEs system with path constraints which explains why they frequently arise in many applications. Indeed, the first equations describe the dynamics of the differential variables while the path constraints define a manifold on which the solutions must lie.

$$\begin{aligned} \mathbf{y}' &= \mathbf{f}(t, \mathbf{x}, \mathbf{y}) \\ \mathbf{0} &= \mathbf{g}(t, \mathbf{x}, \mathbf{y}). \end{aligned} \tag{1.16}$$

In this case, $\mathbf{y}$ is called differential variable while $\mathbf{x}$ is the algebraic variable.

Some common characteristics with linear constant coefficient DAEs can be observed. Let's illustrate this with the following example.

Example 2 *A simple semi-explicit index one DAE is of the form*

$$y' = f(x, y) \tag{1.17a}$$
$$0 = g(x, y) \tag{1.17b}$$

with $\frac{\partial g}{\partial x}$ nonsingular. The differentiation of the algebraic equation (1.17b) is given by:

$$0 = \frac{\partial g}{\partial y} y' + \frac{\partial g}{\partial x} x'. \tag{1.18}$$

After a substitution of (1.17a) for y' into equation (1.18), this gives

$$x' = -\left(\frac{\partial g}{\partial x}\right)^{-1} \frac{\partial g}{\partial y} f. \tag{1.19}$$

The equations (1.17a) and (1.19) form the underlying ODE. In this case, the index of the DAE is one and a consistent initial values must satisfy $0 = g(x_0, y_0)$. There is no hidden constraint since the only constraint to satisfy is explicit in the original equation of the DAEs. However, in the previous section, hidden constraints were mentioned for high index DAEs ($\nu > 1$). In fact, this property is true for any form of DAEs and it can observed that by modifying slightly the previous example:

$$y' = f(x, y) \tag{1.20a}$$
$$0 = g(y) \tag{1.20b}$$

the differential index is two. Consequently, there is one hidden constraint given by

$$0 = \frac{\partial g}{\partial y} f. \tag{1.21}$$

Initial conditions are now consistent if they satisfy both equations (1.20b) and (1.21). Hence, it's more challenging to obtain a consistent initial condition for high index DAEs. Optimal control problems are semi-explicit index 1, 3 or 5 DAEs as Hairer and Wanner ([14] p462) shown on problems given by:

$$\mathbf{y}' = \mathbf{f}(t, \mathbf{x}, \mathbf{u})$$

and a cost

$$J = \int_{t_0}^{t_1} g(s, \mathbf{x}, \mathbf{u}) ds$$

Finally, note that a more general form than the semi-explicit one is described by:

$$\mathbf{M}\, \mathbf{y}' = \mathbf{f}(t, \mathbf{y}) \tag{1.22}$$

with $\mathbf{M}$ the mass matrix which is nonsingular for (implicit) ODEs and singular for a DAEs system. Numerical treatments between this form and the semi-explicit one are very similar and require only small modifications. The numerical solvers studied in this work will be applied on this mass matrix form. In fact, it's a very natural way of modelling mechanical, electrical and chemical problems as it will be shown in section 2.3. Nevertheless, the two physical examples developed in section 1.1.1 already respect this mass matrix form.

Nonlinear systems

Until now, the presented DAEs were linear. However, most DAEs are non-linear. The theory related to this type is similar to non constant coefficient DAEs but non-linearity makes these results true only locally. In addition, it's more difficult to establish necessary and sufficient initial conditions. The general form for the nonlinear DAE is

$$\mathbf{F}(t, \mathbf{y}, \mathbf{y}') = \mathbf{0}. \tag{1.23}$$

Definition 3 *The index ν of (1.23) is the smallest ν such that $\mathbf{F}_{\nu+1}$ uniquely determines the variable $\mathbf{y}'$ as a continuous function of $\mathbf{y}$, t.*

where $\mathbf{F}_j = \mathbf{F}_j(t, \mathbf{y}, \mathbf{y}_j) = 0$ and

$$\mathbf{y}_j = \begin{bmatrix} \mathbf{y}' \\ \vdots \\ \mathbf{y}^{(j+1)} \end{bmatrix}$$

In practice, to solve nonlinear system, the implicit Euler method with step size h can be applied to (1.23)

$$\mathbf{F}(t_n, \mathbf{y}_n, \frac{\mathbf{y}_n - \mathbf{y}_{n-1}}{h}) = 0 \tag{1.24}$$

which can be solved by a nonlinear system of equation solver. This is a simple first order solver but higher order ones exist such as DASSL.

Hessenberg form

High index DAEs in Hessenberg form occur frequently in many applications. For example, Kenneth Clark shows that this type of DAEs often arises in circuit and control theory particularly with differentiators or for the linear-quadratic regulator problem [9].

Definition 4 *A linear time varying DAEs $\mathbf{A}(t)\mathbf{y}'(t) + \mathbf{B}(t)\mathbf{y}(t) = \mathbf{f}(t)$ is in Hessenberg form of size r if it can be rewritten as:*

$$\mathbf{A}(t) = \begin{bmatrix} \mathbf{A_{11}}(t) & 0 & 0 & \dots & 0 \\ \mathbf{A_{21}}(t) & \mathbf{A_{22}}(t) & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \dots & \vdots \\ \mathbf{A_{r1}}(t) & \dots & \dots & \dots & \mathbf{A_{rr}}(t) \end{bmatrix}$$

$$\mathbf{B}(t) = \begin{bmatrix} \mathbf{B_{11}}(t) & 0 & 0 & \dots & 0 \\ \mathbf{B_{21}}(t) & \mathbf{B_{22}}(t) & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \dots & \vdots \\ \mathbf{B_{r1}}(t) & \dots & \dots & \dots & \mathbf{B_{rr}}(t) \end{bmatrix}$$

The most nonlinear general DAEs in Hessenberg form is given by the following definition:

Definition 5 *The DAEs (1.23) is in Hessenberg form of size r if it can be rewritten as:*

$$\begin{aligned} \mathbf{y}_1' &= \mathbf{F}_1(\mathbf{y}_1, ..., \mathbf{y}_r, t) \\ \mathbf{y}_2' &= \mathbf{F}_2(\mathbf{y}_1, ..., \mathbf{y}_{r-1}, t) \\ &\vdots \\ \mathbf{y}_i' &= \mathbf{F}_i(\mathbf{y}_{i-1}, ..., \mathbf{y}_{r-1}, t) \qquad \forall i \in \{3, ..., r-1\} \\ &\vdots \\ 0 &= \mathbf{F}_r(\mathbf{y}_{r-1}, t) \end{aligned}$$

and $\left(\frac{\partial \mathbf{F}_r}{\partial \mathbf{y}_{r-1}}\right)\left(\frac{\partial \mathbf{F}_{r-1}}{\partial \mathbf{y}_{r-2}}\right)\dots\left(\frac{\partial \mathbf{F}_2}{\partial \mathbf{y}_1}\right)\left(\frac{\partial \mathbf{F}_1}{\partial \mathbf{y}_r}\right)$ *is nonsingular.*

One interesting property about the Hessenberg form is that a proposition states its solvability:

Proposition 1 *If all $\mathbf{F}_i$ are sufficiently differentiable, the DAEs in Hessenberg form of size r is solvable and its index is r.*

For example, DAEs in Hessenberg form are the constrained variational problems like constrained mechanical systems whose general equations have been given by

(1.6). The model can be rewritten as:

$$\begin{aligned}\frac{\partial^2 T}{\partial \mathbf{u}^2}\mathbf{u}' &= g(t,\mathbf{q},\mathbf{u}) + \mathbf{G}^T\boldsymbol{\lambda}\\ \mathbf{q}' &= \mathbf{u}\\ \mathbf{0} &= \boldsymbol{\phi}(\mathbf{q})\end{aligned}$$

which are index 3 DAEs in Hessenberg form because they can be rewritten as:

$$\begin{aligned}\mathbf{H}(\mathbf{y}_1,\mathbf{y}_2)\mathbf{y}_1' &= \mathbf{F}_1(\mathbf{y}_1,\mathbf{y}_2,\mathbf{y}_3,t)\\ \mathbf{y}_2' &= \mathbf{F}_1(\mathbf{y}_1,\mathbf{y}_2,t)\\ 0 &= \mathbf{F}_3(\mathbf{y}_2,t)\end{aligned}$$

with $\mathbf{y}_1 = \mathbf{u}$, $\mathbf{y}_2 = \mathbf{q}$ and $\mathbf{y}_3 = \boldsymbol{\lambda}$.

Later, it will be shown that high index DAEs under Hessenberg form have interesting properties when they are solved by the solvers covered in this work.

1.3 Tools for numerical methods

This section will cover the index reduction technique that tries to take a DAEs and turn it into an ODEs. Then, the choice of implicit methods will be argued through a simple example: the Van der Pol oscillator. Finally, some mathematical concepts of numerical stability will be covered to prepare the analysis of the solvers studied in this work.

1.3.1 Index reduction

Numerically solving an algebraic differential equation of index 1 in a semi-explicit form or with a mass matrix is easier than solving higher index DAEs. To solve those high index DAEs, the most common solution is to reduce the index by differentiating the constraints.

The most common and general technique is the Pantelides algorithm [24]. This is a graph-theoretical algorithm that study the structure of the equations to find the equations to be derived in order to reduce the index of the DAEs. However, Pantelides explains that, for some DAEs, his algorithm does not work leaving the high index DAEs with no index reduction.

Differentiating the algebraic constraints modifies them and makes appear a drift-off phenomenon. For example, Hairer shows in [14] that the error in the constraint in a differentiated problem grows linearly while a twice differentiated problem has an error in the original algebraic constraint growing quadratically. In other words, differentiation reduces the accuracy for the algebraic constraints.

Other methods are possible to reduce the index: stabilization by projection, invariants or introducing new constraints or variables. A first attempt proposed by Baumgarte [2] was not to replace the constraint by its derivative but to replace it by a linear combination of its derivatives whose coefficients are carefully chosen. However, choosing such coefficients is not an easy task and can lead to damping effects. Therefore, the stabilization by projection method was proposed to avoid such instabilities using repeated projection of the numerical solution onto the solution manifold [14]. Depending on the invariant of the problem, some integration methods are more suited and projections must be used in a particular way when they are coupled with such methods.

Finally, Gear [12] has proposed a method to reintroduce back some constraints lost by differentiations or by change of variables but nowadays the most general algorithm used is the Pantelides algorithm (see for example the function *reduceDAEIndex* in MATLAB).

1.3.2 The choice of implicit methods

A classical problem to start studying DAE problems is the Van der Pol oscillator. The ODE for this problem is given by:

$$\begin{aligned} y' &= -z && =: f(y,z) \\ \epsilon z' &= y - \left(\frac{z^3}{3} - z\right) && =: g(y,z). \end{aligned} \tag{1.25}$$

To approximate a solution where ϵ is small, one idea is to set $\epsilon = 0$ which implies that the equations become a DAEs:

$$\begin{aligned} y' &= -z && =: f(y,z) \\ 0 &= y - \left(\frac{z^3}{3} - z\right) && =: g(y,z). \end{aligned} \tag{1.26}$$

While the ODE has no analytic solution, the DAE can be solved and the solution

is given by:

$$ln|z| - \frac{z^2}{2} = x + C. \tag{1.27}$$

This problem is often referred as a stiff differential equation. The solution of stiff differential equations requires a special choice of numerical methods. The problem with $\epsilon = 0$ results when the stiffness is pushed to the limit. In other words, a DAE problem is an ODE one but where the stiffness is really considerable. But a big point for numerical methods to deal with is the step size. Explicit methods require small step size because of the stiffness, so implicit methods are generally employed.

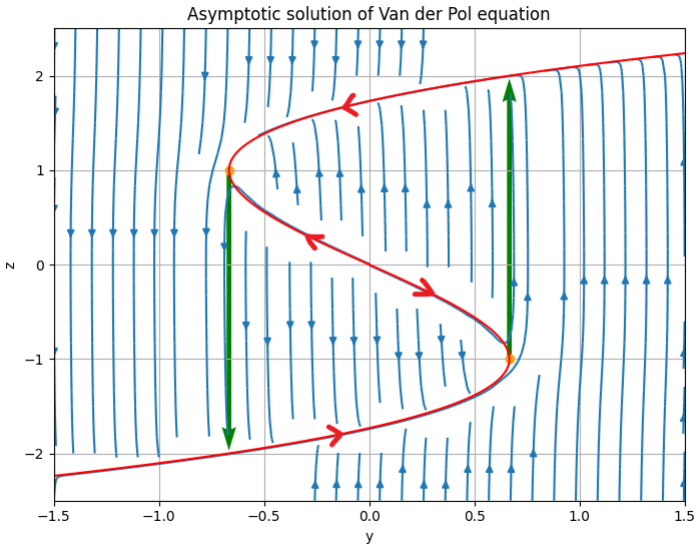

Figure 1.3: Van der Pol problem

The ODEs formulation of this problem relies on a phase plane diagram on which the asymptotic periodic limit cycle is found such that $\epsilon \to 0$. Thus this limit cycle is the solution of the DAEs formulation (equation 1.26). Note that there is a discontinuity at the orange points at which the orbit "jumps" following the direction of the green arrows.

1.3.3 Numerical stability

This section will introduce the different mathematical concepts needed to study the numerical stability of the methods considered in this work. In order to study their numerical stability, their stability region will be plotted and analyzed. The computation of their stability domain is based on the following definition.

Definition 6 *The function $R(z)$ is called the stability function of the method. It can be interpreted as the numerical solution after one step for*

$$y' = \lambda y, \qquad y_0 = 1, \qquad z = h\lambda, \tag{1.28}$$

which is the Dahlquist test equation. The set

$$S = \{z \in \mathbb{C}; |R(z)| \leq 1\} \tag{1.29}$$

is the stability domain of the method.

Based on the numerical stability domain of a method, it's therefore possible to study its numerical stability and more particularly if a method is able to solve all stable problems. Being able to solve stable problems is desirable property and it can be verified using the following definition.

Definition 7 *(Dahlquist 1963). A method, whose stability domain satisfies*

$$S \supset \mathbb{C}^- = \{z;\ Re(z) \leq 0\},$$

is called A-stable.

However, being A-stable is not the only desirable behaviour of a numerical method. Indeed, let's take a look at how to solve

$$y' = -2000(y - cos(t)), \qquad y(0) = 0, \qquad 0 \leq t \leq 1.5, \tag{1.30}$$

by two A-stable implicit methods:

- Implicit Euler method with $R(z) = \frac{1}{1-z}$
- Implicit midpoint method with $R(z) = \frac{1+z/2}{1-z/2}$

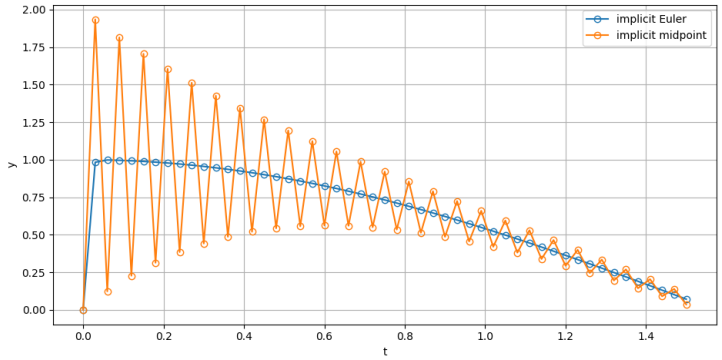

Figure 1.4: Solution of problem (1.30) with implicit Euler and midpoint methods

In fact, even if both methods converge, the implicit midpoint rule does have oscillations and that's an undesirable effect. In order to understand this behaviour, let's take a look at their stability region for large negative values as in this example ($\lambda = -2000$). In Figure 1.5, it's shown that for large negative numbers, implicit Euler has a smaller amplification factor than implicit midpoint method.

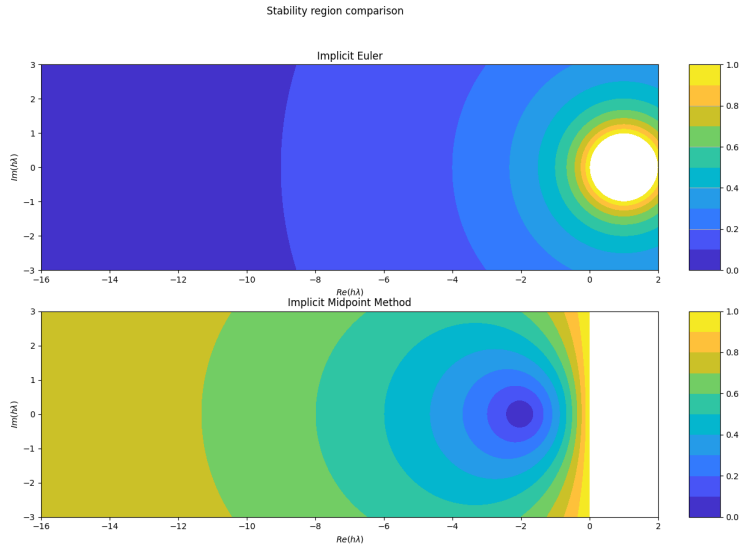

Figure 1.5: Stability regions of implicit Euler and implicit midpoint

This observation leads to the following stronger definition of stability able to deal with stiff problems.

Definition 8 *(Ehle 1969). A method is called L-stable if it's A-stable and if in addition*

$$\lim_{z\to\infty} R(z) = 0. \tag{1.31}$$

Linear multistep methods

A general k-step multistep method is of the form

$$\alpha_k y_{m+k} + \alpha_{k-1} y_{m+k-1} + \cdots + \alpha_0 y_m = h(\beta_k f_{m+k} + \cdots + \beta_0 f_m). \tag{1.32}$$

For this method, the same stability analysis as for one step methods can be performed. Consider the linearized and autonomous system

$$y' = Jy \tag{1.33}$$

this gives

$$\alpha_k y_{m+k} + \cdots + \alpha_0 y_m = hJ(\beta_k y_{m+k} + \cdots + \beta_0 y_m). \tag{1.34}$$

replace J by the corresponding eigenvalue λ. This gives

$$(\alpha_k - \mu\beta_k) y_{m+k} + \cdots + (\alpha_0 - \mu\beta_0) y_m = 0, \qquad \mu = h\lambda \tag{1.35}$$

and then applied to Dahlquist's test equation. The equation (1.35) is solved using Lagrange's method ([13] section $III.3$) and leads to the characteristic equation

$$(\alpha_k - \mu\beta_k)\zeta^k + \cdots + (\alpha_0 - \mu\beta_0) = \varrho(\zeta) - \mu\sigma(\zeta) = 0 \tag{1.36}$$

which depends on the complex parameter μ.

Definition 9 $S = \left\{ \mu \in \mathbb{C};\ \begin{array}{c} \textit{all roots } \zeta_j(\mu) \textit{ of } (1.36) \textit{ satisfy } |\zeta_j(\mu)| \le 1, \\ \textit{multiple roots satisfy } |\zeta_j(\mu)| < 1 \end{array} \right\}$ *is called the stability domain or stability region or region of absolute stability of method (1.32).*

Theorem 2 *(Dahlquist 1963). An A-stable multistep method must be of order $p \le 2$. If the order is 2, then the error constant satisfies*

$$C \le -\frac{1}{12}.$$

The trapezoidal rule is the only A-stable method of order 2 with $C = -1/12$.

This is a clear limitation of linear multistep methods since they have to use a lower order of accuracy to keep the A-stable property. However, these methods remain interesting in particular for their low number of function calls. Therefore, a weaker stability definition has been given to study their stability and to try to get around the problem of the second barrier of Dahlquist (Theorem 2).

Definition 10 *(Widlund 1967). A method is said to be $A(\alpha)$-stable if the sector*

$$S_\alpha = \{z;\ |arg(-z)| < \alpha,\ z \neq 0\}$$

is contained in the stability region.

Note that, in general, this stability definition implies that it's difficult to solve problems with pure imaginary eigenvalues with such methods without using a small step size. Those problems arise when systems rely on oscillations, for example the pendulum whose stability and eigenvalues will be studied later.

In the next chapter, some classical numerical methods used to solve ODEs will be studied through their equations, their characteristics and their numerical stability. Moreover, it will be shown how to turn them into DAEs solvers and a new method will be given.

Chapter 2

Numerical methods

This chapter considers how to solve DAEs using the classical numerical methods for ODEs described in section 2.1 and explains some developments that appear in [14]. Indeed, modifications of those solvers will be needed to solve DAEs of the form $\mathbf{M}\mathbf{y}' = \mathbf{f}(t, \mathbf{y})$. Then subsection 2.2.4 will propose new methods for DAEs: a semi-explicit and a semi-implicit modifications of ABM. Those new methods will be detailed more precisely through their stability analysis in the next chapter. Finally, the end of this chapter will be devoted to experiment those methods on real electrical, mechanical and chemical problems and how these solvers perform depending on the characteristics of the problems.

2.1 Classical numerical methods

The most classical methods used when solving ODEs are Runge-Kutta and linear multistep methods. To begin, it's useful to understand how those methods are built for ODEs which characterizes their behaviour and their stability. After that, those solvers must be modified to solve DAEs of the form $\mathbf{M}\mathbf{y}' = \mathbf{f}(t, \mathbf{y})$. In fact, this work will use two one step and two linear multistep methods such that their ODEs expressions in this section and then their DAEs modifications in the next section will be considered.

2.1.1 One step methods

The most studied one step methods is the family of Runge-Kutta (RK) methods which can be defined by the definitions below. The first definition describes the explicit RK while the second one, describes the more general ones. Note that propositions and definitions of this section are given in [13].

Definition 11 *Let s be an integer (the "number of stages") and $a_{21}, a_{31}, a_{32}, \cdots, a_{s1}, a_{s2}, \cdots, a_{s,s-1}, b_1, \cdots, b_s, c_2, \cdots, c_s$ be real coefficients. Then the method given by:*

$$\begin{aligned} k_1 &= f(t_0, y_0) \\ k_2 &= f(t_0 + c_2 h, y_0 + h a_{21} k_1) \\ k_2 &= f(t_0 + c_3 h, y_0 + h(a_{31} k_1 + a_{32} k_2)) \\ &\cdots \\ k_s &= f(t_0 + c_s h, y_0 + h(a_{s1} k_1 + \cdots + a_{s,s-1} k_{s-1})) \\ y_1 &= y_0 + h(b_1 k_1 + \cdots b_s k_s) \end{aligned} \tag{2.1}$$

is called an s-stage explicit Runge-Kutta method (ERK).

Definition 12 *Let b_i, $a_{ij} (i, j = 1, \cdots, s)$ be real numbers. The method*

$$\begin{aligned} k_i &= f\left(t_0 + c_i h, y_0 + h \sum_{j=1}^{s} a_{ij} k_j\right) \qquad i = 1, \cdots, s \\ y_1 &= y_0 + h \sum_{i=1}^{s} b_i k_i \end{aligned} \tag{2.2}$$

where c_i is defined by

$$c_i = \sum_{j=1}^{i-1} a_{ij}$$

is called an s-stage Runge-Kutta method.

Therefore, any Runge-Kutta method is defined by its Butcher tableau:

$$\begin{array}{c|c} \mathbf{c} & \mathbf{A} \\ \hline & \mathbf{b}^T \end{array}$$

with $\mathbf{A}_{ij} = a_{ij},\ \mathbf{b}_i = b_i,\ \mathbf{c}_i = c_i$. Depending on the form of the coefficient matrix $\mathbf{A}$, RK methods can take many forms:

- Explicit Runge-Kutta (ERK): $a_{ij} = 0$ for $i \leq j$, i.e. $\mathbf{A}$ is lower triangular with 0 on its diagonal.
- Diagonal implicit Runge-Kutta (DIRK): $a_{ij} = 0$ for $i < j$ and at least one $a_{ii} \neq 0$. In other words, a lower triangular matrix with at least one non zero element on its diagonal.
- Simply diagonal implicit Runge-Kutta (SDIRK) : $a_{ij} = 0$ for $i < j$ and all diagonal elements are identical $a_{ii} = \gamma$.
- Implicit Runge-Kutta (IRK): all other cases.

Proposition 2 *If an implicit Runge-Kutta method with nonsingular A satisfies one of the following conditions:*

$$a_{sj} = b_j, \qquad j = 1, \cdots, s, \tag{2.3}$$
$$a_{i1} = b_1, \qquad i = 1, \cdots, s, \tag{2.4}$$

then $R(\infty) = 0$. This makes A-stable methods L-stable.

Proof: Given the stability function of RK methods:

$$R(z) = 1 + z\mathbf{b}^T(\mathbf{I} - z\mathbf{A})^{-1}\mathbb{1}$$

this becomes at infinity:

$$R(\infty) = 1 - \mathbf{b}^T\mathbf{A}^{-1}\mathbb{1}$$

If (2.3), $\mathbf{b} = \mathbf{A}^T\mathbf{e}_s$ for $\mathbf{e}_s = (0, ..., 0, 1)^T$. Thus, $R(\infty) = 1 - \mathbf{e}_s^T\mathbb{1} = 0$.

If (2.4), $b_1\mathbb{1} = \mathbf{A}\mathbf{e}_1$. Thus, $R(\infty) = 1 - \mathbf{b}^T\mathbf{A}^{-1}\mathbf{A}\mathbf{e}_1 b_1^{-1} = 0$. ■

A lot of RK methods exist but this report will cover a major IRK method, the Radau IIA 5th order 3 stages method given by:

$\frac{4-\sqrt{6}}{10}$	$\frac{88-7\sqrt{6}}{360}$	$\frac{296-169\sqrt{6}}{1800}$	$\frac{-2+3\sqrt{6}}{225}$
$\frac{4+\sqrt{6}}{10}$	$\frac{296+169\sqrt{6}}{1800}$	$\frac{88+7\sqrt{6}}{360}$	$\frac{-2-3\sqrt{6}}{225}$
1	$\frac{16-\sqrt{6}}{36}$	$\frac{16+\sqrt{6}}{36}$	$\frac{1}{9}$
	$\frac{16-\sqrt{6}}{36}$	$\frac{16+\sqrt{6}}{36}$	$\frac{1}{9}$

Ehle [10] proved that s-stage Radau IIA methods are A-stable and of order $2s-1$. Moreover, by looking at this tableau, it's clear that (2.3) is satisfied which means that Radau IIA is L-stable. From here, for clarity's sake, RadauIIA will refer to its 5th order and 3 stages method.

Another type of one step methods are the Rosenbrock methods for which the idea is to linearise the RK methods in order to solve a linear system and not a nonlinear one anymore.

From the following DIRK method:

$$k_i = hf\left(y_0 + h\sum_{j=1}^{i-1} a_{ij}k_j + a_{ii}k_i\right), \quad i = 1, ..., s \tag{2.5}$$

$$y_1 = y_0 + \sum_{j=1}^{s} b_j k_i \tag{2.6}$$

applied to the autonomous function: $y' = f(y)$. A first order Taylor polynomial of (2.5) around g_i is given by:

$$\begin{aligned} k_i &= hf(g_i) + hf'(g_i)a_{ii}k_i \\ g_i &= y_0 + h\sum_{j=1}^{i-1} a_{ij}k_j \end{aligned} \tag{2.7}$$

In fact, those equations are one step of Newton's methods with $k_i^{(0)} = 0$. Instead of continuing the iterations, let's consider the system (2.7) as a method. Adding linear combinations of Jk_j and replacing $f'(g_i)$ by $J = f'(y_0)$ for numerical efficiency [5], the s-stage Rosenbrock method is given by:

$$\begin{aligned} k_i &= hf\left(y_0 + \sum_{j=1}^{i-1} \alpha_{ij}k_j\right) + hJ\sum_{j=1}^{i} \gamma_{ij}k_j, \quad i = 1, ..., s \\ y_1 &= y_0 + \sum_{j=1}^{s} b_j k_j \end{aligned} \tag{2.8}$$

with $\alpha_{ij}, \gamma_{ij}, b_i$ the coefficients of the method. Rodas4 is a 6-stages Rosenbrock method developed by Hairer [14] which is L-stable. Note that Rodas4 is the only Rosenbrock method experimented in this work.

Note that for non autonomous systems $y' = f(t, y)$, the method has to take into account the derivative with respect to time, which gives by denoting $\frac{\partial f}{\partial y}(t_0, y_0)$ as J:

$$\begin{aligned} k_i &= hf\left(t_0 + \alpha_i h, y_0 + \sum_{j=1}^{i-1} \alpha_{ij} k_j\right) + hJ \sum_{j=1}^{i} \gamma_{ij} k_j + \gamma_i h^2 \frac{\partial f}{\partial t}(t_0, y_0), \quad i = 1, ..., s \\ y_1 &= y_0 + \sum_{j=1}^{s} b_j k_j \end{aligned} \tag{2.9}$$

with $\alpha_i = \sum_{j=1}^{i-1} \alpha_{ij}$ and $\gamma_i = \sum_{j=1}^{i} \gamma_{ij}$.

2.1.2 Linear multistep methods

The idea for multistep methods is to use previous computations to calculate the new one. For linear multistep, the previous s steps with a combination of y_i and $f(t_i, y_i)$ are used. Thus, the general form for a linear multistep method is given by:

$$\sum_{j=0}^{s} a_j y_{n-j+1} = h \sum_{j=0}^{s} b_j f(t_{n-j+1}, y_{n-j+1}) \tag{2.10}$$

with $a_0 = 1$. The coefficients $a_0, \cdots, a_s$ and $b_0, \cdots, b_s$ determine the method. There exists a lot of choices for these coefficients but there are two big families: Adams and BDF methods.

Adams methods

Definition 13 *Let β_j be real coefficients. Then the method is given by:*

$$y_{n+1} = y_n + h \sum_{j=-1}^{s} \beta_j f_{n-j} \qquad (n \geq s) \tag{2.11}$$

where $f_n = f(x_n, y_n)$. If $\beta_{-1} \neq 0$, the method is implicit and called Adams-Moulton, otherwise the method is explicit and called Adams-Bashforth.

The coefficients β_j differ according to whether the formulation is explicit or implicit. Note that other Adams methods will be developed in the next section.

BDF methods

Definition 14 *Backward differentiation formulas (BDF) are given by:*

$$\sum_{j=1}^{k} \frac{1}{j} \nabla^j y_{n+1} = h f(t_{n+1}, y_{n+1}) \tag{2.12}$$

where the backward differences operator ∇^j *is defined by:*

$$\nabla^0 y_n = y_n, \qquad \nabla^{j+1} y_n = \nabla^j y_n - \nabla^j y_{n-1}$$

Those linear multistep methods suffer from the second barrier of Dahlquist (Theorem 2). Thus, they are not A-stable for orders greater than 2. However, even non A-stable methods do well in practice and that's why a weaker stability has been defined, the $A(\alpha)$ stability (Definition 10) which can be applied to BDF and gives the following table:

order k	α
1	90 °
2	90 °
3	86 °
4	73 °
5	51 °

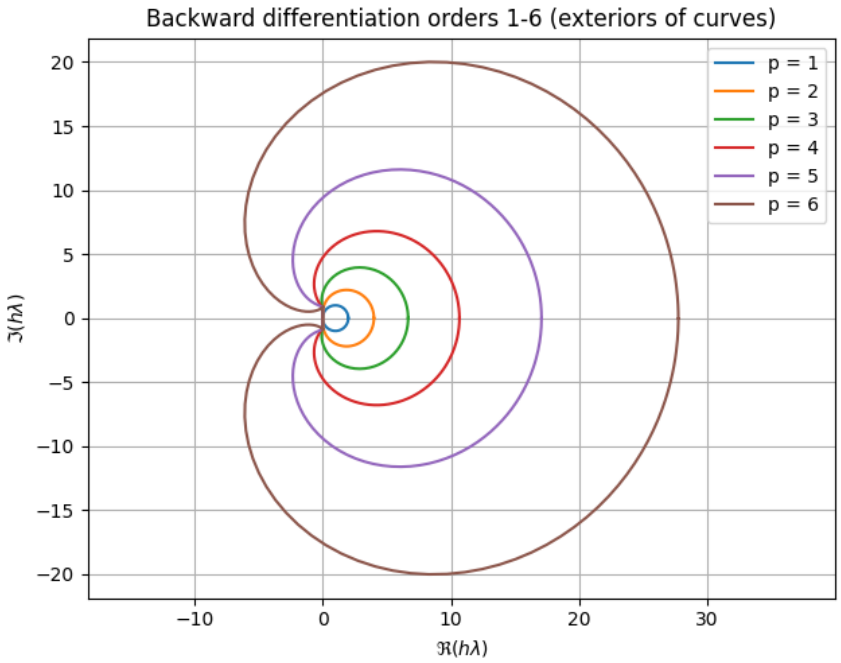

Figure 2.1: Stability regions of BDF

Higher orders are not used in practice. The stability regions of BDF methods from order 1 to 6 are described in the above figure in which the stability region is the exterior of the curves. In this figure, it can be observed that the sixth order has a very small A(α) stability and that's why only orders up to 5 are used in practice.

In this section, the most used methods have been presented but there exists a lot more numerical methods such as symplectic, extrapolation, semi-explicit or semi-implicit ones.

2.2 Modified methods for DAE

This section will cover how to modify BDF, Radau and Rosenbrock methods in order to solve the DAEs $\mathbf{M}\mathbf{y}' = \mathbf{f}(t, \mathbf{y})$. It will be shown how to improve the BDF method using a Numerical Differentiation Formulas (NDF) enhancement. Then a practical point of view will be given explaining the Newton iterations to perform in order to solve such DAEs.

2.2.1 BDF

Recall that in order to solve $y' = F(t, y)$ such that $\mathbf{y}(t_0) = y_0$, BDF can be used as follows:

$$\sum_{i=1}^{k} \frac{1}{i} \nabla^i y_{n+1} = hF(t_{n+1}, y_{n+1}) \tag{2.13}$$

However, in practice, BDF codes use the Numerical Differentiation Formulas (NDF) enhancement which adds an additional term to BDF:

$$\sum_{i=1}^{k} \frac{1}{i} \nabla^i y_{n+1} = hF(t_{n+1}, y_{n+1}) + \kappa\gamma_k(y_{n+1} - y_{n+1}^{(0)}) \tag{2.14}$$

with

- $\kappa \in \mathbb{R}$ a parameter
- $\gamma_k = \sum_{j=1}^{k} 1/j$

In fact, using the following identity:

$$y_{n+1} - y_{n+1}^{(0)} = \nabla^{k+1} y_{n+1}$$

and knowing that the truncation error of k order BDF is:

$$\frac{1}{k+1} h^{k+1} y^{(k+1)} \approx \frac{1}{k+1} \nabla^{k+1} y_{n+1}$$

the parameter κ can be chosen to reduce the truncation error of BDF which would lead to a greater step size and thus a more efficient method. But the goal is to add this new term without degrading too much its numerical stability, i.e. maximising its $A(\alpha)$ stability. This parameter has been derived numerically by Klopfenstein and Reiher giving the following table:

order k	κ	step ratio	α BDF	α NDF
1	-0.185	26 %	90 °	90 °
2	-1/9	26 %	90 °	90 °
3	-0.0823	26 %	86 °	80 °
4	-0.0415	12 %	73 °	66 °
5	0	0%	51 °	51 °

Note that they have chosen not to modify the fifth order since adding the NDF modification would have reduced the numerical stability of an order with a low stability.

Going further, using:

$$\sum_{i=1}^{k} \frac{1}{i} \nabla^i y_{n+1} = \gamma_k (y_{n+1} - y_{n+1}^{(0)}) + \sum_{i=1}^{k} \gamma_i \nabla^i y_n \tag{2.15}$$

allows to extract the unknown y_{n+1} from the sum and by injecting it in the left hand side of (2.14) leading to:

$$(1-\kappa)\gamma_k (y_{n+1} - y_{n+1}^{(0)}) + \sum_{i=1}^{k} \gamma_i \nabla^i y_n = h F(t_{n+1}, y_{n+1})$$

that can be solved by simplified Newton iterations:

$$\left(I - \frac{h}{(1-\kappa)\gamma_k} J \right) \Delta^{(i)} = \frac{h}{(1-\kappa)\gamma_k} F(t_{n+1}, y_{n+1}^{(i)}) - \Psi - (y_{n+1}^{(i)} - y_{n+1}^{(0)})$$

with

- $\Psi = \frac{1}{(1-\kappa)\gamma_k} \sum_{i=1}^{k} \gamma_i \nabla^i y_n$, known at each iteration
- $y_{n+1}^{(i+1)} = y_{n+1}^{(i)} + \Delta^{(i)}$
- $J = \frac{\partial F(t_n, y_n)}{\partial y_n}$

Now, in order to solve $My' = f(t, y)$ with a constant mass matrix, (2.14) becomes:

$$M \sum_{i=1}^{k} \frac{1}{i} \nabla^i y_{n+1} = hf(t_{n+1}, y_{n+1}) + \kappa\gamma_k M(y_{n+1} - y_{n+1}^{(0)})$$

with (2.15) updated:

$$M \sum_{i=1}^{k} \frac{1}{i} \nabla^i y_{n+1} = \gamma_k M(y_{n+1} - y_{n+1}^{(0)}) + M \sum_{i=1}^{k} \gamma_i \nabla^i y_n$$

this gives the new Newton iteration:

$$\left(M - \frac{h}{(1-\kappa)\gamma_k} J\right) \Delta^{(i)} = \frac{h}{(1-\kappa)\gamma_k} f(t_{n+1}, y_{n+1}^{(i)}) - M(\Psi + (y_{n+1}^{(i)} - y_{n+1}^{(0)}))$$

Note that solving this iteration does not require M to be nonsingular !

2.2.2 Radau

The s-stage implicit Runge Kutta method is given by:

$$\begin{aligned} k_i &= y_0 + h \sum_{j=1}^{s} a_{ij} f(t_0 + c_j h, k_j), \quad i = 1, ..., s \\ y_1 &= y_0 + h \sum_{j=1}^{s} b_j f(t_0 + c_j h, k_j) \end{aligned} \tag{2.16}$$

To reduce the influence of rounds off errors, the idea is to work with small quantities $z_i = g_i - y_0$ ($g_i = y_0 + h \sum_{j=1}^{i-1} a_{ij} k_j$, see equation 2.7) and the equation (2.16) becomes:

$$\begin{aligned} z &= A \begin{bmatrix} hf(t_0 + c_1 h, y_0 + z_1) \\ \vdots \\ hf(t_0 + c_s h, y_0 + z_s) \end{bmatrix} \\ y_1 &= y_0 + \sum_{i=1}^{s} d_i z_i \end{aligned} \tag{2.17}$$

with $(d_1, ..., d_s) = (b_1, ..., b_s)A^{-1}$, $\mathbf{d} = (0, 0, 1), b_i = a_{si}$ for RadauIIA5.

The simplified Newton resolution gives the system with matrix:

$$\begin{bmatrix} I - ha_{11}\frac{\partial f}{\partial y}(t_0 + c_1 h, y_0 + z_1) & \dots & -ha_{1s}\frac{\partial f}{\partial y}(t_0 + c_s h, y_0 + z_s) \\ \vdots & \ddots & \vdots \\ -ha_{s1}\frac{\partial f}{\partial y}(t_0 + c_1 h, y_0 + z_1) & \dots & I - ha_{ss}\frac{\partial f}{\partial y}(t_0 + c_s h, y_0 + z_s) \end{bmatrix}$$

where all the Jacobians are approximated by $J \approx \frac{\partial f}{\partial y}(t_0, y_0)$ and $\frac{\partial f}{\partial y}$ is the derivative with respect to the second variable of $f(t, y)$. In fact, the system of the Newton's method is:

$$(I - hA \otimes J)\Delta Z^k = -Z^k + h(A \otimes I)F(Z^k)$$

for $Z^{k+1} = Z^k + \Delta Z^k$ having noted $F(Z^k) = [f(t_0 + c_1 h, y_0 + z_1^k), ..., f(t_0 + c_s h, y_0 + z_s^k)]^T$. Remark, LU decomposition of $I - hA \otimes J$ is expensive but only done once.

Using the structure of $I - hA \otimes J$ and the decomposition $T^{-1}A^{-1}T = \Lambda$, the change of variable $W^k = (T^{-1} \otimes I)Z^k$ is used and the final equation becomes

$$(h^{-1}\Lambda \otimes M - I \otimes J)\Delta W^k = -h^{-1}(\Lambda \otimes M)W^k + (T^{-1} \otimes I)F((T \otimes I)W^k) \quad (2.18)$$

For a matrix A such that the eigenvalues of A^{-1} are $\hat{\gamma}, \hat{\alpha} \pm i\hat{\beta}$, the linear system matrix can be rewritten as:

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

for $\alpha = \hat{\alpha}/h$ as well as for β and γ.

Now, suppose $My' = f(t, y)$, with M a constant mass matrix. The same development as BDF can be done and the linear system matrix of (2.18) becomes:

$$\begin{bmatrix} \gamma M - J & 0 & 0 \\ 0 & \alpha M - J & -\beta M \\ 0 & \beta M & \alpha M - J \end{bmatrix}$$

Note that this matrix is a diagonal block matrix so that the two different blocks can be solved independently.

2.2.3 Rosenbrock

Solving equations (2.8) of Rosenbrock for non autonomous ODEs gives the following Newton iteration:

$$\begin{aligned}\left(\frac{1}{h\gamma_{ii}}I - J\right)u_i &= f\left(t_0 + \alpha_i, y_0 + \sum_{j=1}^{i-1} a_{ij}u_j\right) + \sum_{j=1}^{i-1}\left(\frac{c_{ij}}{h}\right)u_j + \gamma_i h\frac{\partial f}{\partial t}(t_0, y_0), \qquad i = 1, ..., s \\ y_1 &= y_0 + \sum_{j=1}^{s} m_j u_j\end{aligned} \tag{2.19}$$

such that:

- $J = \frac{\partial f}{\partial y}(t_0, y_0)$
- $k_i = \frac{u_i}{\gamma_{ii}} - \sum_{j=1}^{i-1} c_{ij}u_j$ for $C = diag(\gamma_{11}, ..., \gamma_{ss})^{-1} - \Gamma^{-1}$ where $\Gamma_{ij} = \gamma_{ij}$
- $\gamma_i = \sum_{j=1}^{i} \gamma_{ij}$ and $\alpha_i = \sum_{j=1}^{i-1} \alpha_{ij}$
- This is a change of variable such that $u_i = \sum_{j=1}^{i} \gamma_{ij}k_j$ for $i = 1, ..., s$
- $(a_{ij}) = (\alpha_{ij})\Gamma^{-1}$
- $(m_1, ..., m_s) = (b_1, ..., b_s)\Gamma^{-1}$

A small modification is required in order to solve $My' = f(t, y)$:

$$\begin{aligned}\left(\frac{1}{h\gamma_{ii}}M - J\right)u_i &= f\left(t_0 + \alpha_i, y_0 + \sum_{j=1}^{i-1} a_{ij}u_j\right) + \sum_{j=1}^{i-1}\left(\frac{c_{ij}}{h}\right)u_j + \gamma_i h\frac{\partial f}{\partial t}(t_0, y_0), \qquad i = 1, ..., s \\ y_1 &= y_0 + \sum_{j=1}^{s} m_j u_j\end{aligned} \tag{2.20}$$

Note that a more rigorous approach of the change of variable is covered in [14] (p378 and 408).

2.2.4 Semi-explicit and semi-implicit ABM for DAEs

SABM have been developed for ODEs to decrease computations costs while attaining more numerical stability. The article submitted to Mathematics (given in the next chapter) shows the computational and numerical stability advantages of SABM methods which leads to the following question: do SABM perform well for DAEs compared to the classical methods?

The DAE formulation used for this method is given by:

$$\begin{aligned} y' &= f(x, y, t) \\ 0 &= g(x, y, t) \end{aligned} \tag{2.21}$$

where y is the vector of differential variables and x is the vector of algebraic variables. For semi-explicit and semi-implicit ABM methods, the first step is to compute the predictor

$$y^p_{n+1} = y_n + h \sum_{j=1}^{k} B_j f(x_{n-j}, y_{n-j}, t_{n-j}) \tag{2.22}$$

$$0 = g(x^p_{n+1}, y^p_{n+1}, t). \tag{2.23}$$

The equation (2.22) is explicit while determining the predictor of the algebraic variables is implicit. The equation (2.23) is a system, maybe non-linear, to be solved. Then, the predictor is used to compute the corrector:

$$y_{n+1,i} = y_{n,i} + hM_1 f_i\left(x^p_{n+1}, \mathbf{y}^i_{n+1}, t_{n+1}\right) + h \sum_{j=1}^{k} M_{j+1} f_i(x_{n-j}, y_{n-j}, t_{n-j}) \tag{2.24}$$

$$0 = g(x_{n+1}, y_{n+1}, t_{n+1}) \tag{2.25}$$

where $i = 1, \cdots, d$ with d the number of differential equations and

$$\mathbf{y}^i_{n+1} = \begin{bmatrix} y_{n+1,1} & \cdots & y_{n+1,i-1} & y^p_{n+1,i} & y^p_{n+1,i+1} & \cdots & y^p_{n+1,d} \end{bmatrix} \quad \text{if semi-explicit}$$

$$\mathbf{y}^i_{n+1} = \begin{bmatrix} y_{n+1,1} & \cdots & y_{n+1,i-1} & y_{n+1,i} & y^p_{n+1,i+1} & \cdots & y^p_{n+1,d} \end{bmatrix} \quad \text{if semi-implicit}$$

For the semi-explicit method, equation (2.24) is solved using previously computed values of y_{n+1}. In other words, for the first component $y_{n+1,1}$, $\mathbf{y}^1_{n+1}$ is filled with the predicted value since no component has been computed yet. For $y_{n+1,2}$, since we have already computed $y_{n+1,1}$ we can use it instead of the predictor in $\mathbf{y}^2_{n+1}$. On the other hand, the semi-implicit method requires to solve the equation

for $y_{n+1,i}$ for each component. Therefore, for $y_{n+1,1}$, the first value of $\mathbf{y}_{n+1}^1$ is $y_{n+1,1}$ while the other are the predicted values. For the second component $y_{n+1,2}$ we have already computed $y_{n+1,1}$ so that the first component of $\mathbf{y}_{n+1}^2$ is $y_{n+1,1}$, the second is $y_{n+1,2}$ and the other ones are the predicted values.

2.3 Test problems

In order to compare the solvers, six physical/chemical problems will be developed so that they can be used as test problems at the end of this section. Note that open access notebooks have been made to show how to model those problems and how to solve them thanks to the numerical methods considered earlier[1]. Therefore, this work will only give their mathematical model and the results of the code since more details about their modelling process are covered in the notebooks.

2.3.1 Dynamical modelling with DAEs

The examples of this section are not niche problems and they are even representative of existing physical systems for which the approach of this work can be used. Moreover, they have the ability to show how to model electrical, physical and chemical problem easily with DAEs and how to solve them without derivations nor complex algebraic manipulations.

The simple pendulum

The simple pendulum problem is a pendulum without any friction represented as:

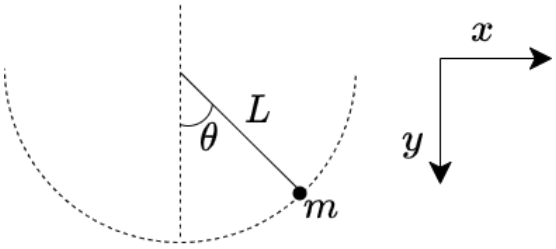

Figure 2.2: The simple pendulum

[1] `https://github.com/ocheffert/Notebooks---Master-thesis`

Its equations are given by:

$$\begin{cases} x' = u \\ y' = v \\ u' = \lambda x \\ v' = \lambda y - g \\ 0 = x^2 + y^2 - L^2 \end{cases} \qquad \begin{cases} x' = u \\ y' = v \\ u' = \lambda x \\ v' = \lambda y - g \\ L^2\lambda = gy - u^2 - v^2 \end{cases} \qquad \begin{cases} \theta' = \omega \\ \omega' = -\dfrac{g}{L} sin(\theta) \end{cases}$$

Index 3 system Index 1 system ODE system

Note that the DAEs equations are obtained following the classical modelling procedure given at the beginning of the work in section 1.1.1 with the Euler-Lagrange equation. The ODEs system has been simplified using a change of variable which shows that even if this change of variable seems trivial, this is not necessarily true for larger and more complex systems. Therefore, having a DAEs solver that can solve the original index 3 Hessenberg DAEs without any manipulations would be easier. The index 3 system is indeed an index 3 Hessenberg DAEs as explained in section 1.2.2. Note that Radau can solve Hessenberg index 3 problems. Furthermore, Proposition 1 stated that this kind of DAEs is solvable.

The stability analysis of the ODEs shows that $(\theta^*, \omega^*) = (0, 0)$ is a center (since there is no dissipation) while $(\theta^*, \omega^*) = (\pi, 0)$ is an unstable equilibrium, more precisely a saddle point. The first equilibrium is a center since the eigenvalues of the linearized system lie on the imaginary axis.

Robotic manipulator with two bodies

The robotic manipulator is a mechanical arm with two bodies without any friction on which a force F is applied and a torque T on its joint:

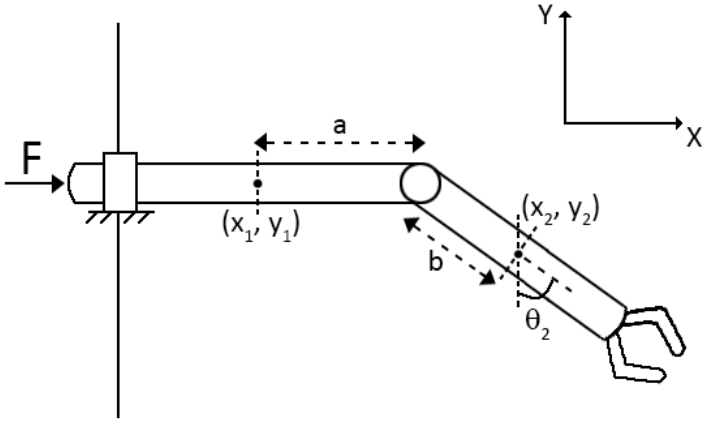

Figure 2.3: Robotic manipulator with two bodies

$$\begin{cases} m_1 x''_1 = F - \lambda_3 \\ I_2 \theta''_2 = -\lambda_3 b \cos\theta_2 - \lambda_4 b \sin\theta_2 + T \\ m_2 x''_2 = \lambda_3 \\ m_2 y''_2 = -m_2 g + \lambda_4 \\ 0 = x_2 - b\sin\theta_2 - x_1 - a \\ 0 = y_2 + b\cos\theta_2 \end{cases}$$

Index 3 system

$$\begin{cases} x'_1 = u \\ m_1 u' = F - \lambda_3 \\ x'_2 = v \\ m_2 v' = \lambda_3 \\ y'_2 = w \\ m_2 w' = -m_2 g + \lambda_4 \\ \theta'_2 = \alpha \\ I_2 \alpha' = -\lambda_3 b \cos\theta_2 - \lambda_4 b \sin\theta_2 + T \\ 0 = v' - b\alpha' \cos\theta_2 + b\alpha^2 \sin\theta_2 - u' \\ 0 = w' - b\alpha' \sin\theta_2 - b\alpha^2 \cos\theta_2 \end{cases}$$

Index 1 system

The DAEs are obtained again with the Euler-Lagrange equation given in 1.1.1 which leads to an index 3 Hessenberg DAEs as explained in section 1.2.2.

For the ODE modelling, the general dynamics of an articulated mechanical system is therefore used $M(\mathbf{q})\mathbf{q}' + f(\mathbf{q}, \mathbf{q}') + g(\mathbf{q}') = G(\mathbf{q}')u$ as explained in the corresponding notebook. The ODEs expression is not very readable, this is a further example in which the DAEs expression is a more natural way to model a dynamical system.

The stability analysis concludes that F^* must be zero since otherwise x_1 would never be in equilibrium (recall that there is no friction). The equilibrium are $(x_1^*, \theta_2^*) = (x_1^*, asin(\frac{T^*}{b*g*m_2}))$ and $(x_1^*, \pi - asin(\frac{T^*}{b*g*m_2}))$. For a nil torque T^*, this becomes exactly the same as the pendulum. Following a similar stability analysis, the first equilibrium is stable while the second is unstable.

Low-pass filter

The following electrical filter is a low-pass filter that reduces the amplitude of the input voltage V_{in} to give a 10 times reduced output signal V_{out}:

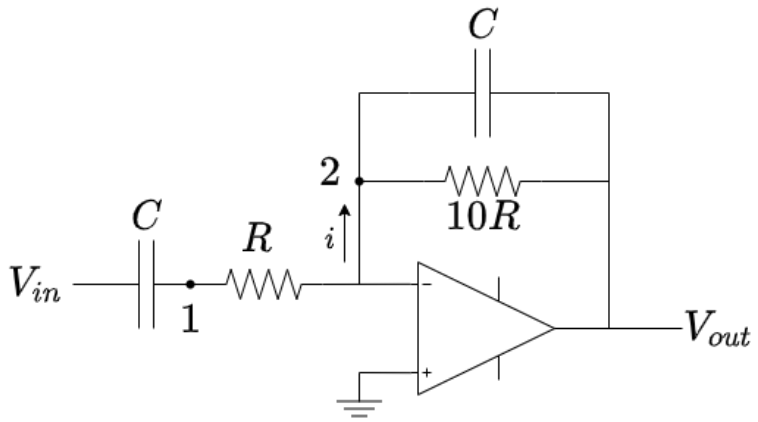

Figure 2.4: Low-pass filter

The equations of this system are obtained following the ideas of 1.1.1, i.e. Kirchhoff's laws:

$$\begin{cases} V'_1 = V'_{in} - \dfrac{i}{C} \\ V'_{out} = -\dfrac{V_{out}}{10RC} - \dfrac{i}{C} \\ 0 = i\ R - V_1 \end{cases} \qquad \begin{cases} V'_1 = V'_{in} - \dfrac{V_1}{RC} \\ V'_{out} = -\dfrac{V_{out}}{10RC} - \dfrac{V_1}{RC} \end{cases} \tag{2.26}$$

Index 1 system ODE system (2.27)

The stability analysis based on the ODEs shows that the system is stable since the eigenvalues $\left(\frac{-1}{RC}\right.$ and $\left.\frac{-1}{10RC}\right)$ are negative real numbers.

Transistor amplifier

The following circuit, based on a transistor, amplifies the input voltage $U_e(t)$ to obtain the output voltage $U_5(t)$:

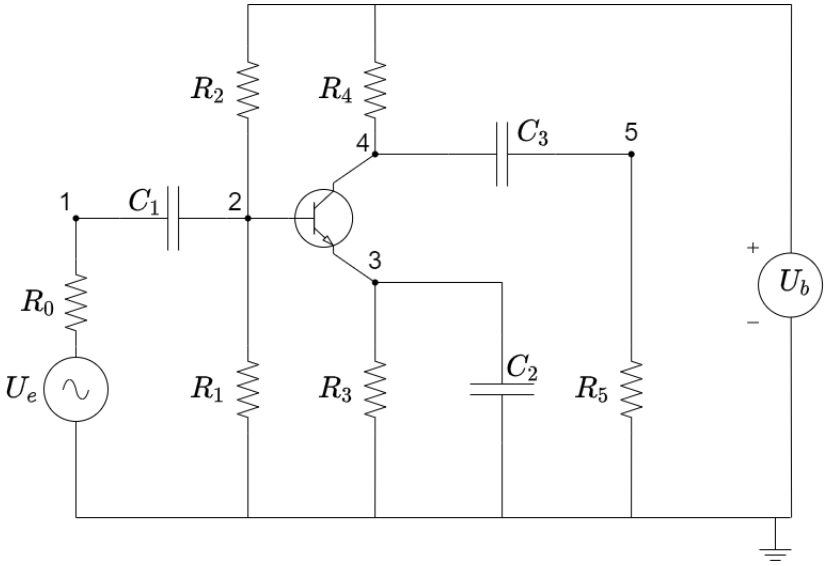

Figure 2.5: Transistor amplifier

This circuit can be explained by the following points:

- U_b is a constant voltage source.
- C_1, C_3 remove the DC signal and can be viewed as filters that only keep the AC signals since the input and output are AC.
- R_1, R_2 keep the transistor in active mode
- C_2 is a decoupling capacitor which is open with DC current and a short circuit for AC current. Therefore, the gain of the transistor is stabilized.

The Kirchhoff's current laws for nodes 1 to 5 are given by:

1. $\frac{U_e - U_1}{R_0} = C_1(U_1' - U_2')$
2. $\frac{U_b - U_2}{R_2} + C_1(U_1' - U_2') = \frac{U_2}{R_2} + 0.01F(U_2 - U_3)$
3. $F(U_2 - U_3) = \frac{U_3}{R_3} + C_2 U_3'$
4. $\frac{U_b - U_4}{R_4} = C_3(U_4' - U_5') + 0.99F(U_2 - U_3)$
5. $C_3(U_4' - U_5') = \frac{U_5}{R_5}$

with

- $F(U) = 10^{-6}\left(exp\left(\frac{U}{0.026}\right) - 1\right)$
- $R_0 = 1000[\Omega], R_i = 9000[\Omega] \quad (i = 1...5)$
- $C_i = i \cdot 10^{-6}[F] \quad (i = 1, 2, 3)$
- the chosen initial signal: $U_e(t) = 0.4 \cdot sin(200\pi t)$

Note that this system is already written under the form $My' = f(t, y)$ with:

$$M = \begin{bmatrix} -C_1 & C_1 & & & \\ C_1 & -C_1 & & & \\ & & -C_2 & & \\ & & & -C_3 & C_3 \\ & & & C_3 & -C_3 \end{bmatrix}$$

This matrix has no null row and thus requires a change of variable to obtain an ODEs system. This ODEs system, its change of variable and the semi-explicit DAE are given in the notebook. However, the ODEs system is complex and composed of non intuitive equations while the DAEs $My' = f(t, y)$ are more intuitive being a direct application of the Kirchhoff's laws. As explained earlier, the change of variable makes appear variables which do not have any physical meaning.

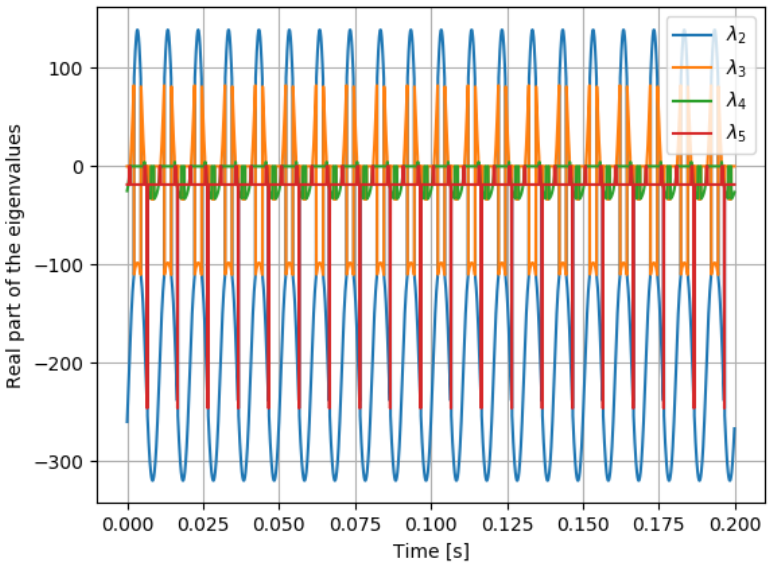

Figure 2.6: Stability of the transistor amplifier

A stability analysis concludes that the system oscillates between stable and unstable states by looking at the time dependent eigenvalues in Figure 2.6. The first eigenvalue is always stable and a large negative number (oscillating between -4000 and -6000) not represented in the plot for clarity.

Robertson's problem

The Robertson problem is a very stiff problem in which there is one very slow reaction and one very fast. It makes sense that L-stable solvers are needed but this needs to be verified. The chemical reactions are given by:

$$A \xrightarrow{k_1} B$$
$$B + B \xrightarrow{k_2} C + B$$
$$B + C \xrightarrow{k_3} A + C$$

A, B, C are the chemical species and k_i are the kinetic coefficients. The mathematical model of the variation of their concentrations C_A, C_B and C_C is given by:

$$\begin{cases} C'_A = -k_1 C_A + k_3 C_B C_C \\ C'_B = k_1 C_A - k_3 C_B C_C - k_2 C_B^2 \\ 1 = C_A + C_B + C_C \end{cases}$$
Index 1 system

$$\begin{cases} C'_A = -k_1 C_A + k_3 C_B C_C \\ C'_B = k_1 C_A - k_3 C_B C_C - k_2 C_B^2 \\ C'_C = k_2 C_B^2 \end{cases}$$
ODE system

The stability analysis of this problem shows that the equilibria are given by $(C_A^*, C_B^*, C_C^*) = (0, 0, a)$ for a real a. It's another example in which there are an infinite number of solutions for ODEs while only one satisfies the DAEs: $(C_A^*, C_B^*, C_C^*) = (0, 0, 1)$. Since their eigenvalues are $(\lambda_1, \lambda_2, \lambda_3) = (0, 0, -k_1 - k_3)$, this is a stable, non isolated attractive equilibrium. The last eigenvalue is a big negative number for a very fast third reaction meaning that the problem becomes very stiff.

Controlled continuous stirred tank reactor

The last example shows how to control a continuous stirred tank reactor (CSTR) thanks to a PID controller. This CSTR can be illustrated as follows:

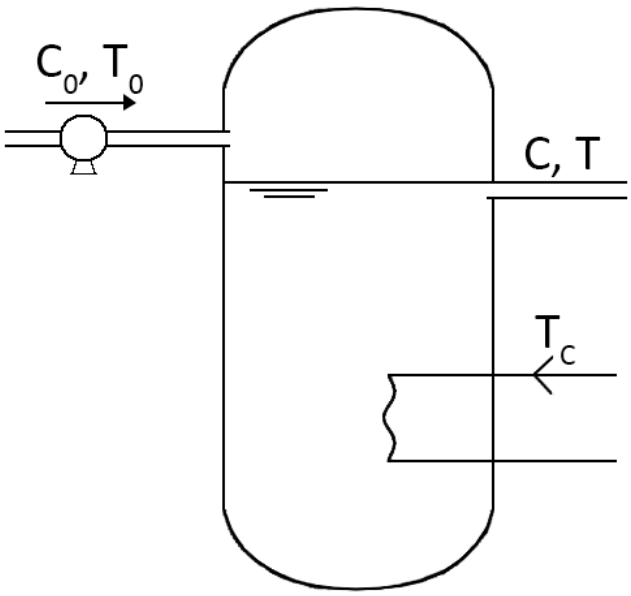

Figure 2.7: Controlled continuous stirred tank reactor

This CSTR has an input of concentration C_0 and temperature T_0 with an output concentration C and an output temperature T. The reaction inside the tank is controlled by a cooling circuit whose temperature T_C is chosen to obtain the desired output concentration C_{sp}.

The equations of the controlled system in index 1 DAE format are given by:

$$\begin{cases} C' = K_1(C_0 - C) - K_3 exp\left(\dfrac{-K_4}{T}\right) C \\ T' = K_1(T_0 - T) + K_2 K_3 exp\left(\dfrac{-K_4}{T}\right) C - K_3(T - T_C) \\ I' = e(t) \\ T_C = K_p e(t) + K_i I(t) + K_d e'(t) \end{cases}$$

with C and T the concentration and temperature of the output, T_C the control variable based on the error $e(t) = C_{sp} - C$ for a set-point C_{sp}. $I(t)$ is the integral of the error over time. Therefore, the last equation clearly states that the system is controlled by a PID.

The ODEs of the controlled CSTR is given by:

$$\begin{cases} C' = K_1(C_0 - C) - K_3 exp\left(\frac{-K_4}{T}\right) C \\ T' = K_1(T_0 - T) + K_2 K_3 exp\left(\frac{-K_4}{T}\right) C - K_3(T - T_C) \\ T'_C = K_p e'(t) + K_i e(t) + K_d e''(t) \end{cases}$$

In the notebook, we have assumed that $K_d = 0$ since it avoids the calculation of $e''(t)$ in the ODEs and the PI controller is sufficient to cancel the static error. This choice has been made for the ODEs since the DAEs does not need the expression $e''(t)$.

The stability analysis has been realized for the particular values chosen and leads to the fact that this system is stable. Indeed, the eigenvalues (computed with SymPy) have negative real parts.

2.3.2 Results

Now, all the necessary tools have been assembled to study the performance of those solvers developed previously on the six test problems. Time is not considered as a performance metric since it can be deceived by the implementation choices (discussed in a later section) and the language (dynamically or statically typed language). In this work, the main metric will be the number of evaluations of the right hand side function(s). Solving ODEs by the Adams-Bashforth-Moulton method allows to integrate computationally heavy functions since the number of function evaluations is lower for this method than the other ones. Thus, this part will check if this property is conserved when solving DAEs.

Note that the results are summarized in the tables presented in this section but more complete results can be found in the excel document of the GitHub repository accompanying this work [2].

The tolerance required by the user will vary to observe its impact on the number of function evaluations. The function evaluated are the right hand side functions for BDF, Radau and Rosenbrock while the SABM solvers uses the number of evaluations of the right hand side function of the differential part. Indeed, it's observed that the constraints function is evaluated much more than for the classical

[2] https://github.com/ocheffert/Notebooks—Master-thesis

methods. Therefore, it's assumed for the rest of this part that the constraints are easier to solve than the functions of the differential parts. For those problems, it's always the case since the algebraic variables appear linearly in the constraints. This topic will be discussed in the next chapter about the implementation and their possible improvements. Furthermore, SABM methods have been tested with a fixed order, BDF uses a variable order method which will be discussed in a later section. The order of SABM methods has been chosen as the second order since this order has the best stability as the article will prove in the next chapter.

Note that all the solvers keep the local error estimate less than $atol + rtol \cdot abs(y)$ at any time for an absolute tolerance $atol$ and a relative tolerance $rtol$. Varying the tolerance means that the relative tolerance will vary while the absolute tolerance will be fixed as a factor of the relative tolerance, i.e. $atol = rtol \cdot k$ for a factor k chosen according to the order of magnitude of the problem.

Pendulum

The behaviour of this mechanical system over one period is plotted in Figure 2.8 using the Radau solver:

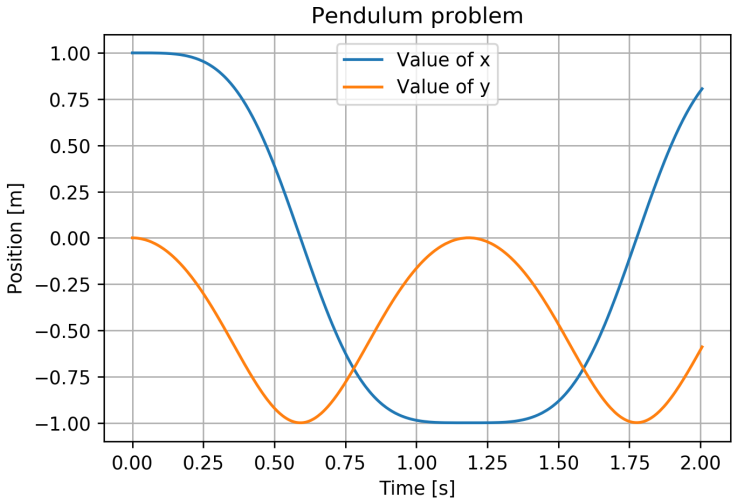

Figure 2.8: Solution with Radau DAE ($rtol = atol = 10^{-3}$)

Table 2.1 allows to compare the numerical methods based on their number of function evaluations taking into account the approximation of the Jacobian. The

column "Bests" describes the minimum number of function evaluations for the required tolerance among the 5 methods. The rest of the table is filled as by factors describing how far is the method with respect to the best number of function evaluations. A low value means that the best number is small with respect to the one of the method while the value 1 means that the method is the one with the minimum number of function evaluations. The value *NA* means that the method does not converge and a grey cell that the method takes too much time.

Rtol	**Bests**	**BDF**	**Radau**	**Rosenbrock**	**SIABM**	**SEABM**
1.00E-01	90	0.346	0.319	0.433	0.372	1
1.00E-02	235	0.635	0.768	0.822	0.366	1
1.00E-03	372	0.622	1	0.853	0.188	0.520
1.00E-04	690	0.791	1	1	0.118	0.324
1.00E-05	1188	0.983	1	0.674	0.065	0.179
1.00E-06	1618	1	0.823	0.287	0.047	NA
1.00E-07	2289	1	0.737	0.119	NA	NA
1.00E-08	3284	1	0.648	0.054	NA	NA

Table 2.1: Function evaluation for pendulum problem ($atol = rtol \cdot 10^{-2}$)

Table 2.1 shows that SEABM reaches the minimum number of function evaluations at high tolerances while BDF and Radau obtained the best number at low tolerances. Rosenbrock needs few function evaluations for medium tolerances compared to the respective optima while SIABM requires much more than the other methods. Those results must be studied by taking a look at the following error table which gives the error between the numerical method and a reference solution. The reference solution is obtained by solving the ODEs with a strict tolerance of 10^{-12}.

Table 2.2 shows that where Radau needs more function evaluations than BDF, it reaches a much better accuracy. Therefore, for a certain accuracy, Radau is more optimal than BDF because it requires a smaller *rtol* and therefore a lower number of function evaluations. Indeed, reaching an error of 10^{-7} is possible with 1188 function evaluations with Radau ($rtol = 10^{-5}$) while BDF needs 3284 evaluations with $rtol = 10^{-8}$. This behaviour is explained by the stability region of the methods since Radau is L-stable which covers the pure imaginary eigenvalues of the pendulum problem while BDF does not cover the imaginary axis for all of its orders. In fact, the Rosenbrock method which is a linearized version of Runge-Kutta reaches a better accuracy with respect to BDF. On the other hand, the SABM methods reach a better accuracy than BDF and a slightly better one compared to Rosenbrock at high *rtol* with an equal one at low *rtol*.

Rtol	BDF	Radau	Rosenbrock	SIABM	SEABM
1.00E-01	4.00E-01	3.50E-02	1.20E-01	3.00E-02	3.00E-02
1.00E-02	1.00E-01	8.00E-04	2.50E-02	7.00E-03	7.00E-03
1.00E-03	2.00E-02	6.00E-05	2.50E-03	2.50E-03	2.50E-03
1.00E-04	1.00E-03	1.50E-06	1.75E-04	3.00E-04	3.50E-04
1.00E-05	6.00E-05	1.50E-07	1.00E-05	4.00E-05	4.00E-05
1.00E-06	1.00E-05	5.00E-09	5.00E-06	8.00E-07	
1.00E-07	1.00E-06	1.00E-10	4.00E-07		
1.00E-08	1.00E-07	1.00E-11	8.00E-08		

Table 2.2: Estimated error for pendulum problem ($atol = rtol \cdot 10^{-2}$)

Note that Radau is able to solve directly the index 3 Hessenberg formulation but with 6 times more function evaluations than when solving the index 1 formulation. However, solving the index 3 formulation requires to choose carefully $rtol, atol$ in order to make the solver converge.

Robot

The behaviour of this mechanical system over a time $[0; 8]$ is plotted in Figure 2.9 using the Radau solver:

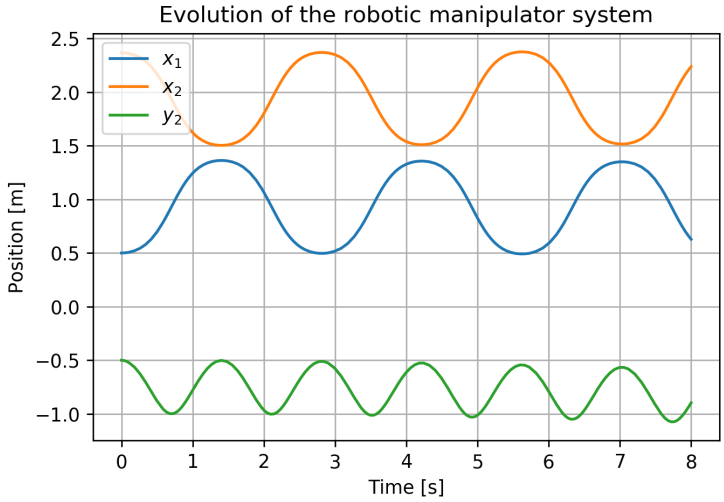

Figure 2.9: Solution with Radau DAE ($rtol = atol = 10^{-3}$)

Note that in this simulation, there is no friction, no torque and no external force applied on the robot. If the applied force is a step function, the BDF solver fails. This is due to the multistep approach of the method. On the other hand, all the other solvers work which is surprising since SABM are linear multistep methods.

Rtol	Bests	BDF	Radau	Rosenbrock	SIABM	SEABM
1.00E-01	234	0.154	0.189	0.260	0.365	1
1.00E-02	577	0.223	0.356	0.407	0.363	1
1.00E-03	1556	0.449	0.847	0.666	0.362	1
1.00E-04	2910	0.503	1	0.544	0.212	0.581
1.00E-05	5084	0.653	1	0.412	0.120	0.331
1.00E-06	8929	0.820	1	0.265	0.068	0,185
1.00E-07	14237	1	0.907	0.137	0.038	0.094
1.00E-08	20497	1	0.739	0.062	0.054	0.135

Table 2.3: Function evaluation for robotic manipulator problem ($atol = rtol \cdot 10^{-2}$)

Table 2.3 shows the same behaviour as the previous problem in which SEABM performs well for high tolerances, Radau for medium tolerances and BDF for low tolerances. But, based on the results of Table 2.4, the BDF method is not a good option to obtain the best accuracy. For example, for a low tolerance $rtol = 10^{-8}$, BDF achieves an estimated error of 2.6 10^{-6} while Radau obtains 6.0 10^{-11}. Rosenbrock, that should be used for high tolerances, does not seem to achieve better performance than SEABM. Similarly to the pendulum problem, it seems that the BDF solver has more difficulty in solving a problem whose eigenvalues are pure imaginary compared to Radau.

Rtol	BDF	Radau	Rosenbrock	SIABM	SEABM
1.00E-01	1.00E+00	1.10E-01	1.40E+00	4.00E-01	4.00E-01
1.00E-02	1.20E-01	2.00E-02	4.00E-02	8.00E-02	8.00E-02
1.00E-03	7.00E-02	3.20E-03	3.60E-03	1.40E-02	1.40E-02
1.00E-04	1.60E-02	1.20E-04	2.30E-04	8.00E-04	1.00E-03
1.00E-05	1.40E-03	2.30E-06	3.00E-05	1.60E-04	1.50E-04
1.00E-06	1.60E-04	6.50E-08	3.00E-06	2.00E-05	2.00E-05
1.00E-07	2.50E-05	1.85E-09	6.90E-07	2.00E-06	2.00E-06
1.00E-08	2.60E-06	6.00E-11	1.60E-07	2.00E-06	2.00E-05

Table 2.4: Estimated error for robotic manipulator problem ($atol = rtol \cdot 10^{-2}$)

Low-pass filter

The behaviour of the low-pass filter computed by Radau is plotted below:

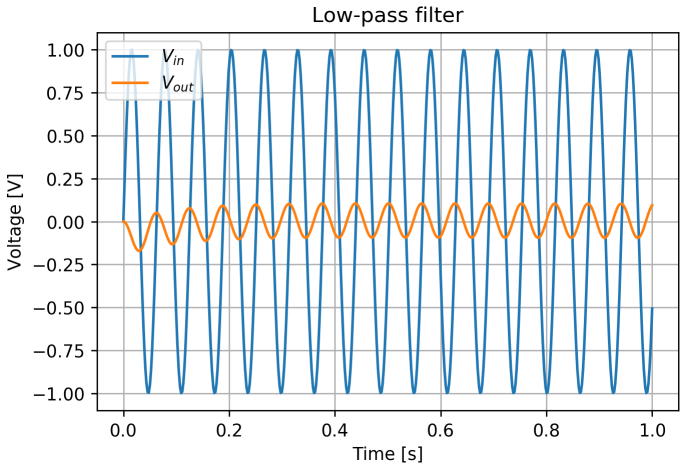

Figure 2.10: Solution with Radau DAE ($rtol = atol = 10^{-5}$)

Table 2.5 shows that Radau performs better than any other solver in terms of function evaluations. Even if BDF seems to satisfy low tolerances with less function evaluations, it's at the cost of a much bigger error than Radau. Surprisingly, neither Rosenbrock nor SEABM competes with Radau at high tolerances. This can be explained by the fact that the eigenvalues (-1 and -10 for the values considered) can be handled easier by Radau which is A-stable than SABM for which the stability region of a triangular matrix $\mathbf{A}$ in $\mathbf{x}' = \mathbf{A}\,\mathbf{x}$ (see equation 2.26) has the smallest coverage (as proven in section 3.3 with $k = 0$) and deals with such eigenvalues with more difficulty. On the other hand, Rosenbrock, a linearized version of Runge-Kutta, does not handle this type of problems with oscillations as well as RK because it's less accurate and thus needs more steps to converge.

Rtol	Bests	BDF	Radau	Rosenbrock	SIABM	SEABM
1.00E-01	419	0.385	1	0.499	0.153	0.313
1.00E-02	665	0.405	1	0.460	0.094	0.192
1.00E-03	1091	0.404	1	0.459	0.049	0.098
1.00E-04	2278	0.612	1	0.576	0.033	0.066
1.00E-05	3688	0.743	1	0.527	0.017	0,035
1.00E-06	6796	1	0.947	0.559	0.010	0,021
1.00E-07	9524	1	0.747	0.437	0.005	0.009
1.00E-08	13625	1	0.694	0.351	0.002	0.004

Table 2.5: Function evaluation for filter problem ($atol = rtol \cdot 10^{-2}$)

Then, by looking at Table 2.6, it seems that the cautious step size strategy of SABM do not perform as well as the eager one of BDF. Indeed, compared to BDF, the accuracy of SABM are one higher order than the one of BDF. Thus, a more aggressive strategy would accept to loose some accuracy to perform the computations faster. As the previous problems, Radau reaches better accuracy than the other methods.

Rtol	BDF	Radau	Rosenbrock	SIABM	SEABM
1.00E-01	7.00E-02	3.50E-04	2.50E-03	8.00E-03	6.50E-03
1.00E-02	1.40E-02	1.40E-05	3.50E-05	2.00E-03	2.00E-03
1.00E-03	6.00E-03	2.50E-06	3.00E-06	2.50E-04	3.50E-04
1.00E-04	4.00E-04	8.00E-08	2.00E-07	7.00E-05	6.65E-05
1.00E-05	3.00E-05	3.00E-09	1.60E-08	1.00E-05	1.04E-05
1.00E-06	3.00E-06	5.00E-10	1.75E-09	1.20E-06	1.24E-06
1.00E-07	2.50E-07	2.00E-11	2.00E-10	1.20E-07	1.20E-07
1.00E-08	4.00E-08	2.00E-12	6.00E-11	1.20E-08	1.15E-08

Table 2.6: Estimated error for filter problem ($atol = rtol \cdot 10^{-2}$)

Transistor amplifier

The behaviour of the transistor amplifier filter is plotted in Figure 2.10 using the Radau solver:

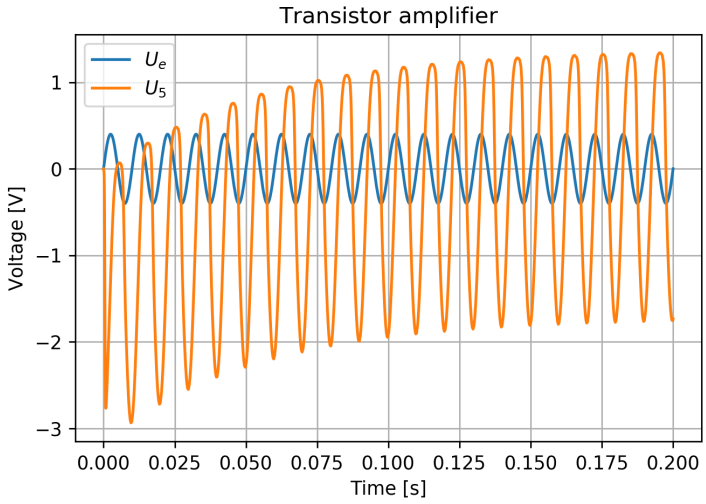

Figure 2.11: Solution with Radau DAE ($rtol = atol = 10^{-5}$)

Recall from section 2.3.1 that this problem has a negative eigenvalue λ large in absolute value and that it has oscillations as shown in the above figure. Thus every method will try to integrate this problem using a time step h such that $h\lambda$ is inside the stability domain but where the amplification factor is near 1 in order to reach a better accuracy. Indeed, an amplification factor near to 0 would cause the method to converge to a value. Since Radau and Rosenbrock are L-stable and BDF has a small amplification factor as $h\lambda$ becomes very large in absolute value (recall the contour plot of implicit Euler which is first order BDF in Figure 1.5), they need a very small time step. On the other hand, the cautious step size strategy of SABM will make the solver use small step size anyway. That's why SEABM performs (as Table 2.7 shows) well compared to others when solving the transistor amplifier while the filter was penalizing its step size strategy and was the worst case for the stability region of SABM.

Rtol	**Bests**	**BDF**	**Radau**	**Rosenbrock**	**SIABM**	**SEABM**
1.00E-01	1917	NA	0.451	0.599	0.626	1
1.00E-02	3755	0.490	0.546	0.589	0.557	1
1.00E-03	7916	NA	1	0.659	0.376	0.719
1.00E-04	12126	0.789	1	0.564	0.180	0.355
1.00E-05	19350	0.829	1	0.501	0.091	0.181
1.00E-06	31233	1	0.982	0.459	0.046	0.092
1.00E-07	54500	NA	1	0.450	0.026	0.051
1.00E-08	94730	NA	1	0.428	NA	NA

Table 2.7: Function evaluation for transistor problem ($atol = rtol \cdot 10^{-2}$)

Again, at low tolerances, Radau achieves the best performance. While SIABM and SEABM solve correctly until low tolerances, BDF struggles to solve this problem at low and high tolerances and can only solve it at medium tolerances. In addition, from Table 2.8, it seems that BDF is less accurate that any other method when it's able to find a solution at medium tolerances. Furthermore, note that for $rtol = 10^{-1}$, only Rosenbrock is able to find a solution with a decent error.

Rtol	BDF	Radau	Rosenbrock	SIABM	SEABM
1.00E-01		7.00E+00	3.00E-01	1.00E+01	1.00E+01
1.00E-02	9.00E-02	1.75E-01	1.00E-01	1.00E-02	3.00E-02
1.00E-03		5.00E-03	1.00E-02	1.00E-03	3.00E-03
1.00E-04	1.75E-03	3.00E-04	4.00E-04	1.00E-04	2.00E-04
1.00E-05	2.50E-04	2.00E-05	3.00E-04	2.00E-05	2.00E-05
1.00E-06	1.20E-05	8.00E-07	3.50E-05	3.00E-06	3.00E-06
1.00E-07		2.50E-08	2.00E-06	4.00E-07	5.00E-07
1.00E-08		1.20E-09	5.50E-08		

Table 2.8: Estimated error for transistor problem ($atol = rtol \cdot 10^{-2}$)

Robertson's problem

Here is the behaviour of the Robertson chemical problem in which the concentration of product B has been scaled up by a factor 10^4:

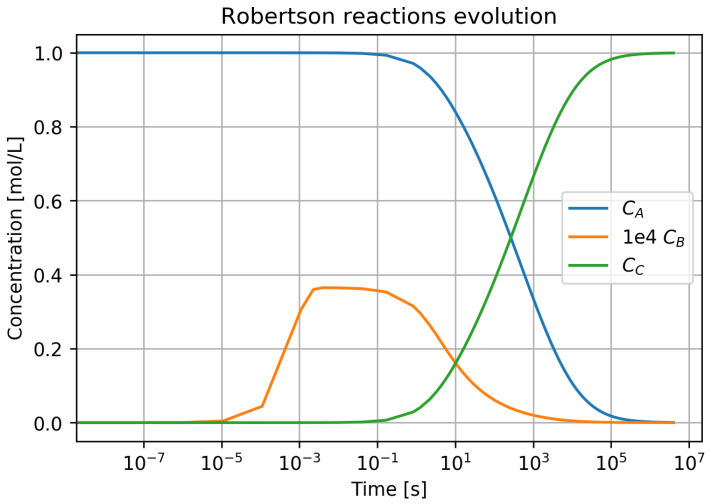

Figure 2.12: Solution with Radau DAE ($rtol = atol = 10^{-5}$)

For the Robertson's problem, the factor between $rtol$ and $atol$ has been chosen as 10^{-4} since C_B is of order 10^{-4}. Indeed, if the local error is kept below $atol + rtol \cdot abs(y)$, $atol$ must be sufficiently small to integrate with accuracy the concentration C_B. This factor has also been used by Hairer in his numerical experiments [14]. However, taking a small $rtol$ will put the program in difficulty because the estimated local error will have to satisfy a very strict condition which explains why BDF and

Radau no longer converge from a certain *rtol* and Rosenbrock does not integrate the problem in less than 1 hour as observed on Table 2.9.

Rtol	Bests	BDF	Radau	Rosenbrock	SIABM	SEABM
1.00E-01	133	1	0.413	0.731	0.002	1.72E-05
1.00E-02	248	0.929	0.858	1	0.004	3.17E-05
1.00E-03	290	0.629	1	0.450	0.005	3.72E-05
1.00E-04	554	0.773	1		0.008	6.02E-05
1.00E-05	66175	NA	NA		1	0.008
1.00E-06	88466	NA	NA		1	0.011
1.00E-07	154112	NA	NA		1	0.020
1.00E-08	354725	NA	NA		1	0.045

Table 2.9: Function evaluation for Robertson's problem ($atol = rtol \cdot 10^{-4}$)

As for SEABM, it manages to reach the best accuracy (see Table 2.10) but at the cost of a huge number of steps which makes sense as ABM methods are not designed for stiff problems. Here, the cautious step size strategy allows to converge towards a solution by using very small step sizes. Note that Runge-Kutta methods such as Radau and Rosenbrock seem to have more difficulty to integrate this stiff problem probably because they need more accurate computations than BDF to converge. Indeed, BDF uses less function evaluations and converges with less accuracy while Radau needs much more accurate values to converge as it can be observed in the error table.

Rtol	BDF	Radau	Rosenbrock	SIABM	SEABM
1.00E-01	1.40E-02	9.30E-04	5.00E-04	2.00E-01	1.00E-01
1.00E-02	1.90E-03	5.00E-05	1.75E-05	2.00E-02	8.00E-03
1.00E-03	2.80E-04	1.10E-05	9.00E-06	1.50E-03	8.00E-06
1.00E-04	4.20E-05	8.30E-07		1.40E-04	8.00E-08
1.00E-05				3.50E-05	1.20E-09
1.00E-06				1.00E-05	6.00E-10
1.00E-07				1.75E-06	5.00E-10
1.00E-08				3.00E-07	5.00E-10

Table 2.10: Estimated error for Robertson's problem ($atol = rtol \cdot 10^{-4}$)

Controlled continuous stirred tank reactor

The behaviour of the controlled continuous stirred tank reactor is plotted in Figure 2.13 using the Radau solver.

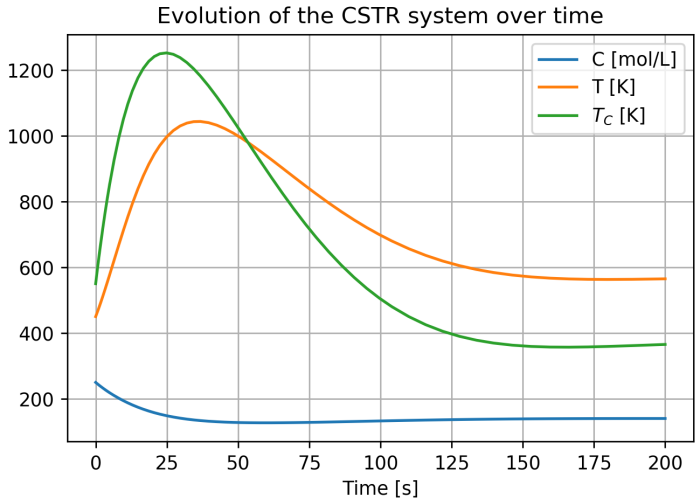

Figure 2.13: Solution with Radau DAE ($rtol = atol = 10^{-5}$)

Note that for a set-point $C_{sp} = 140[mol/L]$, the solver obtains a final value $C_f = 140.31[mol/L]$. Figure 2.13 shows that the curves are smoother than for the previous problems which means that the solvers should solve this problem more easily. Indeed, Radau does only need 184 steps to solve this problem with $rtol = 10^{-8}$. From Table 2.11, it's evident that Radau is the best solver for this problem since it has always the minimal number of function evaluations with a better accuracy than BDF and Rosenbrock as shown on Table 2.12.

Rtol	Bests	BDF	Radau	Rosenbrock	SIABM	SEABM
1.00E-01	78	0.69	1	0.678	0.152	0.311
1.00E-02	90	0.481	1	0.647	0.061	0.123
1.00E-03	137	0.521	1	0,688	0,031	0.062
1.00E-04	193	0.497	1	0.682	0.014	0.028
1.00E-05	291	0.562	1	0.663	0.007	0.013
1.00E-06	470	0.666	1	0.593	0.003	0.007
1.00E-07	772	0.79	1	0.454	NA	NA
1.00E-08	1311	0.958	1	0.226	NA	NA

Table 2.11: Function evaluation for CSTR problem ($atol = rtol$)

However, even if SABM require much more function evaluations, they achieve a better accuracy than every other method (even Radau). This is further evidence that the strategy is too cautious and could afford to be less accurate in order to be faster, especially on such smooth problems as this one. Like the filter problem, which such smooth dynamics, this behaviour is due to the fact that the three classical methods allow themselves to use larger steps. About this issue, ideas for improvement will be discussed in the next part.

Rtol	**BDF**	**Radau**	**Rosenbrock**	**SIABM**	**SEABM**
1.00E-01	5.00E+01	1.40E+00	3.00E+00	2.00E-01	3.00E-01
1.00E-02	7.00E+00	2.20E+00	8.00E-01	3.00E-02	5.00E-02
1.00E-03	1.00E+00	7.00E-02	1.00E-01	5.00E-03	7.00E-03
1.00E-04	1.75E-01	4.00E-03	1.20E-02	7.00E-04	8.00E-04
1.00E-05	2.50E-02	5.00E-04	1.20E-03	7.00E-05	8.00E-05
1.00E-06	2.50E-03	2.50E-05	2.50E-04	7.00E-06	8.00E-06
1.00E-07	4.00E-04	1.60E-06	7.00E-05		
1.00E-08	7.00E-05	7.50E-08	8.00E-06		

Table 2.12: Estimated error for CSTR problem ($atol = rtol$)

Chapter 3

Stability analysis SABM

To complete the analysis of SABM methods, it's necessary to develop the mathematical expressions needed to investigate the stability regions of semi-explicit and semi-implicit ABM and AB-BDF methods in Section 3.1. In Section 3.2, those mathematical developments will be used to plot the stability regions of these methods. Moreover, a performance comparison between those modified methods and conventional linear multistep methods will be given. Finally, a discussion of the obtained results is given in the section 3.3.

Note that all sections (except 3.4) correspond to an article written with Aleksandra tutueva (LETI), Pr. Denis Butusov (LETI) and Pr. Vincent Legat (UCLouvain) submitted to the MDPI journal Mathematics.

3.1 Materials and Methods

Semi-explicit and semi-implicit modifications of the ABM methods have been developed to gain computational efficiency and numerical stability. First, the semi-explicit modification aims to keep the computational efficiency of the explicit scheme while extending its numerical stability [8]. For the semi-implicit modification, the goal is to keep good numerical stability while reducing the computational cost. Indeed, a fully implicit scheme requires solving at each step a large-scale system for each variable while the semi-implicit modification will solve each equation separately for one variable at a time.

Let us consider semi-explicit and semi-implicit modifications of the predictor-

corrector Adams–Bashforth–Moulton formula as described in [33]. In this section, we use the same methodology as in [32]. In the original ABM integration, all calculations are explicit. Furthermore, the stability of such an integration technique is higher than that of the explicit Adams–Bashforth method [33].

3.1.1 Semi-explicit and semi-implicit ABM methods

Following the [32], we apply the two methods on a sample ODE system:

$$\begin{cases} x' = f(x, y, t) \\ y' = g(x, y, t) \end{cases} \tag{3.1}$$

The way how these equations are treated defines the unique properties of these 65 methods: decreasing computations costs while attaining more numerical stability.

The semi-explicit and semi-implicit methods are predictor-corrector methods (like the initial ABM method) and their predictor stage is computed with the Adams-Bashforth method:

$$\begin{aligned} x^p_{n+1} &= x_n + h\sum_{i=1}^{k} B_i f(x_{n-i}, y_{n-i}, t_{n-i}) \\ y^p_{n+1} &= y_n + h\sum_{i=1}^{k} B_i g(x_{n-i}, y_{n-i}, t_{n-i}) \end{aligned}, \tag{3.2}$$

where B_i are the coefficients of the Adams-Bashforth method. For the corrector stage, we use the predicted value and the values obtained during this step to obtain the solution.

Semi-explicit ABM method

The corrector of the semi-explicit scheme is given by:

$$\begin{aligned} x_{n+1} &= x_n + hM_1 f\left(x^p_{n+1}, y^p_{n+1}, t_{n+1}\right) + h\sum_{i=1}^{k} M_{i+1} f(x_{n-i}, y_{n-i}, t_{n-i}) \\ y_{n+1} &= y_n + hM_1 g\left(x_{n+1}, y^p_{n+1}, t_{n+1}\right) + h\sum_{i=1}^{k} M_{i+1} g(x_{n-i}, y_{n-i}, t_{n-i}) \end{aligned}, \tag{3.3}$$

where M_i are the coefficients of the Adams-Moulton method, k the number of stages, h the integration step and t the time moment. Note that x^p and y^p are the predicted value of x and y. Observe that all the computations are explicit because there is no equation to solve since x^p_{n+1}, y^p_{n+1} and x_n are known in the first equation of system (3.3). For the second equation, we use the already obtained and corrected value x_{n+1}. Therefore, the considered method is a semi-explicit integrator.

Semi-implicit ABM method

On the other hand, the semi-implicit method requires an implicit calculation using the Newton's method for each equation. Its corrector is as follows:

$$\begin{aligned} x_{n+1} &= x_n + hM_1 f\left(x_{n+1}, y^p_{n+1}, t_{n+1}\right) + h\sum_{i=1}^{k} M_{i+1} f(x_{n-i}, y_{n-i}, t_{n-i}) \\ y_{n+1} &= y_n + hM_1 g\left(x_{n+1}, y_{n+1}, t_{n+1}\right) + h\sum_{i=1}^{k} M_{i+1} g(x_{n-i}, y_{n-i}, t_{n-i}) \end{aligned} . \tag{3.4}$$

Note that with this method, one does not need to compute the first predictor stage (in this case x^p_{n+1}) which can be an advantage if f is a computationally expensive function. Finally, observe that the method is semi-implicit since all the equations must be solved for the associated variable. As in the previous case, the second equation uses the already obtained x_{n+1}.

General description of semi-explicit and semi-implicit BDF (PEC) methods

Well-known backward differentiation formula (BDF) is a family of $A(\alpha)$-stable linear multistep methods. Using this formula as corrector allows increasing stability of the resulting method. The equation of corrector stage in semi-explicit variant of the BDF-predictor-corrector (BDF PEC) method reads:

$$\begin{aligned} x_{n+1} + \sum_{i=1}^{k} \alpha_i x_{n+1-i} &= \beta_0 f(x^p_{n+1}, y^p_{n+1}, t_{n+1}) \\ y_{n+1} + \sum_{i=1}^{k} \alpha_i y_{n+1-i} &= \beta_0 g(x_{n+1}, y^p_{n+1}, t_{n+1}) \end{aligned} , \tag{3.5}$$

where α_i and β_0 are the coefficients of the BDF method.

Backward differentiation formula can be transformed to semi-implicit variant of the BDF (PEC) method. Then, BDF (PEC) semi-implicit corrector formula reads:

$$\begin{aligned} x_{n+1} + \sum_{i=1}^{k} \alpha_i x_{n+1-i} &= \beta_0 f(x_{n+1}, y^p_{n+1}, t_{n+1}) \\ y_{n+1} + \sum_{i=1}^{k} \alpha_i y_{n+1-i} &= \beta_0 g(x_{n+1}, y_{n+1}, t_{n+1}) \end{aligned}, \tag{3.6}$$

where α_i and β_0 are the coefficients of the BDF method. Below, we present a set of coefficients for the BDF (PEC) methods.

B_1	B_2	B_3	B_4	B_5	B_6
1	0	0	0	0	0
3/2	-1/2	0	0	0	0
23/12	-16/12	5/12	0	0	0
55/24	-59/24	37/24	-9/24	0	0
1901/720	-2774/720	2616/720	-1274/720	251/720	0
4277/1440	-7923/1440	9982/1440	-7298/1440	2877/1440	-475/1440
α_1	α_2	α_3	α_4	α_5	α_6
-1	0	0	0	0	0
-4/3	1/3	0	0	0	0
-18/11	9/11	-2/11	0	0	0
-48/25	36/25	-16/25	3/25	0	0
-300/137	300/137	-200/137	75/137	-12/137	0
-360/147	450/147	-400/147	225/147	-72/147	10/147
β_1	β_2	β_3	β_4	β_5	β_6
1	2/3	6/11	12/25	60/137	60/147

Table 3.1: Coefficients of Adams-Bashforth-Backward Differentiation method

3.1.2 Stability analysis

Since the semi-explicit methods does not exist for ODEs with dimension lower than two, we cannot use the Dahlquist's first order test equation. Instead, we choose the similar approach used in [32] that applies a two-dimensional test problem. Let us compose a matrix A of dimension 2×2 for the studied problem:

$$\mathbf{x}' = A\mathbf{x} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \mathbf{x}, \tag{3.7}$$

where $\mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix}$. We suppose that this matrix A has two conjugate eigenvalues $\lambda_{1,2} = \sigma \pm j\omega$.

The aim of this subsection is to form the polynomial matrix characterizing the stability of the methods and to extract its eigenvalues. Let us define this matrix as

$$P(z) = \begin{bmatrix} P_{11}(z) & P_{12}(z) \\ P_{21}(z) & P_{22}(z) \end{bmatrix} \tag{3.8}$$

with z the delay operator.

To begin, one have to study the stability of the predictor which is used in both semi-explicit and semi-implicit ABM methods. We can form the matrix polynomial of the predictor $p(z)$ as

$$p(z) = (I + B_1 hA)z^{p-1} + B_2 hAz^{p-2} + ... + B_6 hAz^{p-6} = \begin{bmatrix} p_{11}(z) & p_{12}(z) \\ p_{21}(z) & p_{22}(z) \end{bmatrix} \tag{3.9}$$

where I is the identity matrix, h the integration step and we take the terms with a nonnegative power of z according to the order p. Recall that B is the vector of coefficients of the Adams–Bashforth method while M is the vector of coefficients of the Adams–Moulton method, and p denotes the lengths of B and M. In other words, p is the order of the method considered. The formula (3.9) is common to the stability analysis of both methods.

Semi-explicit ABM method

One can rewrite the first equation of system (3.3) as:

$$\begin{aligned} x_{n+1} = \begin{bmatrix} 1 + M_2 hA_{12} & M_2 hA_{11} \end{bmatrix} \mathbf{x}_n + M_3 h \begin{bmatrix} A_{11} & A_{12} \end{bmatrix} \mathbf{x}_n + ... \\ + M_6 h \begin{bmatrix} A_{11} & A_{12} \end{bmatrix} \mathbf{x}_{n-4} + M_1 h \begin{bmatrix} A_{11} & A_{12} \end{bmatrix} \mathbf{x}_{n+1}^p \end{aligned}$$

Recall that $\mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix}$ and $\mathbf{x}_{n+1}^p = \begin{bmatrix} x_{n+1}^p \\ y_{n+1}^p \end{bmatrix}$. According to the order p, we take the terms in which the index of M is not greater than p. It leads to:

$$\begin{aligned} P_{11}(z) &= (1 + M_2 hA_{11})z^{p-1} + M_3 hA_{11}z^{p-2} + ... + M_6 hA_{11}z^{p-5} + M_1 h(A_{11}p_{11} + A_{12}p_{21}) \\ P_{12}(z) &= M_2 hA_{12}z^{p-1} + M_3 hA_{12}z^{p-2} + ... + M_6 hA_{12}z^{p-5} + M_1 h(A_{11}p_{12} + A_{12}p_{22}) \end{aligned}$$

in which we keep the terms with nonnegative power of z. Following the same idea, we can write the second equation of system 3.3 as follows:

$$\begin{aligned} y_{n+1} = \begin{bmatrix} M_2hA_{21} & 1 + M_2hA_{22} \end{bmatrix} \mathbf{x}_n + M_3h \begin{bmatrix} A_{21} & A_{22} \end{bmatrix} \mathbf{x}_{n-1} + ... \\ + M_6h \begin{bmatrix} A_{21} & A_{22} \end{bmatrix} \mathbf{x}_{n-4} + M_1h \begin{bmatrix} 0 & A_{22} \end{bmatrix} \mathbf{x}^p_{n+1} + M_1h \begin{bmatrix} A_{21} & 0 \end{bmatrix} \mathbf{x}_{n+1} \end{aligned}$$

Note that in this equation, we reuse x_{n+1} that we have just computed such that the computation is explicit. From this equation, we obtain the matrix $P(z)$ of equation (3.8) as:

$$\begin{aligned} P_{21}(z) &= M_2hA_{21}z^{p-1} + M_3hA_{21}z^{p-2} + ... + M_6hA_{21}z^{p-5} + M_1h(A_{21}P_{11} + A_{22}p_{21}) \\ P_{22}(z) &= (1 + M_2hA_{22})z^{p-1} + M_3hA_{22}z^{p-2} + ... + M_6hA_{22}z^{p-5} + M_1h(A_{21}P_{12} + A_{22}p_{22}) \end{aligned}$$

with P_{11} and P_{12} already computed.

Semi-implicit ABM method

From the first equation of system (3.4), we can write:

$$\begin{aligned} (1 - M_1hA_{11})x_{n+1} = \begin{bmatrix} 1 + M_2hA_{11} & M_2hA_{12} \end{bmatrix} \mathbf{x}_n + M_3h \begin{bmatrix} A_{11} & A_{12} \end{bmatrix} \mathbf{x}_{n-1} + ... \\ + M_6h \begin{bmatrix} A_{11} & A_{12} \end{bmatrix} \mathbf{x}_{n-4} + M_1h \begin{bmatrix} 0 & A_{12} \end{bmatrix} \mathbf{x}^p_{n+1} \end{aligned} \tag{3.10}$$

It leads to the expressions of $P_{11}(z)$ and $P_{12}(z)$

$$\begin{aligned} P_{11}(z) &= \left((1 + M_2hA_{11})z^{p-1} + M_3hA_{11}z^{p-2} + ... + M_6hA_{11}z^{p-5} + M_1hA_{12}p_{21}\right)(1 - M_1hA_{11})^{-1} \\ P_{12}(z) &= \left(M_2hA_{12}z^{p-1} + M_3hA_{12}z^{p-2} + ... + M_6hA_{12}z^{p-5} + M_1hA_{12}p_{22}\right)(1 - M_1hA_{11})^{-1} \end{aligned}$$

Following the same idea, we can write the second equation of system 3.4 as follow:

$$\begin{aligned} (1 - M_1hA_{22})y_{n+1} = \begin{bmatrix} M_2hA_{21} & 1 + M_2hA_{22} \end{bmatrix} \mathbf{x}_n + M_3h \begin{bmatrix} A_{21} & A_{22} \end{bmatrix} \mathbf{x}_{n-1} + ... \\ + M_6h \begin{bmatrix} A_{21} & A_{22} \end{bmatrix} \mathbf{x}_{n-4} + M_1h \begin{bmatrix} A_{21} & 0 \end{bmatrix} \mathbf{x}^p_{n+1} \end{aligned} \tag{3.11}$$

It leads to the expressions of $P_{21}(z)$ and $P_{22}(z)$:

$$P_{21}(z) = \Big(M_2 h A_{21} z^{p-1} + M_3 h A_{21} z^{p-2} + ... + M_6 h A_{21} z^{p-5} + M_1 h A_{21} P_{11}(z)\Big)(1 - M_1 h A_{22})^{-1}$$

$$P_{22}(z) = \Big((1 + M_2 h A_{22}) z^{p-1} + M_3 h A_{22} z^{p-2} + ... + M_6 h A_{22} z^{p-5} + M_1 h A_{21} P_{12}(z)\Big)(1 - M_1 h A_{22})^{-1}$$

Semi-explicit AB-BDF method

Using the same technique, formulas for stability matrices are obtained for the semi-explicit AB-BDF method (3.5):

$$\begin{aligned}
P_{11}(z) &= \alpha_1 z^{p-1} + \alpha_2 z^{p-2} + ... + \beta_0 h(A_{11} p_{11}(z) + A_{12} p_{21}(z)) \\
P_{12}(z) &= \beta_0 h(A_{11} p_{12}(z) + A_{12} p_{22}(z)) \\
P_{21}(z) &= \beta_0 h(A_{21} P_{11}(z) + A_{22} p_{21}(z)) \\
P_{22}(z) &= \alpha_1 z^{p-1} + \alpha_2 z^{p-2} + ... + \beta_0 h(A_{21} P_{12}(z) + A_{22} p_{22}(z)).
\end{aligned}$$

Semi-implicit AB-BDF method

Stability matrices for the semi-implicit AB-BDF method (3.6) reads in the similar way:

$$\begin{aligned}
P_{11}(z) &= \Big(\alpha_1 z^{p-1} + \alpha_2 z^{p-2} + \cdots + \beta_0 h A_{12} p_{21}(z)\Big)(1 - \beta_0 h A_{11})^{-1} \\
P_{12}(z) &= \beta_0 h A_{12} p_{22}(z)(1 - \beta_0 h A_{11})^{-1} \\
P_{21}(z) &= \beta_0 h A_{21} P_{11}(z)(1 - \beta_0 h A_{22})^{-1} \\
P_{22}(z) &= \Big(\alpha_1 z^{p-1} + \alpha_2 z^{p-2} + ... + \beta_0 h A_{21} P_{12}(z)\Big)(1 - \beta_0 h A_{22})^{-1}.
\end{aligned}$$

3.1.3 Test problem for estimating method stability

Following the ideas of [32] and particularly [4] for a two-dimensional problem, the matrix A of the test problem is built such that the conjugate eigenvalues $\lambda_{1,2}$ defined in section 3.1.2 are the eigenvalues of this matrix A. This will allow us to

study the same stability regions with values $\mathcal{R}(|\lambda_{1,2}|)$. Moreover, the matrix A is built such that every possible 2×2 matrix with those two conjugate eigenvalues must be easily derived. In order to obtain such a matrix, we must first ensure that $A_{12} = A_{21}$. Then, all those 2-dimensional matrices can be described by a ratio between the first entry of the diagonal and the second one. In fact, there exist two extreme cases for such matrices: an asymmetric triangular matrix and the symmetric one with respect to the diagonal elements. Indeed, for 2×2 matrices, we have two diagonal entries such that we can express the first one as a multiple of the second one.

Taking into account those considerations gives us the following coefficients of the test matrix for problem (3.7):

$$\begin{aligned} A_{11} &= kA_{22} \\ A_{12} &= -\sqrt{\lambda^2 - (1+k)A_{22}\lambda + kA_{22}^2} \\ A_{21} &= A_{12} \\ A_{22} &= \frac{2\sigma}{1+k} \end{aligned}$$

where we define $k = A_{11}/A_{22}$ called the symmetry coefficient. This matrix has been developed such that every matrix with conjugate eigenvalues can be expressed by this factor k with the two extreme cases. On one hand, the coefficient k can be 0 or ∞ for an asymmetric triangular matrix (Frobenius normal form). On the other hand, for a matrix in Jordan normal form, the coefficient k is equal to 1. All the other matrices that have not one of these two forms have an intermediate value of k. The two extreme cases can be viewed as limits for stability functions.

Finally, in order to study the stability regions of the two methods, we need to take the largest eigenvalue of $P(z)$ in modulus. We can do this with the characteristic polynomial of $P(z)$ given by $(h\lambda - P_{11}(z))(h\lambda - P_{22}(z)) - P_{21}(z)P_{12}(z)$. These roots must be less than 1 in modulus to make the method stable.

3.1.4 Test problems for estimating method performance

We use well-known test problems for estimating the method performance. The following methodology is used. The test ODE is solved with a given time step. In the final point of the simulation, an error is estimated through a comparison with a higher-order method, and the corresponding CPU time is measured. Experiments in several points give the line describing the method performance allowing comparing

with the other methods. More efficient methods are distinguished by a simple criterion: the lower the line lies the more efficient the method is. We use DOPRI8 as the high-order method for obtaining a reference solution. All investigated algorithms are of order 4. In order to avoid the influence of the operating system processes during CPU time measurements, this time was averaged over 10 independent simulations. The following methods are compared: Adams-Bashforth (AB), Adams-Moulton (AM), BDF, Semi-implicit predictor-corrector BDF (BDF (PEC)), Semi-explicit predictor-corrector BDF (BDF (PEC, SE)).

The first investigated test problem is the reknown Rössler chaotic equation [30]:

$$\begin{cases} \dot{x} &= -y - z, \\ \dot{y} &= x + ay, \\ \dot{z} &= b + z(x - c), \end{cases} \tag{3.12}$$

In our experiments, we used classical parameters of the Rössler system $a = 0.2, b = 0.2, c = 5.7$. Initial conditions are $(0.1, 0, -0.1)^\top$. Period of simulation is 50 sec. Step sizes under investigation are $\{5 \cdot 10^{-4}, 10^{-3}, 5 \cdot 10^{-3}, 0.01\}$ sec.

The second investigated test problem is the Nosé-Hoover chaotic equation [27]:

$$\begin{cases} \dot{x} &= y, \\ \dot{y} &= -x - ayz, \\ \dot{z} &= b(y^2 - 1), \end{cases} \tag{3.13}$$

The parameters used are $a = 1, b = 1, c = 5.7$. Initial conditions are $(0.1, 0, -0.1)^\top$. Period of simulation is 15 sec. Step sizes under investigation are $\{5 \cdot 10^{-4}, 10^{-3}, 5 \cdot 10^{-3}, 0.01\}$ sec.

The third test problem is the Van der Pol oscillator equation [34]:

$$\begin{cases} \dot{x} &= y, \\ \dot{y} &= \mu(1 - x^2)y - x, \end{cases} \tag{3.14}$$

The parameter used is $\mu = 55$. Initial conditions are $(1, 0)^\top$. Period of simulation is 15 sec. Step sizes under investigation are $\{10^{-5}, 5 \cdot 10^{-5}, 10^{-4}, 5 \cdot 10^{-4}\}$ sec.

3.2 Results

In this section, we provide the stability regions of the semi-explicit, semi-implicit ABM and AB-BDF predictor-corrector methods for $k = [0, 0.5, 1]$. In each figure, the stability regions of the methods of order 1 to 6 are depicted.

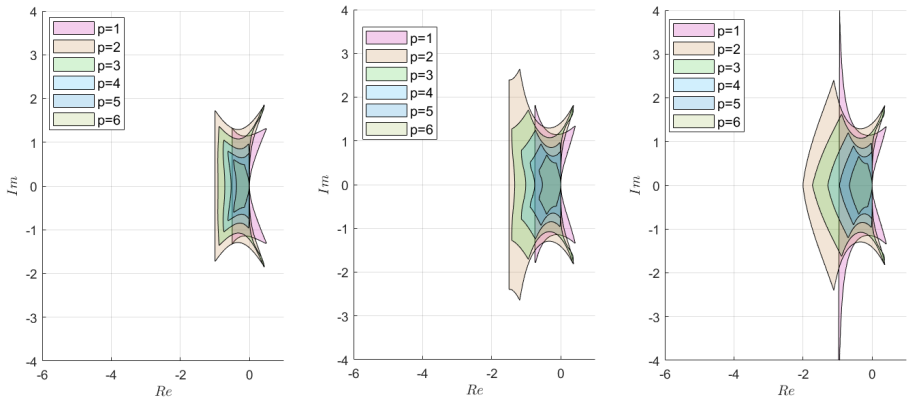

Figure 3.1: Stability regions of the semi-explicit ABM method with for $k = [0, 0.5, 1]$

In Figure 3.1, one can see that the region of stability for the semi-explicit method is smaller for the order 1 than the order 2 near the real axis. The method is, therefore, less interesting to be used in the order 1. The region decreases as the order of accuracy increases and the higher-order regions are included in the smaller order ones (except for $p = 1$).

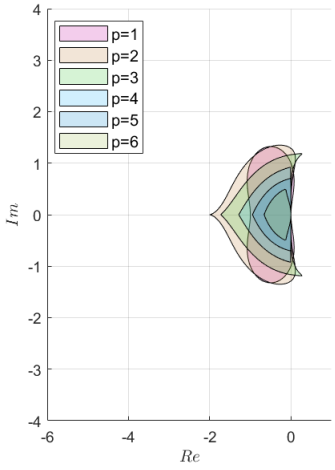

Figure 3.2: Stability regions of the ABM predictor-corrector method

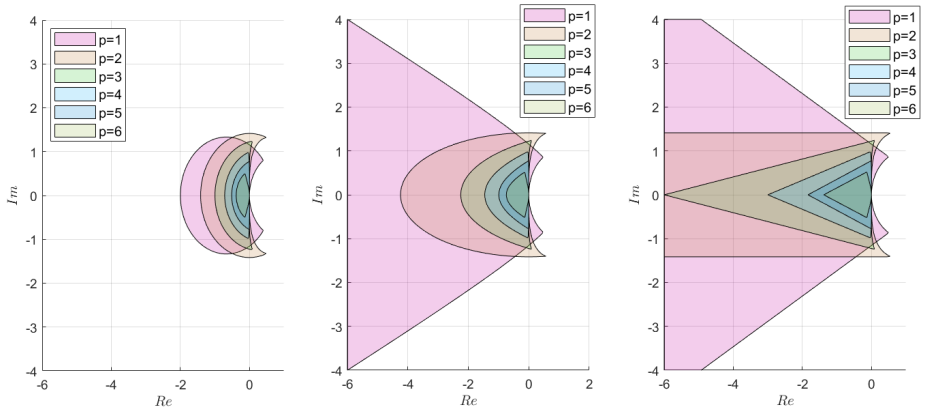

Figure 3.3: Stability regions of the semi-implicit ABM method for $k = [0, 0.5, 1]$

For the semi-implicit method, it seems that order 1 covers the negative real axis or at least large negative numbers in absolute value for $k = 0.5, 1$. Contrary to the previous method, the first order of semi-implicit ABM method has greater stability region than the following orders. However, it keeps some common characteristics: from order 4 the stability region does not include the positive real part. Moreover, the stability region decreases when the order of the method increases and the stability region of an order is included in the one of the previous order. These characteristics are common to all three methods which is reasonable since the semi-explicit (Figure 3.1) and semi-implicit (Figure 3.3) methods are modifications of the ABM method (Figure 3.2). Note that the stability regions of the semi-implicit method is strictly included in the stability region of the implicit AM method (Figure 3.4) in the left part of the complex plane (i.e. for stable problems).

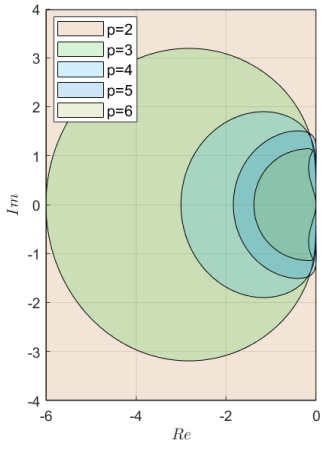

Figure 3.4: Stability regions of the Adams-Moulton method

Similar conclusions can be drawn for $k = 0$ and $k = 0.5$ as we see in Figure 3.1 for the semi-explicit method and Figure 3.3 for the semi-implicit one. Moreover, based on the stability regions for those three value of k, we can stress that $k = 0.5$ is an intermediate case between $k = 0$ and $k = 1$. Finally, comparing the two extreme cases shows that case $k = 0$ is the worst case for the numerical stability of the two modifications of ABM whereas $k = 1$ is the best case. Obviously, the ABM method is not impacted by the value of k.

Figure 3.5 shows the Adams-Bashforth-BDF predictor-corrector semi-explicit method stability regions, and Figure 3.6 shows the Adams-Bashforth-BDF predictor-corrector semi-implicit method stability regions. The stability regions of the AB-BDF semi-explicit method are given in Figure 3.5. Its size in imaginary axis is sufficiently greater than the size of any other investigated methods which makes it more feasible for moderately stiff problems. The AB-BDF semi-implicit method with the stability region shown in Figure 3.6 has approximately similar size in imaginary axis as its semi-explicit counterpart, has a much greater size in real axes. So, among all considered semi-explicit and semi-implicit methods, a family of semi-implicit AB-BDF methods has the largest stability regions, and for high orders, they are even greater than the stability regions of Adams-Moulton methods.

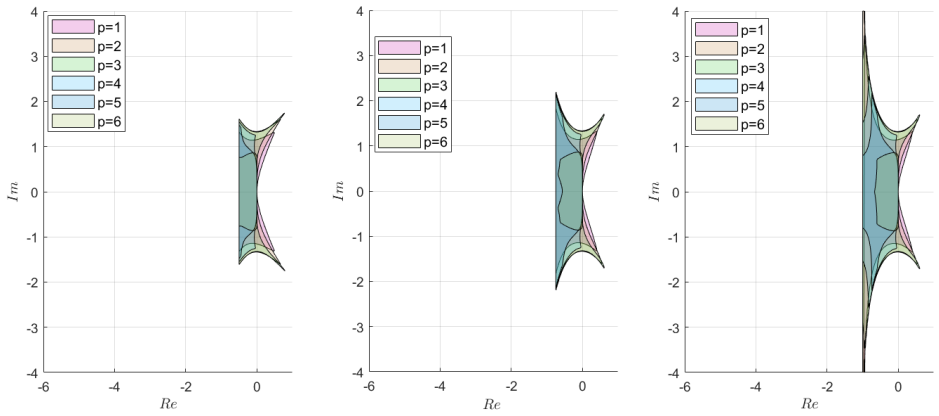

Figure 3.5: Stability Regions of Adams-Bashforth-BDF semi-explicit methods with $k = [0, 0.5, 1]$ (from left to right)

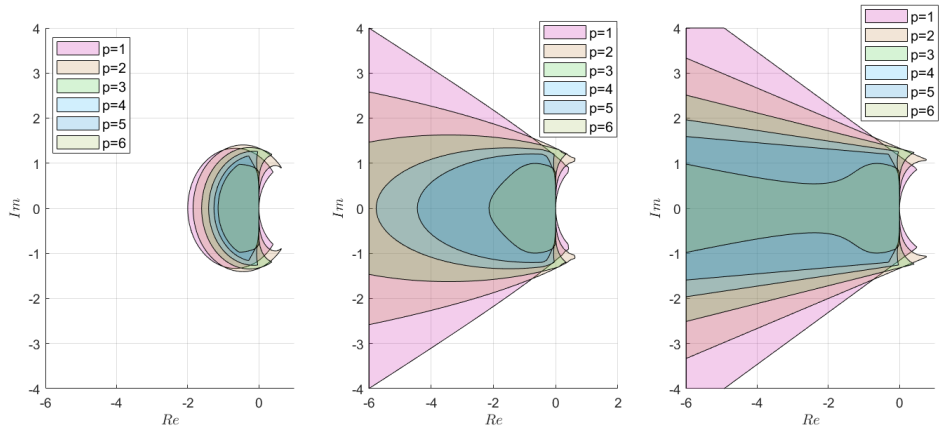

Figure 3.6: Stability Regions of Adams-Bashforth-BDF semi-implicit methods with $k = [0, 0.5, 1]$ (from left to right)

Performance tests show that on the test problems described above the semi-implicit predictor-corrector BDF (BDF (PEC)) and semi-explicit predictor-corrector BDF (BDF (PEC, SE))) shows the best error to CPU time ratio in comparison with conventional multistep methods: Adams-Bashforth (AB), Adams-Moulton (AM), and backward differentiation formula (BDF). The first test on the Rössler test problem shows that BDF and AB methods are the best for the problem. This is due to the low stiffness of the problem and using constant stepsizes. The Nosé-Hoover test problem is a type of conservative chaotic problem. While the AB method preserved its high performance, the proposed predictor-corrector BDF methods have the advantage over it and other methods. The Van der Pol oscillator with $\mu = 15$ is moderately stiff but it also allows BDF PEC methods with constant stepsizes to outperform all the other methods.

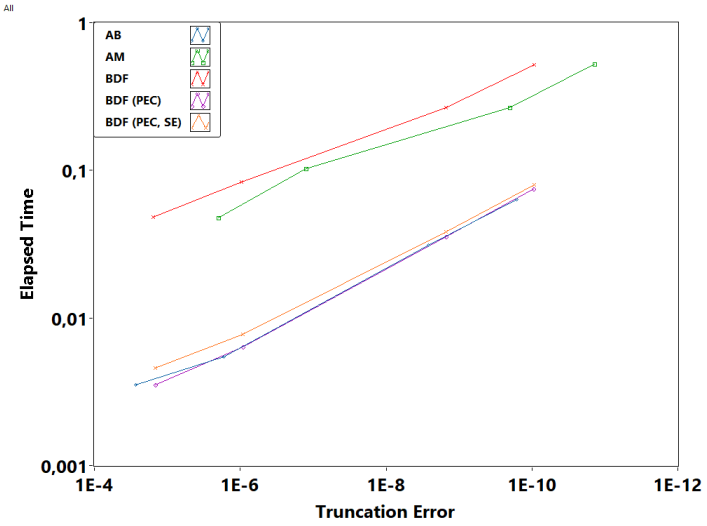

Figure 3.7: Comparison of error versus elapsed time for the family of investigated methods on the test Rössler problem

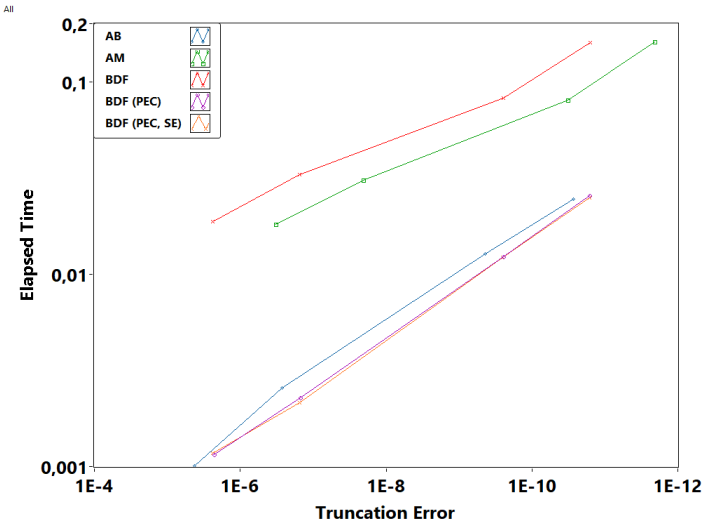

Figure 3.8: Comparison of error versus elapsed time for the family of investigated methods on the Nosé-Hoover problem

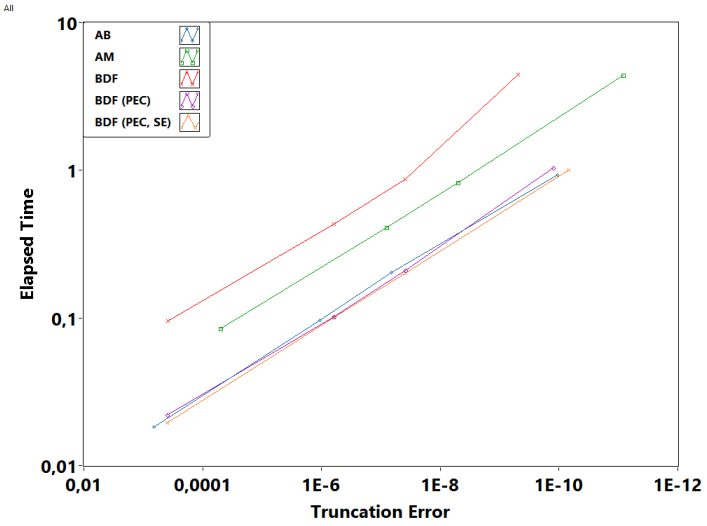

Figure 3.9: Comparison of error versus elapsed time for the family of investigated methods on the Van der Pol oscillator problem

3.3 Discussion

In this section, a stability analysis of semi-implicit and semi-explicit predictor-corrector methods has been performed. We clearly show, that the semi-explicit and semi-implicit ABM and AB-BDF methods expand the stability regions over the conventional ABM method and thus have better numerical stability. As one can see in Figures 3.1−3.6, for the test problem with a matrix in the Jordan normal form ($k = 1$), the stability regions of all considered methods are wider in the complex plane than those of the ABM method no matter whatever the order and method are. For the intermediate case ($k = 0.5$), we observe a similar situation. For the worst case $k = 0$ when the matrix is in the Frobenius form it's not true. For example, in Figure 3.1, the semi-explicit ABM method obtains a smaller stability region than the ABM method. In a similar case $k = 0$ the semi-implicit ABM method obtains stability regions relatively comparable to that of ABM.

It's well-known that the original BDF is $A(\alpha)$-stable, while this stability is bought by high costs of fully implicit computation involving Newton's method. In the case of the semi-explicit method, no truly implicit computation is needed, while the stability regions are almost as wide in imaginary directions as the stability regions of the AM methods for $p \geq 3$. This allows using AB-BDF semi-explicit method in many problems where the conventional AM method is used, with less

computational efforts. As for the semi-implicit AB-BDF method, it has a very large negative real part of stability regions in case of values of k close to 1, which make it especially attractive for the problems with symmetric Jacobian matrices. It should be noted, that using AB-BDF methods requires twice more memory, that ABM. However, these algorithms can be further optimized using the approach given in [32].

For the problems at the boundary of stability, when the eigenvalues are pure imaginary, the ABM methods are interesting up to the order of 5 and AB-BDF up to the order 6 because they have a part of the imaginary axis in their region of stability. A simple example of the problem with purely imaginary eigenvalues is the pendulum without friction. However, a key difference in the stability of the semi-explicit and semi-implicit methods is that BDF is not stable for orders greater than 6, while Adams-Bashforth and Adams-Moulton methods can be used with much higher orders with sufficiently small step sizes.

The overall results show that in the worst case we lose less stability than we can gain in intermediate and best cases, which is evidence of the potential superiority of semi-explicit and semi-implicit methods over conventional AM and ABM algorithms.

In terms of computational efficiency, paper [32] has shown that semi-explicit and semi-implicit ABM methods are efficient methods compared to other linear multistep methods (such as ABM and BDF). However, the comparison of semi-explicit and semi-implicit AB-BDF methods has not been performed yet to our knowledge, so it would be of interest in future studies. Furthermore, it was shown in [32] that semi-explicit and semi-implicit ABM methods are flexible enough to optimize the computations according to the problem features. The comparison of the error to CPU time ratio of semi-explicit and semi-implicit predictor-corrector BDF methods has been performed in the current paper with the other multistep methods. It was shown that these methods have superior error to CPU time ratio in experiments with constant step size, and this superiority could be even more valuable for variable step sizes. Further investigation is needed to compare variable step size implementations.

The motivation of semi-implicit computation is that each equation can be solved separately, so implicit algebraic equations have to be solved for one variable, and the computation of the full Hessian matrix in the Newton method is not required. Moreover, some one-dimensional variants of root-finding algorithms can also be used such as Brent's method or Steffensen's method. Note that in some cases, the computational cost can be optimized further. For instance, the work [33] describes

a problem for which an equation of the corrector can be solved independently of the previous computations of the corrector.

Finally, applying the semi-explicit and semi-implicit methods to other problems such as fractional-order differential equations could be one of the possible directions of further investigations. Numerical approaches to solving this type of problem have been studied in [36, 16] and [15] applied Adams-Bashforth explicit method. Thus, using semi-explicit and semi-implicit modified ABM methods may be a new application in the continuation of these articles.

3.4 Adaptative step size strategy

To maintain the local error below a tolerance ϵ, common strategies use the order of the method and an estimate of the local error committed at each step to calculate the step size in an adaptative way. With the previous methods which are predictor-corrector methods, the local error can be estimated with the value computed by the predictor and the corrector. In other words, the estimation of the local error θ_i is given by:

$$\theta_i = \max_j |y_{i,j} - y^p_{i,j}|$$

with $y_{i,j}$ is the value of the corrector at step i for the variable j. By imposing $\theta_{i+1} \leq \epsilon$, a bound for the new step size reads:

$$h_{i+1} \leq h_i \left(\frac{\epsilon}{\theta_i}\right)^{\frac{1}{r+1}} = \hat{h} \tag{3.15}$$

with r the order of the method. An example of prudent choice is then given by:

- if $\theta_i > \epsilon$, repeat the current step i with a smaller time step: $h = \frac{h_i}{2}$
- if $\hat{h} > h_i$: $h_{i+1} = \min(\hat{h}, \frac{3}{2}h_i, h_{max})$
- if $\hat{h} < h_i$: $h_{i+1} = \max(\hat{h}, \frac{1}{2}h_i, h_{min})$

Chapter 4

Scipy

From a more practical point of view, a first section will deal with implementation and the choices that have been made. Understanding the choices and framework in which the solvers work will be a key characteristic to identify the possible ways of improvements. Indeed, it will be explained that even if the Scipy framework has some advantages, it cannot be changed in depth freely and this creates problems when new types of solver have to be added within a not entirely suitable environment.

4.1 Scipy submission

All solvers have been realized to satisfy the Scipy formalism and more precisely, the *solve_ivp* function[1]. This function allows to solve initial value problems under the form $y' = f(t, y)$ which means that additional parameters will be required to solve DAEs. The advantage of this framework is that only the step implementation must be done while the process between steps and the user information gathering is already implemented and works under the hood. However, this function is internally built to solve ODEs and some issues have been encountered while developing the DAEs solvers.

[1] `https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html`

4.1.1 Implementation of the DAE solvers

Recall that the different implementations of the following solvers are available in the Github repository [2].

BDF, Radau and Rosenbrock

BDF and Radau DAE solvers apply the DAE modification on their current Scipy ODEs implementation as explained in section 2.2. Those modification have been realized in such a way that no computation of the ODE part is modified. The Newton iterations have been updated accordingly to the theory explained in sections 2.2.1 and 2.2.2. Therefore, the user is allowed to provide a mass matrix to use the DAEs solver mode. Note that this mass matrix can be sparse, exactly like the Jacobian which implies that the linear solvers will switch to sparse linear solvers.

For Rosenbrock, Yuanlong Huang has translated a part of the Fortran Rosenbrock solver for autonomous ODEs into Python[3]. However, the goal of this work is to solve DAEs (possibly non autonomous), therefore some improvements have been realized. First, the solver has to take into account the time derivative as shown in section 2.1.1. The original Fortran code[4] applies a first order finite difference with a small time step that depends on the magnitude of the present time and the machine epsilon. Then, the ODEs solver has been transformed into a DAEs solver by allowing the user to give a mass matrix and by modifying the Newton iterations accordingly as explained in section 2.2.3. Finally, the code is able to approximate the Jacobian if the user does not give it. This has been realized by using the Scipy *num_jac* function while writing the Rosenbrock method according to the Scipy framework. This function works in a similar way as in MATLAB and approximates the Jacobian by finite differences. Solving the linear systems for K involves *numpy.linalg.solve* or *scipy.sparse.linalg.spsolve* in case of a sparse jacobian or sparse mass matrix. Those solvers relies on the LU decomposition.

SABM

This method solves differential-algebraic equations (DAEs) and ODEs. This is a fixed order method with the order chosen between 1 to 5. The general framework

[2] `https://github.com/ocheffert/Notebooks---Master-thesis`
[3] `https://github.com/YuanlongHuang/NNEIT/blob/main/ROSsolver_e.py`
[4] `http://www.unige.ch/~hairer/prog/stiff/Oldies/ros4.f`

of the SABM algorithm is described in 2.2.4. This class implements a varying step size strategy which depends on the estimated error shown in section 3.4. The estimated error is the difference between the predictor and the corrector. This solver can not be applied in the complex domain.

SABM are multistep methods. Therefore, it needs to store information from previous steps. First, the user must choose the order of the method, for example 3. To use this order, he needs the information of the 2 previous steps. At the beginning of the algorithm, only one piece of information is available: the initial conditions. To get more information, it will start by calculating the first step with a first order Euler method and then increase its order as it goes along. To store these previous steps efficiently, the choice of a *deque* (doubly ended queue) was made. This allows faster append and pop operations which provide an $\mathcal{O}(1)$ time complexity. The maximum order for this algorithm is 5 because the stability region is less interesting for higher orders as shown in the results of the article described in section 3.2.

To solve the differential part, the ODE SABM methods described in 3.1 are used. For the semi-explicit version, this part is solved explicitly in contrast to the constraints that are implicitly solved using *fsolve*. This choice was made to remain general in the form of constraints. Indeed, constraints can be both linear or non-linear. For the semi-implicit version, both parts are solved using the *fsolve* function for sake of clarity. The function *fsolve* finishes whenever the relative error between two consecutive iterates is at most $rtol$.

To compute the new step size, the estimated error must be derived from the predictor and the corrector as explained in section 3.4. But in the case of multiple variables, the question is: which error must be used to compute the new step size ? The choice made in this implementation is to determine the biggest relative error and use the associate error in the computations described in 3.4.

Some new parameters have been added to provide the solver enough information to solve the problem:

- fun: callable
  Right-hand side of the system for the differential variables. There is two options for the signature of fun : it can be $fun(t, y, x)$ to solve DAE. Here t is a scalar, y is a *ndarray* with shape $(n,)$ and x is a *ndarray* with shape $(m,)$; then fun must return an *array_like* with shape $(n,)$. Alternatively it can be $fun(t, y)$ to solve an ODE.

- *gun*: callable, optional
  Algebraic equations. Default is *None* which means that is an ODE. Otherwise, it's the right-hand side of the system for the algebraic variables. The calling signature is $gun(t, y, x)$. Here t is a scalar, y is a *ndarray* with shape $(n,)$, and x is a *ndarray* with shape $(m,)$; then *gun* must return *array_like* with shape $(m,)$.

- mode: string, optional
  The user can choose between the *Explicit* and *Implicit* form of the method. Default is *Explicit*.

- order: int, optional
  The user can choose the order of the method. Default is 2. The order is bounded by *MAX_ORDER* which is equal to 5.

- num_diff: int, optional
  The number of differential variables. Must be set to n for a DAE.

Putting all this together, the implementation can be described by the following diagrams:

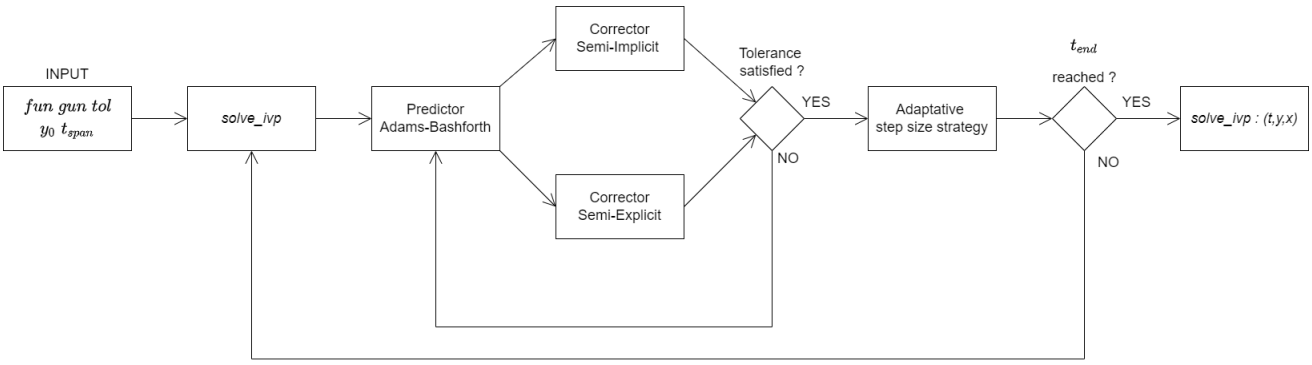

Figure 4.1: High level description of the implementation

In this figure, the user interaction is described by the input that the user must give to *solve_ivp* (some of the inputs are explained above) and he receives the output which is the solution of the differential variables $\mathbf{y}$ and the algebraic ones $\mathbf{x}$ evaluated at each t. Therefore, the Scipy solvers are used only in backend and return to *solve_ivp* the result of one step. Since *solve_ivp* is the frontend function, it will store the result of the step for the user and ask the solver to proceed to the next step.

In Figure 4.2 below, the Python implementation of one step of the SABM methods is shown. This is the direct implementation of the Figure 4.1. Checking if t_{end} is reached is handled by *solve_ivp* but not the size of the step which must not overshoot t_{end}. This implementation allows to solve ODEs as well by leaving *gun* as *None* as explained above. At each step, it's mandatory to check if the step is too small which means that the solver cannot converge.

4.1.2 Submission

A GitHub pull request has been done to incorporate the SABM solver into the *solve_ivp* environment. Posting such a pull request requires to install a Scipy Developer Environment in which all the Fortran, C/C++, Cython and Python codes must compile. Therefore, after having added a code, it must be ensured that all Scipy codes work as before and the added code solves correctly what it's aimed to solve. In order to prove that it works, many unit tests must be done. However, the existing tests were concerning the resolution of ODEs, which had to be extended to the resolution of DAEs. Since no Scipy method resolves DAEs, new unit tests have been added. The new tests check if the solver works for simple DAEs with known analytical solutions both with *Explicit* and *Implicit* mode, if events work, if the integration interval is respected, etc ... Moreover, existing Scipy unit tests for ODEs solvers have all been modified for this DAEs solver when applicable.

Finally, other considerations must be taken into account when adding a code to Scipy such as the license of the code, the documentation (docstrings) of the added/modified functions, the coding style that must satisfy the PEP8 rules, etc... All those rules ensure that the code is maintainable for future modifications or improvements. Even if all solvers have been implemented in the Scipy fashion, only SABM satisfy all those requirements and have been submitted. A further work may submit the other ones by doing complete Pull Requests but they can already be used by any user in the same way as the one of the notebooks since a solver does not need to be pushed into Scipy to work with *solve_ivp*. All those solvers are able to solve ODEs and a Scipy submission will ensure that the solvers pass all the existing ODEs unit tests.

```python
def _step_impl(self):
    # apply SABM
    if self.mode == "Implicit":
        (y_new, x_new, err) = self.Semi_Implicit_Adams_Bashforth_Moulton()
    elif self.mode == "Explicit":
        (y_new, x_new, err) = self.Semi_Explicit_Adams_Bashforth_Moulton()
    else:
        return False, "Error mode. Need 'Implicit' or 'Explicit' for mode"

    # error, the step does not converge
    if y_new is False:
        return False, x_new

    # adaptative step size strategy
    h = self.h_abs
    i = np.argmax(abs(err/y_new))
    if err[i] < 1e-14:
        h_new = min([3/2 * h, self.max_step])
    else:
        h_hat = h*(self.atol[i]/err[i])**(1/(self.p+1))
        if h_hat > h:
            h_new = min([h_hat, 3/2 * h, self.max_step])
        else:
            h_new = max([h_hat, 1/2 * h, MIN_H])

    # method fails to converge
    if h_new < MIN_H:
        return False, self.TOO_SMALL_STEP

    # ensure to not overshoot the boundary
    h_new = min(h_new, abs(self.t_bound - (self.t + h*self.direction)))
    # update the time t
    self.t += self.h_abs*self.direction
    self.h_abs = h_new
    self.prev_f_y.append(self.f(self.t, y_new, x_new))
    self.nfev += 1
    # if ODE take y, if DAE take y and x
    if self.ode:
        self.y = y_new
    else:
        self.y = np.concatenate((y_new, x_new), axis=None)
    # increase order if possible
    if self.p < self.order and self.p < MAX_ORDER:
        self.p += 1

    return True, None
```

Figure 4.2: Python implementation of one step of SABM

4.2 Ways of improvement

This section will discuss how to improve the Scipy SABM code. Some of the restrictions come from the Scipy *solve_ivp* framework whereas other ways of improvement may require further investigations.

4.2.1 Step size strategy

By taking a look at the results shown in a previous section, it seems that the adaptative step size strategy is too cautious while the other Scipy methods use more aggressive strategies and allow a larger step size. Indeed, it has been observed that compared to BDF and Rosenbrock, SABM reach a high accuracy at the cost of more function evaluations. Therefore, it's legitimate to wonder if the SABM solver can achieve better performance by using a more aggressive strategy and by allowing more room for the error. Selecting the appropriate strategy for this solver requires further investigations.

An adaptative step size strategy is given in [23]. In this paper, the step size is estimated with combined absolute (atol) and relative (rtol) errors in the predictor. For that, they use consecutive order AB formulas for estimating the error in the lower order:

$$\gamma = \min_i \left[\frac{atol + rtol|P(m+1)(i)|}{2|[P(m+1) - P(m)](i)|} \right]^{1/(m+2)} \tag{4.1}$$

where $P(m)$ is the value obtained from the predictor at the order m. The new step size is computed as $h_{n+1} = \gamma h_n$.

4.2.2 First step

Scipy codes are able to find a first initial step depending on the initial values and the function to integrate. However, when testing, it's possible to observe that for some DAEs with very stiff behaviour such as Robertson and low tolerances, the user can help the solver to converge by providing manually a first step. Since the first step is not based on the mass matrix, the initial step algorithm should be modified to incorporate this matrix.

Currently, the algorithm chooses a first step candidate based on the norm of the initial values $\mathbf{y}_0$ and $\mathbf{f}(t, \mathbf{y}_0)$. This step is used to perform an Explicit Euler step to obtain $\mathbf{y}_1$ and thus $\mathbf{f}(t, \mathbf{y}_1)$ which are used to obtain a second candidate and the minimum between them is chosen (see [13], p169 for more details). All norms are computed by scaling their input by a scale factor based on tolerances.

4.2.3 Order strategy

The BDF solver uses an adaptative order strategy whereas the SABM implementation considered in this study is based on a fixed order strategy. Taking the advantage of the accuracy of the higher orders or the stability of the lower orders should be tested in further studies and may enhance the SABM performance. In paper [23], they propose a variable order and variable step size algorithm that can be investigated.

4.2.4 Reduce the number of function evaluations

In the SABM methods, the differential variables are computed one by one:

$$y_{n+1,i} = y_{n,i} + hM_1 f_i(\mathbf{x}^p_{n+1}, \mathbf{y}^i_{n+1}, t_{n+1}) + h\sum_{j=1}^{k} M_{j+1} f_i(\mathbf{x}_{n-j}, \mathbf{y}_{n-j}, t_{n-j}) \qquad (4.2)$$

In other words, when only one element of the function $\mathbf{f}$ in $\mathbf{y}' = \mathbf{f}(\mathbf{x}, \mathbf{y}, t)$ is needed, the Scipy framework imposes to call the function $\mathbf{f}$ which computes all elements but there is only one useful value. For a DAE with n differential variables, this implementation computes n times n equations instead of 1 time for each equation. This greatly reduces the performance of the SABM solver which consists of only one calling of the function $\mathbf{f}$ for each iteration.

A way to solve this problem is to give a list of functions as input for $\mathbf{f}$ and not only one function but in another framework than Scipy.

4.2.5 Solving the constraints

Solving the constraints is equivalent to solve a potential non-linear system. Indeed, the constraints are often linear and simple. With this finding, there are two possible improvements.

Jacobian

Some clever ways to compute or use the Jacobian might be used to improve the performance of the codes. A first way would be to reuse the Jacobian of the previous step and use it as an approximation of the Jacobian of this step. This technique is already used in the BDF algorithm and leads to better performance when this approximated Jacobian is sufficient to satisfy the expected tolerances. Indeed, running the examples with the BDF codes shows that the number of Jacobian evaluations is lower than the number of steps. In practice, the BDF solver computes the Jacobian only when a step is rejected due to a lack of accuracy and needs to restart the step. The W-Rosenbrock methods exist to deal with an approximated Jacobian even if no implementation of those methods has been tested in this work.

When solving the linear constraints that arise from the conservation laws or the Kirchhoff's laws, the Jacobian is constant and it's a serious lack of efficiency to compute it at each step.

Pre-compute

An other possibility is to pre-compute the constraints. For that, the constraints

$$0 = \mathbf{g}(\mathbf{x}_{n+1}, \mathbf{y}_{n+1}, t_{n+1}) \tag{4.3}$$

must be rewritten as

$$\mathbf{x}_{n+1} = \tilde{\mathbf{g}}(\mathbf{y}_{n+1}, t_{n+1}). \tag{4.4}$$

With this new formulation, the method that solves the constraints is totally explicit and this reduces the computation of solving (4.3). Depending on the structure of $\mathbf{g}$, this can be easy or difficult but in case of linear constraints (conservation laws, Kirchhoff's laws, ...) it should be straightforward. Note that a linear constraint refers to a linear equation with respect to the algebraic variables. For example, for the Robotic manipulator with two bodies, the algebraic variables are the Lagrange multipliers λ_3, λ_4 so that $sin(\theta_2)$ and $cos(\theta_2)$ have no impact on the linearity of the equations. It's also the case for the simple pendulum whose the index 1 formulation is linear with respect to the algebraic variable λ.

Moreover, if the user gives the information that constraints are linear, it would be possible to use a linear solver instead of nonlinear one which would lead to better performance. In fact, this would lower the number of constraint function evaluations that appears to be significant.

4.2.6 Implement Newton step

In sections 2.2.1, 2.2.2 and 2.2.3, the Newton iterations are given. Those Newton iterations require to solve a linear system whose matrix is defined by the mass matrix $\mathbf{M}$ and the Jacobian $\mathbf{J}$. In fact, the structure of the linear system matrix is fully defined by those of the mass matrix and the Jacobian. Thus, a user who knows the structure of these two matrices could provide a solver that takes advantage of this structure. This could be a sparse solver, a band solver, a hermitian banded solver, ... This choice would lead to better performance than the general LU decomposition used currently.

4.2.7 Parallel Radau

In section 2.2.2 about how to apply Radau, the structure of the matrix of the Newton's iterations has not been discussed. However, the structure can be exploited using the fact that solving this system can be done by solving two independent linear systems, one of size n (for n the number of equations of the model) and one of size $2n$. In fact, using a greater number of stages will not change the fact that the maximum size of each independent subsystem is $2n$. Taking this observation into account, it's possible to parallelize the resolutions of the subsystems and reduce the impact of one major drawback of implicit Runge-Kutta methods: the size of the system to solve.

Conclusion and Perspectives

In the introduction, we mentioned three goals: studying mathematically DAEs, implementing DAEs solvers within the Scipy environment and comparing the performance of those solvers. Chapter 1 addressed the question of the usefulness of DAEs through practical applications. In this chapter, the notion of DAE and its index have been developed as a response to the first goal as well as numerical tools such as index reduction and stability concepts that apply on the numerical methods covered in this work. Then, chapter 2 has considered some classical numerical methods (BDF, RadauIIa and Rosenbrock) and new modifications of ABM. This chapter also studied the performance of those solvers on six practical tests problems with different characteristics which validates the third objective. Note that these results are still to be discussed in this final part. Chapter 3 tackled the problem of the stability analysis of a 2D Dhalquist test and discusses on how the structure of the modified ABM can be used to gain in performance. This chapter is an article that we have submitted to Mathematics in collaboration with other scientists. Finally, chapter 4 describes the implementation of those solvers within the Scipy environment through the *solve_ivp* function and how the code may be improved to reach better performance such that the second goal is reached. Our SABM code has been submitted to Scipy via a pull request.

Main results

- It's not mandatory to turn DAEs into ODEs. Classical solvers does not need a lot of modifications to be able to solve directly DAEs. However, not all solvers are able to solve high index DAEs. One of them is the RadauIIa method which is able to solve index 3 Hessenberg DAEs such as the mechanical problems. Solving a high index DAE is more difficult than an index 1 DAE as we shown with the pendulum problem. In addition, the error tables indicate the maximum error reached by each method for each test problem but you can

find figures in the different notebook of the GitHub repository that describe more the behaviour of the error.

- SEABM was the best performing solver at high tolerances for mechanical problems and for the transistor amplifier. However, this solver has been beaten by the classical methods for the other three problems including the Roberston stiff problem. Indeed, we studied the stability of this one and shown that it's very stiff. Explicit methods are not used in practice for this kind of problems and Adams-Bashforth-Moulton is not recommended for stiff problems. Thus, we have observed that even semi-explicit and semi-implicit modifications that expand the stability region of ABM does not seem to be adequate to solve stiff problems. Pure implicit methods must be preferred for those problems even if we have shown that all solvers eventually fail to integrate Robertson at medium tolerances since stiff equations are challenging problems.

- Concerning the filter and CSTR problems, we have given an explanation of why SEABM obtains worse performance than classical solvers. The idea is that since those problems have small eigenvalues in absolute value, they should be easier to solve but classical solvers having a more aggressive adaptative step size strategy allow themselves to loose accuracy to achieve better performance. This explanation is reinforced by the error tables in which BDF has always a worse accuracy than SABM but integrates with a smaller number of function evaluations. For the CSTR, SEABM has even achieved a better accuracy than Radau which has not been observed elsewhere.

- Nevertheless, SIABM has never beaten any classical method and its performance are clearly worse in all our tests. However, unlike the classical methods, it has always been able to integrate the Robertson problem even at low tolerances. We think that our very cautious strategy allows to solve this problem but at the cost of a huge number of steps. Note that, for this problem, Radau achieves the best error order of SIABM with higher tolerances and far fewer steps.

- Both methods do not seem to work well with low and medium tolerances but we think that it comes from the fact that we have used a fixed order method with the second order. Using a variable order strategy like BDF (order 1 to 5) may lead to other conclusions about these tolerances. Moreover, Radau is a fifth order method which can explain why this method achieves the best accuracy in general.

- On one hand, BDF seems less robust than SABM with high tolerances since BDF failed for the transistor amplifier while SABM never failed neither at

high nor at medium tolerances. This might be due to the cautious strategy step size that prevents SABM from failing even with stiff problems as shown with the Robertson's problem. On the other hand, Rosenbrock seems to perform better for high tolerances than low tolerances except for the transistor amplifier for which Radau performs as well as the linearized Runge-Kutta. This may seem paradoxical since Rosenbrock is built to be efficient at high tolerance, but in reality if we compare the execution time, Rosenbrock is faster than Radau at high tolerances except for electrical problems. Comparing Rosenbrock with Radau based on the execution time makes sense in this case since Rosenbrock is built to solve only a linear system instead of the non linear one of IRK methods which leads to better performance at high tolerances in most cases.

- Finally, in the article we shown that SEABM extends the stability region of ABM and explicit AB while SIABM extends the one of ABM but shrinks the stability region of implicit AM. However, unlike AM, SIABM does not have to solve the full non linear system but n non linear functions of 1 variable if n is the number of variables of the problem.

Improving paths

In addition to the improvements proposed in the Scipy part, we can also look at other aspects to make our methods more general.

- In MATLAB, the DAEs BDF solver *ode*15*s* solves problems with a varying mass matrix $\mathbf{M}(t, \mathbf{y})$ which requires, after having approximated the derivative of the matrix with respect to $\mathbf{y}$, minor modifications. Note that by change of variables, we can already write most problems $\mathbf{M}(t, \mathbf{y})\mathbf{y}' = \mathbf{f}(t, \mathbf{y})$ with a constant mass matrix but this increases the size of the system.

- Another step could be to implement a more general DAEs solver for the non linear DAEs described in section 1.2.2. Linda Petzold has already implemented such a solver in Fortran called DASSL. Therefore, a further step may be to implement a Python wrapper for this Fortran solver.

- An implementation of a technique able to turn a high index DAE into an index 1 can be realized to use the different solvers correctly. In MATLAB, the function *reduceDAEIndex* implements the Pantelides algorithm. Some techniques have been briefly explained in section 1.3.1.

- In the article, two other new methods called semi-explicit and semi-implicit AB-BDF have been introduced. They can be turned into DAEs solvers in the same way as SABM which would allow to test, benchmark and compare these new methods to the other ones of this work as it has been done for ODEs in the article.
- A future study could implement the proposed improvements and compare all these solvers on the basis of execution time. For larger systems, we should see a longer computation time for Radau which solves systems twice as large and SABM should take better advantage of solving each function one by one rather than the whole system.

Bibliography

[1] Jan Awrejcewicz. *Ordinary Differential Equations and Mechanical Systems*. 01 2014.

[2] J. Baumgarte. Stabilization of constraints and integrals of motion in dynamical systems. *Computer Methods in Applied Mechanics and Engineering*, 1(1):1–16, 1972.

[3] K.E. Brenan, S.L. Campbell, S.L.V. Campbell, and L.R. Petzold. *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*. Classics in Applied Mathematics. Society for Industrial and Applied Mathematics, 1996.

[4] Denis Butusov, Valerii Ostrovskii, Artur Karimov, and Valery Andreev. Semi-explicit composition methods in memcapacitor circuit simulation. *International Journal of Embedded and Real-Time Communication Systems*, 10:37–52, 04 2019.

[5] D.A. Calahan. A stable, accurate method of numerical integration for nonlinear systems. *Proceedings of the IEEE*, 56(4):744–744, 1968.

[6] Luca Cardelli. From processes to odes by chemistry. In Giorgio Ausiello, Juhani Karhumäki, Giancarlo Mauri, and Luke Ong, editors, *Fifth Ifip International Conference On Theoretical Computer Science – Tcs 2008*, pages 261–281, Boston, MA, 2008. Springer US.

[7] Luca Cardelli, Mirco Tribastone, and Max Tschaikowski. From electric circuits to chemical networks. *Nat. Comput.*, 19(1):237–248, 2020.

[8] François Cellier. Continuous system modeling / francois e. cellier. *SERBIULA (sistema Librum 2.0)*, 02 1991.

[9] Kenneth D. Clark. A structural form for higher-index semistate equations. i. theory and applications to circuit and control theory. *Linear Algebra and its Applications*, 98:169–197, 1988.

[10] Byron L. Ehle. *On pade approximations to the exponential function and a-stable methods for the numerical solution of initial value problems*. PhD thesis, Univeristy of Waterloo, 1969.

[11] Boris Faleichik. Minimal residual multistep methods for large stiff non-autonomous linear problems. *Journal of Computational and Applied Mathematics*, 387:112498, 09 2019.

[12] C. W. Gear. Differential-algebraic equation index transformations. *SIAM Journal on Scientific and Statistical Computing*, 9(1):39–47, 1988.

[13] Ernst Hairer, Syvert Norsett, and G. Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*, volume 8. 01 1993.

[14] Ernst Hairer and Gerhard Wanner. *Solving Ordinary Differential Equations II. Stiff and Differential-Algebraic Problems*, volume 14. 01 1996.

[15] H. Jafari, R.M. Ganji, N.S. Nkomo, and Y.P. Lv. A numerical study of fractional order population dynamics model. *Results in Physics*, 27:104456, 2021.

[16] Nematollah Kadkhoda, Hossein Jafari, and R.M. Ganji. A numerical solution of variable order diffusion and wave equations. *International Journal of Nonlinear Analysis and Applications*, 12(1):27–36, 2021.

[17] Artur Karimov, Denis Butusov, and Aleksandra Tutueva. Adaptive explicit-implicit switching solver for stiff odes. pages 440–444, 01 2017.

[18] Matt Keeling, Pejman Rohani, and Babak Pourbohloul. Modeling infectious diseases in humans and animals. *Clinical infectious diseases : an official publication of the Infectious Diseases Society of America*, 47:864–865, 10 2008.

[19] Aditya Kumar and Prodromos Daoutidis. *Control of Nonlinear Differential Algebraic Equation Systems : An Overview*, pages 311–344. Springer Netherlands, Dordrecht, 1998.

[20] Meng Li, Boshan Chen, and Huawen Ye. A bioeconomic differential algebraic predator–prey model with nonlinear prey harvesting. *Applied Mathematical Modelling*, 42:17–28, 2017.

[21] Chunfeng Liu, Haiming Wu, Li Feng, and Aimin Yang. Parallel fourth-order runge-kutta method to solve differential equations. volume 7030, pages 192–199, 10 2011.

[22] Salvatore Lopez. A predictor–corrector time integration algorithm for dynamic analysis of nonlinear systems. *Nonlinear Dynamics*, 101, 07 2020.

[23] Bruce N. Lundberg and Aubrey B. Poore. Variable order adams-bashforth predictors with an error-stepsize control for continuation methods. *SIAM Journal on Scientific and Statistical Computing*, 12(3):695–723, 1991.

[24] Constantinos C. Pantelides. The consistent initialization of differential-algebraic systems. *SIAM Journal on Scientific and Statistical Computing*, 9(2):213–231, 1988.

[25] L. R. Petzold. Description of dassl: a differential/algebraic system solver. 9 1982.

[26] A Polynikis, Stephen Hogan, and Mario Di Bernardo. Comparing different ode modeling approaches of gene regulatory networks. *Journal of theoretical biology*, 261:511–30, 09 2009.

[27] Harald A. Posch, William G. Hoover, and Franz J. Vesely. Canonical dynamics of the nosé oscillator: Stability, order, and chaos. *Phys. Rev. A*, 33:4253–4265, Jun 1986.

[28] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, 3 edition, 2007.

[29] Thomas Rauber and Gudula Rünger. Parallel implementations of iterated runge-kutta methods. *The International Journal of Supercomputer Applications and High Performance Computing*, 10(1):62–90, 1996.

[30] O. E. Rössler. An equation for continuous chaos. *Physics Letters A*, 57(5):397–398, July 1976.

[31] Peter Stechlinski, Michael Patrascu, and Paul I. Barton. Nonsmooth differential-algebraic equations in chemical engineering. *Computers Chemical Engineering*, 114:52–68, 2018. FOCAPO/CPC 2017.

[32] Aleksandra Tutueva and Denis Butusov. Stability analysis and optimization of semi-explicit predictor–corrector methods. *Mathematics*, 9:2463, 10 2021.

[33] Aleksandra Tutueva, Timur Karimov, and Denis Butusov. Semi-implicit and semi-explicit adams-bashforth-moulton methods. *Mathematics*, 8, 05 2020.

[34] Balthasar Van Der Pol. The nonlinear theory of electric oscillations. *Proceedings of the Institute of Radio Engineers*, 22:1051–1086.

[35] A. J. van der Schaft. *Port-Hamiltonian Differential-Algebraic Systems*, pages 173–226. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[36] Anqi Zhang, Roghayeh Moallem Ganji, Hossein Jafari, Mahuli Naisbitt Ncube, and Latifa Agamalieva. Numerical solution of distributed-order integro-differential equations. *Fractals*.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**