On the selection of increments in Jacobian approximations for numerical solvers of ordinary differential equations

An experimental study

Thomas Renström

Master's thesis
2024:E8

Lund University

Faculty of Science
Centre for Mathematical Sciences
Numerical Analysis

CENTRUM SCIENTIARUM MATHEMATICARUM

Abstract

Some numerical methods approximate the Jacobian using finite differences as part of their process. The finite difference method of approximation resembles the limit definition of the derivative. Since a computer cannot handle the mathematical concept of limits, a sufficiently small number has to be chosen as an increment. However, when the increments are chosen outside of certain acceptable ranges, we need to account for error stemming from issues regarding representability. In this thesis we will look at the mathematical limitations of the increments. We will also go over how they are chosen in 3 different solvers, and suggest ways to improve upon them.

Populärvetenskaplig sammanfattning

Inom matematik så används något som kallas derivator för att beräkna förändring. Ett exempel är hastighet som är positionens derivata. När man uppskattar hastighet så dividerar man förflyttningssträckan med tidsåtgången. För att ge en mer precis uppskattning för hastigheten i stunden så kan man då minska tiden man beräknar över. När detta görs på datorer så stöter man på problem när stegen blir för små. Detta grundar sig i hur datorer lagrar information. Det blir helt enkelt problem när man försöker summera ett stort och litet tal. Lite förenklat så kan datorn inte se skillnaden på hastigheten vid 15 minuter efter fyra och 15 minuter och 113 miljontedels sekunder efter fyra. Detta arbete ser över de matematiska begränsningarna för hur denna tidsskillnad kan väljas, och hur det praktiskt görs i 3 olika simuleringsalgoritmer. Vi föreslår sedan hur förbättringar skulle kunna göras.

In loving memory of K.G. & Ingrid Andersson
and Anita Renström

Acknowledgements

I want to thank everybody at Modelon for treating me like one of them during my time at the office. Also a big thanks to my advisors Peter and Philipp for having patience with my incoherent ramblings. Of course an extremely huge thanks to my parents and to my dear wife Sofie who has been working hard to support me both emotionally and financially during my studies.

Contents

1 Introduction **1**

2 Preliminaries **3**
2.1 Newton's method 4
2.2 Radau IIA 6

3 Floating point numbers **9**

4 Error control **15**
4.1 Data error for the finite differences method 16
4.2 Computational error for the finite difference method 17
4.3 Total error of the finite difference method 18
4.4 The representability of the increments 19

5 Software stack **21**
5.1 Creating and compiling a model 22
5.2 PyFMI 22
5.3 CVode 25
5.4 Radau5 28

6 Models **33**
6.1 Model 1 - Van der Pol oscillator 33
6.2 Model 2 - Pneumatics 36
6.3 Model 3 - Hydraulics 40
6.4 Model 4 - Double wishbone suspension 43

7 Numerical experiments **47**
7.1 Experiment 1 - Post-processing algorithms 48
7.2 Experiment 2 - Smaller nominal values 48
7.3 Experiment 3 - No nominal values 52
7.4 Experiment 4 - Different curvature scale 55
7.5 Experiment 5 - Powers of 2 58

8 Summary and conclusion **61**
8.1 Regarding nominal values 61
8.2 General improvements 62
8.3 Other considerations 63

A The design of $sigma_0$ in CVode **69**

Chapter 1

Introduction

Simulations play an increasingly important role in industry. There are several advantages to being able to test a theory or model in a computer simulation instead of manufacturing prototypes. Most obvious are the financial and ecological savings that follow from having less prototypes. Another is the possibility of more technical progress when more ideas can easily be tested.

One widely used standard for setting up simulations is the *Functional Mock-Up Interface* (FMI) [1] standard. The FMI standard has been in development for over a decade. Version 1.0 was released in October 2010 and version 3.0 was released in May of 2022. It is developed as a Modelica Association Project [2] along the Modelica language [3] and the Modelica libraries [4]. The problems are stored as *Functional Mock-Up Units* or FMUs. It is a well established standard which means that if a model is defined as an FMU it can easily be handled by a range of different softwares without the need to rebuild it. One important advantage is also that FMUs do not reveal the inner workings of the model which could very well contain trade secrets.

FMI defines two types of models [5, p.3]. In this thesis we will consider *model exchange FMUs*, these contain a model to be solved using external tools. These FMUs make all the information necessary for a solver to perform a simulation available. The other type, *co-simulation FMUs*, are used for coupled models. They contain not only the model but also a solver. Because of this the FMUs can keep much more information hidden, exposing only the ability to make time steps.

The idea for this thesis comes from the company Modelon. Modelon develops PyFMI [5], an open-source Python package for interacting with FMUs. They had experienced performance issues with two of the ODE solvers they use, called CVode [6, p.366] and Radau5 [7, p.118]. Both of these solvers build on implicit methods and at each timestep they use versions of New-

ton's method. Newton's method utilizes the Jacobian, and when it is not supplied by the user an approximation of it has to be made. The issue arose when, instead of using the Jacobian approximation native to the methods, an approximation defined in PyFMI was used. This led to the simulation needing to take a lot more steps in order to finish. Since both the solvers and PyFMI use a difference approximation [8, p.79], what sets the them apart is how they choose the increments in the approximation.

Since the Jacobian is costly to approximate due to the number of function evaluations, we instead use a simplified version of Newton's method. The simplified Newton's method, or Chord method [8, p.76], reuses the Jacobian as long as the result is satisfactory.

PyFMI has access to problem specific information from the FMU and uses this to improve its performance. For example, PyFMI compresses the process of approximating Jacobians by calculating directional derivatives of independent state variables simultaneously [5, p.8]. PyFMI also utilizes the nominal values of state variables stored in the FMU for choosing the increments in the difference approximation.

An initial look at the models provided by Modelon indicated that some of the simulations using PyFMIs Jacobian approximations got stuck and attempted smaller and smaller timesteps. For some of the models this resulted in the simulations needing a lot more steps to finish than when using the methods Jacobians. Some of the models could not finish at all.

In this thesis we will make an in-depth analysis of the different error sources for the difference approximation of the Jacobian. We will present a mathematical motivation for how the increments in the approximation should be chosen in order to control the error. We shall also see how the increments are chosen in PyFMI, CVode and Radau5.

We will show that PyFMIs choice of increments based upon nominal values is sub-optimal and suggest several ways it can be improved upon.

To this end we will first explain floating point numbers in chapter 3 and the different errors in chapter 4. Then we will look at the different choices of increments as they are made in PyFMI as well as in the solvers CVode and Radau5 in chapter 5. The models used and the numerical experiments performend are explained in chapters 6 and 7. Finally our findings are presented in chapter 8.

Chapter 2

Preliminaries

In this thesis we consider a time dependent ODE of the form

$$\dot{y}(t) = f(t, y), \quad y^0 = y(t_0)$$

with $t \in [t_0, t_{END}]$ and $y \in \mathbb{R}^N$. Using numerical methods we attempt to approximate the result

$$\begin{bmatrix} y^1 \\ \vdots \\ y^{END} \end{bmatrix} \approx \begin{bmatrix} y(t_1) \\ \vdots \\ y(t_{END}) \end{bmatrix}.$$

Numerical methods can be separated into two categories, explicit and implicit. An s-step implicit scheme will depend on previous points as well as the point solved for,

$$y^n = \Phi(t_n, \dots, t_{n-s}, y^n, \dots, y^{n-s}), \tag{2.1}$$

while an explicit would only use previously calculated points. As an example, for the 1-step Implicit Euler method we have that

$$\Phi_{IE}(t_n, t_{n-1}, y^n, y^{n-1}) = y^{n-1} + h_n f(t_n, y^n),$$

with step size $h_n = t_n - t_{n-1}$.

In order to solve Equation (2.1) we reformulate it into the problem of finding the value y^n that is a root of

$$G(y) = y - \Phi(t_n, \dots, t_{n-s}, y, y^{n-1}, \dots, y^{n-s}).$$

There are several root finding algorithms to solve such problems. In this thesis we will discuss Newton's method and the simplified Newton's method.

2.1 Newton's method

Newton's method [8, p.71] is an iterative root finding algorithm. For a function $G(y) = [G_1(y), \ldots, G_N(y)]^{\mathrm{T}}$, $y \in \mathbb{R}^N$, it finds a root y^* of G, with an initial guess $y^{n(0)}$ through the iterative process

$$y^{n(m)} = y^{n(m-1)} - G'(y^{n(m-1)})^{-1} G(y^{n(m-1)}) \tag{2.2}$$

where $y^{n(m)} \to y^*$ as $m \to \infty$. The Jacobian, $G'(y)$, is defined as

$$G'(y) := \begin{bmatrix} \frac{\partial G_1}{\partial y_1} & \cdots & \frac{\partial G_1}{\partial y_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial G_N}{\partial y_1} & \cdots & \frac{\partial G_N}{\partial y_N} \end{bmatrix}.$$

Since the inversion of a matrix is very costly, an algorithm would use a reformulation of Equation (2.2). The process instead becomes

$$y^{n(m)} = y^{n(m-1)} + \Delta y^{n(m-1)},$$

where $\Delta y^{n(m-1)}$ is found by solving

$$G'(y^{n(m-1)}) \Delta y^{n(m-1)} = -G(y^{n(m-1)}) \tag{2.3}$$

using a linear solver [8, p.74].

With an initial guess close enough to a root, Newton's method converges q-quadratically [8, p.71]. This type of convergence is specified in Definition 2.1.1.

Definition 2.1.1. *[8, p.65] Let $\{x^n\} \subset \mathbb{R}^N$ and $x^* \in \mathbb{R}^N$. We say that x^n converges q-quadratically to x^* if $x^n \to x^*$ and there is $K > 0$ such that*

$$\left\| x^{n+1} - x^* \right\| \leq K \left\| x^n - x^* \right\|^2,$$

for n sufficiently large.

It can be shown that a suitable stopping condition for the iterative process is the relation $\left\| G(y^{n(m)}) \right\| / \left\| G(y^{n(0)}) \right\|$ [8, p.72]. However, if the initial guess $y^{n(0)}$ is too close to a root, or if there is an error in the evaluation of G, the process may be stopped too late or not stopped at all. To account for this we combine a relative and an absolute tolerance, as is common for numerical

methods for ordinary differential equations [8, p.73]. We stop the Newton iteration if

$$\left\|G(y^{n(m)})\right\| \leq \gamma_R \left\|G(y^{n(0)})\right\| + \gamma_A,$$

for some relative tolerance γ_R and absolute tolerance γ_A. Another possible termination condition would be if the Newton iteration step

$$\left\|y^{n(m)} - y^{n(m-1)}\right\|$$

is sufficiently small [8, p.73].

2.1.1 Finite differences method

When we approximate the Jacobian of the vector valued function

$$G(y) = [G_1(y), \ldots, G_N(y)]^{\mathrm{T}} \qquad y \in \mathbb{R}^N$$

using the finite difference method [8, p.79] we have that the element at row i and column j of the Jacobian is

$$J_{ij}(y) = \frac{G_i(y + \sigma_j \delta_{ij}) - G_i(y)}{\sigma_j} \approx G'(y)_{ij}$$

for some non-zero $\sigma_j \in \mathbb{R}$, and δ_{ij} the Kronecker delta [9],

$$\delta_{ij} := \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{otherwise.} \end{cases}$$

When $\sigma_j > 0$ we can also use the term *Forward differences method* and otherwise *Backward differences method.*

In order to reduce the number of evaluations of the function G we evaluating one column at a time. The jth column is approximated as

$$J_j(y) = \frac{G(y + \sigma_j e_j) - G(y)}{\sigma_j}$$

for some non-zero $\sigma_j \in \mathbb{R}$, and e_j the unit vector. Since $J \in \mathbb{R}^{N\times N}$ we thus need $N+1$ evaluations of G in order to approximate the Jacobian. However, since $G(y)$ is also used in the right hand side of Newton's method, it will already have been evaluated. This means we only need to make N function evaluations for the Jacobian.

2.1.2 Simplified Newton's method

If we work with dense $N \times N$ Jacobians, each iteration of Newton's method requires $N + 1$ evaluations of the function G. One for the right hand side of Equation (2.3), and N for the Jacobian on the left hand side. For M iterations on Newton's method this means a total of $M(N + 1)$ function calls. The simplified Newton's method, or Chord method [8, p.76], attempts to reduce the number of function calls by only calculating the Jacobian at the first Newton iteration and then reusing it. This loss of precision that this entails could lead to more work performed, but this should be outweighed by the savings made by reducing the number of function calls.

This method is given by

$$y^{n(m)} = y^{n(m-1)} - J_G^{-1}(y^{n(0)})G(y^{n(m-1)}).$$

The drawback with using this simplified version of Newton's method is that the convergence rate is reduced. Instead of q-quadratic, the convergence is q-linear [8, 76], a rate of convergence specified in Definition 2.1.2.

Definition 2.1.2. *[8, p.65] Let $\{x^n\} \subset \mathbb{R}^N$ and $x^* \in \mathbb{R}^N$. Then $x^n \to x^*$ q-linearly with q-factor $\upsilon \in (0, 1)$ if*

$$\left\|x^{n+1} - x^*\right\| \leq \upsilon \left\|x^n - x^*\right\|,$$

for n sufficiently large.

By the triangle inequality we have that

$$\begin{aligned}\left\|x^n - x^*\right\| - \left\|x^{n+1} - x^*\right\| &\leq \left\|x^{n+1} - x^* - (x^n - x^*)\right\| \\ &\leq \left\|x^{n+1} - x^*\right\| + \left\|x^n - x^*\right\|.\end{aligned}$$

If the iteration is q-linearly convergent, it implies that

$$(1 - \upsilon)\left\|x^n - x^*\right\| \leq \left\|x^{n+1} - x^n\right\| \leq (1 + \upsilon)\left\|x^n - x^*\right\|.$$

Thus we have that the size of the iteration step is a reliable indicator of the error, as long as υ is not too near 1 [8, p.73].

2.2 Radau IIA

The Radau methods [7, p.118] are a class of implicit Runge-Kutta methods. Radau IIA are s-stage implicit Runge-Kutta methods of order $2s - 1$ that are A-stable [7, p.118].

As an implicit RK method they use the scheme

$$g^i = y^{n-1} + h\sum_{j=1}^{s} a_{ij} f(t_{n-1} + c_j h, g^j), \quad i = 1, \ldots, s \tag{2.4}$$

$$y^n = y^{n-1} + h\sum_{j=1}^{s} b_j f(t_{n-1} + c_j h, g^j), \tag{2.5}$$

with the coefficients given by the Butcher's tableau

$$\begin{array}{c|c} c & A \\ \hline & b \end{array} = \begin{array}{c|ccc} c_1 & a_{11} & \cdots & a_{1s} \\ \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s1} & \cdots & a_{ss} \\ \hline & b_1 & \cdots & b_s. \end{array}$$

In order to minimize influence of round-off errors, Equation (2.4) is redefined with the smaller quantities $z^i = g^i - y^{n-1}$ [7, p.118] yielding

$$z^i = h\sum_{j=1}^{s} a_{ij} f(t_{n-1} + c_j h, y^{n-1} + z^j), \quad i = 1, \ldots, s. \tag{2.6}$$

If the solution $z^1, \ldots, z^s$ of Equation (2.6) is known, then Equation (2.5) is an explicit formula for y^n with s function evaluations. However, if the matrix A is non-singular, these can be avoided. We assume this to be true and rewrite Equation (2.6) as

$$\begin{pmatrix} z^1 \\ \vdots \\ z^s \end{pmatrix} = A \begin{pmatrix} hf(t_{n-1} + c_1 h, y^{n-1} + z^1) \\ \vdots \\ hf(t_{n-1} + c_s h, y^{n-1} + z^s) \end{pmatrix}$$

and Equation (2.5) as

$$y^n = y^{n-1} + \sum_{i=1}^{s} d_i z^i \tag{2.7}$$

where

$$(d_1, \ldots, d_s) = (b_1, \ldots, b_s)A^{-1}.$$

Because of the way the Butcher's tableaus of Radau IIA methods are constructed, the vector d will consist of $s-1$ zeros and a one at the end. For example, the 3-stage Radau IIA methods has $d = (0, 0, 1)$. This means we can further rewrite Equation (2.7) as

$$y^n = y^{n-1} + z^s.$$

Chapter 3

Floating point numbers

Non-integer numbers on a computer are typically as *floating point numbers*, or *floats* for short. Floating point numbers save on memory by instead of using all the bits available to store significant numbers, a set number of bits are used to store the exponent. The term floating point number comes from the fact that the exponent E varies, or *floats*, between a lower and upper limit, L and U. This is contrast to fixed point numbers, where the exponent is static.

By [10, p.16] a system $\mathbb{F}$ of float numbers is characterized by a base, β; a precision, p; and an exponent range, $[L, U]$.

An element $x \in \mathbb{F}$ then has the form

$$x = (-1)^S \left(c_0 + \frac{c_1}{\beta^1} + \frac{c_2}{\beta^2} + \cdots + \frac{c_{p-1}}{\beta^{p-1}} \right) \beta^E \tag{3.1}$$

for $S \in \{0, 1\}$, $E \in [L, U] \subset \mathbb{Z}$ and $c_i \in [0, \beta - 1] \subset \mathbb{Z}$ for $i = 0, \ldots, p - 1$.

Alternatively, as in [11, p.16], we can write $x \in \mathbb{F}$ as

$$x = \frac{s}{\beta^{p-1}} \beta^E = s\beta^{E-(p-1)}, \tag{3.2}$$

where $s = (-1)^S \left(\beta^{p-1}c_0 + \beta^{p-2}c_1 + \cdots + c_{p-1}\right) \in \mathbb{Z}$. Since $c_i \in [0, \beta - 1]$ it follows that $s \in [1 - \beta^p, \beta^p - 1] \subset \mathbb{Z}$.

Modern computing follows the IEEE 754 standard for binary formats, which defines five different float numbers precisions. In this thesis we will use double precision floating point numbers. A double precision float takes up 64 bits in total. One bit is used for the sign S, 11 bits are used to store the exponent E leaving 52 bits to store the significand. However, in the normal range, the leading 1 is implicit and not stored, meaning that $p = 53$.

When we say that x is in the *normal range*, we mean that

$$\beta^{e_{min}} \leq |x| \leq \left(\beta - \beta^{p-1}\right) \beta^{e_{max}}.$$

When $|x| < \beta^{e_{min}}$, we say that x is *subnormal.*

In this thesis we will use $\mathrm{fl}(x)$ to indicate the floating point representation of x.

In order to define the error that occurs when we perform floating point operations we first define the notion of exact representability for floats.

Definition 3.0.1. *If for a real number $x \in \mathbb{R}$*

$$\mathrm{fl}(x) = x$$

we say that x is exactly representable and that $\mathrm{fl}(x)$ is exact.

A problem that follows using floating point numbers is that the result of simple operations between two exact floats might not be correctly represented. As a simple example, using two numbers in base 10 with 3 digit precision, and representing the result with the same precision we get

$$1.23 - 0.00456 = 1.23.$$

In 1974, Sterbenz presented a theorem for subtraction of floating point numbers specifically for the IBM/System 360 [12, 137]. We use here a version of what is known as Sterbenz lemma from a more recent source.

Lemma 3.0.2. *[11, p.101]*

If a, b are non-negative finite floating point numbers such that

$$b \leq a \leq 2b,$$

then $a - b$ is exactly representable.

Proof. [11, p.101] We first note that if $b = 0$ then $a - b = a$ and is therefore exact. Also, if $a = b$ then $a - b = 0$ and also exact. We assume that $0 < b < a \leq 2b$ and represent a and b as

$$a = s_a \beta^{e_a - p + 1} \qquad b = s_b \beta^{e_b - p + 1},$$

where

$$\begin{aligned} e_{min} \leq & e_a \leq e_{max} \\ e_{min} \leq & e_b \leq e_{max} \\ 0 \leq & s_a \leq \beta^p - 1 \\ 0 \leq & s_b \leq \beta^p - 1. \end{aligned}$$

Since $b < a$ it follows that $e_b \leq e_a$, and thus we can write

$$a = s_a \beta^{e_a - e_b} \beta^{e_b - p+1},$$

where $\beta^{e_a - e_b}$ is a positive integer. With this we can expand $a - b$,

$$\begin{aligned} a - b &= s_a \beta^{e_a - p+1} - s_b \beta^{e_b - p+1} \\ &= \left(s_a \beta^{e_a - e_b} - s_b \right) \beta^{e_b - p+1}. \end{aligned}$$

We define $s' := s_a \beta^{e_a - e_b} - s_b$ and note that $s' > 0$ since $a - b > 0$. By assumption we also have that $a - b \leq b$, implying

$$s' \beta^{e_b - p+1} \leq s_b \beta^{e_b - p+1}$$

or equivalently

$$s' \leq s_b.$$

Since $s_b \leq \beta^p - 1$ it follows that

$$s' \leq \beta^p - 1.$$

Since $a - b = s' \beta^{e_b - p+1}$ with

$$\begin{aligned} e_{min} \leq & e_b \leq e_{max} \\ 0 \leq & s' \leq \beta^p - 1, \end{aligned}$$

it follows that $a - b$ is a floating number. Thus $a - b = \text{fl}(a - b)$ and therefore $a - b$ is exact with the stated assumptions. □

This lemma 3.0.2 has further implications that Sterbenz presented in a corollary [12, p.138]. In this thesis we will first show these implication in two separate remarks, before arriving at the corollary.

Remark 3.0.3. *If a, b are two positive floating point numbers of the same sign such that*

$$\frac{b}{2} \leq a \leq 2b,$$

then the differences $a - b$ and $b - a$ are exactly representable.

Proof. We have from lemma 3.0.2 that $a - b$ is exactly representable if

$$b \leq a \leq 2b.$$

In the same way, $b - a$ is exactly representable if

$$b \leq a \leq 2b,$$

or equivalently,

$$\frac{b}{2} \leq \frac{a}{2} \leq b.$$

Together, these inequalities gives

$$\frac{b}{2} \leq \frac{a}{2} \leq b \leq a \leq 2b,$$

or

$$\frac{b}{2} \leq a \leq 2b.$$

□

The second remark deals with negative floating point numbers.

Remark 3.0.4. *If a, b are two finite floating point numbers of the same sign such that*

$$|b| \leq |a| \leq 2\,|b|\,,$$

then $a - b$ is exactly representable.

Proof. This follows from lemma 3.0.2 and the fact that for all floating point numbers, their additive inverse is also a floating point number. □

With these two remarks we now reconstruct Sterbenz corollary.

Corollary 3.0.5. *[12, p.138] If a, b are two finite floating point numbers of the same sign such that*

$$\frac{|b|}{2} \leq |a| \leq 2\,|b|\,,$$

then the differences $a - b$ and $b - a$ are exactly representable.

Proof. This follows from remarks 3.0.3 and 3.0.4. □

With a firm grasp of what it means for a number to be exact, we can now look at the smallest representable difference of two floating point numbers.

Theorem 3.0.6. *The minimal exactly representable difference of two distinct floating points in the same system with base β, precision p and exponent E is*

$$\beta^{E-(p-1)}.$$

Proof. We take $x_1, x_2 \in \mathbb{F}$ to be two neighboring floating points. Without loss of generality we choose $x_2 > x_1$. By Equation (3.2) we have that

$$\begin{aligned} x_1 &= s_1 \beta^{E-(p-1)} \\ x_2 &= s_2 \beta^{E-(p-1)} = (s_1 + 1)\beta^{E-(p-1)}. \end{aligned}$$

It follows that

$$\begin{aligned} |x_2 - x_1| &= \left|(s_1 + 1)\beta^{E-(p-1)} - s_1 \beta^{E-(p-1)}\right| \\ &= \left|\beta^{E-(p-1)}\right|. \end{aligned}$$

□

It follows from Theorem 3.0.6 that the smallest step from 1 is $\beta^{-(p-1)}$, since in that case $E = 0$.

Closely related to the smallest representable step from a floating point number is the relative error of a floating point rounding [11, p.26].

Theorem 3.0.7. *[11, p.26] For x in the normal range, the floating point rounding error for a floating point system with radix β and precision p satisfies*

$$\varepsilon_f = \left|\frac{x - \mathrm{fl}(x)}{x}\right| < \begin{cases} \frac{1}{2}\beta^{-(p-1)} & \text{using rounding to nearest,} \\ \beta^{-(p-1)} & \text{using directed rounding.} \end{cases}$$

Proof. [11, p.26] Since x is in the normal range, there is a unique integer e such that $\beta^e \leq |x| < \beta^{e+1}$ for some $e \in \mathbb{Z}$. If $|x| = \beta^e$ then $\varepsilon_f = 0$. We assume now that $|x| > \beta^e$. The distance between two neighboring floating points is exactly β^{e-p+1} so the absolute error due to rounding satisfies

$$|x - \mathrm{fl}(x)| < \frac{1}{2}\beta^{e-p+1}, \quad \text{for rounding to nearest,}$$

and

$$|x - \mathrm{fl}(x)| < \beta^{e-p+1}, \quad \text{for directed rounding.}$$

Dividing by $|x| > \beta^e$ completes the proof. □

We define the unit roundoff [13] as the relative error.

Definition 3.0.8. *[11, p.27] The unit roundoff of a floating point system with radix β and precision p is defined as*

$$U := \begin{cases} \frac{1}{2}\beta^{-(p-1)} & \text{using rounding to nearest,} \\ \beta^{-(p-1)} & \text{using directed rounding.} \end{cases}$$

In order to be applicable to all types of rounding we will use the larger $U = \beta^{-(p-1)}$ in this thesis.

From $\left|\frac{x-\mathrm{fl}(x)}{x}\right| < U$ it follows that

$$\mathrm{fl}(x) = x + \mathcal{O}\left(|x|\,U\right), \tag{3.3}$$

where $\mathcal{O}\left(\cdot\right)$ is the Big-$\mathcal{O}$ notation [14], indicating the magnitude of the error.

When dividing two floating point numbers,

$$x_1 = s_1\beta^{e_x-p+1} \qquad x_2 = s_2\beta^{e_y-p+1},$$

the operation [15] is performed as

$$\frac{x_1}{x_2} = \frac{s_1}{s_2}\beta^{e_x-e_y}.$$

We know from Equation (3.1) that x_1 and x_2 each have p digits. If x_1 has $p-k_1$ sequential zeros at the end, and x_2 has $p-k_2$, that is:

$$x_1 = (-1)^{S_1}\left(c_0 + \frac{c_1}{\beta} + \cdots + \frac{c_{k_1-1}}{\beta^{k_1-1}} + \frac{0}{\beta^{k_1}} + \cdots \frac{0}{\beta^{p-1}}\right)\beta^{e_1}$$
$$x_2 = (-1)^{S_2}\left(c_0 + \frac{d_1}{\beta} + \cdots + \frac{d_{k_2-1}}{\beta^{k_2-1}} + \frac{0}{\beta^{k_2}} + \cdots \frac{0}{\beta^{p-1}}\right)\beta^{e_2}.$$

Then x_1/x_2 is exact if $k_1 + k_2 \leq p$ [11, p.103]. Division is also exact if the denominator is a power of the radix, β, assuming the division does not result in over- or underflow [11, p.103].

Chapter 4

Error control

Our aim is to control the *total error* through control of the increments of the finite difference method, since we have no way of changing the other parameters of the error. We will achieve this by assuring that the error stemming from the finite difference is larger then any other errors that might occur. The first step to this end is to have an overview of all the different sources of error.

We assume a 1-dimensional function $g : \mathbb{R} \to \mathbb{R}$ defined around a point $z \in \mathbb{R}$. g is approximated by $\hat{g}$, and the point z is approximated by $\hat{z}$.

Definition 4.0.1. *For a computed approximation $\hat{g}$ to the function g and a inexact representation $\hat{z}$ of the point z we define the total error as*

$$\varepsilon_{TOT} := \|\hat{g}(\hat{z}) - g(z)\| .$$

Using the terminology used in [10, p.6] we then separate the total error using the following conjecture.

Conjecture 4.0.2. *For the total error ε_{TOT} as in Definition 4.0.1, we can perform the following separation.*

$$\begin{aligned} \varepsilon_{TOT} &= \|\hat{g}(\hat{z}) - g(z)\| \\ &\leq \|\hat{g}(\hat{z}) - g(\hat{z})\| + \|g(\hat{z}) - g(z)\| \\ &= \varepsilon_{COMP} + \varepsilon_{DATA}, \end{aligned}$$

where ε_{COMP} is the computational error and ε_{DATA} is the data error.

Definition 4.0.3. *For a computed approximation $\hat{g}$ to the function g and the point $\hat{z}$ we have the computational error*

$$\varepsilon_{COMP} := \|\hat{g}(\hat{z}) - g(\hat{z})\| ,$$

and

Definition 4.0.4. *For a function g and $\hat{z}$ approximating the point z we have the data error*

$$\varepsilon_{DATA} := \|g(\hat{z}) - g(z)\| .$$

By separating the total error in this fashion we can see that the computational error depends only on $\hat{z}$ while the data error depends on both $\hat{z}$ and z. This means that we will see no propagation of the error in data input in ε_{COMP}, and therefore not in the finite difference method. Error propagation [16] would otherwise mean that a simple linear error in the input data could grow depending on the unknown structure of $\hat{g}$ and could overtake the error of the method.

We can also see that the data error does not depend on the method of approximation. This means that we cannot control the data error by improving the method. We can however use it as a lower limit.

The computational error is a combination of rounding errors and *truncation error* [10, p.8], where the truncation error is the difference between the true result and approximated result produced by a numerical algorithm. The name comes from the truncation of infinite series into finite ones.

4.1 Data error for the finite differences method

When we use the finite difference method to approximate the derivative of the function $F(u)$, we have by Definition 4.0.4 that

$$\varepsilon_{DATA} = \|F'(\mathrm{fl}(u)) - F'(u)\| . \tag{4.1}$$

By Equation (3.3) we have that $\mathrm{fl}(u) = u + \mathcal{O}(|u|\, U)$. Substituting this into Equation (4.1) and applying Taylor's theorem we get

$$\begin{aligned} \varepsilon_{DATA} &= \|F'(\mathrm{fl}(u)) - F'(u)\| \\ &= \|F'(u + \mathcal{O}(|u|\, U)) - F'(u)\| \\ &= \mathcal{O}(|F''(u)|\, |u|\, U) \end{aligned}$$

It follows that if the input is exact, i.e. $\mathrm{fl}(u) = u$, then $\varepsilon_{DATA} = 0$.

4.2 Computational error for the finite difference method

The truncation error of the finite difference method approximation of the derivative $F'(\mathrm{fl}(u))$ is

$$\varepsilon_t = \left\| \frac{F(\mathrm{fl}(u)+\sigma) - F(\mathrm{fl}(u))}{\sigma} - F'(\mathrm{fl}(u)) \right\|. \tag{4.2}$$

As previously mentioned the computational error is a combination of the truncation error and rounding error, so we add the rounding errors to Equation (4.2), yielding

$$\varepsilon_{COMP} = \left\| \mathrm{fl}\left(\frac{\mathrm{fl}(\mathrm{fl}(F(\mathrm{fl}(\mathrm{fl}(u)+\mathrm{fl}(\sigma)))) - \mathrm{fl}(F(\mathrm{fl}(u))))}{\mathrm{fl}(\sigma)} \right) - F'(\mathrm{fl}(u)) \right\|.$$

The outermost floating point rounding is for the division, so if we assume that the division is exact, we can remove it.

$$\varepsilon_{COMP} = \left\| \frac{\mathrm{fl}(\mathrm{fl}(F(\mathrm{fl}(\mathrm{fl}(u)+\mathrm{fl}(\sigma)))) - \mathrm{fl}(F(\mathrm{fl}(u))))}{\mathrm{fl}(\sigma)} - F'(\mathrm{fl}(u)) \right\|.$$

We separate the error from the outermost floating point rounding of the nominator according to Equation (3.3), giving us

$$\begin{aligned}\varepsilon_{COMP} \leq & \left\| \frac{\mathrm{fl}(F(\mathrm{fl}(\mathrm{fl}(u)+\mathrm{fl}(\sigma)))) - \mathrm{fl}(F(\mathrm{fl}(u)))}{\mathrm{fl}(\sigma)} - F'(\mathrm{fl}(u)) \right\| \\ & + \mathcal{O}\left(\frac{|F(\mathrm{fl}(u))|\, U}{\mathrm{fl}(\sigma)} \right).\end{aligned}$$

This is the rounding that would give rise to the so called *cancellation error.*

Since we have separated the data error we have no error propagation in the evaluation of F, this would otherwise have been a more significant source of error [8, p.80]. The evaluation of F might still lose information due to floating point rounding however. Expanding this rounding, again by Equation (3.3), we get

$$\begin{aligned}\varepsilon_{COMP} \leq & \left\| \frac{F(\mathrm{fl}(\mathrm{fl}(u)+\mathrm{fl}(\sigma))) - F(\mathrm{fl}(u))}{\mathrm{fl}(\sigma)} - F'(\mathrm{fl}(u)) \right\| \\ & + \mathcal{O}\left(2\frac{|F(\mathrm{fl}(u))|\, U}{\mathrm{fl}(\sigma)} \right) + \mathcal{O}\left(\frac{|F(\mathrm{fl}(u))|\, U}{\mathrm{fl}(\sigma)} \right) \\ = & \left\| \frac{F(\mathrm{fl}(\mathrm{fl}(u)+\mathrm{fl}(\sigma))) - F(\mathrm{fl}(u))}{\mathrm{fl}(\sigma)} - F'(\mathrm{fl}(u)) \right\| + \mathcal{O}\left(3\frac{|F(\mathrm{fl}(u))|\, U}{\mathrm{fl}(\sigma)} \right).\end{aligned}$$

If we assume now that $\mathrm{fl}(\mathrm{fl}(u) + \mathrm{fl}(\sigma))$ is exact we can write this as

$$\varepsilon_{COMP} \leq \left\| \frac{F(\mathrm{fl}(u) + \mathrm{fl}(\sigma)) - F(\mathrm{fl}(u))}{\mathrm{fl}(\sigma)} - F'(\mathrm{fl}(u)) \right\| + \mathcal{O}\left(3\frac{|F(\mathrm{fl}(u))|\,U}{\mathrm{fl}(\sigma)}\right). \tag{4.3}$$

Now we have reached the point where the truncation error has been entirely separated. As $\varepsilon_t \leq \mathcal{O}\left(\frac{|F''(\mathrm{fl}(u))|}{2}\,\mathrm{fl}(\sigma)\right)$ we have

$$\varepsilon_{COMP} \leq \mathcal{O}\left(3\frac{|F(\mathrm{fl}(u))|\,U}{\mathrm{fl}(\sigma)}\right) + \mathcal{O}\left(\frac{|F''(\mathrm{fl}(u))|}{2}\,\mathrm{fl}(\sigma)\right).$$

Assuming further that $\mathrm{fl}(\sigma)$ is exact, this is the same as

$$\varepsilon_{COMP} \leq \mathcal{O}\left(3\frac{|F(\mathrm{fl}(u))|\,U}{\sigma}\right) + \mathcal{O}\left(\frac{|F''(\mathrm{fl}(u))|}{2}\sigma\right). \tag{4.4}$$

Discarding the constants we are left with

$$\varepsilon_{COMP} \leq \mathcal{O}\left(\frac{|F(\mathrm{fl}(u))|\,U}{\sigma}\right) + \mathcal{O}\left(|F''(\mathrm{fl}(u))|\,\sigma\right).$$

4.3 Total error of the finite difference method

Combining the data error and the computational error from the previous sections we get the total error

$$\begin{aligned} e_{TOT} &\leq e_c + e_d \\ &\leq \mathcal{O}\left(\frac{|F(\mathrm{fl}(u))|\,U}{\sigma}\right) + \mathcal{O}\left(|F''(\mathrm{fl}(u))|\,\sigma\right) + \mathcal{O}\left(|F''(u)|\,|u|\,U\right) \end{aligned}$$

In order for the truncation error to be greater than the data error, we need the following inequality to hold

$$|F''(\mathrm{fl}(u))|\,\sigma \geq |F''(u)|\,|u|\,U.$$

Or equivalently

$$\sigma \geq |u|\,U. \tag{4.5}$$

For the truncation error to be the dominant error in the computational error we require the inequality

$$|F''(\mathrm{fl}(u))|\,\sigma \geq \frac{|F(\mathrm{fl}(u))|\,U}{\sigma}$$

to hold. This equates to the condition

$$\sigma \geq \sqrt{U \frac{|F(\text{fl}(u))|}{|F''(\text{fl}(u))|}}. \tag{4.6}$$

With these conditions we construct the following theorem.

Theorem 4.3.1. *In order for the truncation error to be the dominating error in a finite difference approximation of $F'(u)$ we need to select increments σ such that*

$$\sigma \geq \max\left\{ |u|\, U, \sqrt{U}\sqrt{\frac{|F(\text{fl}(u))|}{|F''(\text{fl}(u))|}} \right\}$$

Proof. By equations (4.5) and (4.6). □

While we know $F(\text{fl}(u))$, it is unreasonable for us to know $F''(\text{fl}(u))$. In [17, p.187], $\sqrt{\frac{|F(\text{fl}(u))|}{|F''(\text{fl}(u))|}}$ is called the 'curvature scale' of F. It is stated that in the absence of any other information it is usually assumed to be $|u|$. For a nice and polynomial-like F, this is true as $u \to \infty$.

4.4 The representability of the increments

In equations (4.3) and (4.4) we made the assumption that both $\text{fl}(\text{fl}(u)+\text{fl}(\sigma))$ and $\text{fl}(\sigma)$ were exact, that is

$$\text{fl}(\text{fl}(u) + \text{fl}(\sigma)) = \text{fl}(u) + \text{fl}(\sigma) = \text{fl}(u) + \sigma.$$

We also made the assumption that the division was exact. If we can select σ in such a way that that the assumptions hold, then these floating point roundings will not generate additional errors. What sets these floating point roundings apart from the ones of the function evaluations is that we have full control of σ.

In [17, p.186] this is achieved by applying Algorithm 1. This algorithm will ensure that the σ used in the division is the same as in the addition by removing the digits that would be rounded away in the addition otherwise. Also, as long as $\sigma < |y|$, it potentially increases the number of sequential zeros digits at the end of the significand of σ, making the division more likely to be exact.

Note that some optimizing compilers might automatically concatenate these two lines, making it necessary to include a dummy function between

Data: y, σ_{in}
Result: σ_{out}
$temp \leftarrow \sigma_{in} + y$
$\sigma_{out} \leftarrow temp - y$

Algorithm 1: Ensure σ is representable

them. Since we will at some point make a function call with `temp` as input, this could be done here instead of using a dummy function. The effects of this algorithm is demonstrated in example 4.4.1.

Example 4.4.1. *For two floating point numbers of precision $p = 5$,*

$$x_1 = 5432.1 \quad \textit{and} \quad x_2 = 1.2345$$

we see that

$$\frac{\mathrm{fl}(x_1 + x_2)}{x_2} = \frac{5433.3}{1.2345} \neq \frac{x_1 + x_2}{x_2}.$$

If we however use $\hat{x}_2 = \mathrm{fl}(x_1 + x_2) - x_1 = 1.2$, we get

$$\frac{\mathrm{fl}(x_1 + \hat{x}_2)}{\hat{x}_2} = \frac{5433.3}{1.2} = \frac{x_1 + \hat{x}_2}{\hat{x}_2}.$$

However, for Corollary 3.0.5 to be applyable for the finite difference method we need $\mathrm{sgn}(y + \sigma) = \mathrm{sgn}(y)$. For $|\sigma| \leq |y|$ this always hold, otherwise we need to choose σ such that $\mathrm{sgn}(\sigma) = \mathrm{sgn}(y)$. We can modify a chosen σ after the fact, as shown in Algorithm 2.

Data: y, σ_{in}
Result: σ_{out}
if $y \neq 0$ **then**
  if $\sigma_{in} \geq |y|$ **then**
    $temp \leftarrow \sigma_{in}\,\mathrm{sgn}(y) + y$
  else
    $temp \leftarrow \sigma_{in} + y$
  end
  $\sigma_{out} \leftarrow temp - y$
end

Algorithm 2: Ensure σ is representable

Note that Algorithm 2 could change forward differences into backwards differences, and vice versa.

Chapter 5

Software stack

The Functional Mock-up Interface standard for model exchange specifies methods for computing the derivatives and setting of states and time. Model exchange FMUs encode ODEs as a compressed archive containing a model description as well as binaries and libraries needed for its functionality. The model description is a `.xml` file that contains metadata such as names of variables, parameters, constants and inputs. A simulator tool will read the .xml file, allowing a user to interact with it. The files for functionality are implementations of the functions defined by FMI contained in one, or several, `.dll` or `.so` files.

The FMI standard is in C, an efficient, low-level language [18]. It does not specify how the FMUs are to be generated. The FMUs in this thesis are generated from models written in the Modelica language [3].

PyFMI is a Python package developed by Modelon for interaction and simulations of FMUs. Python is a high-level language [19], more user friendly than C. PyFMI implements Cython in order to facilitate interaction between the Python and C.

PyFMI [5] imports its solvers from Assimulo [20], a set of general purpose solvers for ODEs and differential algebraic equations. Assimulo includes some solvers from the SUNDIALS suite [6], one of which is the variable-order, variable-step multistep solver CVode [6, p.366]. CVode is written in C and is an evolution of the Livermore Solver for Ordinary Differential Equation (LSODE) [21], a FORTRAN subroutine package. Assimulo also includes codes by Hairer [22] such as RADAU5, which is also a FORTRAN routine. Besides these two, Assimulo also has their own implementations of Euler and Runge-Kutta.

5.1 Creating and compiling a model

The models used in this thesis are written in the Modelica language [3], and contained as `*.mo` files. As an example we show in listing 1 the code for a model of the Van der Pol oscillator with nominal value $y_{NOM} = 10^6$. Firstly, the parameters and states of the model are defined. Then the equations for the model, in this case the time derivatives of the state variables, are defined.

```
model VanDerPol1e6
        parameter Real mu = 1.e6;
        parameter Real y1_0 = 2;
        parameter Real y2_0 = -2/3;
        parameter Real y_nom = 1.e6;
        Real y1(start=y1_0, nominal=y_nom);
        Real y2(start=y2_0, nominal=y_nom);
equation
        der(y1) = mu*(y1-y1^3/3-y2);
        der(y2) = y1/mu;
end VanDerPol1e6;
```

Listing 1: Modelica code for the Van der Pol oscillator with $y_{NOM} = 10^6$.

The models were then compiled into FMUs, contained as `*.fmu` files.

5.2 PyFMI

PyFMI [5] is a Python package for interacting with FMUs. For simulations of ordinary differential equations, PyFMI uses the solvers available in Assimulo [20], an open source suite of general purpose solvers for ODEs. When solving an FMU using Radau5 [7, p.118] or CVode [6, p.366], the solution requires the Jacobian. If the FMU does not contain the Jacobian, PyFMI can either use the different solvers internal design of Jacobian approximation or use its own derivative approximation.

PyFMIs can use information available within the FMU that is not directly available to the solvers. For example, FMUs contain information on which states impact derivatives [5, p.6]. With this information an adjacency matrix, A_{adj}, can be constructed with the elements

$$A_{adj}(F)_{ij} = \begin{cases} 1 & \text{if } F_i(y) \text{ depends on } y_j, \\ 0 & \text{otherwise.} \end{cases}$$

With this information the number of function evaluations for the Jacobian approximation can potentially be reduced.

Example 5.2.1. *The function* $F : \mathbb{R}^3 \to \mathbb{R}^3$,

$$F(y) = \begin{bmatrix} y_1 \\ y_2 + y_3 \\ y_2 - y_3 \end{bmatrix}$$

has the adjacency matrix

$$A_{adj}(F) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}.$$

This means that the first column of a difference approximation of the Jacobian J_F *can be computed at the same time as one of the other two columns since*

$$F_1\left(y + \begin{bmatrix} \delta_1 \\ \delta_2 \\ \delta_3 \end{bmatrix}\right) = F_1\left(y + \begin{bmatrix} \delta_1 \\ 0 \\ 0 \end{bmatrix}\right),$$

for all $\delta_2, \delta_3 \in \mathbb{R}$ *and*

$$F_2\left(y + \begin{bmatrix} \delta_1 \\ \delta_2 \\ 0 \end{bmatrix}\right) = F_2\left(y + \begin{bmatrix} 0 \\ \delta_2 \\ 0 \end{bmatrix}\right),$$

for all $\delta_1 \in \mathbb{R}$.

With this information, a lower bound for the number of zeros in the Jacobian, as well as their positions, will also be known beforehand.

PyFMI allows options to be set for the simulations [23]. Some of these are described in table 5.1. The options are provided as an object of the class `AssimuloFMIAlgOptions` when invoking the simulation. To set these options an existing dictionary can be extracted from the model object and then modified as demonstrated in the listing 2.

```
from pyfmi import load_fmu

fmu = load_fmu('VanDerPol1e6.fmu')
opts = fmu.simulate_options()
opts['solver'] = 'Radau5ODE'
res = fmu.simulate(options=opts)
```

Listing 2: Python code for simulating the FMU `VanDerPol1e6.fmu` with the solver Radau5.

Option	Description
solver	Specifies the solver to be used in the simulation. In this thesis we use 'Radau5ODE' or 'CVode'.
ncp	Number of communication points to be returned. If ncp is zero, the solver will return the internal steps taken.
with_jacobian	If set to 'True' the PyFMI Jacobian is used. If 'False' the solver specific Jacobian is used instead.
{solver}_options	Solver-specific options that are passed on to the solver.

Table 5.1: Options available for PyFMI simulations. Note that this list only contains the options used for the simulations in this thesis, it is in no way exhaustive.

PyFMI can further pass on options to the solver. The options used in this thesis are described in table 5.2. A complete list of solver-specific options can be found in the Assimulo documentation [24].

Option	Description
rtol	The relative tolerance.
atol	The absolute tolerance.
time_limit	Number of seconds spent on the simulation before aborting.
maxsteps	The maximum number of steps allowed before aborting.

Table 5.2: Solver options. Note that this list only contains the options used for the simulations in this thesis, it is in no way exhaustive.

5.2.1 Simulating an FMU model

An example of the Python code used to run a simulation of an FMU using PyFMI can be seen in listing 3.

```
from pyfmi import load_fmu

fmu = load_fmu('VanDerPol1e6.fmu')
opts = fmu.simulate_options()
opts['solver'] = 'Radau5ODE'
res = fmu.simulate(0, 2*1e6, options=opts)
```

Listing 3: Python code for simulating the FMU `VanDerPol1e6.fmu` using Radau5 with starting time 0 and final time $2 \cdot 10^6$, using the `pyfmi` package.

5.2.2 σ in PyFMI

For the PyFMI Jacobian approximation the increments

$$\sigma_j = \sqrt{U} \max \left\{ \left| y_j^{n(0)} \right|, y_{NOM}^j \right\}$$

are used for forward or backward difference, where

$$y_{NOM} = \left[y_{NOM}^1, \dots, y_{NOM}^N \right]^{\mathrm{T}}$$

is the *nominal value* of y. The nominal value [25] of a state is an expected value. A good example is electrical mains that in some areas have the nominal voltage of $230V$, but are allowed a variance of 10%. The nominal values are stored in the FMU. If no value is defined in the model, or if it is set to zero, it will default to one.

5.3 CVode

CVode [6, p.366] solves ODEs using variable-order, variable-step methods. For stiff problems CVode uses Backwards Differentiation Formulas which in fixed-leading coefficient form are given by

$$\sum_{i=0}^{q} \alpha_{n,i} y^{n-i} + h_n \beta_{n,0} f(t_n, y^n) = 0$$

for some order $q = 1, \ldots, 5$ with $\alpha_{n,0} = -1$. The coefficients are uniquely determined and well documented [26, p.349].

For each time step CVode uses a version of Newton's method,such as simplified Newton as presented in section 2.1.2, to solve the root finding problem

$$G(y) \equiv y - h_n \beta_{n,0} f(t_n, y) - \sum_{i=1}^{q} \alpha_{n,i} y^{n-i} = 0.$$

Throughout the code CVode uses a weighted root mean square norm $\|\cdot\|_{\text{WRMS}} : \mathbb{R}^n \to \mathbb{R}$ defined as

$$\|v\|_{\text{WRMS}} := \sqrt{\frac{1}{N} \sum_{1}^{N} \left(\frac{v_i}{W_i} \right)^2}.$$

With the weights $W_j := \text{RTOL} \left| y_j^{n(m)} \right| + \text{ATOL}_j$ defined in [27].

5.3.1 Stopping conditions and error test

After each timestep CVode makes a local error test. The local truncation error (LTE) for the time integration for order q and stepsize h_n is by [6, p.369]

$$\text{LTE} = C h_n^{q+1} \frac{\partial^{q+1} y}{\partial t^{q+1}} + \mathcal{O}\left(h_n^{q+2}\right)$$

for a method- and order dependent constant C. There is a similar relation for the error in the predictor $y^{n(0)}$ leading to the relation

$$\text{LTE} = C' \left[y^n - y^{n(0)} \right] + \mathcal{O}\left(h_n^{q+2}\right)$$

for a method- and order dependent constant C'.

Since $\|v\|_{\text{WRMS}} = 1$ at the tolerance limit, CVode uses the error test $\|\text{LTE}\|_{\text{WRMS}} \leq 1$. With $y^{n(m)}$ being the last step and thus $y^{n(m)} \approx y^n$, the error test is performed on the predictor-corrector difference $y^{n(m)} - y^{n(0)}$. This means that if the relation

$$\left\| y^{n(m)} - y^{n(0)} \right\|_{\text{WRMS}} \leq \varepsilon := \frac{1}{|C'|}$$

holds, the step is successful. If the relation does not hold, the step is rejected and redone with a different stepsize.

A fraction of ε is also used in the stopping condition for the Newton iteration. Here the difference between the last step taken, $y^{n(m)}$, and the true solution, y^*, is considered. This difference need to be controlled so that $\left\|y^* - y^{n(m)}\right\|_{\mathrm{WRMS}} < \chi\varepsilon$. The default setting is $\chi = 0.1$, but can be set by the user when solving.

For this CVode also uses an estimated convergence rate constant R which is initiated as $R = 1$ whenever $\mathrm{I} - h_n\beta_{n,0}J_G$ is updated. R itself is updated as

$$R = \max\left\{0.3R, \frac{\left\|\Delta y^{n(m)}\right\|_{\mathrm{WRMS}}}{\left\|\Delta y^{n(m-1)}\right\|_{\mathrm{WRMS}}}\right\}$$

after each iteration. And with the estimation

$$\left\|y^* - y^{n(m)}\right\|_{\mathrm{WRMS}} \approx R\left\|y^{n(m)} - y^{n(m-1)}\right\|_{\mathrm{WRMS}}$$

the stopping test is

$$R\left\|y^{n(m)} - y^{n(m-1)}\right\|_{\mathrm{WRMS}} < \chi\varepsilon.$$

By default the iteration stops after a maximum of 3 iterations, this can however be changed to a different number by the user. Futhermore, the iteration is deemed to be diverging and aborted if

$$\frac{\left\|y^{n(m)} - y^{n(m-1)}\right\|_{\mathrm{WRMS}}}{\left\|y^{n(m-1)} - y^{n(m-2)}\right\|_{\mathrm{WRMS}}} > 2,$$

for any $m > 1$.

5.3.2 σ in CVode

From article [6, p.369] we have that the increments, σ_j, in CVode are chosen as

$$\sigma_j = \max\left\{\sqrt{U}\left|y_j^{n(0)}\right|, \sigma_0 W_j\right\}, \tag{5.1}$$

where σ_0 is a dimensionless value involving the unit roundoff U and the norm of f [6, p.369].

The term $\left|y_i^{n(0)}\right|\sqrt{U}$ is described as "the standard choice" in the precursor to CVode [21, p.66]. But since it will not work close to zero, an alternative term must be available.

From the CVode code at Sundials [28] we find that σ_0 is defined as

$$\sigma_0 = \begin{cases} 10^3 \, |h| \, UN \, \|f\|_{\text{WRMS}} & \text{if } \|f\|_{\text{WRMS}} \neq 0, \\ 1 & \text{otherwise.} \end{cases}$$

σ_0 was defined in [21, p.66] and will be shown in appendix A.
With this definition of σ_0, Equation (5.1) becomes

$$\sigma_j = \begin{cases} \max\left\{\sqrt{U}\left|y_j^{n(0)}\right|, 10^3 \, |h| \, UN \, \|f\|_{\text{WRMS}} \, W_j\right\} & \text{if } \|f\|_{\text{WRMS}} \neq 0, \\ \max\left\{\sqrt{U}\left|y_j^{n(0)}\right|, W_j\right\} & \text{otherwise.} \end{cases}$$

5.4 Radau5

Radau5 is a code based on the 3-stage, order 5 Radau IIA method described in section 2.2. It has the Butcher's tableau [7, p.74]

$$\begin{array}{c|c} c & A \\ \hline & b \end{array} = \begin{array}{c|ccc} \frac{4-\sqrt{6}}{10} & \frac{88-7\sqrt{6}}{360} & \frac{296-169\sqrt{6}}{1800} & \frac{-2+3\sqrt{6}}{225} \\ \frac{4+\sqrt{6}}{10} & \frac{296+169\sqrt{6}}{1800} & \frac{88+7\sqrt{6}}{360} & \frac{-2-3\sqrt{6}}{225} \\ 1 & \frac{16-\sqrt{6}}{36} & \frac{16+\sqrt{6}}{36} & \frac{1}{9} \\ \hline & \frac{16-\sqrt{6}}{36} & \frac{16+\sqrt{6}}{36} & \frac{1}{9} \end{array}$$

and requires solving the root finding problem

$$G(Z) = Z - A \begin{pmatrix} hf(t_n + c_1 h, y^n + z^1) \\ hf(t_n + c_2 h, y^n + z^2) \\ hf(t_n + c_3 h, y^n + z^3) \end{pmatrix}.$$

Note that

$$Z = \begin{pmatrix} z^1 \\ z^2 \\ z^3 \end{pmatrix} = \begin{pmatrix} z_1^1 & \cdots & z_N^1 & z_1^2 & \cdots & z_N^2 & z_1^3 & \cdots & z_N^3 \end{pmatrix}^{\text{T}} \in \mathbb{R}^{3N}.$$

Solving the system with Newton's method that would mean that for each iteration m it would be necessary to solve a linear system with the matrix

$$\begin{pmatrix} \text{I} - ha_{11}\frac{\partial f}{\partial z^{1(m)}}(t_n + c_1 h, y^n + z^{1(m)}) & \cdots & -ha_{13}\frac{\partial f}{\partial z^{3(m)}}(t_n + c_3 h, y^n + z^{3(m)}) \\ -ha_{21}\frac{\partial f}{\partial z^{1(m)}}(t_n + c_1 h, y^n + z^{1(m)}) & \cdots & -ha_{23}\frac{\partial f}{\partial z^{3(m)}}(t_n + c_3 h, y^n + z^{3(m)}) \\ -ha_{31}\frac{\partial f}{\partial z^{1(m)}}(t_n + c_1 h, y^n + z^{1(m)}) & \cdots & \text{I} - ha_{33}\frac{\partial f}{\partial z^{3(m)}}(t_n + c_3 h, y^n + z^{3(m)}) \end{pmatrix}.$$

Using a simplified Newton's method, the number of Jacobian approximations can be reduced to 3 by replacing all Jacobians

$$\frac{\partial f}{\partial z^{i(m)}}(t_n+c_ih, y^n+z^{i(m)}) \quad \text{with} \quad \frac{\partial f}{\partial z^{i(0)}}(t_n+c_ih, y^n+z^{i(0)}), \quad \text{for } i = 1, \ldots, 3.$$

This is then further reduced by replacing

$$\frac{\partial f}{\partial z^{i(0)}}(t_n + c_ih, y^n + z^{i(0)}) \quad \text{with} \quad J_f \approx \frac{\partial f}{\partial y}(t_n, y^n), \quad \text{for } i = 1, \ldots, 3.$$

This results in the simplified Newton iteration

$$(\mathrm{I} - hA \otimes J_f)\Delta Z^{(m)} = -Z^{(m-1)} + h(A \otimes \mathrm{I})F(Z^{(m-1)})$$
$$Z^{(m)} = Z^{(m-1)} + \Delta Z^{(m)}$$

where $\otimes$ is the tensor product[1] and $Z^{(m)} = (z^{1(m)}, \ldots, z^{3(m)})^{\mathrm{T}}$ is the mth iteration of Z. Each iteration requires 3 evaluations of f and solving a $3N$ dimensional linear system. Since the matrix $\mathrm{I} - hA \otimes J$ is the same in all iterations it is only calculated once, and because it is normally reused, an LU-decomposition is performed as it will simplify its use.

5.4.1 Stopping conditions and error test

Assuming at least linear convergence we have that for some constant $0 < \Theta_k < 1$

$$\left\|\Delta Z^k\right\| \leq \Theta \left\|\Delta Z^{k-1}\right\|,$$

where Θ_k is called the rate of convergence.

In [7, p.120] the convergence rate is approximated in each iteration as

$$\Theta_k := \frac{\left\|\Delta Z^k\right\|}{\left\|\Delta Z^{k-1}\right\|},$$

for $k \geq 1$.

[1]The tensor product $Q \otimes R$ of two square matrices of dimensions $p \times p$ and $s \times s$ is a square matrix of size $ps \times ps$ structured as

$$Q \otimes R = \begin{bmatrix} q_{11} & q_{12} \\ q_{21} & q_{22} \end{bmatrix} \otimes \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} = \begin{bmatrix} q_{11} \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} & q_{12} \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \\ q_{21} \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} & q_{22} \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \end{bmatrix}.$$

Note however that in the code for Radau5, the definition

$$\Theta_k = \begin{cases} \frac{\|\Delta Z^1\|}{\|\Delta Z^0\|} & \text{for } k = 1, \\ \sqrt{\frac{\|\Delta Z^k\|}{\|\Delta Z^{k-2}\|}} & \text{for } k \geq 2, \end{cases}$$

is used instead [22, 29].

In [7, p.120], an upper estimate of the error between the step Z^{k+1} and the actual solution Z^* is found.

$$\begin{aligned} \left\|Z^{k+1} - Z^*\right\| &= \left\|\Delta Z^{k+1} + \Delta Z^{k+2} + \cdots\right\| \\ &\leq \left\|\Delta Z^{k+1}\right\| + \left\|\Delta Z^{k+2}\right\| + \cdots \\ &\leq \Theta \left\|\Delta Z^k\right\| + \Theta^2 \left\|\Delta Z^k\right\| + \cdots \\ &= \left\|\Delta Z^k\right\| \sum_{j=1}^{\infty} \Theta_k^j \\ &= \left\|\Delta Z^k\right\| \frac{\Theta_k}{1 - \Theta_k}. \end{aligned}$$

Since the iteration error should be smaller than the local discretization error, which is usually kept close to Tol, the following stopping condition for the iteration process is constructed.

$$\eta_k \left\|\Delta Z^k\right\| \leq \kappa \cdot Tol, \quad \text{with} \quad \eta_k = \frac{\Theta_k}{1 - \Theta_k}.$$

Since Θ_k is only defined for $k \geq 1$, this leaves η_0 undefined as well. We instead use the definition

$$\eta_k := \begin{cases} \left(\max\left\{\eta_{OLD}, U\right\}\right)^{0.8} & \text{for } k = 0, \\ \Theta_k / \left(1 - \Theta_k\right) & \text{for } k \geq 1, \end{cases}$$

with η_{OLD} being the last η_k of the previous time step. This new definition allows for the iteration to be stopped after only one step.

For the parameter κ experiments performed for $\kappa \in [10^{-4}, 10]$ showed that the code worked best for values around 10^{-1} and 10^{-2} [7, p.121]. In the code for Radau5, the whole term $\kappa \cdot Tol$ can be entered by the user. If a value lower than $0.1\,\mathrm{RTOL}^{2/3}$ is entered, an error is returned. If no value is entered it is set as

$$\kappa \cdot Tol = \max\left\{\frac{10U}{0.1\,\mathrm{RTOL}^{2/3}}, \min\left\{0.03, \sqrt{0.1\,\mathrm{RTOL}^{2/3}}\right\}\right\},$$

with a user-entered RTOL $\leq 10U$ returning an error [22, 29].

It is stated in [7, 121] that the iteration is aborted and restarted with a smaller stepsize if Θ_k is greater than 1, as that indicates that the iteration diverges. Looking at the actual code however, we see that this condition is applied stricter than this. In fact, it aborts when $\Theta_k \geq 0.99$. This makes sense considering that, as stated in section 2.1.2, a convergence rate too close to 1 would mean the iteration step size is no longer a reliable indicator of the error.

Radau5 also aborts the iteration if

$$\frac{\Theta_k^{k_{MAX}-k}}{1-\Theta_k}\left\|\Delta Z^k\right\| > \kappa \cdot Tol,$$

for some k. This is an estimate of what the iteration error will be after the maximum number of iterations, k_{MAX}, which is suggested to be relatively high, the examples given are 7 or 10 [7, p.121].

5.4.2 σ in Radau5

The choice of increments in Radau5 goes back to the original FORTRAN code [22] where they are chosen as

$$\sigma_j = \sqrt{U}\max\left\{\sqrt{\left|y_j^{n(0)}\right|}, \sqrt{10^{-5}}\right\}.$$

The selection of increments in the Assimulo code [29] for Radau5 are presented in listing 4.

```
ysafe = y[i];
delt = sqrt(rmem->input->uround * radau_max(1e-5,
↪  radau5_abs(ysafe)));
```

Listing 4: C code for choice of increments in Radau5.

No mathematical background is provided, but it is possible that the assumption

$$\frac{|G|}{|G_{yy}|} \approx |y|$$

for Theorem 4.3.1 was used. However, if this is true, the candidate $\left|y_j^{n(0)}\right|U$ should still be included for σ, as it would be the greatest of the candidates

when $\left|y_j^{n(0)}\right| > \frac{1}{U} \approx 10^{16}$. As we shall see in chapter 6 however, this does not effect the models used in this thesis.

Presumably, since this choice will not work when $\left|y_j^{n(0)}\right|$ is close to or equal to zero, $\sqrt{10^{-5}}$ is added as an alternative increment.

We note here that $\sqrt{10^{-5}} = 0.01/\sqrt{10}$ is irrational, since 10 is not a perfect square. This means that if $\mathrm{fl}(\sqrt{10^{-5}})$ ends with zeros, they all stem from rounding. The chance that the division

$$\frac{G(y + \sqrt{10^{-5}U}) - G(y)}{\sqrt{10^{-5}U}}$$

is exact is therefore very slim.

Chapter 6

Models

The first model, one modelling the Van der Pol oscillator, was designed for this thesis. The rest were supplied by Modelon and are three of their models with issues as mentioned in the introduction. These were supplied as compiled FMUs, meaning we had a limited insight into them. These pre-made models would take more steps to simulate using the PyFMI Jacobian, if they were able to finish at all.

6.1 Model 1 - Van der Pol oscillator

To test the hypothesis about the nominal values we set up a model of a Van Der Pol oscillator,

$$\dot{y} = \begin{bmatrix} \dot{y_1} \\ \dot{y_2} \end{bmatrix} = \begin{bmatrix} \mu \left(y_1 - \frac{1}{3} y_1^3 - y_2 \right) \\ \frac{1}{\mu} y_1 \end{bmatrix}, \quad y^0 = \begin{bmatrix} 2 \\ -\frac{2}{3} \end{bmatrix},$$

with $\mu = 10^6$ and $t \in [0, 2\mu]$.

When a high value of μ is supplied, the y_1 term has a slow buildup followed by a sudden change, see Figure 6.3.

We found that the number of steps needed to finish the simulation increased significantly with the size of the nominal values, see Figure 6.1.

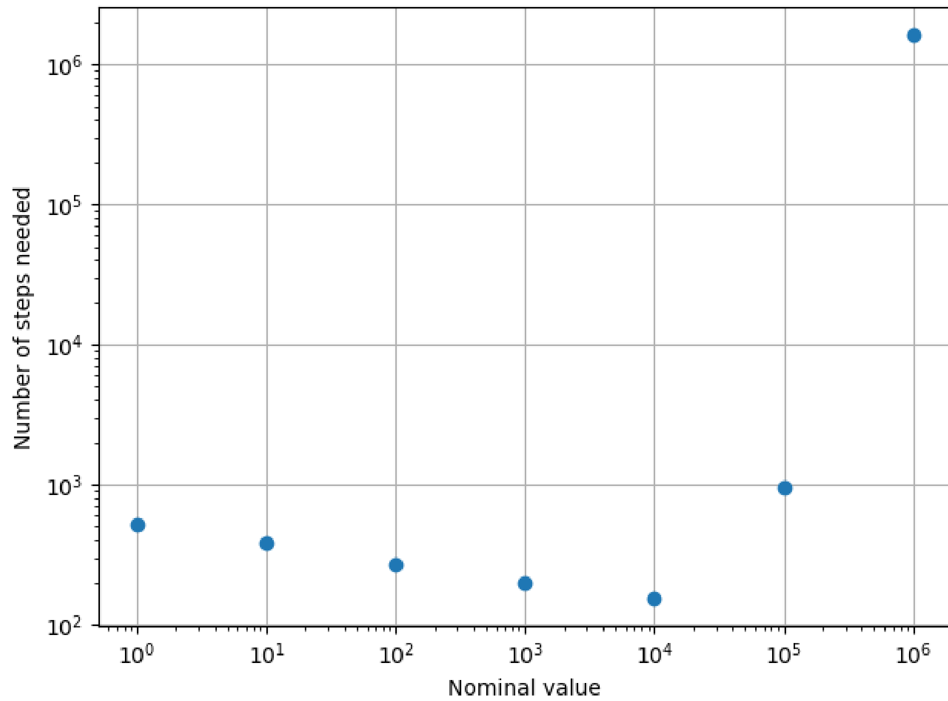


Figure 6.1: Comparison of number of steps needed as an effect of nominal value. The values are observed from successful simulations using Radau5ODE with PyFMI Jacobian. The FMU was constructed with $\mu = 10^6, y_0 = [2, -0.7], t_0 = 0$ and $t_f = 2\mu$.

Comparing the plots of y_1 from two simulations with radically different nominal values we can see that an erroneous y_{NOM} still produce the correct result, see Figure 6.2.

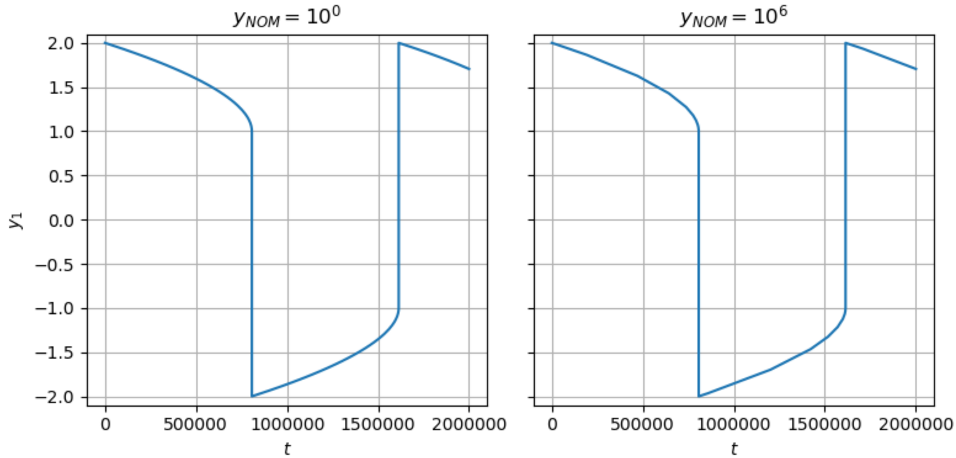


Figure 6.2: Plots of y_1 over time for two successful simulation of Van Der Pol oscillator with $\mu = 10^6, y_0 = [2, -0.7,], t_0 = 0$ and $t_f = 2\mu$. Data retrived from successful simulations using Radau5 with PyFMI Jacobian.

Simulations with $y_{NOM} \geq 10^7$ could not finish within exceeding the maximal number of steps. This maximum could not be set higher than $2147483647 \approx 2 \cdot 10^9$ without resulting in an `OverflowError` as the Python int could not be converted into a C long.

Method	Steps needed
CVode, internal Jacobian	Did not finish
CVode, PyFMI Jacobian	Did not finish
Radau5, internal Jacobian	105
Radau5, PyFMI Jacobian	1620212

Table 6.1: Steps needed to finish Model 1 with $y_{NOM} = 10^6$.

The relative tolerance was set to 10^{-9} and the absolute tolerance was set to RTOL y_{NOM} for both components.

Looking more closely at the timesteps we can in Figure 6.3 identify the issue arises when there is an abrupt change in a component.

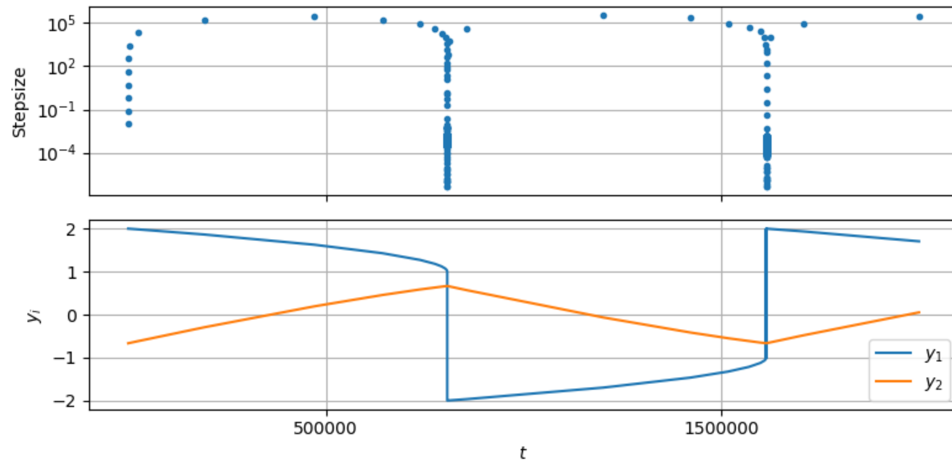


Figure 6.3: The top plot shows steplength as a function of time while the bottom shows y_1 and y_2 as functions of t. Observations were made after simulating the Van der Pol oscillator with Radau5 using PyFMI Jacobian. The FMU was constructed with $\mu = 10^6, y_0 = [2, -0.7], y_{NOM} = 10^6, t_0 = 0$ and $t_f = 2\mu$

State	$\|y\|_{MIN}$	$\|y\|_{AVG}$	$\|y\|_{MAX}$
y_1	0.0993	1.4	2
y_2	0.0518	0.638	0.667

Table 6.2: The two continuous states of Model 1. The magnitude of the minimum, maximum and average of the states were observed in a successful simulation using Radau5 with internal Jacobian.

6.2 Model 2 - Pneumatics

This is a simulation of a piece of pneumatically operated lock valve. The application of such a component could for example be to lock the bucket of an excavator in place.

The simulations were run with the relative tolerance RTOL $= 10^{-8}$ and the absolute tolerance, ATOL $=$ RTOL $\cdot y_{NOM}$ where the nominal values, y_{NOM}, are presented in table 6.3. The model has a total of 21 unknowns in very different ranges. This is due to the fact that the states are given in a range of units, such as pressures in Pascal or temperatures in Kelvin.

State	$\lvert y_i\rvert_{MIN}$	$\lvert y_i\rvert_{AVG}$	$\lvert y_i\rvert_{MAX}$	y^i_{NOM}
y_1	$1 \cdot 10^{-7}$	$1 \cdot 10^{-7}$	$1 \cdot 10^{-7}$	10^{-7}
y_2	$1 \cdot 10^{-7}$	$1 \cdot 10^{-7}$	$1 \cdot 10^{-7}$	10^{-7}
y_3	0.137	0.23	0.318	–
y_4	290	292	297	293.0
y_5	$4.79 \cdot 10^5$	$4.93 \cdot 10^5$	$5.11 \cdot 10^5$	10^5
y_6	291	293	297	293.0
y_7	$4.83 \cdot 10^5$	$4.96 \cdot 10^5$	$5.16 \cdot 10^5$	10^5
y_8	0	0.372	1	–
y_9	0	0.361	1	–
y_{10}	0	0.658	1	–
y_{11}	293	295	384	293.0
y_{12}	$1 \cdot 10^5$	$6.14 \cdot 10^5$	$6.2 \cdot 10^5$	10^5
y_{13}	$1 \cdot 10^5$	$3.02 \cdot 10^5$	$6.2 \cdot 10^5$	10^5
y_{14}	$1 \cdot 10^5$	$3.05 \cdot 10^5$	$6.2 \cdot 10^5$	10^5
y_{15}	$1 \cdot 10^5$	$4.79 \cdot 10^5$	$5.12 \cdot 10^5$	10^5
y_{16}	$1.3 \cdot 10^{-13}$	0.00798	0.01	–
y_{17}	$1 \cdot 10^5$	$4.82 \cdot 10^5$	$5.16 \cdot 10^5$	10^5
y_{18}	$1.12 \cdot 10^{-14}$	0.00814	0.01	–
y_{19}	0	0.034	0.0474	–
y_{20}	0	0.0277	0.593	–
y_{21}	0	0.0407	1.25	–

Table 6.3: The different continuous states of Model 2. The magnitude of the minimum, maximum and average of the states were observed in a successful simulation using CVode with internal Jacobian. The nominal values are set beforehand by the user when constructing the FMU. The nominal values represented by a dash were not set and will therefore default to 1.

We can note from table 6.3 that for all the undefined nominal values that are defaulted to 1, the nominal value is always used to calculate the Jacobian increment. For y_{11} or the states with nominal values set to 10^{-7} or 10^5, the nominal values are always less than or equal to the minimum measured value and therefore never used. Only for y_4, y_6 or y_{21} does it actually vary whether or not the nominal value is used to calculate σ.

The unknown values, represented as dashes, defaulted to 1.0. While the number of communication points was intended to be 10000, we will set them to 0 in order to better compare the efficiency of the methods. Simulations were done with using both Radau5 and CVode as solvers, using internal Jacobians, as well as those from PyFMI and the performance of the original

methods can be seen in table 6.4.

Method	Steps needed
CVode, internal Jacobian	5967
CVode, PyFMI Jacobian	Did not finish
Radau5, internal Jacobian	1145
Radau5, PyFMI Jacobian	Did not finish

Table 6.4: Steps needed to finish simulations of Model 2.

Note that the methods using the PyFMI Jacobians did not finish no matter how high the number of maximum steps was set. In the case with Radau5 used as a solver, it surpassed the maximum number of steps, while in the case of CVode it was manually aborted after not finishing within reasonable time.

In Figure 6.4 we see the results of a failed simulation of Model 2 using Radau5 with PyFMI Jacobian, while in Figure 6.5 we have a successful simulation using Radau5 with internal Jacobian. The figures show the ratio $\left|y_j^n\right| / y_{NOM}^j$ of the states y_{16} and y_{20}. Note that the states y_8, y_9, y_{10}, y_{18}, y_{19} and y_{21} also have ratios that break Sterbenz lemma at some point. The states that do exhibit this behavior all have a nominal value defaulting to 1.

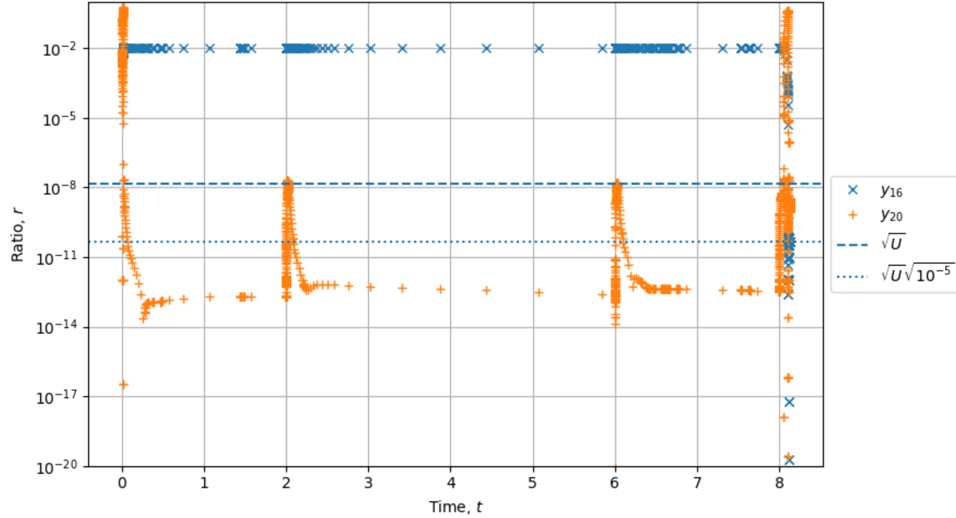


Figure 6.4: The ratio $r = \left|y_j^n\right| / y_{NOM}^j$ over time for the states y_{16} and y_{20}. When below $\sqrt{U}$ the ratio breaks the assumptions required for Sterbenz lemma. From a simulation of Model 2 using Radau5 with PyFMI Jacobian.

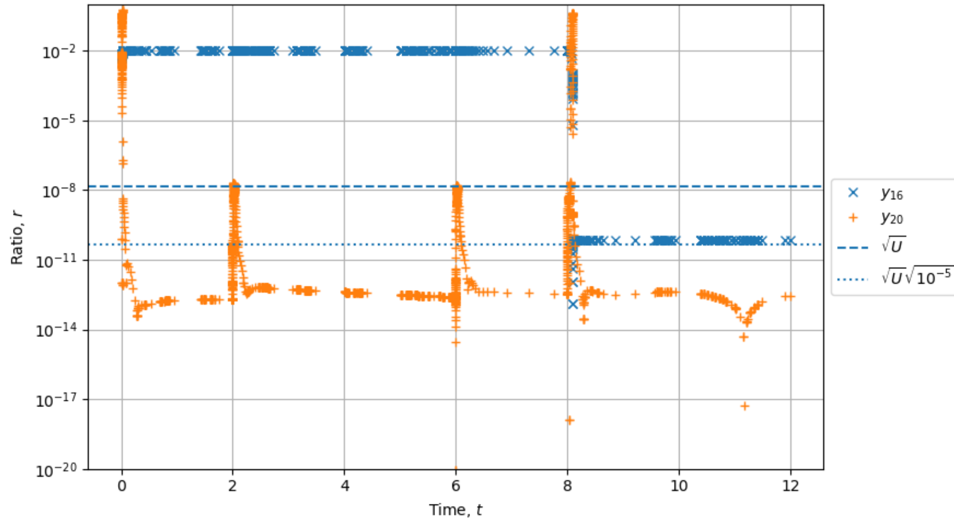


Figure 6.5: The ratio $r = \left|y_j^n\right| / y_{NOM}^j$ over time for the states y_{16} and y_{20}. When below $\sqrt{U}\sqrt{10^{-5}}$ the ratio breaks the assumptions required for Sterbenz lemma. From a simulation of Model 2 using Radau5 with internal Jacobian.

In Figures 6.6 and 6.7 we can see the behavior of the state y_{16} and y_{20} over time from a successful simulation using Radau5 with internal Jacobian. We can see that both states exhibit sudden changes around $t \approx 8$.

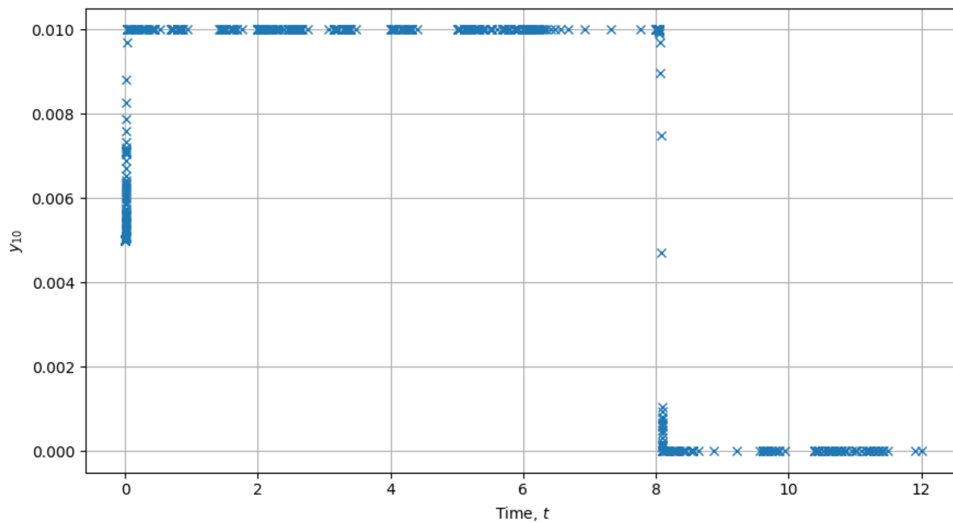


Figure 6.6: Values of y_{16} over time as observed in a successful simulation of Model 2 using Radau5 with internal Jacobian.

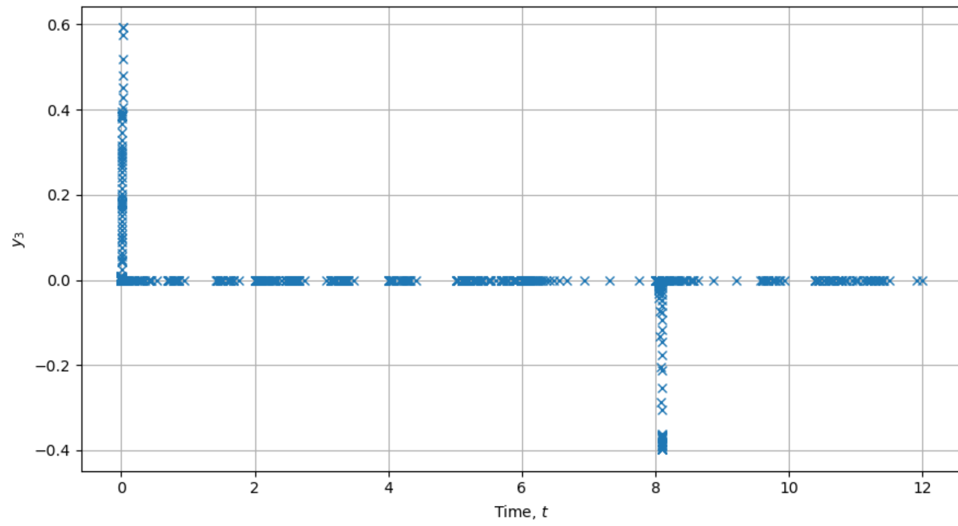


Figure 6.7: Values of y_{20} over time as observed in a successful simulation of Model 2 using Radau5 with internal Jacobian.

6.3 Model 3 - Hydraulics

This a model of a component similar to that of Model 2. However, this component utilizing liquids instead of gases.

State	$\lvert y_i \rvert_{MIN}$	$\lvert y_i \rvert_{AVG}$	$\lvert y_i \rvert_{MAX}$	y^i_{NOM}
y_1	0	0.798	1.03	–
y_2	0	180	580	–
y_3	0	0.795	0.981	–
y_4	0.179	0.367	0.45	–
y_5	1.26	2.86	4	–
y_6	0	2.63	4.17	–
y_7	0.999	151	200	1.0
y_8	0.999	1.18	1.46	1.0
y_9	0.999	4.23	5.04	1.0
y_{10}	0	0.384	1.11	–
y_{11}	0.38	102	200	1.0
y_{12}	4	28.7	107	1.0

Table 6.5: The different continuous states of Model 3. The magnitude of the minimum, maximum and average of the states were observed in a successful simulation using CVode with internal Jacobian. The nominal values are set beforehand by the user when constructing the FMU. The nominal values represented by a dash were not set and will therefore default to 1.

We can note from table 6.5 that all of the nominal values are either set to 1 or unset and therefor defaulted to 1. Since the states y_3 and y_4 are never greater than 1, the nominal value is always used to calculate the Jacobian increments. The continuous state y_5 and y_{12} is always greater than 1 and therefore the nominal values are never used. The same is almost true for y_7, y_8 and y_9 that barely goes below 1. For the other five states it does actually vary whether or not the nominal value is used to calculate σ.

This model finished using both solver internal Jacobians and PyFMI Jacobian, see table 6.6.

Method	Steps needed
CVode, internal Jacobian	13486
CVode, PyFMI Jacobian	10927
Radau5, internal Jacobian	1807
Radau5, PyFMI Jacobian	66345

Table 6.6: Steps needed to finish the simulations of Model 3.

In Figure 6.8 we see the ratio $\left|y^n_j\right| / y^j_{NOM}$ for the state y_2. Whenever the ratio dips below $\sqrt{U}$ it breaks the condition from Sterbenz Lemma when

using PyFMI Jacobian. When it dips below $\sqrt{U}\sqrt{10^{-5}}$ it breaks the condition for Radau5. The states y_1, y_3, y_6 and y_{10} also breaks the condition for PyFMI at some point, but none as much as y_2. Note that all of these states have a nominal value that is defaulted to 1, meaning that we can easily make a comparison between the condition for PyFMI and Radau5, as $\left|y_j^n\right| / y_{NOM}^j = \left|y_j^n\right|$.

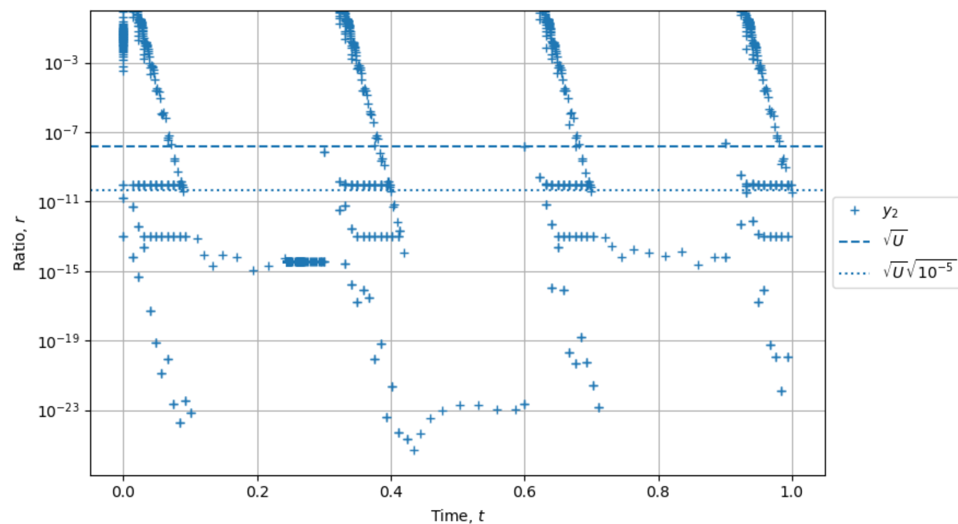


Figure 6.8: The ratio $r = \left|y_j^n\right| / y_{NOM}^j$ over time for the state y_2. When the ratio dips below $\sqrt{U}$ it would break Sterbenz lemma for PyFMI Jacobians, under $\sqrt{U}\sqrt{10^{-5}}$ for Radau5. From a simulation of Model 3 using Radau5 with internal Jacobian.

In Figure 6.9 we can see the behavior of the state y_2 over time from a successful simulation using Radau5 with internal Jacobian.

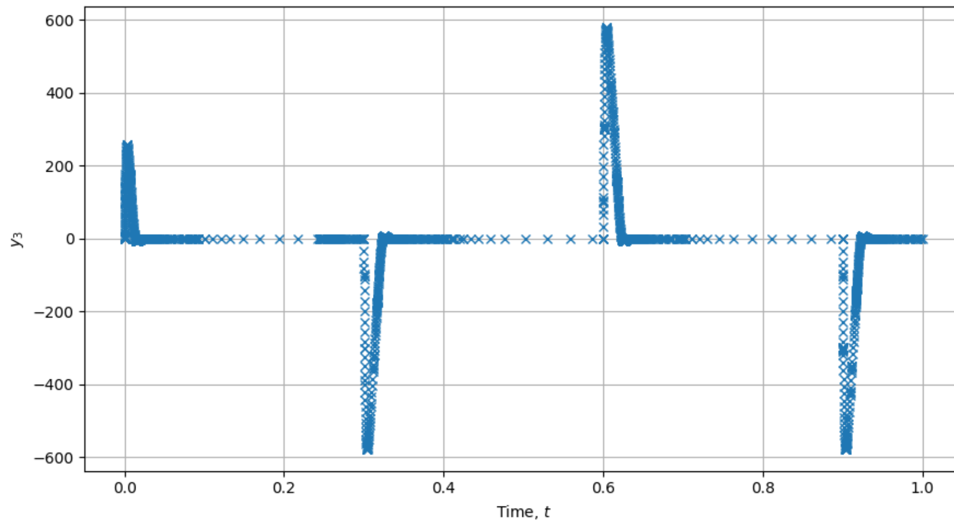


Figure 6.9: Values of y_2 over time as observed in a successful simulation of Model 3 using Radau5 with internal Jacobian.

6.4 Model 4 - Double wishbone suspension

This is a model for a type of vehicular suspension called double wishbone suspension. Named for the two wishbone shaped arms with two connection points each to the body and one to the "upright", which in turn connects to the wheel.

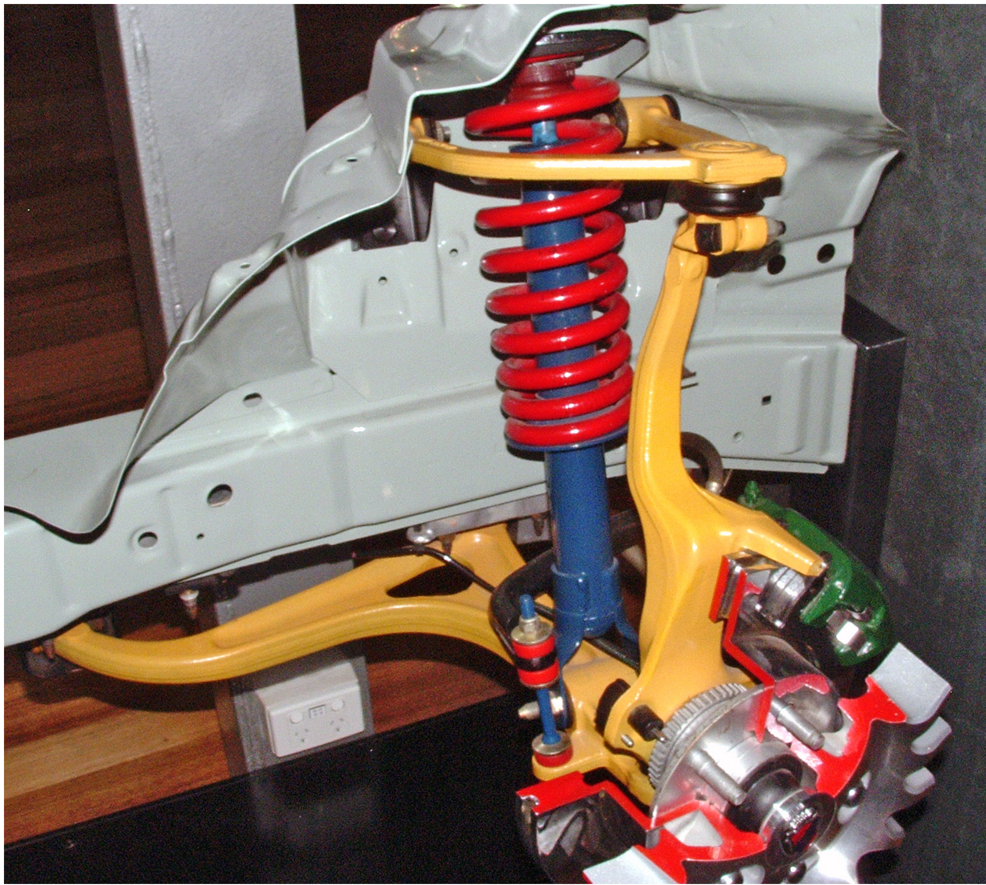


Figure 6.10: Wishbones and upright painted yellow.
"Double wishbone suspension" by RB30DE is licensed under CC BY-SA 3.0

The model was run with a relative tolerance of 10^{-6} over the time interval $[0, 10]$. As we can see in table 6.7 no nominal value was set and therefor all of them will default to 1. We can further see that none of the components ever reach a magnitude above 1, meaning that with the PyFMI Jacobian approximations we will always use the same increment.

State	$\lvert y_i\rvert_{MIN}$	$\lvert y_i\rvert_{AVG}$	$\lvert y_i\rvert_{MAX}$	y^i_{NOM}
y_1	0	0.00558	0.00646	–
y_2	0	0.0142	0.0164	–
y_3	0	0.00142	0.00165	–
y_4	0	0.00062	0.00479	–
y_5	0	0.0014	0.0116	–
y_6	0	0.000176	0.0018	–
y_7	0	0.22	0.241	–
y_8	0	0.0368	0.0402	–
y_9	0	0.0256	0.029	–
y_{10}	0	0.00728	0.0511	–
y_{11}	0	0.00109	0.00922	–
y_{12}	0	0.00182	0.0141	–
y_{13}	0	0	0	–
y_{14}	0	0	0	–

Table 6.7: The different continuous states of Model 4. The magnitude of the minimum, maximum and average of the states were observed in a successful simulation using CVode with internal Jacobian. The nominal values are set beforehand by the user when constructing the FMU. The nominal values represented by a dash were not set and will therefore default to 1.

When run we can see in table 6.8 that the original simulations with PyFMI Jacobian required more steps than the internals, in particular with CVode as a solver.

Method	Steps needed
CVode, internal Jacobian	594
CVode, PyFMI Jacobian	563463
Radau5, internal Jacobian	468
Radau5, PyFMI Jacobian	1425

Table 6.8: Steps needed to finish Model 4

In Figure 6.11 we see the results of a successful simulation using Radau5 with internal Jacobian. The points shows are all the states where at some point the ratio $\left|y^n_j\right| / y^j_{NOM}$ breaks the condition from Sterbenz Lemma, which for this model means all states. Note that all of these states have a nominal value that is defaulted to 1, meaning that we can easily make a comparison between the condition for PyFMI and Radau5, as $\left|y^n_j\right| / y^j_{NOM} = \left|y^n_j\right|$. We can

see that the ratio for several states fulfill the condition of Sterbenz Lemma for Radau5 σ_j but not for PyFMI σ_j for the timespan $t \in [8, 10]$.

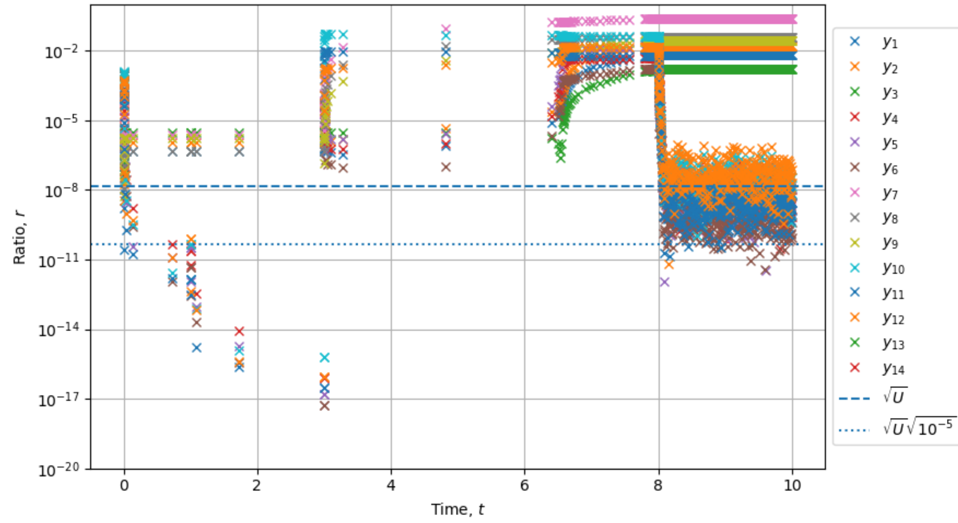


Figure 6.11: The ratio $r = \left|y_j^n\right| / y_{NOM}^j$ over time for the states where the ratio at some point dips below $\sqrt{U}$. From a simulation of Model 4 using Radau5 with internal Jacobian.

Chapter 7

Numerical experiments

These are experiments conducted on the models described in chapter 6. The models all have the same tolerances and nominal values as described in that section. For the Van der Pol model, the nominal values were set to 10^6.

In the plots for Experiments 2 to 4, the number of steps taken are represented with a dot. All simulations were run for at least one hour before being cancelled, if they had not finished by then or if the solver aborted with an error, there is no dot. To make it clearer where simulations have failed, the successful simulations will be connected by a line, while a failed experiment will break this line.

The Cython code for PyFMI, `fmi.pyx` from [23], was modified to take an extra option. This was then set at simulation by adding

```
opts['sigma_0'] = c_0  # Float, default value 1.
```

to the simulation options in listing 3.

Then the code for the Jacobian increments in `fmi.pyx`

```
for i in range(len_v):
        eps_pt[i] = RUROUND*(max(abs(v_pt[i]),
        ↪  nominals_pt[i]))
```

was modified in some way. Directly and/or by appending some code to with an additional algorithm.

The algorithms used will be the two presented in section 4.4 and Algorithm 3, which will be more aggressively swap between forward and backward differences than Algorithm 2.

A Python script for running the simulations were set up, going through the various experiments, post-processing algorithms, models, solvers and constants. The results were saved in `.json` files.

Data: y, σ_{in}
Result: σ_{out}
if $y \neq 0$ **then**
  $temp \leftarrow \sigma_{in}\,\mathrm{sgn}(y) + y$
  $\sigma_{out} \leftarrow temp - y$
end

Algorithm 3: Ensure σ is representable

7.1 Experiment 1 - Post-processing algorithms

For this experiment we used the original definition of increments in PyFMI, but before evaluation we applied the three post-processing algorithms presented in section 4.4.

	Model 1	Model 2	Model 3	Model 4
Internal	105	1145	1807	468
PyFMI (Original)	1620212	Did not finish	66342	1425
PyFMI (Algorithm 1)	1620326	Did not finish	66414	1470
PyFMI (Algorithm 2)	1620326	Did not finish	66354	1440
PyFMI (Algorithm 3)	16049	Did not finish	66069	1449

Table 7.1: Experiment 1, Radau5ODE

	Model 1	Model 2	Model 3	Model 4
Internal	Did not finish	5967	13486	594
PyFMI (Original)	Did not finish	Did not finish	10927	563463
PyFMI (Algorithm 1)	Did not finish	Did not finish	11365	547584
PyFMI (Algorithm 2)	Did not finish	Did not finish	11403	544824
PyFMI (Algorithm 3)	273	Did not finish	11838	581669

Table 7.2: Experiment 1, CVode

We can see in tables 7.1 and 7.2 that except for Model 1, the different algorithms seem to not have any greater effect on the number of steps.

7.2 Experiment 2 - Smaller nominal values

In this experiment we change the formula for selecting the increments to

$$\sigma_i = \sqrt{U}\max\left\{\left|y_i^{n(0)}\right|, c_0 y_{NOM}^i\right\}$$

for different positive values for the constant $c_0 \in \mathbb{R}$.

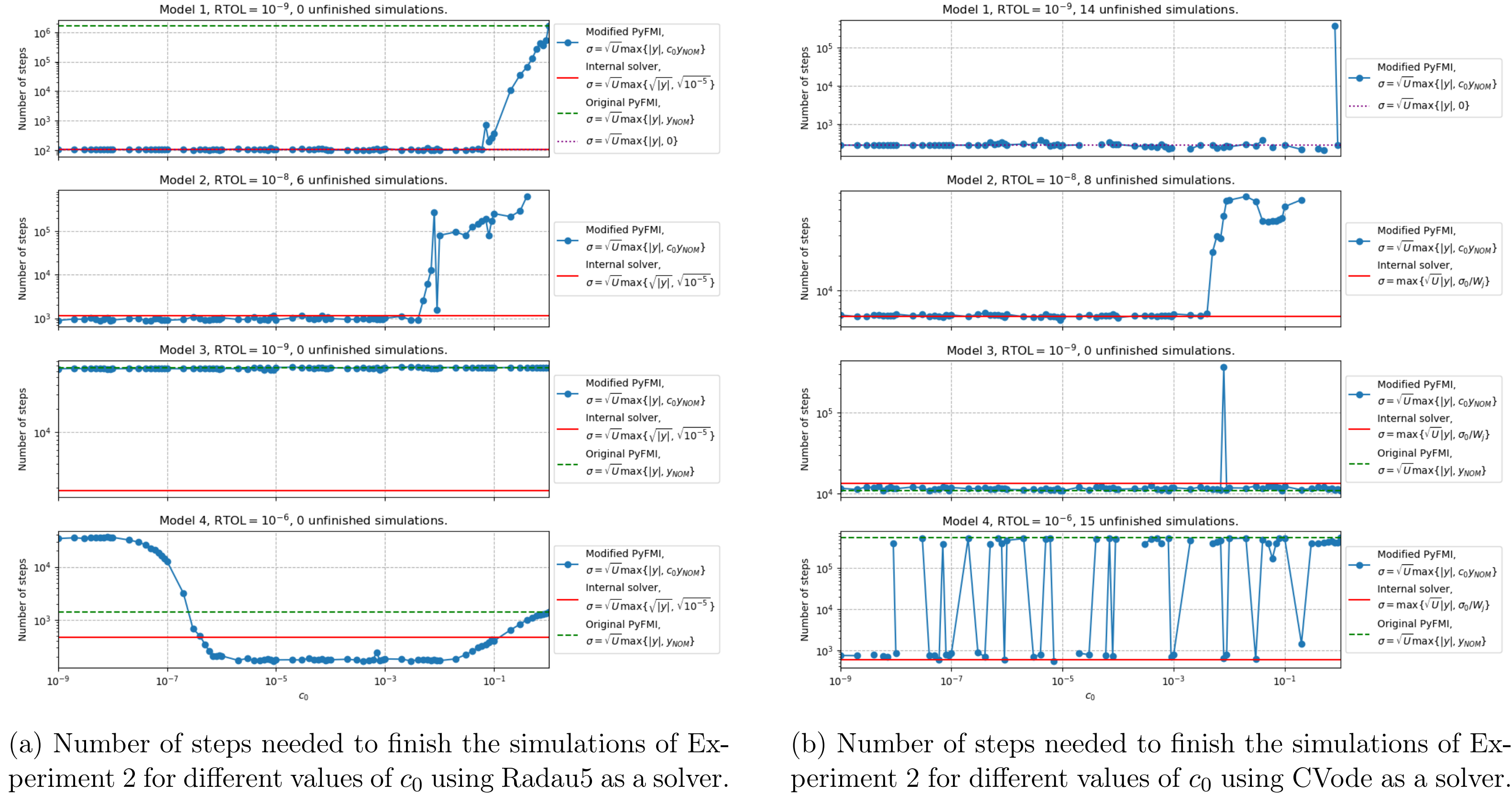
We used the following Cython code for the Jacobian increments.

```
for i in range(len_v):
        eps_pt[i] = eps_pt[i] = RUROUND*(max(abs(v_pt[i]),
        ↪  self._sigma_0*nominals_pt[i]))
```

We see from the results in Figure 7.1a that when using Radau5 as a solver Model 1 and 2 show a significant improvement for constants below $6 \cdot 10^{-2}$ and $4\dot{1}0^{-3}$ respectively. Model 4 performs best with a constant in the range $[7 \cdot 10^{-7}, 1 \cdot 10^{-2}]$. For Model 3 we see no improvement what so ever.

In Figure 7.1b we see that for Model 1 we get a good result for constants under $5 \cdot 10^{-1}$, where before we would get no result at all. Model 2 shows a good result for $c_0 \leq 4 \cdot 10^{-3}$. Apart from an outlier, Model 3 does not seem to improve at all, just as was the case with Radau5 as a solver. Model 4 exhibits a strange behavior where about a third of the constants results in a behavior like original PyFMI, a third like the internal solver and a third fails to finish.

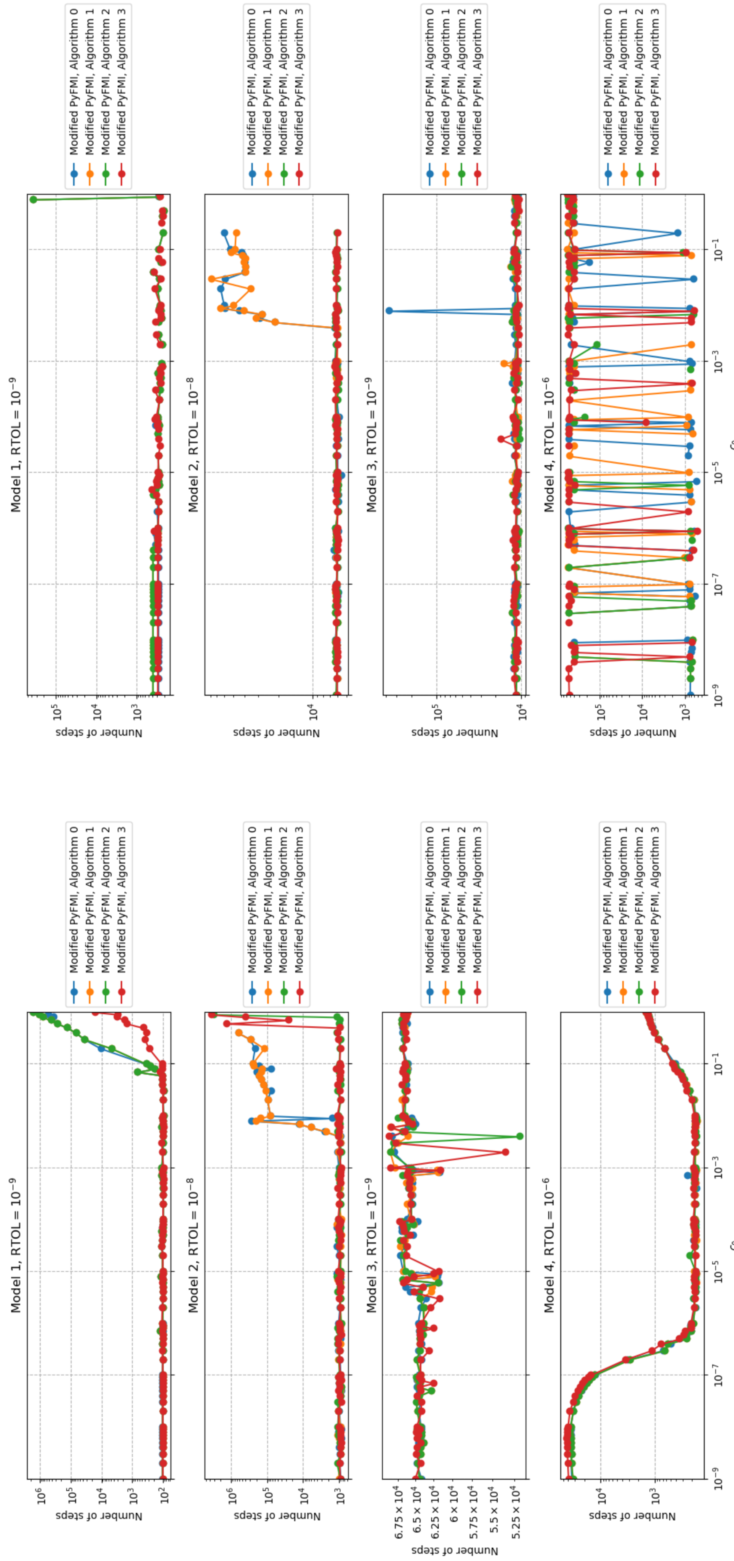
In Figures 7.2a and 7.2a we can see that prepending the additional algorithm seems to have no effect on Models 3 and 4, while Model 1 and 2 gives a better result with larger constants.



(a) Number of steps needed to finish the simulations of Experiment 2 for different values of c_0 using Radau5 as a solver.

(b) Number of steps needed to finish the simulations of Experiment 2 for different values of c_0 using CVode as a solver.

Figure 7.1: The result of Experiment 2 with no additional algorithm.



(a) Number of steps needed to finish the simulations of Experiment 2 for different values of c_0 using Radau5 as a solver.

(b) Number of steps needed to finish the simulations of Experiment 2 for different values of c_0 using CVode as a solver.

Figure 7.2: Comparison the result of Experiment 2 using different post-processing algorithms.

7.3 Experiment 3 - No nominal values

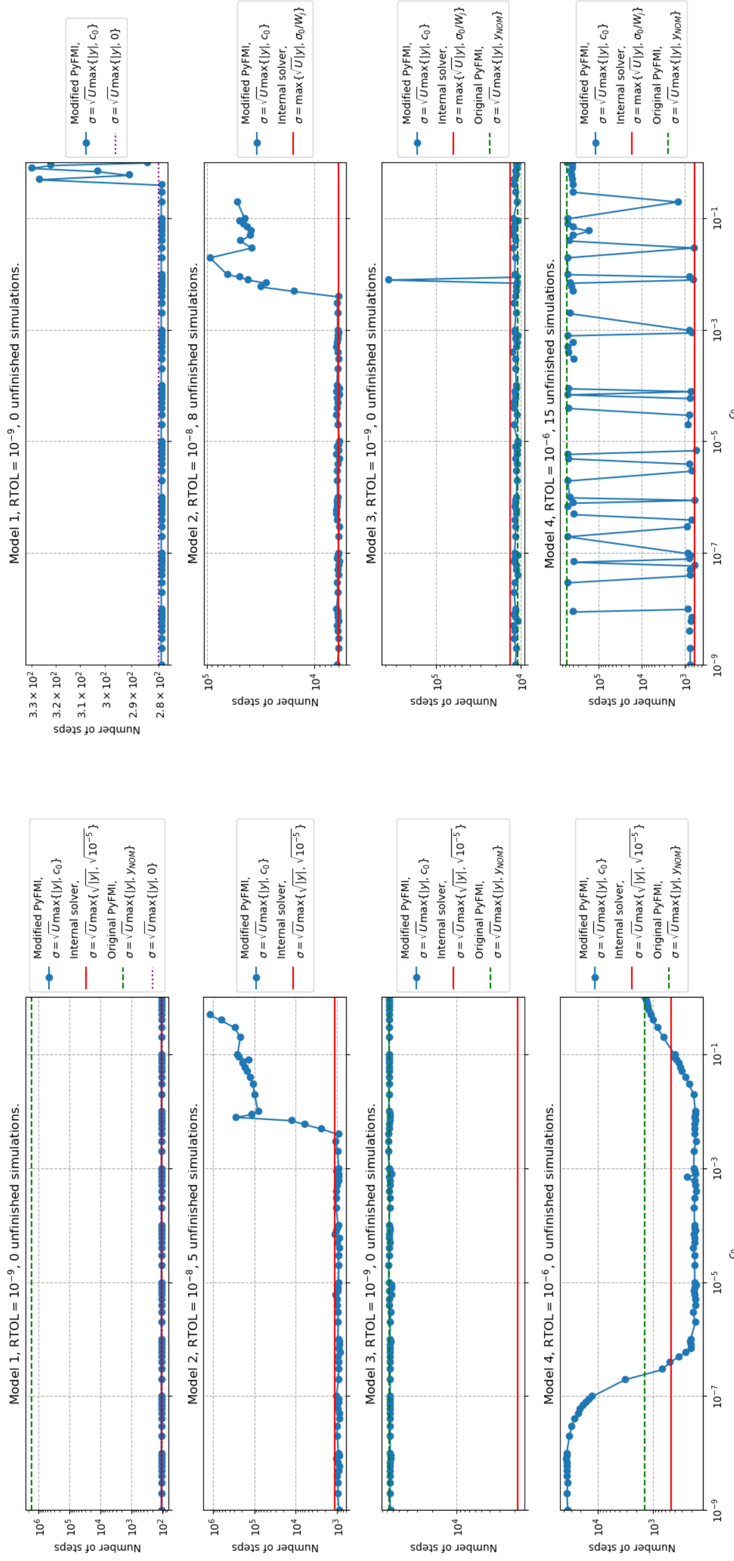
As a third experiment we disregard the nominal values completely, and instead use

$$\sigma_i = \sqrt{U} \max\left\{\left|y_i^{n(0)}\right|, c_0\right\}.$$

For this experiment we used the following code for the Jacobian increments.

```
for i in range(len_v):
        eps_pt[i] = RUROUND*(max(abs(v_pt[i]),
        ↪  self._sigma_0))
        temp = v_pt[i] + eps_pt[i]
        eps_pt[i] = temp - v_pt[i]
```

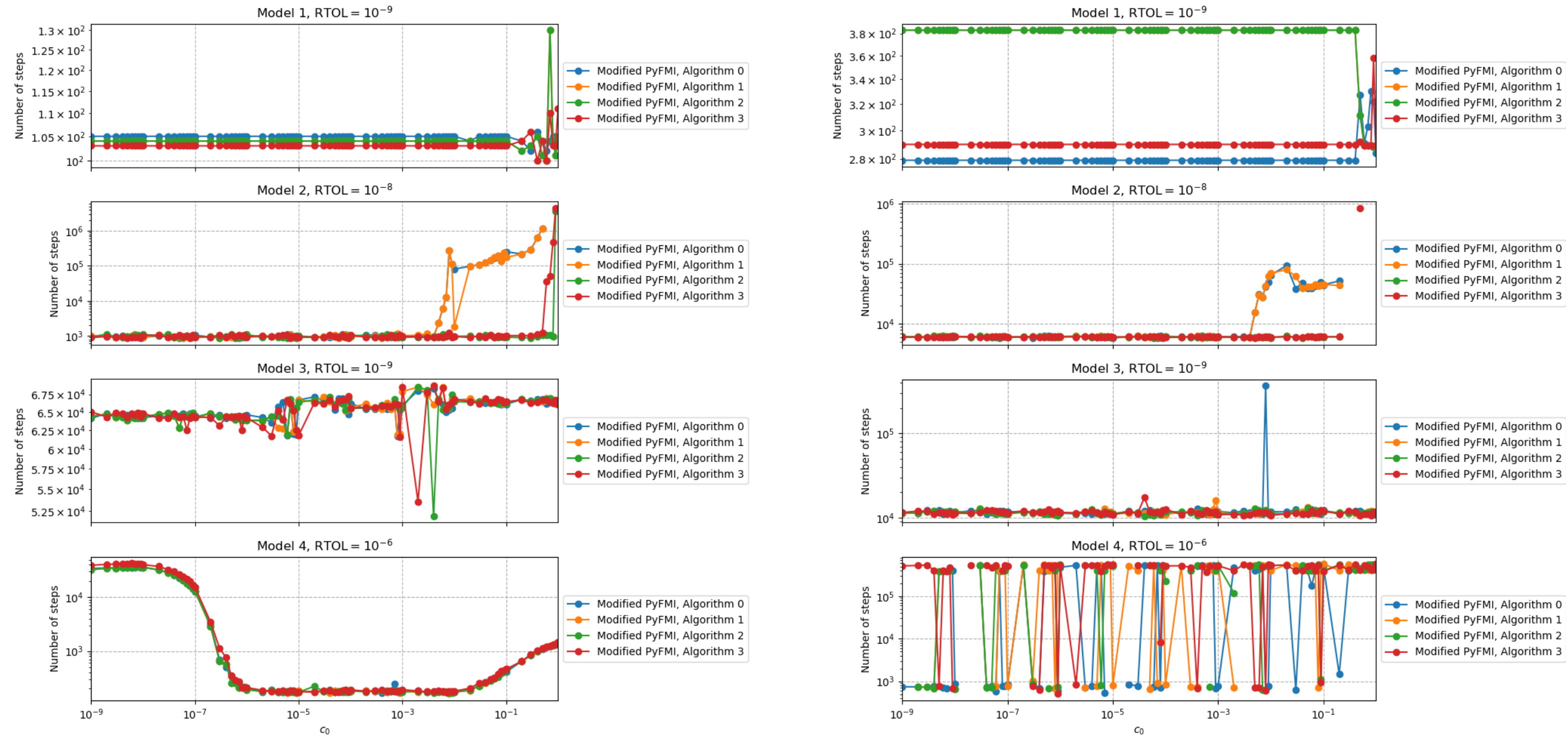
Looking at the results in Figure 7.3a we can see that Model 1 behaves very well. This is to be expected as instead of using an oversized nominal value we now use values below 1. The results from Model 2 are more interesting. However the result does not seem to differ much from those in Experiment 2.



(a) Number of steps needed to finish the simulations of Experiment 3 for different values of c_0 using Radau5 as a solver.

(b) Number of steps needed to finish the simulations of Experiment 3 for different values of c_0 using CVode as a solver.

Figure 7.3: The result of Experiment 3.



(a) Number of steps needed to finish the simulations of Experiment 3 for different values of c_0 using Radau5 as a solver.

(b) Number of steps needed to finish the simulations of Experiment 3 for different values of c_0 using CVode as a solver.

Figure 7.4: Comparison the result of Experiment 3 using different post-processing algorithms.

7.4 Experiment 4 - Different curvature scale

As a fourth experiment we used the increments

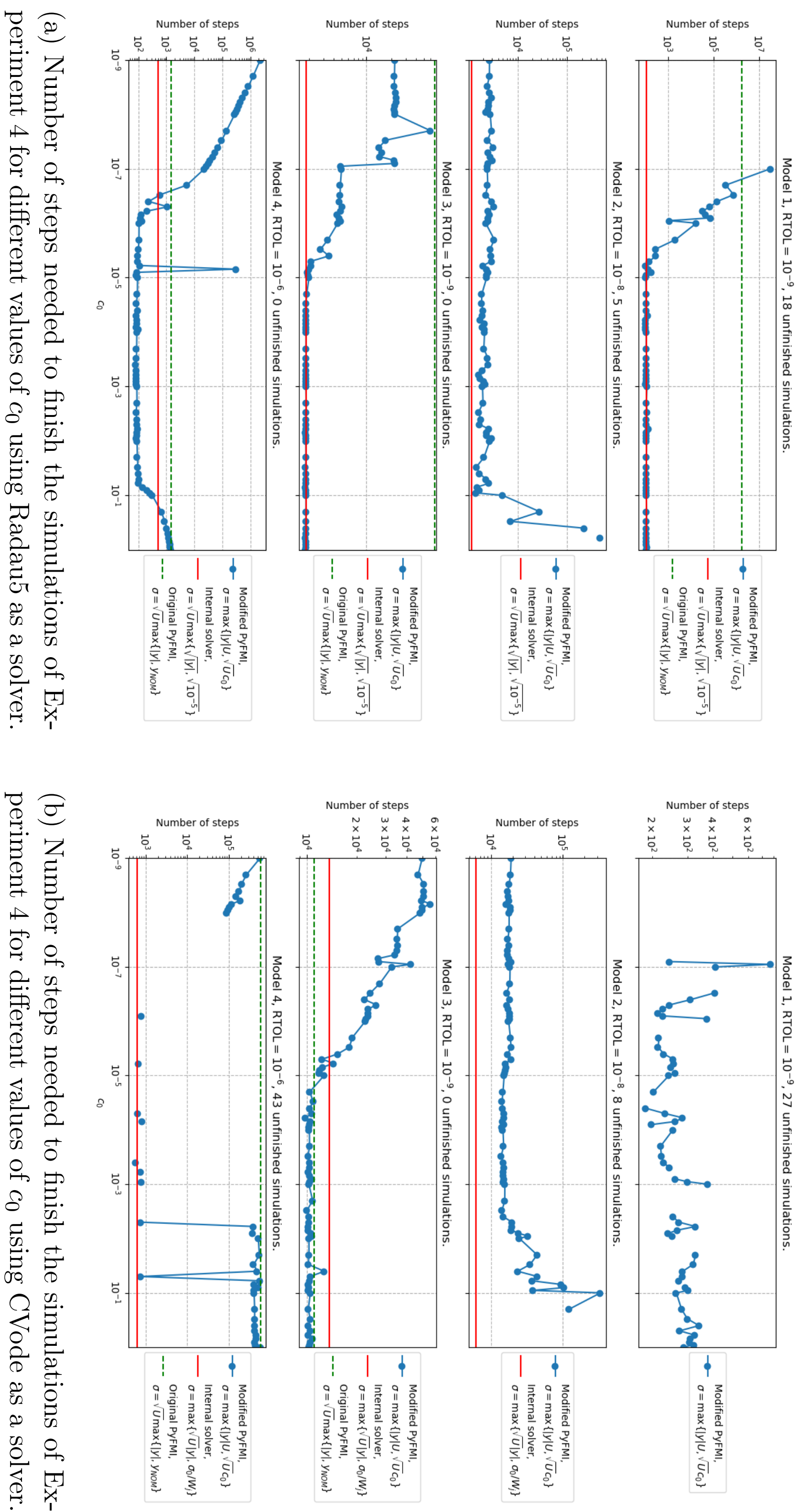
$$\sigma_i = \max\left\{\left|y_i^{n(0)}\right| U, c_0\sqrt{U}\right\},$$

meaning we make the assumption that $\frac{|G|}{|G_{yy}|}$ is constant instead of $\frac{|G|}{|G_{yy}|} = |y|^2$ as in previous experiments.

For this experiment we used the following code for the Jacobian increments.

```
for i in range(len_v):
        eps_pt[i] = RUROUND*(max(abs(v_pt[i])*RUROUND,
         ↪  self._sigma_0))
        temp = v_pt[i] + eps_pt[i]
        eps_pt[i] = temp - v_pt[i]
```

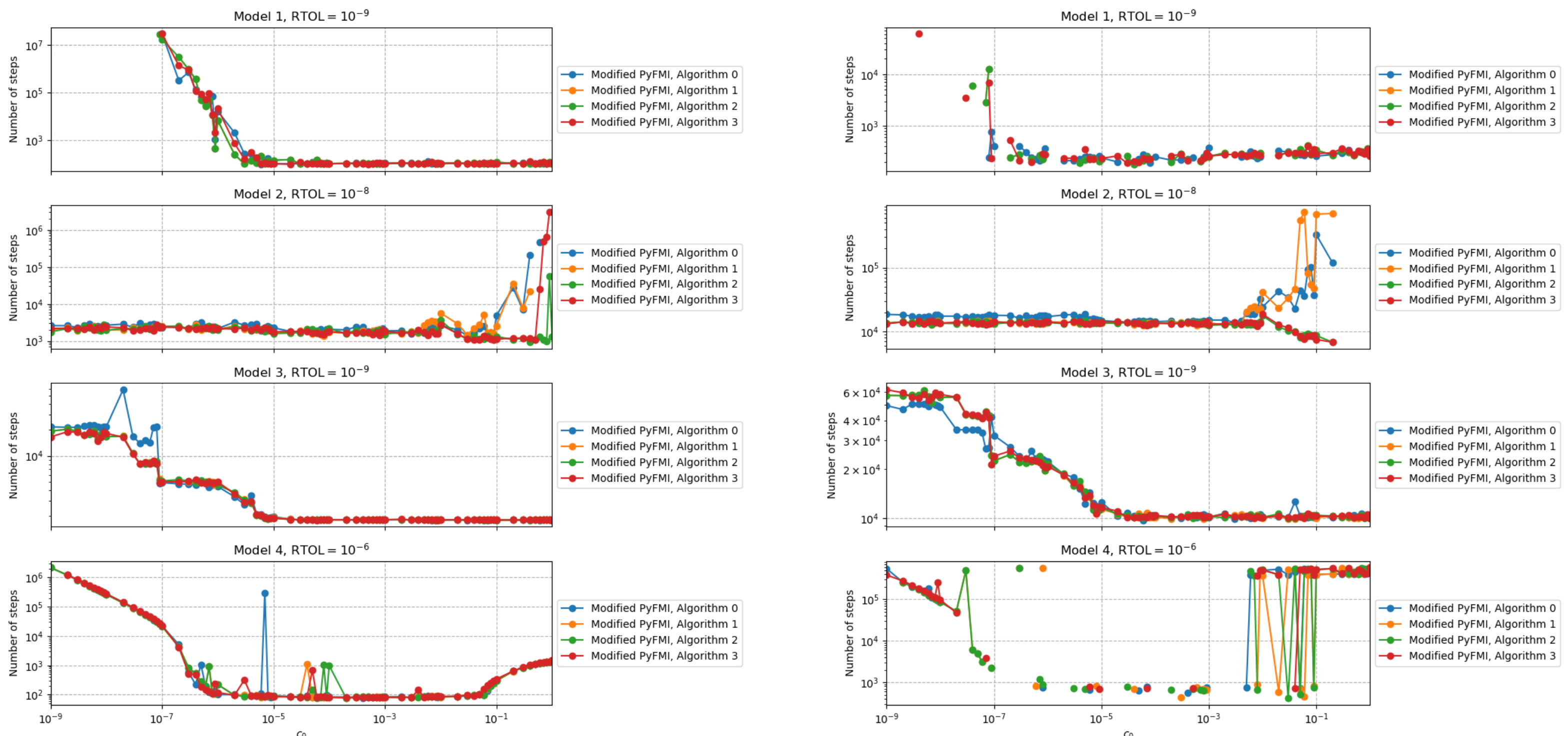
In Figure 7.5a we see that, when using Radau5 as a solver, Model 1 gives a good result for higher constants but bad for lower, in contrast to Experiment 2. Most notable however is the behavior for Model 3, where we get a good result for $c_0 \geq 6\dot{1}0^{-6}$. In Figure 7.5b we see no improvement with CVode as a solver.



(a) Number of steps needed to finish the simulations of Experiment 4 for different values of c_0 using Radau5 as a solver.

(b) Number of steps needed to finish the simulations of Experiment 4 for different values of c_0 using CVode as a solver.

Figure 7.5: The result of Experiment 4.



(a) Number of steps needed to finish the simulations of Experiment 4 for different values of c_0 using Radau5 as a solver.

(b) Number of steps needed to finish the simulations of Experiment 4 for different values of c_0 using CVode as a solver.

captionComparison the result of Experiment 4 using different post-processing algorithms.

7.5 Experiment 5 - Powers of 2

This experiment is a essentially the same as Experiment 3, but ensuring that the increments are always selected as powers of 2.

This means that for some $y_i^{n(0)} = (-1)^S s_y \beta^{E_y - p+1}$ we choose the increments as

$$\sigma_i = \sqrt{U} \max \left\{ \beta^{E_y}, c_0 \right\},$$

for $c_0 \in \left\{ 2^{-k}; k \in \mathbb{N} \right\}$.

The replacement of $|y_j| \sqrt{U}$ from Experiment 3 with the similar β^{E_y-p+1} is done using the function `nextafter()` from the Python package numpy [30]. This function is used to find the next representable floating point number. `nextafter()` takes two non-optional arguments, the starting point and the direction. This way both alternatives for increment is a power of 2, ensuring that the float division is exact.

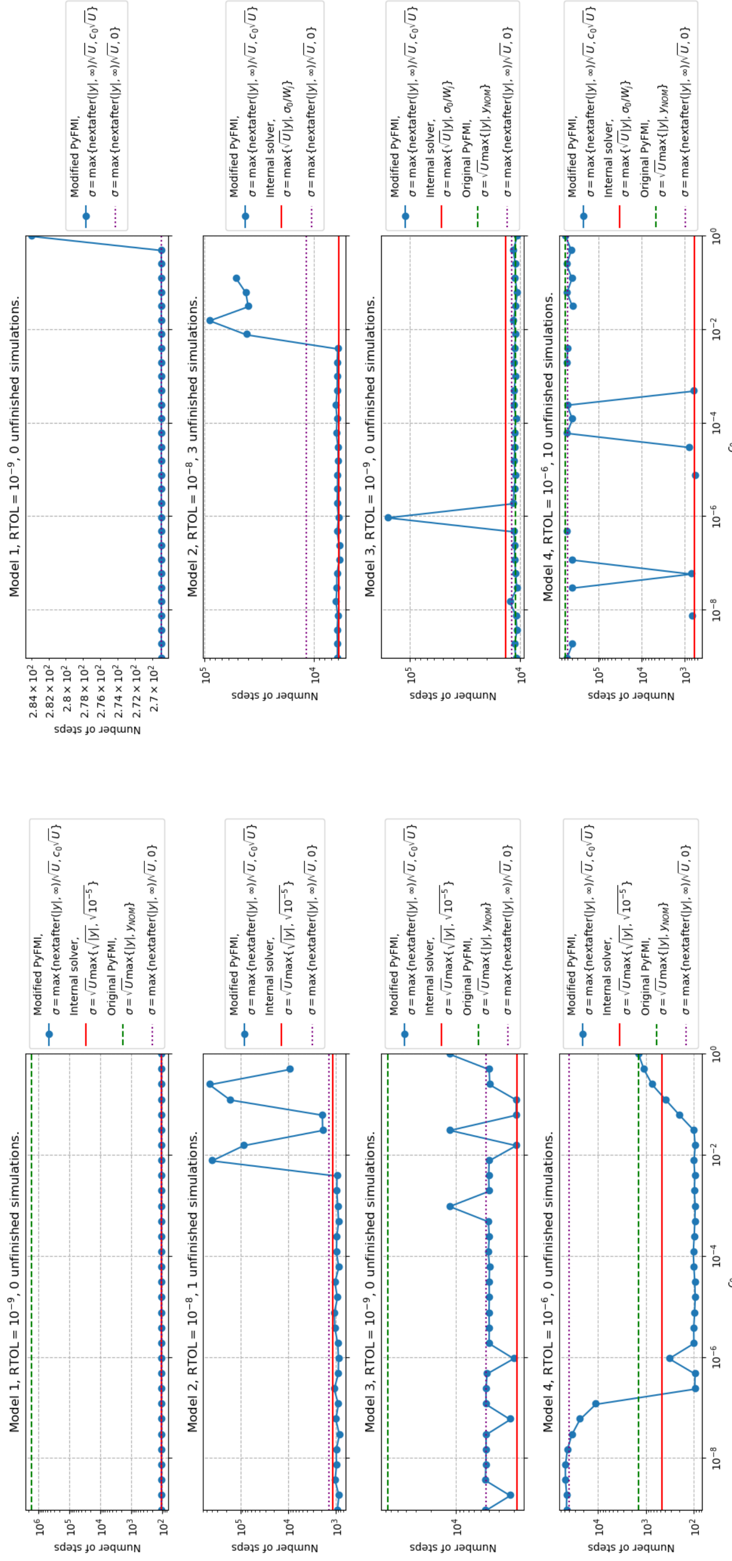
In C the equivalent `nextafterf` defined in the header `math.h` [31] can be used.

We used the following Cython code for the Jacobian increments:

```
import numpy as N

for i in range(len_v):
        eps_pt[i] = max((N.nextafter(abs(v_pt[i]),
         ↪   N.inf)-abs(v_pt[i]))/RUROUND,
         ↪   RUROUND*self._sigma_0)
```

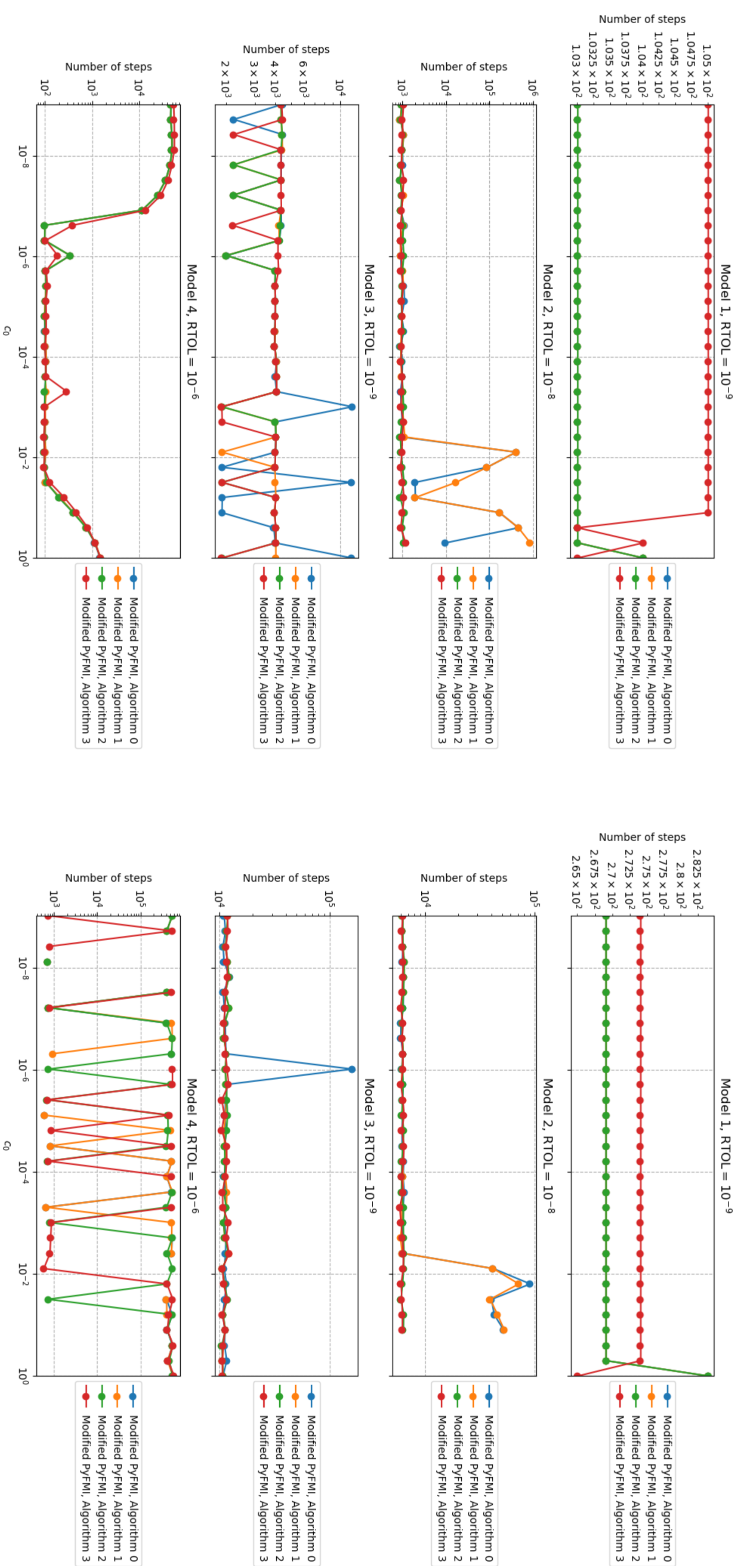
We see in Figures 7.7a and 7.7b we can see that using $c_0 = 0$ is no longer an issue. This is because we are using `nextafter()` towards infinity, meaning it will always be non-zero, and gets a build-in minimum of $\beta^{E_{MIN}-p+1}$, the minimal subnormal value. This means the alternative term is no longer needed to avoid division by zero.



(a) Number of steps needed to finish the simulations of Experiment 5 for different values of c_0 using Radau5 as a solver.

(b) Number of steps needed to finish the simulations of Experiment 5 for different values of c_0 using CVode as a solver.

Figure 7.7: The result of Experiment 5.



(a) Number of steps needed to finish the simulations of Experiment 5 for different values of c_0 using Radau5 as a solver.

(b) Number of steps needed to finish the simulations of Experiment 5 for different values of c_0 using CVode as a solver.

Figure 7.8: The result of Experiment 5.

Chapter 8

Summary and conclusion

From the numerical experiments performed we can see several ways to improve upon finite difference approximations, not only for PyFMI by for other solvers as well. Experiments 2 and 3 mainly concerns the use of nominal values in PyFMI, while the other experiments suggests the possibility of general improvements. We suggest that moving forward PyFMI uses the following code for selecting Jacobian increments:

```
import numpy as N

for i in range(len_v):
        eps_pt[i] = max((N.nextafter(abs(v_pt[i]), N.inf) -
         ↪  abs(v_pt[i])) / RUROUND, RUROUND * 2**(-9))
        if N.sign(v_pt[i]):
                if N.sign(v_pt[i]) != N.sign(v_pt[i] +
                 ↪  eps_pt[i]):
                        eps_pt[i] *= N.sign(v_pt[i])
```

The value 2^{-9} is chosen as it the lower power of 2 closest to $\sqrt{10^{-5}}$. It is possible that quantitative studies would suggest a better alternative.

8.1 Regarding nominal values

While Radau5 and CVode uses an alternative increment when $|y|$ is close to zero, PyFMI uses an alternative when $|y| < y_{NOM}$, where y_{NOM} is a strictly positive value that is set at model creation. y_{NOM} defaults to 1.0 if not entered. If y at this point is considered to be zero, there are possible benefits to this, as we shall see in section 8.3, but PyFMI does not do this. From

the experiments we can see that moving forward some adjustment ought to be done to PyFMIs way of choosing increments. One possibility would be to stop using nominal values altogether, and replace the choice of increments in PyFMI to be

$$\sigma_i = \sqrt{U} \max \left\{ \left| y_i^{n(0)} \right|, c_0 \right\},$$

as in Experiment 3. From the experiments we see that $c_0 \approx \sqrt{10^{-5}}$, as used in Radau5, would be a good constant. Further testing with more models would be necessary to confirm whether this is a good choice. Especially since all the models used in this thesis had a relatively large nominal value.

If this is to radical of a change, there are other possible improvements such as using a default value lower than 1.0, for example $\sqrt{10^{-5}}$, for the nominal values. This change cannot be applied to the FMU, but the `.xml` of the FMU should contain the information of which states have a nominal value entered. An easier way to achieve this could to, as in Experiment 2, applying a multiplier to all the nominal values.

Another possible alteration, not tested in these experiments, could be to use nominal values when entered, and otherwise default to use the solver increments.

8.2 General improvements

From the results of Experiment 4, where we replaced the common assumption $\frac{|G|}{|G_{yy}|} = \left| y_j^{n(m)} \right|^2$ with a constant, we get good results in the span $[10^{-5}, 10^{-2.5}]$. This was the only experiment that improved upon Model 3. The internal Radau5 Jacobian, which uses the curvature scale $\frac{|G|}{|G_{yy}|} = \left| y_j^{n(m)} \right|$, performs much better in all other experiments. This could be seen as middle ground of the standard assumption and a constant. Moving forward it could be beneficial to let the Jacobian increments be dependent on a user entered curvature scale. This could either be available as an argument to the solver, or be added to the FMI-standard. However, as this information is reasonably unknown to the user, experiments with a curvature adaptive solver would also be interesting. Such a solver could test whether or not a change to the curvature scale would be beneficial after a number of discarded time steps.

From Experiment 5 we can see that it can be advantageous to change all increments to powers of 2 as this makes all divisions exact. We also see that by replacing $|y|\,U$, with programming language functions for retrieving the next step away from zero, we no longer have a need to have an alternative

term in order to avoid division by zero. It can however still be a good idea to have such a term in order to be able to pre-calculate and store function evaluations for when $|y|$ is close to zero. It is possible that the magnitude of this alternative could be based on the tolerances. If so, the tolerances would have to be imported from the solver as they are not properties of the FMU. Both Radau5 and CVode would theoretically benefit from the same improvements that were made to PyFMI in Experiment 5. Especially Radau5 that uses a floating point representation of an irrational number instead of a power of 2.

Some of the improvements we see from the various post-processing algorithms might lie in the fact that the increments get "chopped", setting a lot of significand digits to zero. This increased number of zeros make it more likely for the division to be exact. Using increments like the ones in Experiment 5, the division is always exact. The necessary post-processing is sign adjustment, preserving the assumptions necessary for Sterbenz lemma to hold.

8.3 Other considerations

Although it does not directly affect the number of steps, there is also reason to consider another equation presented in [8, p.80], as it could reduce the number of function calls. There it is suggested for a chosen increment σ_j, to use the following equation for the finite difference approximation for the Jacobian elements

$$J_{ij} = \begin{cases} \frac{f_i(u+\sigma_j e_j)-f_i(u)}{\sigma_j} & u \neq 0, \\ \frac{f_i(\sigma_j e_j)-f_i(u)}{\sigma_j} & u = 0, \end{cases}$$

with e_j the unit vector.

With σ_j designed to use a static value when the state variable is approaching zero, as is the case with Radau5, the evaluation $f(\sigma_j e_j)$ can be done beforehand.

If this equation was used for the Jacobian elements in PyFMI, with σ_j chosen as it is now, the Jacobian approximations would not need a single function evaluation for Model 4, since it always uses the nominal values for the increments.

Bibliography

[1] The Modelica Association et al., "Website for fmi-standard," 2023, [Accessed 16-August-2023]. [Online]. Available: https://fmi-standard.org

[2] ——, "Modelica association projects," [Accessed 02-October-2023]. [Online]. Available: https://modelica.org/projects

[3] ——, "Modelica language specification version 3.6," 2023, [Accessed 27-September-2023]. [Online]. Available: https://specification.modelica.org/maint/3.6/MLS.html

[4] ——, "Modelica libraries," 2024, [Accessed 3-January-2024]. [Online]. Available: https://modelica.org/libraries/

[5] C. Andersson, C. Führer, and J. Åkesson, "Pyfmi: A python package for simulation of coupled dynamic models with the functional mock-up interface," *Technical Report in Mathematical Sciences*, vol. LUTFNA-5008-2016, no. 2, 2016.

[6] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward, "SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers," *ACM Transactions on Mathematical Software (TOMS)*, vol. 31, no. 3, pp. 363–396, 2005.

[7] E. Hairer and G. Wanner, *Solving Ordinary Differential Equations II*. Berlin: Springer Berlin, Heidelberg, 2010.

[8] C. T. Kelley, *Iterative Methods for Linear and Nonlinear Equations*. Society for Industrial and Applied Mathematics, 1995. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611970944

[9] Wikipedia contributors, "Kronecker delta — Wikipedia, the free encyclopedia," 2023, [Accessed 27-November-2023]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Kronecker_delta&oldid=1178239896

[10] M. T. Heath, *Scientific Computing.* Philadelphia, PA: Society for Industrial and Applied Mathematics, 2018. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611975581

[11] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol, and S. Torres, *Handbook of Floating-Point Arithmetic*, 2nd ed. Cham, Switzerland: Birkhäuser, 2018.

[12] P. H. Sterbenz, *Floating-Point Computation.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1974.

[13] Wikipedia contributors, "Machine epsilon — Wikipedia, the free encyclopedia," 2023, [Accessed 16-November-2023]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Machine_epsilon&oldid=1182899489

[14] ——, "Big o notation — Wikipedia, the free encyclopedia," 2023, [Accessed 27-November-2023]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Big_O_notation&oldid=1184240535

[15] ——, "Floating-point arithmetic — Wikipedia, the free encyclopedia," 2023, [Accessed 20-November-2023]. [Online]. Available: https://en.wikipedia.org/wiki/Floating-point_arithmetic#Multiplication_and_division

[16] ——, "Propagation of uncertainty — Wikipedia, the free encyclopedia," 2023, [Accessed 29-September-2023]. [Online]. Available: https://en.wikipedia.org/wiki/Propagation_of_uncertainty

[17] W. H. Press, *Numerical Recipes in C: The Art of Scientific Computing*, 2nd ed. Cambridge University Press, 2002.

[18] Wikipedia contributors, "Low-level programming language — Wikipedia, the free encyclopedia," 2023, [Accessed 24-November-2023]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Low-level_programming_language&oldid=1182236747

[19] ——, "High-level programming language — Wikipedia, the free encyclopedia," 2023, [Accessed 24-November-2023]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=High-level_programming_language&oldid=1184757006

[20] C. Andersson, C. Führer, and J. Åkesson, “Assimulo: A unified framework for {ODE} solvers,” *Mathematics and Computers in Simulation*, vol. 116, no. 0, pp. 26 – 43, 2015.

[21] K. Radhakrishnan and A. C. Hindmarsh, “Description and use of lsode, the livemore solver for ordinary differential equations,” NASA, United States, Tech. Rep., Dec. 1993. [Online]. Available: https://ntrs.nasa.gov/citations/19940030753

[22] Ernst Hairer, “Fortran and matlab codes,” http://www.unige.ch/~hairer/software.html, [Accessed 2-May-2023].

[23] Modelon Community, “Pyfmi repository,” https://github.com/modelon-community/PyFMI, [Release 2.11.0].

[24] Various contributors, “Assimulo homepage,” https://jmodelica.org/assimulo, [Accessed 12-October-2023].

[25] Wikipedia contributors, “Real versus nominal value — Wikipedia, the free encyclopedia,” 2023, [Accessed 8-January-2024]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Real_versus_nominal_value&oldid=1170666205

[26] E. Süli and D. F. Mayers, *An Introduction to Numerical Analysis*. Cambridge University Press, 2003.

[27] A. C. Hindmarsh, R. Serban, C. J. Balos, D. J. Gardner, D. R. Reynolds, and C. S. Woodward, “User documentation for cvode,” 2023, v6.5.1. [Online]. Available: https://sundials.readthedocs.io/en/latest/cvode

[28] Lawrence Livermore National Laboratory, “Sundials repository,” https://github.com/LLNL/sundials, [Accessed 3-May-2023].

[29] Modelon Community, “Assimulo repository,” https://github.com/modelon-community/Assimulo, [Release 3.4.3].

[30] NumPy Developers, “Numpy documentation,” [Version 1.26]. [Online]. Available: https://numpy.org/doc/1.26/index.html

[31] cppreference, “Common mathematical functions,” [Accessed 22-November-2023]. [Online]. Available: https://en.cppreference.com/w/c/numeric/math

Appendix A

The design of $sigma_0$ in CVode

In [21, p.66] the authors reason that the standard choice of $\sigma = \sqrt{U}\,|y|$ cannot be used if $|y|$ is very small or equal to zero and therefore base an alternative value on noise level.

Since the roundoff error of each f_i is of order $U\,|f_i|$ the equation

$$J_{ij} \approx \frac{f_i(t_n, y + \sigma_j e_j) - f_i(t_n, y)}{\sigma_j}$$

has an error of order $U\,|f_i|\,/\sigma_j$.

Because the method coefficient $\beta_0 (= l_0)$ is of order unity the error δP_{ij} of the iteration matrix $P = \mathrm{I} - h_n \beta_0 J$ is approximately

$$\delta P_{ij} \approx |h|\,U\,|f_i|\,/\sigma_j.$$

We introduce the vector $s \in \mathbb{R}^N$ with elements

$$s_j = 1/\sigma_j, \quad j = 1, \ldots, N$$

and $|f| \in \mathbb{R}^N_+$ with

$$|f|_j = |f_j|\,, \quad j = 1, \ldots, N$$

and rewrite δP as

$$\delta P = |h|\,U\,|f|\,s^{\mathrm{T}}.$$

The idea is then to find σ_j by bounding δP.

If we construct a diagonal matrix $D \in \mathbb{R}^{N \times N}$ with the elements

$$D_{ii} = W_i, \quad i = 1, \ldots, N$$

we see that the weighted RMS-norm can be rewritten as

$$\begin{aligned}\|v\|_{WRMS} &= \sqrt{\frac{1}{N}\sum_{i=1}^{N}(v_i W_i)^2} \\ &= \frac{1}{\sqrt{N}}\sqrt{\sum_{i=1}^{N}(D_{ii}v_i)^2} \\ &= \frac{\|Dv\|_E}{\sqrt{N}},\end{aligned}$$

where $\|\cdot\|_E$ is the Euclidean norm.

The norm of δP is given by

$$\|\delta P\|_{WRMS} = \max_v \frac{\|\delta P v\|_{WRMS}}{\|v\|_{WRMS}},$$

where

$$\begin{aligned}\|\delta P v\|_{WRMS} &= |h|\, U \left\| |f|\, s^{\mathrm{T}} v \right\|_{WRMS} \\ &= |h|\, U \|f\|_{WRMS} \left| s^{\mathrm{T}} v \right|,\end{aligned}$$

for some $v \in \mathbb{R}^N$. Hence

$$\begin{aligned}\|\delta P\|_{WRMS} &= |h|\, U \|f\|_{WRMS} \max_v \frac{\left|s^{\mathrm{T}} v\right|}{\|v\|_{WRMS}} \\ &= |h|\, U \|f\|_{WRMS} \max_v \frac{\left|(D^{-1}s)^{\mathrm{T}}(Dv)\right|}{\|v\|} \\ &= |h|\, U \|f\|_{WRMS} \max_v \frac{\|D^{-1}s\|_E \|Dv\|_E}{\|Dv\|_E / \sqrt{N}} \\ &= |h|\, U \|f\|_{WRMS} \sqrt{N} \left\|D^{-1}s\right\|_E \\ &\le |h|\, U N \|f\|_{WRMS} \max_i \left(\frac{1}{\sigma_i W_i}\right).\end{aligned}$$

Or equivalently

$$\|\delta P\| \le \frac{|h|\, U N \|f\|}{\min_i(\sigma_i W_i)}.$$

To establish the maximum allowable error in P, we consider the linear system $Px = b$, which is the form to be solved in each Newton iteration. To first order, the error δx in x due to the error δP in P is given by

$$\frac{\|\delta x\|}{\|x\|} \leq \frac{\|\delta P\|}{\|P\|} = \left\|P^{-1}\right\| \|\delta P\| .$$

$\|P^{-1}\|$ is unknown but is expected to be unity since $P \to \mathrm{I}$, as $h \to 0$. Therefore, a reasonable strategy is to bound $\|\delta P\|$ alone by selecting a suitably small value for the relative error that can be tolerated in the Newton correction vector. By using a value of 10^{-3} for this tolerance, we obtain

$$\min_i \sigma_i W_i \geq \frac{|h|\, UN\, \|f\|}{10^{-3}} =: \sigma_0.$$

In the case where $\sigma_0 = 0$, it is instead reset to 1.

Master’s Theses in Mathematical Sciences 2024:E8
ISSN 1404-6342

LUNFNA-3038-2024

Numerical Analysis
Centre for Mathematical Sciences
Lund University
Box 118, SE-221 00 Lund, Sweden

http://www.maths.lu.se/