Numerical integration of differential-algebraic systems with recommendations to improve the efficiency of multibody simulations

Nijhuis, T.R.T.

Award date:
2013

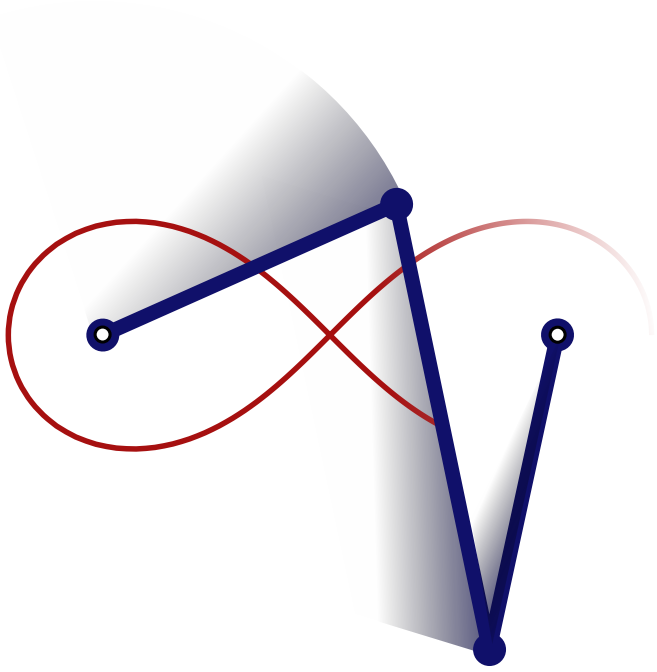
EINDHOVEN University of Technology
Department of Mathematics & Computer science

MASTER'S THESIS

Numerical integration of differential-algebraic systems

with recommendations to improve the efficiency of multibody simulations

T.R.T. NIJHUIS



Supervised by | prof. dr. W.H.A. SCHILDERS
dr. G.H.K. HEIRMAN

Abstract

This project provides an exploratory answer to the question of how to speed up the software package *LMS Virtual.Lab Motion*, by LMS International. This software package is a simulator for mechanical devices that offers a variety of time integration algorithms, the main one is based upon the general-purpose integrator DASSL. Our results can reasonably be adapted to fit other applications that make use of DASSL or a similar time-integrator.

Three approximate Newton methods (Simplified, Simplified with an approximate LU-decomposition, and full Newton with an approximate Krylov solver) are studied to investigate whether or not they are likely to increase the speed of DASSL. This study is not performed by implementing them but rather by providing an estimate on how well they would perform when they are implemented.

We define the concept of refresh-cycle for integrators that use the simplified method. This is the number of time-steps for which the integrator reuses an old Jacobian before refactoring. We exploit the fact that the error in the Jacobian is likely to be non-decreasing during such a refresh-cycle in our application.

The results of the study on the separate approximate Newton methods suggests the concept of a three-phase time-integrator that cycles through the three methods. This way, the benefits from each of the methods are exploited on the right moment in the cycle. The expected speed gain of the three-phase cycle is investigated. And we conclude that the addition of the approximate LU-decomposition is always beneficial and the addition of the Krylov solver is beneficial for mechanical models with many sudden changes in the system's dynamics (e.g. collisions).

Finally we exploit some features of the Jacobian matrix to provide a way of updating it without the need to refactor. These are specifically designed for *LMS Virtual.Lab Motion* but other time-integrators might benefit from similar ideas.

- Do not disclose this thesis before August 2014 -

Keywords: computational efficiency; DASSL; differential-algebraic equations (DAE); Krylov methods; multibody dynamics; Newton methods; numerical time-integration; simulation

Cover illustration: A four-bar linkage. The midpoint of the centre bar draws out a lemniscate.

Acknowledgements

"The only way to atone for being occasionally a little over-dressed is by being always absolutely over-educated."

OSCAR WILDE
PHRASES AND PHILOSOPHIES FOR THE USE OF THE YOUNG (1894)

For this project I have had the pleasure to be supervised by Wil Schilders, from the TU/e, and Gert Heirman, from LMS. The two of them have shown great trust and confidence in me by allowing me to chart my own way. Moreover, they have regarded me as an equal and were as eager to learn from me as I from them. This working relationship motivated me greatly and my supervisors deserve my fond gratitude for this approach. Further thanks goes to the two other member of my graduation committee, Sjoerd Rienstra and Michiel Hochstenbach. They provided us with additional angles and insights that surely are worthwhile to peruse.

This thesis is the result of a nine month internship at LMS International in Leuven. I thank LMS for their impressive hospitality and for an altogether pleasant working environment. The R&D staff with whom I worked daily was very inspiring and I am thankful for the care that everybody took to help me out in the world of multi-body dynamics. The fact that LMS intends to apply for patent on my work is a great honour. I am flattered that my work is taken so seriously and by their intention to follow up on it.

Leisure in Leuven has been amazing. Not in the least because of the presence of Alba, Raquel, Andreas and Fabio. Thanks guys, it was a blast!

This graduation project is the crown upon my studies in Eindhoven. The staff of the department, teachers or otherwise, have crafted a very pleasant environment which I am sad to leave behind. Moreover, I'd like to thank my fellows at Dispuut "In Vino Veritas", my fellow board members, fellow committee members and all my other friends in Eindhoven for making my student life more than studious.

Finally, my family deserves a lion's share of my gratitude. My parents, for their love and care, for their support and for their genuine interest in whatever it is that occupies me; my sister-in-law-to-be, Anne, for being a splendid addition to the family; and my dearest brother, Rik, for being there always, for his relentless effort in helping me with the problems he himself overcame just a year ago and for being my best friend.

Tom Nijhuis
August, 2013

Contents

Chapter 1 – Introduction

"Sense, sure, you have. Else could you not have motion."

W. SHAKESPEARE
HAMLET (~1600)

1.1 – Background and motivation

1.1.1 – Mechanical devices

When designing a complicated mechanical device like, for example, a car, a robotic arm or a landing gear, it is often advisable to beforehand simulate the movement of the device. This practice not only gives insight in whether or not the device will work properly but also provides information on various important quantities like force and friction. In the early phase, manufacturers will typically simulate and test the various components of a device individually but at some point it is important to gain knowledge about the interaction between a greater number of components. In this phase, multibody simulations come into play.

A multibody simulator is a computer program which is capable of predicting the movement of a set of interconnected components under the influence of external and internal forces. To this end, one simulates the well known equations for mechanical movement (i.e. Newton's second law). But in addition the interaction between objects is modelled. This is less straightforward. An interaction between two object is, most commonly, the fact that they are linked in a way. This can be by joint, a spring, an actuator or other connections. A single joint between two bodies, say a ball joint like *Figure* 1.1, is modelled by an additional equation which states that the respective points where both bodies are linked are to be equal. This is effectively an imposed constraint on the motion of the bodies. Springs and actuators constitute extra equations stating that there is some force exerted on the two bodies as a of their relative position. Further terms and equations arise with the extension to more complicated interactions like damping, friction and even controls and hydraulics. The arising set of equations is called a system of differential-algebraic equations (DAEs).

Figure 1.1: A ball joint asserts that two corresponding points on two bodies have the same position.
(Image taken from `https://en.wikipedia.org/wiki/File:Srjoint.jpg`, licensed as CC-BY-SA-3.0)

1.1.2 – Virtual.Lab Motion

The company LMS, head-quartered in Leuven, Belgium, produces *LMS Virtual.Lab Motion* (or *VL Motion* for short), a software package capable of performing these simulations. Typical customers are manufacturers of cars, aeroplanes and machinery. *VL Motion* contains a variety of time-integrators, the one on which we will focus in this report is an integrator called DASSL, explained in *Section* 2.2.3.

1.2 – Goals and philosophy

The original goal of this project was to test whether or not efficiency can be gained for the time-integrator by switching from a direct solver to a Krylov solver. This goals was quickly expanded to investigating the possibilities to adapt a general-purpose time integrator to better fit a subclass of equations. The goal is to find a series of changes and additions (that may or may not include Krylov solvers) to the current integrator that are expected to increase overall efficiency. Each of these is formally motivated in terms of computational effort and preliminary testing is done to further solidify the claims made.

An important task of a simulation product is to perform repeated simulations with slightly varying parameters. This is an ubiquitous application for two reasons. The first being that some mechanical models are optimised for a certain characteristic (like passenger comfort or noise level) by changing a set of parameters. Then an optimisation is performed on a function that basically consists of performing the simulation. This optimisation can require many function calls. The second reason is that for safety reasons many manufacturers need to run very similar simulations many times with just slightly different conditions to be notified early of possible faults. This repetition makes it worthwhile to optimise an integrator for a single specific model.

The proposed changes are designed in such a way that they are strictly additions to the solver. It will always fall back to the current solution in the case that there is no benefit from the additions. This way we are assured that the augmented program will be at least as successful as the original one.

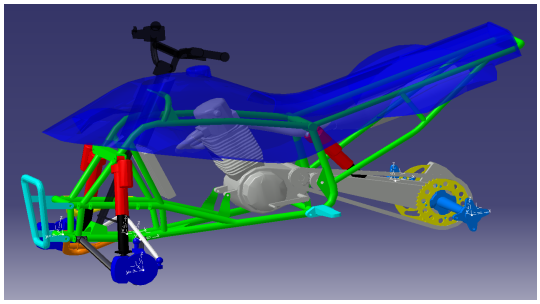
No implementation of the additions has been performed due to the time investment this requires. By avoiding copious amounts of programming we managed to investigate a large number of options, the most viable of which are presented here. Nonetheless, throughout the thesis many remarks are given on implementation.

We strive to base the reasoning and motivation as much as possible on theoretical results but experimental results are not neglected. In some cases they prove to be more useful than the theoretical ones.

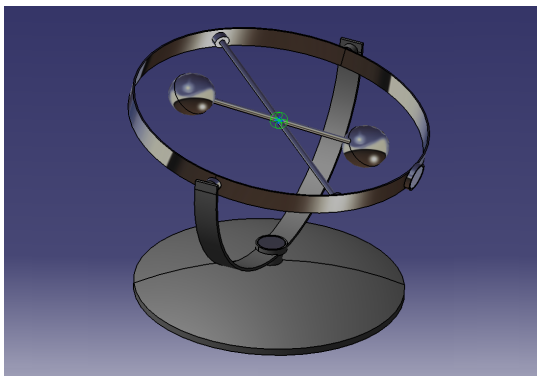
Furthermore, we identify some important characteristics of multibody models and we discuss how they affect the usefulness of the additions.

Throughout we use data from three models in *LMS Virtual.Lab Motion* to test various ideas.

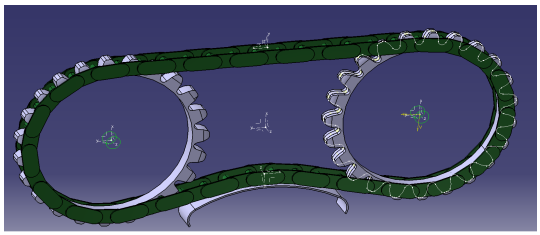
- **All terrain vehicle (ATV)**
  A model for a small four-wheeled vehicle driving over rough terrain. Although the motion is very wild and profound, the problem is rather smooth and not stiff. This model is similar to one a real client might be interested in simulating.



- **Gimbal**
  This is a model of a small desk toy consisting of two gimbals supporting an eccentric weight. The model is very small and very smooth. This model has very few equations and hence allows for quick testing.



- **Chain**
  A chain riding over two sprockets and one chain guide. The motion is smooth, but there are many collisions. This makes the problem very challenging.



1.3 – Outline and reading guide

This thesis consists of six chapters, three of which contain the original research.

- *Chapter* 2 contains preliminary knowledge on multibody simulations and time-integrators. There is no original research in this chapter. We do, however, call attention to the definition of the refresh cycle (*Definition* 2.2) which is a recurring motive throughout.
- *Chapter* 3 provides a general form of approximate Newton methods. This is used as a framework to define three specific method (one of which is the currently implemented method). Results are provided to compare the three methods. These results are the basis for the motivations in the next chapter.
- *Chapter* 4 contains the most important part of original research: the concept of the three-phase cycle. This concept is motivated from the results in the chapter before and extensive estimates are performed on the efficiency that can be gained by it.
- *Chapter* 5 can be seen as an encore. It provides some ideas to further speed up the integrator. The ideas presented in this chapter work with or without the three-phase cycle.

At the beginning of every chapter is a short overview of its structure and the main conclusion.

Chapter 2 – Preliminaries

"They, instead of remaining fixed in their places, move freely about, on or in the surface, but without the power of rising above or sinking below it."

E.A. ABBOTT
FLATLAND (1884)

Overview

This chapter contains two parts. In the first part the equations of motion are derived in the form of an n-dimensional evolution equation. The second part contains a description of DASSL, the time-integrator that is used to numerically solve the evolution equation. The concept of refresh cycle is defined as the set of time-steps that are solved with the same Jacobian matrix. The second part ends with definitions for computational effort and simulation rate.

2.1 – Modelling multibody dynamics

In this section the equations of motion will be derived in the form of a system of differential-algebraic equations. We will begin by defining rigid body motion and the coordinate system we use, then we will apply Newton's laws to these coordinates to obtain rigid body dynamics, finally, we add constraint equations to a system of multiple bodies to obtain multibody dynamics.

2.1.1 – Rigid body motion and generalised coordinates

A rigid body is a single, free moving object in space which does not deform. The model we use for it is a bounded, connected set of points in Euclidean space, say $\Sigma \in \mathbb{R}^n$. The movement of the body during the course of time is represented by a mapping,

$$\xi_t : \mathbb{R}^n \to \mathbb{R}^n \tag{2.1}$$

$$\xi_t(\Sigma_0) = \Sigma_t \,, \tag{2.2}$$

which maps every point in some initial reference set Σ_0 to its corresponding location in Σ_t which is the set of points that make up the body on time t. The mapping should be regarded as a series of spatial maps in the time domain. This is emphasised by time t not being an argument but a subscript parameter. By the definition of rigid body motion, this mapping is restricted in that it should not deform the body. In other words, it should preserve internal distances and angles

$$\forall_{c,v,w\in\mathbb{R}^n} \forall_{t\in\mathbb{R}^+} : \big(\xi_t(v) - \xi_t(c)\big)^{\mathsf{T}} \big(\xi_t(w) - \xi_t(c)\big) = \big(v - c\big)^{\mathsf{T}} \big(w - c\big) \,. \tag{2.3}$$

Lemma 2.1. *A mapping that preserves angles and distances is an affine map of the form*

$$\xi_t(x) = R_t x + b_t \,, \tag{2.4}$$

with R_t an orthogonal matrix and b_t some vector.

Proof. Equality (2.3) can be differentiated in the directions v and w to give

$$\frac{\partial}{\partial w}\frac{\partial}{\partial v}\Big[\big(\xi_t(v)-\xi_t(c)\big)^{\mathsf{T}}\big(\xi_t(w)-\xi_t(c)\big)\Big] = \frac{\partial}{\partial w}\frac{\partial}{\partial v}\Big[\big(v-c\big)^{\mathsf{T}}\big(w-c\big)\Big] \tag{2.5}$$

$$\frac{\partial \xi_t(v)}{\partial v}^{\mathsf{T}}\frac{\partial \xi_t(w)}{\partial w} = I \,, \tag{2.6}$$

for all v and w. The right hand is the identity matrix. Thus the total derivative $\partial_x \xi_t(x)$ is an orthogonal matrix R_t, independent of x. Integrating in some direction x with integration constant b_t yields (2.4) □

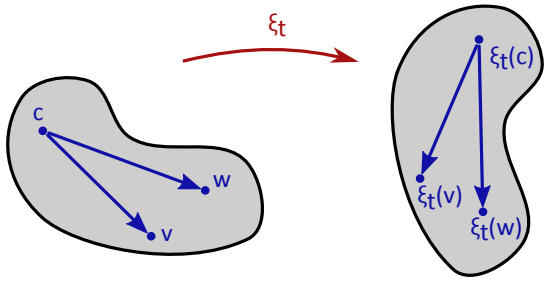


Figure 2.1: Rigid body motion, a visual interpretation of equation (2.3)

Furthermore, $\xi_t(x)$ is continuous in t and at time zero it equals the identity since no movement has yet taken place. Therefore, we know that the determinant of R_t is equal to one, not minus one. Thus R_t is strictly a rotation. This kind of mapping is called a *direct isometry*. The group of all such mappings is called the *Euclidean motion group* or *rigid motion group* and is denoted by $\mathsf{SE}(n)$.

By *Lemma* 2.1, the movement from Σ_0 to Σ_t is completely characterised by the components of R_t and b_t. This leads to the notion of generalised coordinates for rigid bodies. We can associate a body with a long column like

$$x = [\underbrace{s_1,\ldots,s_n}_{\text{position}},\underbrace{\theta_1,\ldots,\theta_k}_{\text{angle}}]^{\mathsf{T}} \,. \tag{2.7}$$

For the position vector s we typically use the position of the centre of mass. This choice is technically a convention, but it does simplify the equations because a freely moving body will rotate about its centre of mass. That means we can easily decouple the equations for translation and those for rotation. The vector θ, with the angular information, is of length $k = \frac{1}{2}(n^2-n)$, the number of coordinates needed to uniquely determine the rotation matrix R_t.

In three dimensions: The notion of rotation is not commonly encountered in spaces of over three dimensions. Fortunately, users of multibody simulations are rarely interested in such spaces. So henceforth in this thesis, we will restrict ourselves to $\mathbb{R}^3$. Then k equals 3 and angular components are typically referred to by the names of the Tait-Bryan angles: roll, pitch and yaw (ϕ,θ and ψ). They are defined as follows: Choose an arbitrary axis for the body through the centre of mass. Then yaw is its angle with the fixed xy-place; pitch with

the xz-plane and roll is the rotation of the body about this chosen axis. These three figures are sufficient to describe any configuration of a three dimensional body. However, they are discontinuous. More specifically, there exist a configurations for a body such that an arbitrarily small movement leads to a jump in the Tait-Bryan angles. This phenomenon is known as gimbal lock and makes Tait-Bryan angles unsuitable for applications where movement is involved.

The problem of gimbal lock is elegantly solved by regarding the three-dimensional space in which ψ,θ and ϕ lie as a spherical surface in four-dimensional space. Every point on this sphere can be indicated by $e = (e_1, e_2, e_3, e_4)$ with $\|e\| = 1$. The numbers $\{e_i\}_{i=1}^4$ are called Euler-Rodrigues parameters and they are obtained from the Tait-Bryan angles as follows

$$\begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \end{bmatrix} = \begin{bmatrix} \cos(\phi/2)\cos(\theta/2)\cos(\psi/2) + \sin(\phi/2)\sin(\theta/2)\sin(\psi/2) \\ \sin(\phi/2)\cos(\theta/2)\cos(\psi/2) - \cos(\phi/2)\sin(\theta/2)\sin(\psi/2) \\ \cos(\phi/2)\sin(\theta/2)\cos(\psi/2) + \sin(\phi/2)\cos(\theta/2)\sin(\psi/2) \\ \cos(\phi/2)\cos(\theta/2)\sin(\psi/2) - \sin(\phi/2)\sin(\theta/2)\cos(\psi/2) \end{bmatrix} = \begin{bmatrix} \cos(\chi/2) \\ u_1\sin(\chi/2) \\ u_2\sin(\chi/2) \\ u_3\sin(\chi/2) \end{bmatrix}, \tag{2.8}$$

with u and χ yet another description of rotation, namely decomposition in an axis of rotation u and an angle χ about that axis [13]. If the four parameters e_i are used instead of the three angles, then an additional equation is required to avoid an under-determined system. This additional equation is given by the fact that the Euler-parameters lie on a sphere, that is $\|e\| = 1$. This additional equation is a constraint equation that must be satisfied. It will be expanded below in *Section* 2.1.2.

It is interesting to remark that the addition of a fourth angle-like quantity is not solely a theoretical trick. A mechanical way to measure angles in aircraft and the likes thereof is by a system of gimbals. Indeed, adding a fourth gimbal and actively controlling it at a ninety degree angle from the second one avoids gimbal lock. See [14] for an amusing example from the 1963 Apollo mission.

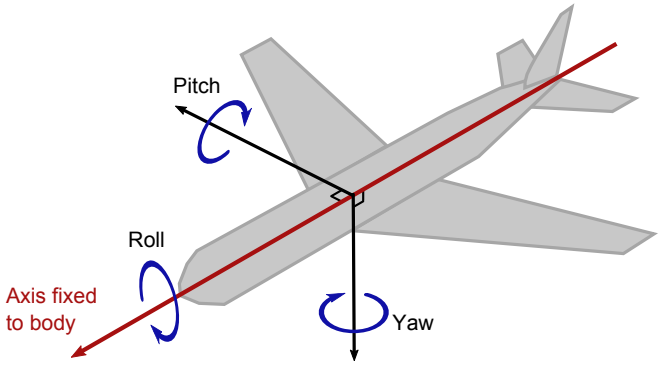


Figure 2.2: The Tait-Bryan angles for an aeroplane. The direction of flight is the obvious choice as a main axis. If the plane were flying up- or downwards (a pitch of $\pm 90°$) it would suffer from gimbal lock, manifested by the fact that then roll and yaw are interchangeable.
(This is a cosmetic adaptation of `https://en.wikipedia.org/wiki/File:Yaw_Axis_Corrected.svg`, licensed as CC-BY-SA-3.0)

We now have a convenient way to characterise a rigid body in $\mathbb{R}^3$: By the position of its centre of mass and the four Euler angles. We can use Newton's laws of motion to figure out how the body moves when subject to a certain force.

2.1.2 – Constrained dynamics and multibody systems

The derivation for constrained dynamics is given in the introductory literature. For our application, the examples of E.J. Haug[13] and [21] were followed. If the position of a body is described with its generalised

coordinates, we can formulate equations on these coordinates to describe the movement of the whole body. The basic equations of motion we use are Newton's second law for both translational and angular motion:

$$m\ddot{s} = f \tag{2.9}$$
$$\mu\ddot{e} = \tau\ , \tag{2.10}$$

with f an external force vector, τ the torque resulting from this force, m a scalar denoting the mass of the body and μ the 4×4 inertial tensor. Due to the similar form of the two equations, we can collect them into one vector equation

$$\begin{pmatrix} mI & \\ & \mu \end{pmatrix}\begin{pmatrix} \ddot{s} \\ \ddot{e} \end{pmatrix} = \begin{pmatrix} f \\ \tau \end{pmatrix} \quad \in \mathbb{R}^7\ . \tag{2.11}$$

This is the equation to describe the motion of a single body. To describe more bodies (say l of them) simultaneously they can be collected within a single vector equation (in $k = 7l$ dimensions). Moreover, we can now add the constraints on the bodies by requiring a second set of equations to be satisfied. This second set is given by a function $\Phi : \mathbb{R}^k \to \mathbb{R}^m$ which only acts on x. From the definition of the Euler parameters above, we already know one constraint equation that is in Φ, namely the normalisation of the Euler parameters: $\|e\| = 1$.

$$M\ddot{x} = Q \quad \in \mathbb{R}^k \tag{2.12}$$
$$\Phi(x) = \mathbf{0} \quad \in \mathbb{R}^m\ , \tag{2.13}$$

with Q_A a generalised load function that contains applied forces, torques and inertial forces. The additional equation is just algebraic rather than an differential equation. This means we are now considering a differential equation on a manifold given by $\mathscr{M} = \{x \in \mathbb{R}^k \mid \Phi(x) = 0\}$. A system like this is called a differential-algebraic equation (DAE). Part of the loading force Q is responsible for making x satisfy the constraint equation. This is an unknown internal reaction force that we want to distinguish from the external applied forces Q_A. The reaction force is only unknown to a certain extent. In fact, we know that its direction is perpendicular to the face of the manifold because of Newton's third law. The magnitude of the reaction forces will be collected in the unknown vector λ and then the system looks like (see *Appendix* B). The vector λ is a collection of Lagrange-multipliers, see also the paper by Karabulut [15].

$$M\ddot{x} + \Phi_x^\mathsf{T}\cdot\lambda = Q_\mathrm{A} \tag{2.14}$$
$$\Phi(x) = 0\ . \tag{2.15}$$

At this point we stress that the occurrence of internal forces results from a modelling assumption. We do not choose to 'manually add forces' so as to constrain the bodies to Φ, but rather we make use of the modelling assumption that the bodies are constraint to Φ and deduce what the internal force necessarily must look like [1].

2.1.3 – Preparing for numerical integration

The system in (2.14 - 2.15) is a complete description of multibody dynamics. It is, however, not yet suitable for integration with DASSL for two reasons: It is an index-three system and it is second order.

[1] This assumption does not hold anymore when the magnitude of internal forces (i.e. λ) gets so large that the joints break of. Then the bodies will stray away from the manifold.

Index reduction

The concept of index is not an easy one. There are different definitions: Differentiation-index, kronecker-index, perturbation-index and more. These all address different aspects of similar problem. The definition used here is the differentiation-index found in [2].

Definition 2.1. *The **differentiation-index** of a differential-algebraic system*

$$\boldsymbol{f}(x, \dot{x}, t) = 0 \tag{2.16}$$

is the smallest k such that

$$\frac{\partial^k}{\partial t^k}\boldsymbol{f}(x, \dot{x}, t) = 0 \tag{2.17}$$

is a system of ordinary differential equations.

It can be thought of as a measure of the dominance of the algebraic part in the systems. The general definition is the number of times one needs to differentiate a system so as to end up with a differential equation for every variable. For constrained dynamics the index is usually three. This is a problem since DASSL only allows for differential-algebraic systems of index one. Therefore it is key to change the system so as to decrease this index. Remark that the repeated differentiation in *Definition* 2.1 does reduces the index of the system. It is, however, not a suitable means to obtain a lower index system since it also removes valuable information. A more subtle approach is to differentiate the constraint equations once and to interchange the occurrence of λ with its derivative, $\dot{\lambda}$.

$$M\ddot{x} + \Phi_x^{\mathsf{T}}\dot{\lambda} = Q_{\mathsf{A}} \tag{2.18}$$
$$\Phi_y(x)\dot{x} + \Phi_t(x) = 0 \tag{2.19}$$
$$. \tag{2.20}$$

This is a system of index one. The loss of information in the original system is minimal but not negligible: where (2.15) fixed the relative position of the bodies, (2.19) fixes their relative speeds. Theoretically, this is not a problem given proper initial conditions, but in numerical practice this leads to a phenomenon called *drift*, in which the bodies slowly float away from each-other. This is remedied by actively monitoring the constraint equation $\Phi(x) = 0$ and stabilising if its residual exceeds a certain tolerance.

Order reduction

The integration order of the system is two. Most generic numerical integrators only solve equations of order one. This is easily solved by the well known trick of adding an additional equation for the derivative of x

$$\dot{x} = v \tag{2.21}$$
$$M\dot{v} + \Phi_x^{\mathsf{T}}\dot{\lambda} = Q_{\mathsf{A}} \tag{2.22}$$
$$\Phi_y(x)\dot{x} + \Phi_t(x) = 0\,, \tag{2.23}$$

in matrix form

$$\begin{bmatrix} I & 0 & 0 \\ 0 & M & \Phi_x^{\mathsf{T}} \\ \Phi_x & 0 & 0 \end{bmatrix} \begin{bmatrix} \dot{x} \\ \dot{v} \\ \dot{\lambda} \end{bmatrix} = \begin{bmatrix} v \\ Q_{\mathsf{A}} \\ -\Phi_t \end{bmatrix}\,, \tag{2.24}$$

and cast as a general nonlinear system

$$f(y,\dot{y},t) = \begin{bmatrix} \dot{x} - v \\ M\dot{v} + \Phi_x^{\mathsf{T}}\dot{\lambda} - Q_{\mathsf{A}} \\ \Phi_y v + \Phi_t \end{bmatrix} = 0\,. \tag{2.25}$$

with $y = \left[x, v, \dot{\lambda}\right]^{\mathsf{T}}$.

2.2 – Time integrators

2.2.1 – General description

A time integrator is an algorithm capable of approximating the solution to an evolution equation. In the most general cases, an integrator is capable of approximating the solution of a system like

$$F(y,\dot{y},t) = 0 \quad \text{for } t \in [0,T] \tag{2.26}$$

$$y(0) = y^0\,, \tag{2.27}$$

for example the system (2.25). The output is a list of time values $\{t_i\}$ and a finite set of points $\{y_i\}$ that aim to sample the exact solution $y^\star$ within a certain tolerance margin τ_n:

$$0 = t_0 < t_1 \ldots < t_N = T \tag{2.28}$$

$$y^n = y^\star(t_n) + \tau_n\,. \tag{2.29}$$

Note that we can not expect an absolute tolerance in general. Since future sample points are calculated from the current ones, the error accumulates over time.

There are many different integrators available. On the highest level they can be classified as either a Runge-Kutta method or a multi-step method. DASSL belongs to the latter category. A nice introduction is provided by Hairer, Nørsett & Wanner in [11].

2.2.2 – Backward difference formulae

A time integrator yields a discrete set of points. Therefore the notion of derivative is non-existent and we need an approximation for it given this set of points. The method that is used by DASSL is an implicit backward difference on k points. That is, the derivative is approximated by

$$\dot{y}^n \approx \alpha y^n + c_1 y^{n-1} + \ldots c_k y^{n-k} \tag{2.30}$$

$$= \alpha y^n + \beta\,. \tag{2.31}$$

The factors $\alpha, c_1, \ldots, c_k$ are chosen according to the step sizes in-between the various points. Since the scheme is implicit, the values for y^n and possibly for the time step h are yet unknown. This is stressed by setting them apart. For example, the traditional backward Euler difference is given by $k = 1$, $\alpha = 1/h$ and $c_1 = 1/h$. In general, α is always some constant depending on k divided by the time step h. This is important later on when we discuss the scaling of the matrix.

With the approximation of (2.31), the system (2.25) becomes

$$f(y,\alpha y + \beta_y, t) = \begin{bmatrix} \alpha x + \beta_x - v \\ M(\alpha v + \beta_v) + \Phi_x^{\mathsf{T}}\dot{\lambda} - Q_{\mathsf{A}} \\ \Phi_y v + \Phi_t \end{bmatrix} = 0\,. \tag{2.32}$$

2.2.3 – DASSL

A very generic integrator is DASSL (Differential algebraic system solver). The development and maintenance of this algorithm is led by L.R. Petzold who released it into the public domain in 1982 [17]. A full description of the algorithm is available in her book [2]. This section contains merely an overview of the different steps it takes. Moreover, we will not regard the initial time-steps.

The following steps are looped during the execution. Below this list is a more detailed explanation.

1. Choose **order** k and **time-step size** h
2. Set $t_n = t_{n-1} + h$
3. **Predictor:** given $\{y^{n-1}, y^{n-2}, \ldots, y^{n-k}\}$, extrapolate to y_0^n
4. **Corrector:** Try to solve for y^n in $f(y^n, \alpha y^n + \beta, t_n) = 0$ with Newton iterations
   If failure: Go to step 1
5. **Increment** n
6. **Go to** step 1

The choice of the order and the time-step size in *Step* 1 depends on how well the previous time step performed. When there is fast convergence, the time-step is increased and for poor or no convergence the time-step is decreased. In *Step* 2 the time is increased. Note that the value for n stays unchanged until *Step* 5. *Step* 3 calculates the predictor y_0^n. This is an extrapolation from a polynomial on the last k points. In *Step* 4 the predictor is used as an initial guess for the Newton algorithm to solve the non-linear system. If this fails, the flow is redirected to *Step* 1. If the Newton iteration succeeded, then the step number is incremented in *Step* 5 and the new point y^n becomes a part of the output.

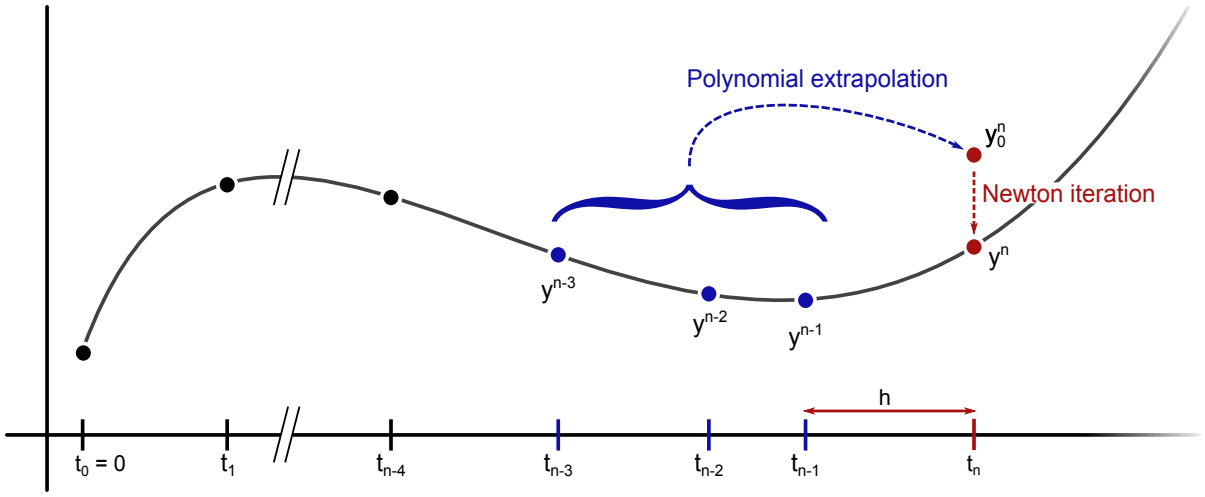


Figure 2.3: A graphical explanation of DASSL. In this example k is equal to three

Newton iterations in *Step* 4

By far most of the computational resources are spent in *Step* 4. Therefore we will concentrate on this step in the remainder of the thesis. That is not to say that there is nothing to gain by reviewing the other steps. See the recommendations for further work in *Section* 6.3.

To perform Newton iterations, one needs the Jacobian of (2.32)

$$\partial_y f(y, \alpha y + \beta, t) = G = = \begin{pmatrix} \alpha I & -I & 0 \\ \alpha \partial_x \left[M\dot{v} + \Phi_x^\mathsf{T} \dot{\lambda} - Q_\mathrm{A} \right] & \partial_v Q_\mathrm{A} - \alpha \left[M - \partial_{\dot{v}} Q_\mathrm{A} \right] & \Phi_x^\mathsf{T} \\ \partial_x (\Phi_x v - \Phi_t) & \Phi_x & 0 \end{pmatrix}. \tag{2.33}$$

This Jacobian is scaled on some of the rows with a factor α^{-1}. This improves the scale and condition of the matrix and allows for faster LU-factorisation. In this thesis we will not consider the scaling of the matrix, but a current outlook on it is given in the paper by Botasso e.a. [**?**]. We do, however call attention to the fact that this scaling factor α^{-1} is dependent on the time-step size. This will return in *Section* 5.2.

The Newton method that DASSL uses is in fact an approximate Newton method called *the simplified Newton method*. The simplification entails that the Jacobian is not recalculated every step but rather kept constant for a number of steps. This is efficient since the LU-decomposition that is needed to solve for the Jacobian can then be kept throughout. Details about this and other approximate methods are explained in *Chapter* 3. We will define the concept of *refresh cycle*. These refresh cycles are the focus of this thesis and can be viewed as the unit cell on which the research is based.

Definition 2.2. *A **refresh cycle** in* DASSL *(or similar) is a subsequent series of time-steps for which the Jacobian is kept constant. A **(Jacobian) refresh** is the act of re-assembling and refactoring the current Jacobian.*

2.2.4 – Computational effort and simulation rate

For any numerical application we strive to minimise the effort the process takes. To quantify this, the effort κ is usually expressed in the number of flops (floating point operations) or more practically in CPU time or wall-clock time. The desired goal of a time integrator is to simulate a model as it advances trough time τ as a result of the computational effort. Both of these quantities are non-decreasing, so they are suitable to be plotted against one-another. We call this the time/effort-plot.

A convenient notion is the *simulation rate* of an integration process. We define this as the simulated time divided by the effort it took to get there.

Definition 2.3. *The global (or average) **(simulation) rate** of a time integrator is defined as*

$$\sigma := \frac{\tau_{end}}{\kappa_{total}}. \tag{2.34}$$

the local rate of a time integrator on the interval $T = [\tau_1, \tau_2]$ is defined as

$$\sigma_T := \frac{\tau_2 - \tau_1}{\kappa_2 - \kappa_1}. \tag{2.35}$$

The above definition of local rate would suggest to take the limit $\tau_2 \to \tau_1$ to obtain an instantaneous rate as the effort-derivative of time. However, this limit is not defined because the sample points are discrete.

In this thesis, no absolute quantitative research on the computational effort is performed. Algorithms will be compared to one-another. In other words, we set the unit-time and unit-effort as those for the current algorithm (in *Section* 4.2). That way, the efficiency of changes and additions is calculated as a speed gain on the

current algorithm. It is, however, important to note that a time-integrator is well capable of keeping track of its own computational effort. This means that an integrator can be designed to adjust its parameters accordingly (see *Section* 4.3)

See *Figure* 2.4 for a schematic time/effort-plot of DASSL.

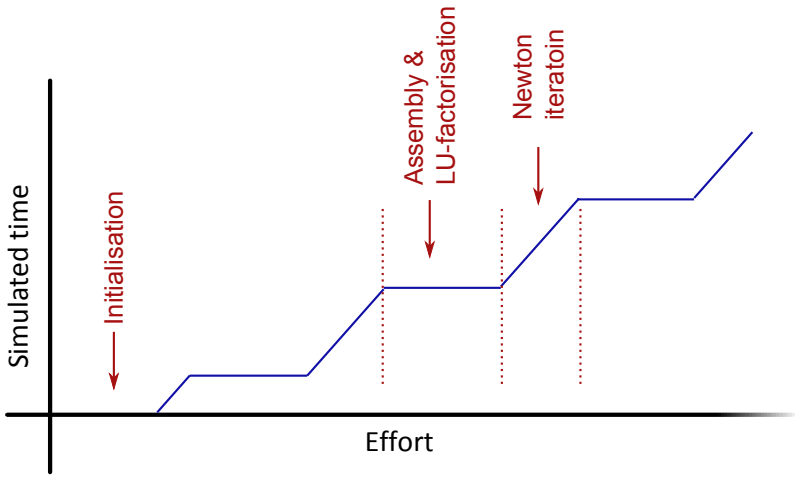


Figure 2.4: A schematic view of how the simulated time progresses as a function of the computational effort. The factorisation and integration are alternated. One refresh cycle consists of the factorisation and the subsequent time integration together. The drawing is not to scale.

Chapter 3 – Newton methods

"Can't repeat the past? . . . Why of course you can!"

F.S. FITZGERALD
THE GREAT GATSBY (1922)

Overview

This chapter defines our framework for approximate Newton methods after a short introduction of the exact Newton method. The framework consists of methods in which the solution of the Jacobian system is approximated by approximating the matrix, simplifying the solving algorithm or both. From the framework follow three approximate methods: Simplified (as used currently), simplified with dropped-LU (SDLU, defined below) and truncated Krylov iterations.

The convergence speed (counted in Newton steps) of such approximated methods is given by the contraction factor of the process. This contraction is estimated by the spectral radius of the approximate Newton-function when applied to a linear problem.

Finally numerical results are provided. From this it follows that the contraction rates of SDLU and Krylov are respectively worse and better. Krylov shows exceptional performance for the irregular model.

3.1 – The exact Newton method

The Newton-Raphson method, or just Newton method, is a means to find a root $y^\star$ of a differentiable vector-valued function $f : \mathbb{R}^n \to \mathbb{R}^n$. It is based on the expectation that a function is well enough approximated by its linearisation and thus that the root of the linearisation is close to the root of the function itself. Repeating the act of finding this root gives rise to an iterative process of the following form

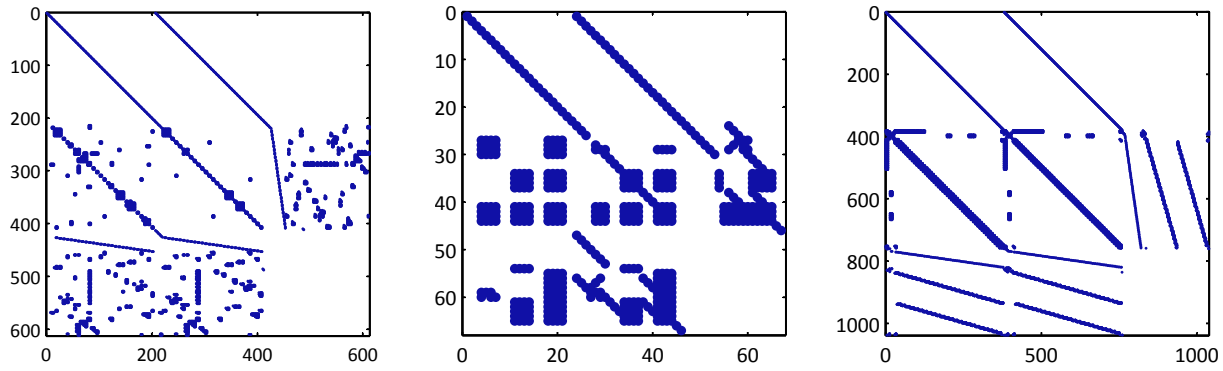
$$y_{k+1} = y_k - G(y_k)^{-1} f(y_k)\,, \tag{3.1}$$

with $G(y) := \partial_y f(y)$ the Jacobian in point y. See *Figure* 3.1 for sparsity plots of the Jacobians of the three test models.

For sufficiently smooth functions, this process is guaranteed to find a root given that the initial guess is close enough. That is, there exists an open set around $y^\star$ such that a Newton process starting from within this set is guaranteed to converge to $y^\star$. Moreover, the convergence is quadratic. That is

$$\|y_{k+1} - y^\star\| \leq C\|y_k - y^\star\|^2\,. \tag{3.2}$$

In practical applications, one never makes use of the inverse G^{-1}. Instead, we solve $Gz = f$ by two backward



(a) ATV
(612 eqns, 8216 nonzeros)

(b) Gimbal
(67 eqns, 522 nonzeros)

(c) Chain
(1038 eqns, 11309 nonzeros)

Figure 3.1: Sparsity plots of the Jacobian for the three test models.

substitutions via the LU-decomposition $G = LU$.

$$\text{factorise: } G = LU \tag{3.3}$$

$$\text{solve: } Ls = f \tag{3.4}$$

$$\text{solve: } Uz = s\,. \tag{3.5}$$

Throughout the thesis the notation G^{-1} will be used as shorthand for these backward substitutions.

3.2 – Approximate Newton methods

The Newton method as presented above has two major downsides: one, dependence on the quality of the initial guess, and two, numerical effort. The first of these is not regarded in this thesis. We assume the initial guess to be close enough to the root for the Newton method to achieve quadratic convergence. This assumption is justified by the fact that DASSL decreases the time step if the corrector loop fails to converge. The second downside stems from the fact that in in full Newton the Jacobian matrix G needs to be assembled and factorised at every step. This is usually a prohibitive computational expense. This leads to a class of approximate Newton methods that aim to mitigate this expense. A large number of approximate Newton methods is provided by Deuflhard in [6].

We will consider iteration functions in this general form

$$y_{k+1} = \mu(y_k) = y_k - \mathscr{G}^{-1}(y_k; f)\,. \tag{3.6}$$

Where $\mathscr{G}^{-1}$ is a general operator acting on y_k that is dependent on the function f. Given the Banach contraction theorem (see *Appendix* A), we have a lot of leeway in choosing this mapping. It is sufficient if the fixed point of μ is equal to $y^\star$ and if μ is a contraction.

1. $\mathscr{G}^{-1}(y^\star; f) = 0$
2. $\rho\left(\partial_y \mu(y)\right) < 1$

The second of these is troublesome. For differentiable functions it is easy to check whether or not they are a contraction, but in general we do not know enough about the operator $\mathscr{G}^{-1}$, or even about the function f, to ascertain facts about the derivative even if it exists. In *Section* 3.3 we sidestep this issue by a linear approximation which still offers a consistent estimate of the convergence speed.

In this thesis we look for operators $\mathcal{G}^{-1}(y;f)$ that aim to approximate $G^{-1}f$. Heuristically, one can think of $\mathcal{G}^{-1}$ as an algorithm. In the next section, we will consider two distinct approaches in approximating the solution of such a system.

Lastly we remark that the convergence speed is no longer quadratic but linear for these approximate Newton methods (see Deuflhard [6]). That is

$$\|y_{k+1} - y^\star\| = C\|y_{k+1} - y^\star\| \,. \tag{3.7}$$

Nonetheless, the solution $y^\star$ is found by an approximate method equally well as by the exact Newton method. The fact that the system's solutions are being approximated does not affect the accuracy of the final solution, just the path towards it (see *Figure* 3.2).

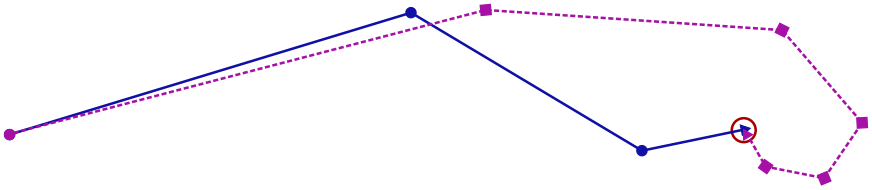


Figure 3.2: Two Newton paths to the same solution. The blue path takes three iterations, the purple one takes six. The purple method is faster if the speed gain on the iterations is more than a factor two compared to the blue method. The achieved tolerance (as a red circle) is independent of the path that was taken.

3.2.1 – Approximating a linear system

The operator $\mathcal{G}^{-1}$ is a means to approximate the solution of $Gy = f$. Depending on the problem, one can think of many ways to make such an approximate solution. We will identify methods based on two options.

1. **Matrix approximation:** Solve for an approximation of G rather than the correct one.
   We will denote a matrix approximation by an over-set tilde: $\tilde{G}$. In this case the operator is simply a backward substitution: $\mathcal{G}^{-1} = \tilde{G}^{-1}$

2. **Solving approximation:** Simplify the solving method.

We shall refer to the errors that arise from these approximation as the *matrix error* and *solving error* respectively. The methods can be used independently so there are four combinations possible. They can be conveniently laid out like in *Table* 3.1. In discussing the various Newton approximations below, we will refer to their place in this table.

Either option of the two contains a few methods. Before we give a more detailed definitions and test results, for both options short lists are given that describe the main ideas for each method.

Matrix approximation

- Use a dated matrix:
  This method is suitable for applications in which a series of slowly changing systems is solved. This is an obvious feature of the Newton method.

- Structural simplification:
  In some cases, one can approximate a matrix well with a structural simplification. For example by using

	Exact matrix	Approx. matrix
Exact solution	**1**	**2**
Approx. solution	**3**	**4**

Table 3.1: Four options to approximate the solution of a linear system. This table appears more elaborate in *Figure* 3.4.

just a diagonal matrix (which means that the system is nearly decoupled already) or by averaging out small blocks to single elements.

- Low rank approximation:
  When the distribution of eigenvalues (or singular values) of a matrix is skew, then most of its behaviour is driven by the largest eigenvalues.

In our application, by far the best matrix approximation has proved to be the usage of a dated matrix. A structural simplification, like lumping or domain decomposition, is usually suitable for discretisation-type problems. For general multibody applications, no such simplifications were found. The low rank approximation was experimented with but then quickly discarded due to the fact that over 30% of the singular values were needed to give an approximation that can compete with the dated matrix. Besides the relatively poor quality of the other matrix approximation is the fact that the computational overhead for them is far bigger than for the dated matrix. Therefore we will consider no other option than this.

Solution approximation

- A truncated iterative solver:
  Iterative solvers repeat a process that improves a certain initial guess at every step. If the method is monotone we are assured of an approximate solution that is at least as good as the initial guess after a finite, fixed number of steps.

- Inexact decomposition:
  Instead of performing a full LU-decomposition to solve the system, an inaccurate one can be used. By this we mean obtaining two matrices $\tilde{L}$ and $\tilde{U}$ such that $G \approx \tilde{L}\tilde{U}$. The benefit of this approach is found in one or both of the following reasons: one, the process of factoring might be cheaper, and two, the process of the backward substitution might be cheaper.
  The most well-known one is the incomplete-LU decomposition (ILU). An alternative, described below, is what we will call the dropped-LU (DLU) decomposition.

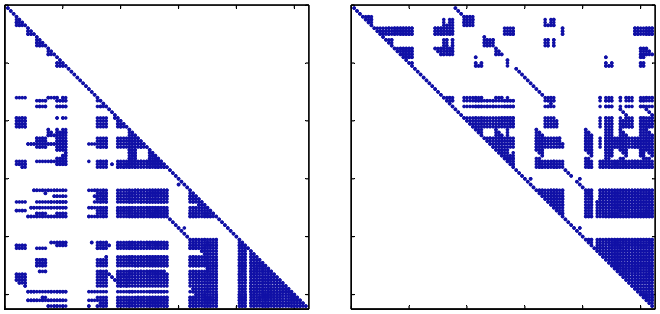
Of the solution approximations we will first consider the truncated iterative solver. For generic linear systems, the prime choice for an iterative solver is a Krylov-subspace method. An introduction can be found in the standard reference by Y. Saad [19].

Definition 3.1. ***Dropping*** *is the act of ignoring small, off-diagonal elements in a matrix. For a given drop-parameter* $\delta \in [0,1]$*, every matrix slot* $\{i\,j\}$ *gets ignored if and only if*

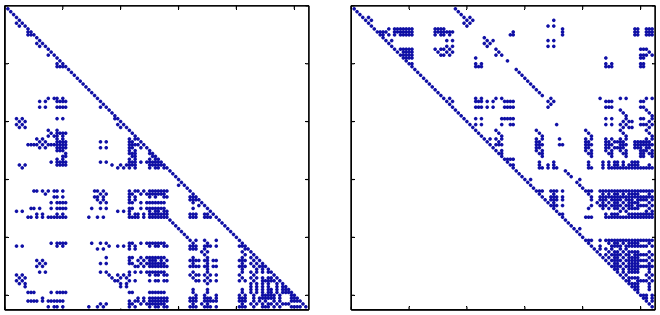
$$\left[|A_{ij}| < \delta \max_{1\le k\le n} |A_{ik}|\right] \;\wedge\; \left[i \neq j\right] . \tag{3.8}$$

The choice to compare with the largest element in a row as opposed to a column is arbitrary. However, in applications with more specific matrix characteristics (like symmetry or diagonal dominance) it is worth it to make a more systematic choice on which elements to drop.

When we perform dropping on the triangular matrices L and U from the exact LU-decomposition to obtain $\tilde{L}_\delta$ and $\tilde{U}_\delta$. This is called the dropped-LU factorisation (DLU). The benefit of dropping is the fact that fewer nonzero values are used for the backward substitution. Hence the solver will perform faster. The computational cost of the backward substitution is linear in the number of nonzero values, so if a fraction ϕ of the values remains, the substitution will be $1/\phi$ times faster. See *Figure* 3.3 for nonzero-plots of a typical triangular matrix and the dropped version of it.



(a) L and U



(b) L_δ and U_δ, about one third of the original non-zeros are maintained.

Figure 3.3: Sparsity plots of L, U and L_δ, U_δ.

Note on implementation: We deliberately use the word ignore. This reflects the implementation of using a boolean list alongside the row/column-pointers that state whether or not this value is contained within the dropped variant of the matrix. Dropping is also known as ILU-threshold or ILUT (see Saad [20]). In this thesis, however, we choose to use the terminology of dropping and ignoring to stress that the original LU-factors are not discarded. Rather we see the matrices $\tilde{L}_\delta$ and $\tilde{U}_\delta$ as a byproduct of creating L and U.

3.2.2 – Approximate Newton methods

Simplified Newton

Approximation nature	A dated Jacobian G_0 and exact LU
Justification	Slow change in the Jacobian
Computational effort per step	Two major backward substitutions
Overhead computational effort	Occasional LU-factorisations
Parameters	*none*

This method is the one currently used in DASSL. It is a matrix approximation with $\tilde{G} = G_0 = LU$

Simplified Newton with dropped LU (SDLU)

Approximation nature	A dated Jacobian G_0 and DLU
Justification	Slow change in the Jacobian A very skew distribution of values in L and U
Computational effort per step	Two minor backward substitutions
Overhead computational effort	Occasional LU-factorisations and comparing row elements in the factorisation
Parameters	δ, the drop tolerance

Instead of using the exact LU-decomposition, we use the dropped version DLU. This will introduce extra inaccuracies in the Newton path, but the steps are calculated faster. It is a matrix approximation with $\tilde{G} = L_\delta U_\delta$

Approximate Newton with a truncated Krylov solver

Approximation nature	An exact Jacobian and a truncated Krylov solver
Justification	The availability of an excellent preconditioner and initial guess
Computational effort per step	Assembling the Jacobian and performing a fixed number of Krylov iterations
Overhead computational effort	Refreshing the preconditioner
Parameters	n, the number of iteration steps

As opposed to the two methods above, this is a solution approximation. In this discussion the nature of the Krylov solver is intentionally kept generic but details about different Krylov subspace methods are found in [19]. A typical choice is GMRES (which is used for the numerical results in this thesis), but for specific Jacobian matrices there might be better options. We will denote the action of the truncated Krylov solver as $\mathcal{K}_n^{-1}$, with the subscript n indicating the number of Krylov steps. One might want to consider restarting the iteration, but as it turns out later a single iteration is already enough so restarting is not needed.

As stressed by [19], Krylov subspace methods are incredibly dependent of a suitable preconditioner and, to a lesser extent, a suitable initial guess. A preconditioner P is a matrix that resembles G in some way and that is easy to invert. Note that these requirements are contradictory. Now, the main idea, leading to the three-phase cycle in *Chapter* 4, is the insight that an old Jacobian might very well be a fine preconditioner even if it

is no longer suitable as a solver. It clearly resembles G since it is used successfully as a substitute of G in the simplified method and it is easily invertible if we have the LU-factors calculated beforehand.

A potential downside of Krylov methods is the fact that the matrix needs to be assembled at every step. Even though it is not factorised, the assembly might still take up considerable time that needs to be accounted for in the analysis of the speed gain. In our application, however, the matrix assembly takes negligible time. This is due to the fact that the Jacobian is assembled from differentiation rules in a look-up table for which no calculation is required.

Finally, we refer to the integrator DASPK as introduced by Brown, Hindmarsh and Petzold [3]. This is an integrator like DASSL that employs Krylov methods for solving the system. It is not a truncated Krylov solver, but can nonetheless offer a great deal of background on how to implement a truncated solver within DASSL.

Map of Newton methods

The succinct schematic overview of *Table* 3.1 can be expanded to include the matrix- and solving-error. See *Figure* 3.4 for a more detailed schematic of the various Newton methods with their place in the approximation landscape.

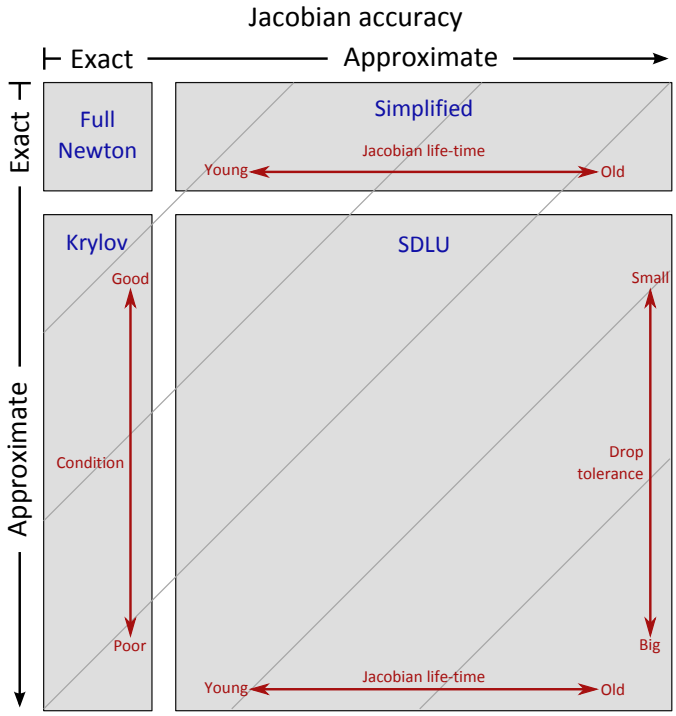


Figure 3.4: The four Newton methods indicated in the approximation landscape. Full Newton approximates neither the matrix nor the solution. The other three methods approximate one or both. The degree of approximation is controlled by the various parameters for the methods (indicated by the small axes within the blocks). Lastly, the total error (matrix plus solution) is indicated by the diagonals.

3.3 – Convergence speed

To estimate the convergence speed, we will use the contraction factor of the iterative process. The contraction factor κ is the rate at which the distance between successive iterates is expected to decline (or the distance to the solution). For approximate Newton methods we have linear convergence, so the contraction looks like

$$\|y_{n+1} - y_n\| \leq \kappa \|y_n - y_{n-1}\| \quad \forall_n \,, \tag{3.9}$$

with $\kappa \in (0,1)$. See also *Appendix* A

3.3.1 – Estimates for the contraction factor

We consider the following two estimates for the contraction factor:

1. Theoretical estimate: The spectral radius of $I - \mathscr{G}^{-1}G$.
2. Experimental results: The rate at which $\|y_{k+1} - y_k\|$ decreases.

The first one, the spectral radius, is proposed by Brenan e.a. [2]. We shall denote the spectral radius of a matrix A with $\rho(A)$. The term $\mathscr{G}^{-1}G$ is a matrix that is assembled by applying $\mathscr{G}^{-1}$ to the columns of G. This figure is the contraction factor for a linear problem if the Newton method uses the approximate solution $\mathscr{G}^{-1}f$ instead of $G^{-1}f$. Hence it is a sensible choice to use this method to compare the approximate Newton process to the exact process. Remark, however, that the function $x \to \rho\left(I - x^{-1}G\right)$ is *not* a matrix norm. We discard the theoretical advantages of using a proper norm in favour of the practical viewpoint that we can readily equate this spectral radius with the expected convergence speed of the Newton-process.

This spectral radius estimate for the true contraction factor κ is obtained from the derivative of the iterating function as follows

$$y_{k+1} = y_k - \mathscr{G}^{-1} f(y_k) = \mu(y_k) \tag{3.10}$$

$$\Rightarrow \kappa \leq \rho\left(\frac{\partial \mu(y)}{\partial y}\right), \tag{3.11}$$

then

$$\kappa \leq \rho\left(I - \mathscr{G}^{-1}G - \frac{\partial \mathscr{G}^{-1}}{\partial y} f\right). \tag{3.12}$$

Now $\frac{\partial \mathscr{G}^{-1}}{\partial y}$ is a troubling term. It is a Hessian-like tensor (of size $n \times n \times n$) with the second order effects. Indeed, the exact version (G^{-1} instead of $\mathscr{G}^{-1}$) relates to the Hessian $\underline{H}$ as follows:

$$\frac{\partial G^{-1}}{\partial y} = -G^{-1}\frac{\partial G}{\partial y}G^{-1} \tag{3.13}$$

$$= -G^{-1}\underline{H}G^{-1}. \tag{3.14}$$

This Hessian is definitely not readily available like the Jacobian. Moreover, for various choices of $\mathscr{G}^{-1}$, we cannot even be sure that this derivative exists. This is the reason that (like Brenan e.a. [2]) we approximate by using a linear version of the problem. That means that $\underline{H} \equiv 0$ and we choose $\partial_y \mathscr{G}^{-1} \equiv 0$ as well. Now, we finally get the spectral radius estimate as

$$\rho\left(I - \mathscr{G}^{-1}G\right). \tag{3.15}$$

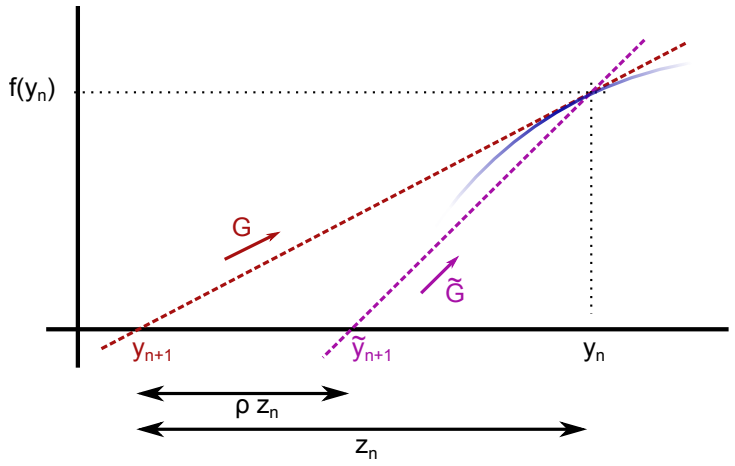


Figure 3.5: A geometrical interpretation of the meaning of the spectral radius ρ. The function (indicated locally in blue) has a gradient is equal to G, approximated by $\tilde{G}$. The distance between $\|y_{n+1} - \tilde{y}_{n+1}\|$ is at most ρ times $\|y_n - y_{n+1}\| = \|z_n\|$.

A geometric interpretation in one dimension of this spectral radius is given in *Figure* 3.5

The second of the two estimates mentioned above, the experimental value, follows from (3.9). It is calculated in DASSL as follows

$$r = \left(\frac{\|y_{i+1} - y_i\|}{\|y_1 - y_0\|} \right)^{1/m} . \tag{3.16}$$

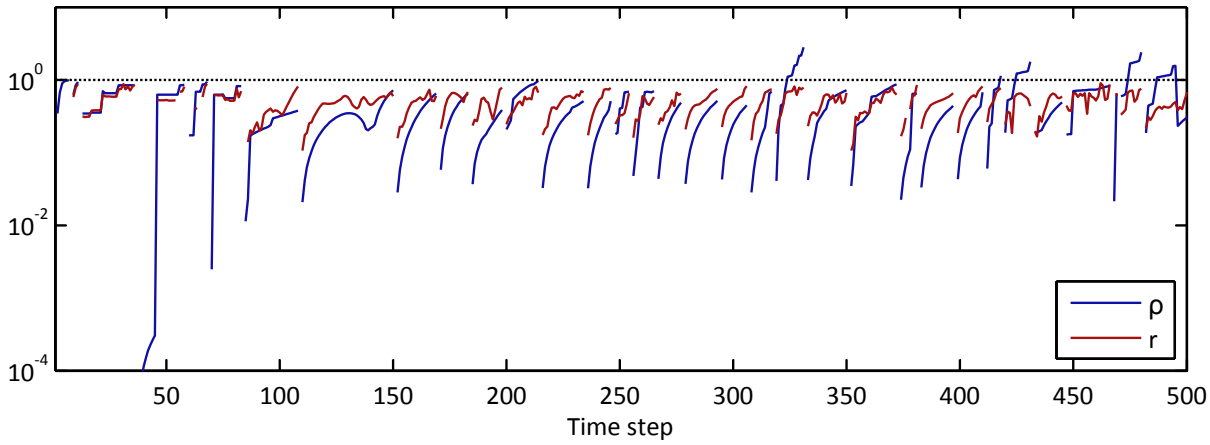
This figure is purely experimental and cheaply computed during time integration. We can, however, not use it as a comparison base since the algorithm needs to be implemented to obtain it.

3.3.2 – Numerical results

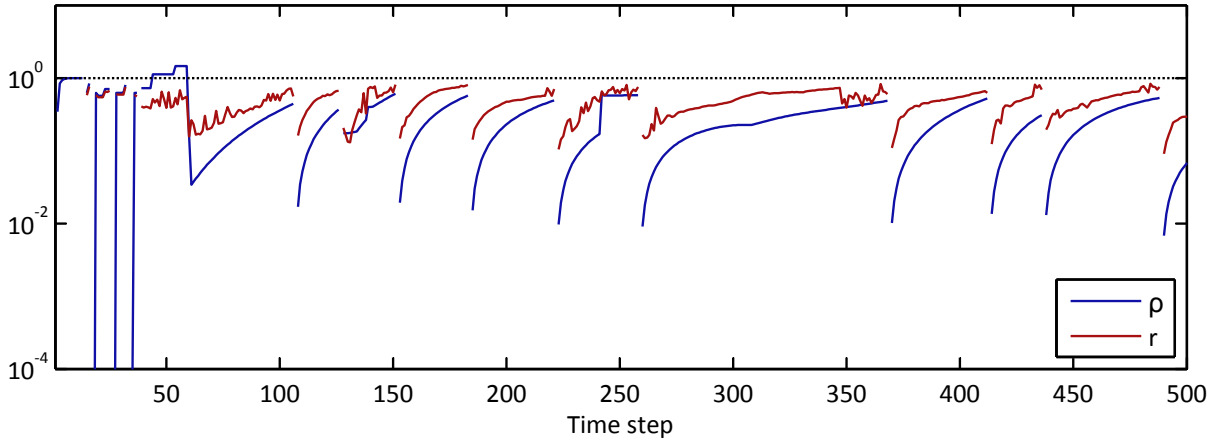
First, the two approximations for the contraction rate are compared. That is, the spectral radius ρ and the experimental value r. The plots are shown in *Figure* 3.6 for the three test models. Note that the spectral radius is an underestimation of the spectral radius estimate, save for some exceptions. The plots show individual refresh cycles by virtue of the contraction factor (the error, if you will) starting out very small and then slowly rising. DASSL rejects a Jacobian once the experimental contraction rate exceeds the hard-coded threshold 0.9. For the chain model both the spectral radius and the experimental result are very chaotic. Only 150 time steps are plotted[1] but already many refreshes take place. The error is not nearly as monotone as for the smoother models. Lastly note the initialisation phase in the first 60-odd time steps.

Now we will compare SDLU with the simplified method. As mentioned, due to the lack of implementation we will only compare the spectral radius ρ and no longer the experimental value r. The plots for SDLU with various drop tolerances are shown in *Figure* 3.7. The error of SDLU is indeed higher than the error of the simplified method (in dashed blue. LU equals SDLU with $\delta = 1$). Again, the chain model shows unexpected results. For example: at the refresh cycle starting at time step 60 the smaller drop tolerance has a larger error. Finally, note that the drop tolerance is chosen such that the error of SDLU is roughly similar on all models. To accomplish this, wildly varying values had to be chosen for the different models. As of yet, the reason for this is not found

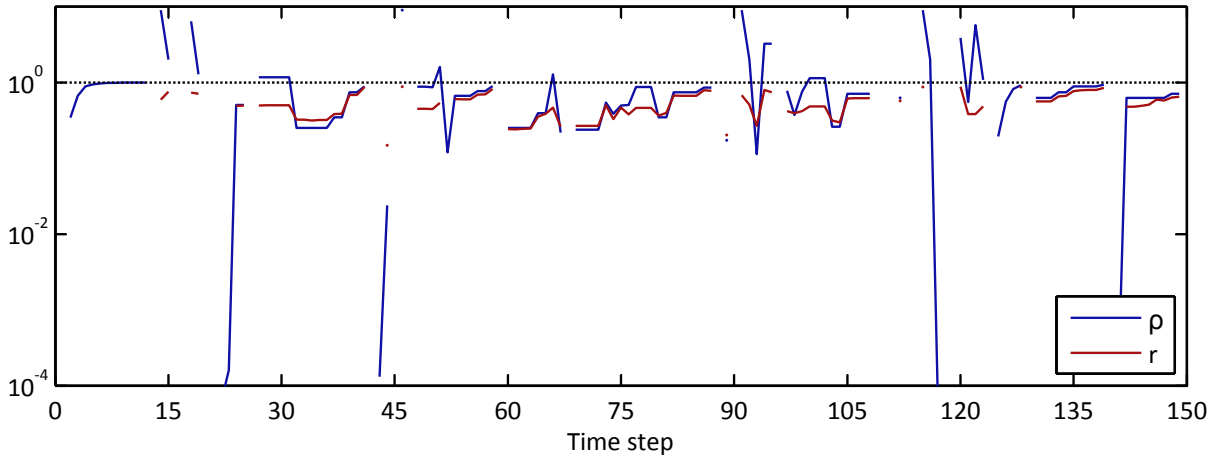
[1] due to computational restrictions, not more time steps were analysed



(a) ATV



(b) Gimbal



(c) Chain

Figure 3.6: The spectral radius and the experimental contraction for the three models respectively.

since no research has been performed to understand the characteristics of the matrices L and U for various models.

Finally, the Krylov iterations are compared with the simplified method. In this project we have chosen to use the Krylov method GMRES as implemented in Matlab[2]. This is a generic solver and it is a sensible choice for a preliminary analysis of the usefulness of Krylov methods. For a single Krylov iteration of GMRES the spectral radius, $\rho\left(I-\mathscr{K}_n^{-1}G\right)$, is plotted in *Figure* 3.8. It is once more stressed that the matrix $\mathscr{K}_n^{-1}G$ is obtained by applying $\mathscr{K}_n^{-1}$ to each of the columns of G. As expected the error of the Krylov iteration is lower than the error of the original simplified method. It is very noteworthy that the reduction in error is more profound for the chain model than for the other two models. To understand this, we refer back to the nature of the simplified method, versus the nature of the Krylov approximation (see *Figure* 3.4): The simplified method has no recent information whereas the Krylov approximation does use the most recent matrix. Apparently for the fast-changing model of the chain, the more recent data in the Krylov method gives a great advantage over the simplified method.

A single Krylov iteration can be regarded as an update of the solution of the dated system along the one-dimensional Krylov space that is spanned by the residual of the current system.

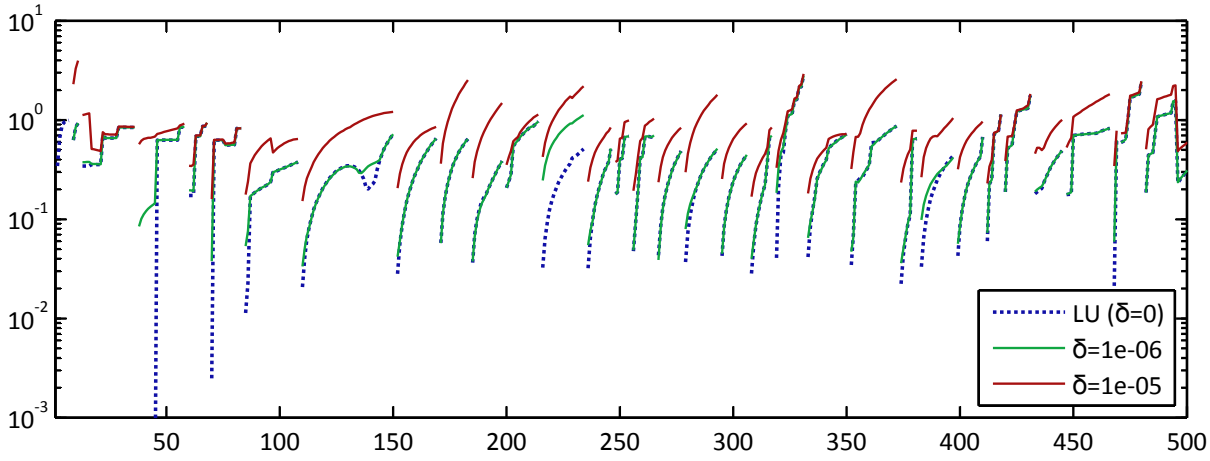
3.4 – Observations and conclusions

We have investigated three approximate Newton methods that fall within the approximation framework. The simplified method with dropped-LU can be seen as a quick but inaccurate variant of the regular simplified method and the truncated Krylov method, on the other hand, is a slower method that incorporates recent information from the Jacobian as opposed to the dated information in the other two methods.

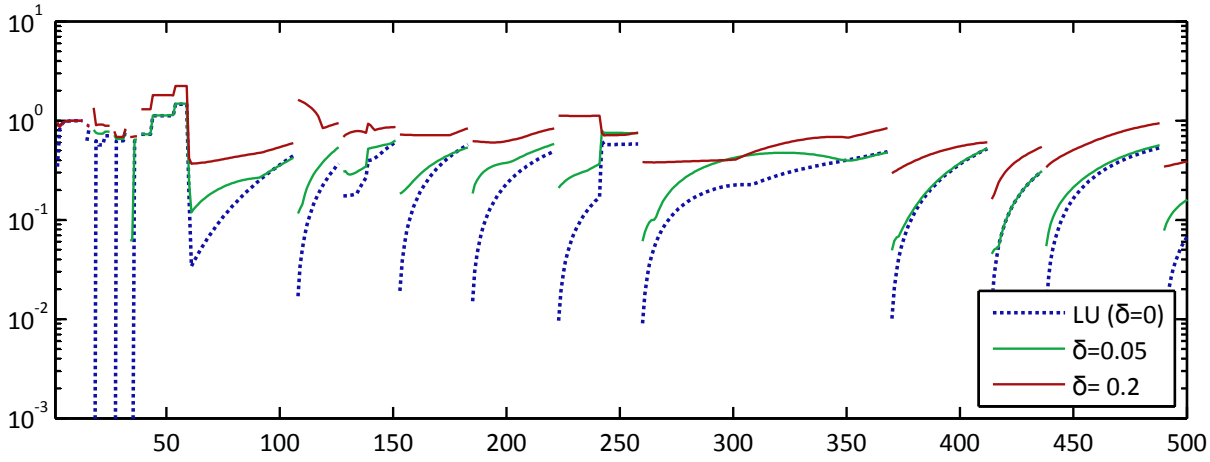
Choosing the drop-parameter for SDLU is somewhat of a conundrum. Its value seems highly model dependent and can vary about nine orders of magnitude. For now, sensible values are found via trial and error.

Krylov is a slower but more accurate method. The results for single iteration with the dated Jacobian as preconditioner already show good results. It keeps on performing well after the original cycle ends. Krylov shows great potential for models that change quickly due to the fact that they are not very smooth. The original simplified method needs to use very small refresh cycles (sometimes of just a single step) to accommodate to the rapidly changing Jacobian. The Krylov method incorporates more of the new information and outperforms the simplified method by about two orders of magnitude.

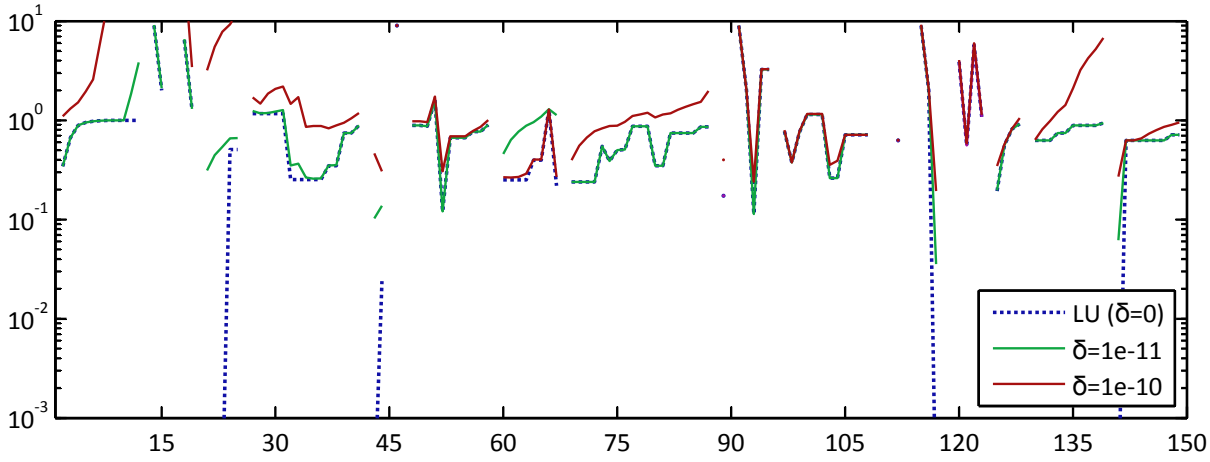
[2] the function `gmres` in Matlab R2012b



(a) ATV

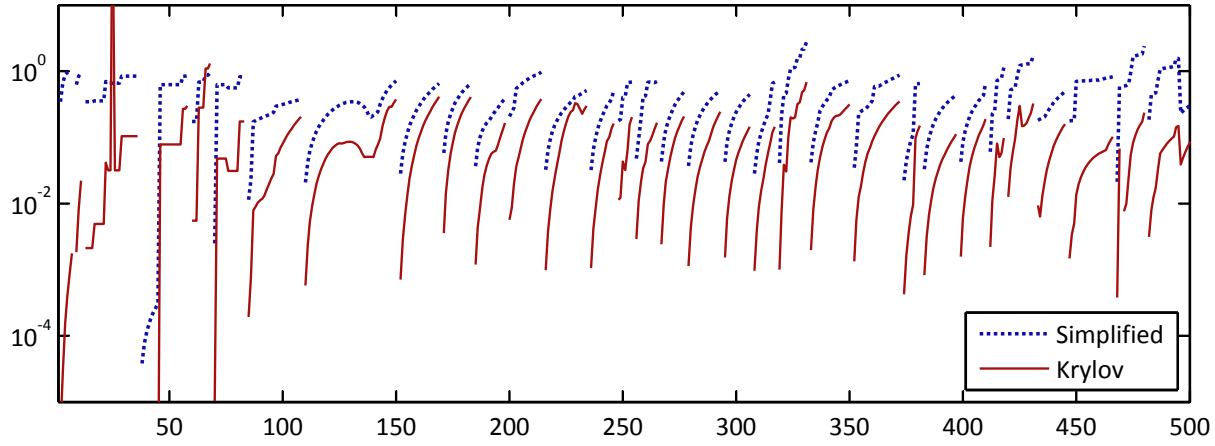


(b) Gimbal

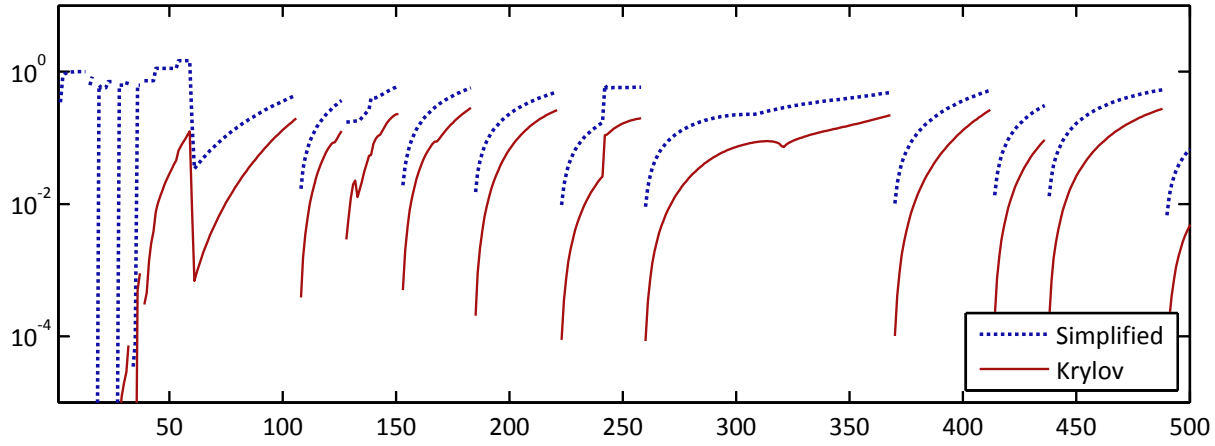


(c) Chain

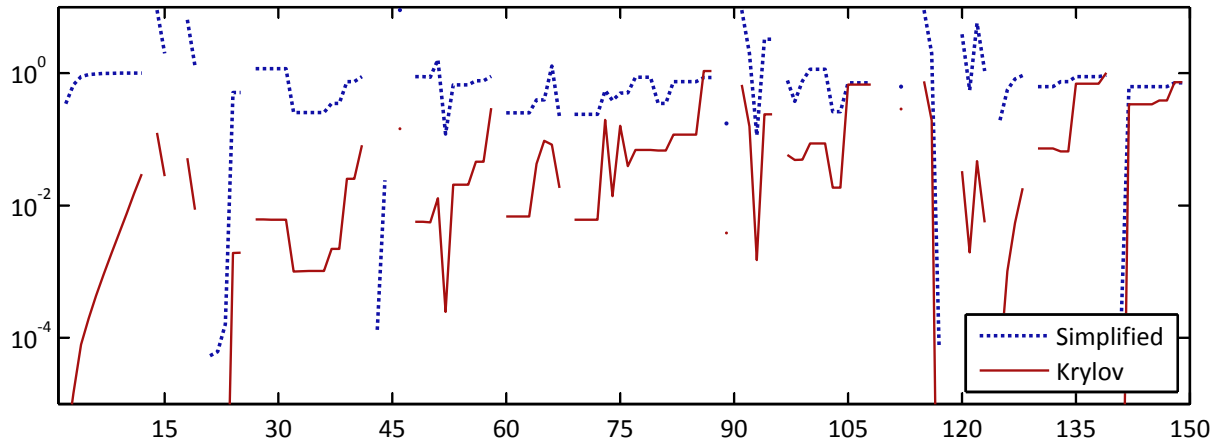
Figure 3.7: The spectral radius for the DLU. In dashed blue the spectral radius for full LU is plotted as well for comparison.



(a) ATV



(b) Gimbal



(c) Chain

Figure 3.8: The spectral radius for the Krylov method compared to the simplified method.

Chapter 4 – Refresh cycles

"Now is no time to think of what you do not have. Think of what you can do with what there is."

E. HEMMINGWAY
THE OLD MAN AND THE SEA (1952)

Overview

From the results in the previous section we derive the concept of the multi-phase cycle, a refresh cycle that is traversed by the SDLU method, the simplified method and the truncated Krylov iterations in that order. The motivation is the roughly monotone behaviour of the spectral radius during a cycle.

Since the goal is to compare the new methods with the current one, the speed gain is analysed as a factor of the current speed. To that end, the current average speed per Newton cycle is scaled to one.

The speed gain is estimated via two chief values: the expected length and the expected speed of each phase. The length is estimated by assuming that a phase ends when its spectral radius hits a certain threshold value and the speed is estimated from the speed per Newton step and the expected number of Newton steps.

Results are provided that are favourable for the first two phases. Phase one and two adhere to the philosophy only adding aspects that improve the efficiency. For the third phase, no final result is reached since there is no quantified data on the cost of the factorisation process. Moreover, Krylov is not guaranteed to improve the efficiency. Nonetheless, the outlook is promising.

The chapter ends with an outline on how to design the time-integrator such that it automatically chooses the optimal parameters (the drop tolerance for SDLU and the decision whether or not to include the Krylov-phase).

4.1 – The multi-phase cycle

In *Chapter* 2 the concept of *refresh cycle* was introduced. We observed, for some models, a nearly monotonously increasing Jacobian-error during a refresh cycle. Furthermore, we have seen in *Chapter* 3 that the integrator uses a modified Newton method known as the simplified Newton method. We introduced two new methods (SDLU and truncated Krylov) and compared these with the simplified method.

In this chapter we propose a novel approach, namely switching from one Newton method to another within a refresh cycle. This way, we strive to benefit from various aspects of the different methods at various moments in the refresh cycle. The goal is twofold: one, we aim to speed up the cycle, and two, we aim to increase its life-time.

The speed-up can be gained in the beginning of the cycle when the error is still small. This is motivated heuristically by the idea that we spend too much time on solving easy Newton iterations. We expect to be able to make do with the faster, but less accurate, SDLU. Formally, we will justify this by comparing the speed gain we

get from the faster iteration steps to the speed loss we suffer from the longer Newton path. The increased life-time can be achieved by succeeding the simplified method with the Krylov-method. In the overall process an increased life-time will result in fewer factorisations and thus in less time spent on the expensive factorisation algorithm. It is formally justified by comparing the cost of the LU-factorisation to the speed that is lost on the Krylov iterations.

These observations suggest the following three-phase cycle: Start with SDLU; when this fails, fall back to the regular simplified method; when this fails, do not refactorise immediately but append the cycle with Krylov iterations. *Figure* 4.1 gives a simple interpretation of the original single-phase cycle and the three-phased cycle on a time-line of simulated time.

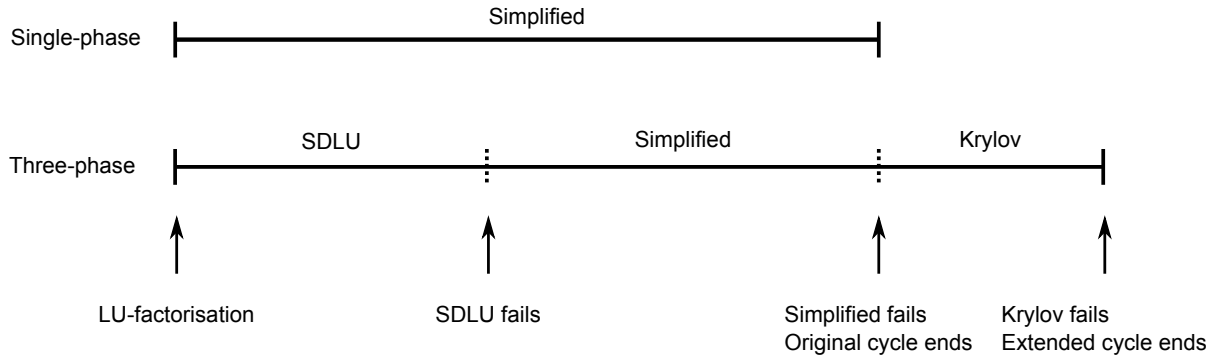


Figure 4.1: A simple graphical interpretation of the three phase cycle as proposed.

We can compare this three-phase cycle with the single-phase in the diagram of *Figure* 3.4. During a cycle, the integrator follows a path through this diagram. For both the single-phase and the three-phase cycle, this path is indicated in *Figure* 4.2. Note that the path for the current single-phase process, in blue, only traverses the simplified method. The path for the three-phase cycle, in red, crosses all three of the approximate methods. As a result, the total error (indicated by the diagonals) is expected to be better contained within a small bandwidth.

To analyse the expected speed gain, we start by considering the error of the various approximate methods. In *Figure* 3.7 and *Figure* 3.8 the spectral radii for the simplified method, SDLU, and Krylov were indicated. *Figure* 4.3 contains a sketch of the error of all three methods in a single refresh cycle. This sketch is only viable for the smooth models and not for the chain model since it does not show the monotone behaviour. Note how in this sketch the Krylov method is extended until after the simplified method fails. This indicates how far it can extend the cycle length. The plot indeed shows the error to be better contained within a certain bandwidth.

4.2 – Estimating the speed gain

The difference in speed between the single-phase and three-phase processes will be derived as a scale factor compared with the current single-phase process. First we identify the factors that contribute to the speed gain. In accordance with *Section* 2.2.4 the average computational efforts for various processes in the cycle will be denoted by κ and the resulting increase in simulated time with τ. The simulation rate is indicated by σ. Moreover, as scaled unit-time and unit-effort we choose the average over a single-phase cycle without the LU-factorisation. The other relevant factors will then be a multiple of these. All these values are annotated in the time/effort-plots of *Figure* 4.4.

All these values can be readily estimated from the results of *Chapter* 3. We start by estimating the values for τ, that is the duration of the approximate method in simulated time. DASSL discards a Newton process

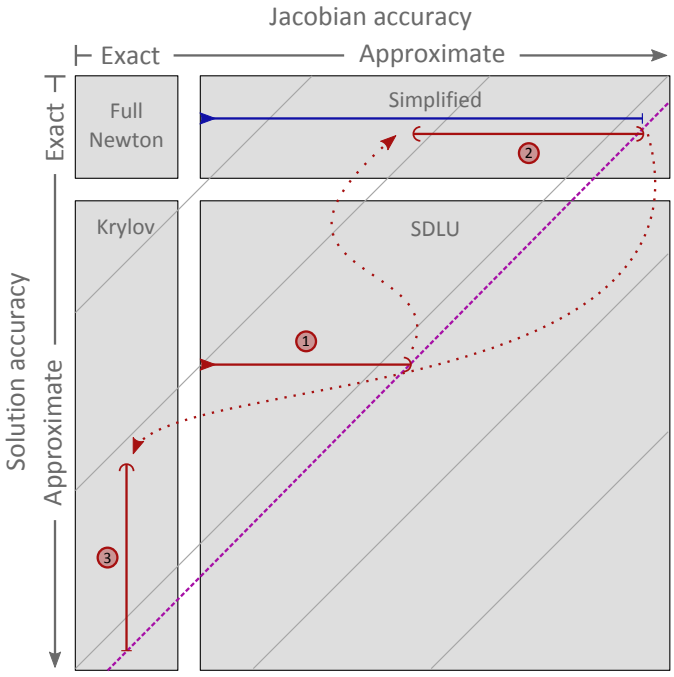


Figure 4.2: The diagram from *Figure* 3.4 with the path for the single-phase cycle in blue and the path for the three-phase in red. The dashed purple line indicates the refresh-threshold taken from the single-phase process.

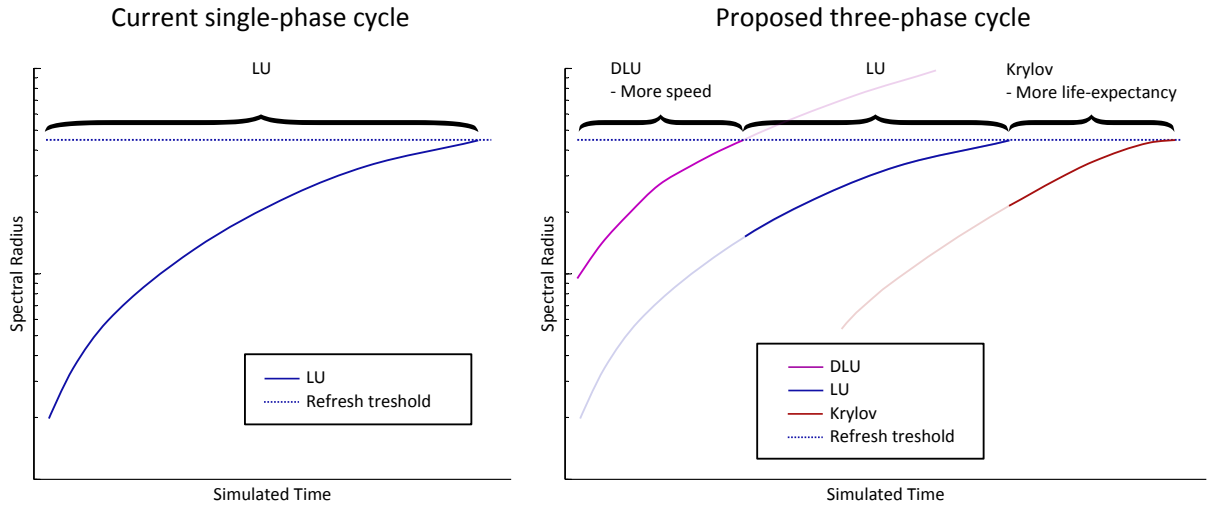


Figure 4.3: A sketch of the error in the original single-phase cycle (left) and the expected error in the three-phase cycle. Note how the total error is more constant during a cycle.

if the experimental contraction factor exceeds 0.9 [17, 18]. To predict when a method fails, we can therefore mirror this with the spectral radius by deeming the process to have failed when the spectral radius exceeds some hard-coded threshold. This seems, however, not very robust. We can gather from *Figure* 3.6 that this supposed threshold for the spectral radius is not easily pointed out: Sometimes the process fails already at ρ equal to 0.5 and sometimes it even exceeds 1. Instead of a hard-coded threshold, we will use the final spectral radius of the simplified method to capture this dynamic behaviour. In *Figure* 4.3 this refresh-threshold is already indicated

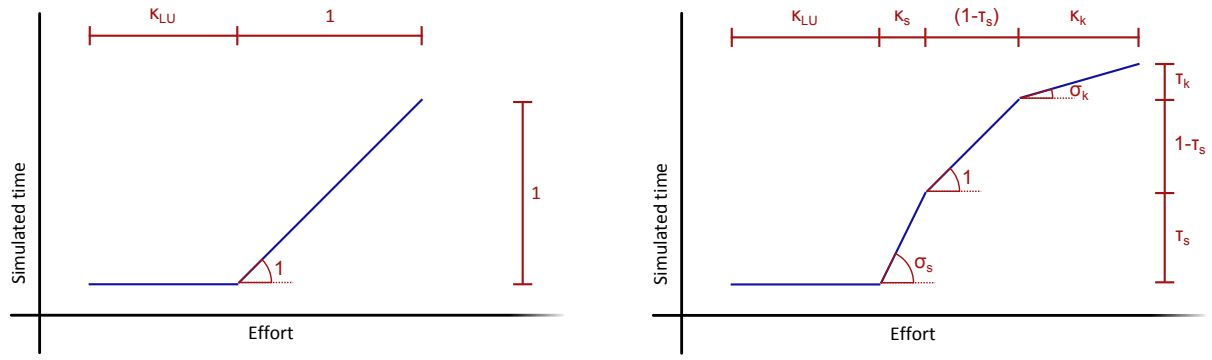


Figure 4.4: Time/effort-plots for the single-phase (left) and the three-phase cycle (right). κ indicates an effort and τ an interval of simulated time. σ is the slope $\frac{\Delta\tau}{\Delta\kappa}$.

in dashed blue.

For the chain model, however, this method can not be relied upon. The estimate of when a method breaks down is used under the assumption that the spectral radius is non-decreasing. As evidenced by *Figure* 3.6 this is not the case for the chain. On a related note, the aforementioned threshold of 0.9 for DASSL also might need to be revisited for models like these (see *Section* 6.3 for suggestions for future research).

Then we estimate the values for σ, the speed of the various Newton methods per time-step. These are determined by two numbers: the number of Newton iterations and the cost of one such iteration. The number of iterations that are needed for an alternative process will be estimated from the number that was needed for the current method. We propose three methods to do this. The first with a theoretical motivation and the second and third with an experimental motivation.

- **Match the total contraction:**
  The total contraction of a mapping with contraction rate ρ after m steps is ρ^m. Thus given that a certain method with spectral radius ρ_1 converges in m_1 steps, then an alternative process with spectral radius ρ_2 will convergence in m_2 steps such that

$$\rho_1{}^{m_1} = \rho_2{}^{m_2} \tag{4.1}$$

$$\Rightarrow m_2 = \frac{\log\rho_1}{\log\rho_2} m_1 \,. \tag{4.2}$$

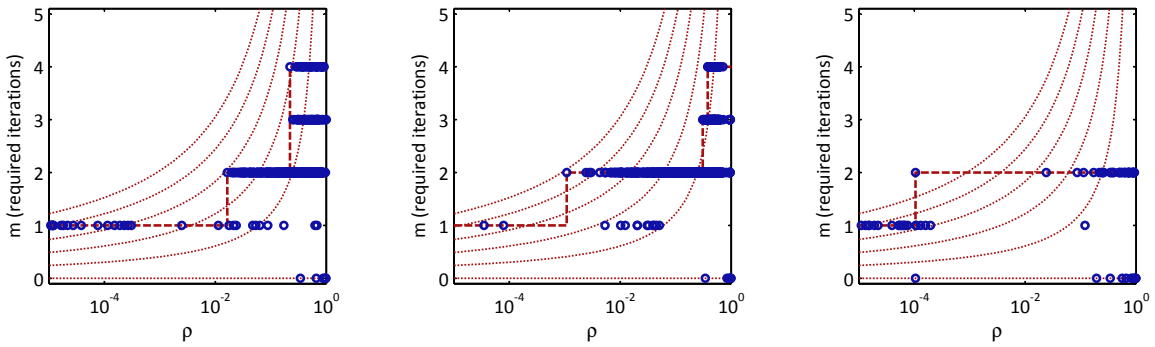
- **The worst-observed-case bound:**
  Given an alternative process with spectral radius ρ_2, then we expect this method to take as many steps as the maximum of steps that were needed by the original method for a spectral radius smaller or equal to ρ_2. That is

$$m_2 = \max\{m_1 \mid \text{for all time-steps where } \rho_1 \leq \rho_2\} \,. \tag{4.3}$$

- **The nearest-observed-case:**
  Given an alternative process with spectral radius ρ_2, then we expect this method to take as many steps as the number of steps that were needed by the original method for the nearest time step with a spectral radius larger or equal to ρ_2.

The downside of the first method is the fact that the values for m are integer. For Newton processes with more iterations, this distinction gets increasingly less important. But for our application only a handful of iterations

are used. Then the act of rounding m up to an integer seriously exaggerates the required contraction when ρ_1 is small. This method proved unsuitable for predicting the number of iterations required for the current method for which we can verify the prediction. The second method is an experimental worst-case bound. In *Figure* 4.5 the number of iterations as a function of the spectral radius is indicated together with the theoretical contraction-matching and the worst-case bound. It shows indeed that the experimental values (blue dots in the plot) do not coincide with one single level curve for constant ρ^m (dashed red). Especially for m equal to 1 or 2 the required contraction is exaggerated. In addition, note that some time steps occur that use no Newton step at all, regardless of ρ. These steps correspond with the initial phase where the time step is so small that the polynomial extrapolation of the estimator in DASSL already yields a sufficient result for the next time step. The third method is maybe too lenient. For the methods tested in this thesis, it almost always gave the same number of Newton iterations as the original method. It will not be investigated further.



(a) ATV (b) Gimbal (c) Chain

Figure 4.5: The blue dots are a scatter plot of m versus ρ, in dotted red some of the theoretical level curves where ρ^m is constant are indicated, and in heavier dashed red the experimental worst-case scenario is indicated.

Besides the number of Newton iterations, the speed per iteration is also needed. For SDLU, this iteration speed factor is equal to the reciprocal sparsity of the dropped matrices (see *definition* 3.1) and for Krylov every iteration is deemed about as expensive as one backward substitution. So a single-iteration Krylov approximation is twice as expensive as the original simplified method.

Finally, the values for κ are calculated by the division $\frac{\sigma}{\tau}$. Then the total simulation rate, including the LU-factorisation, for a three-phase cycle is calculated by

$$\sigma_{\text{3-phase}} = \frac{1+\tau_k}{\kappa_{\text{LU}}+\kappa_s+(1-\tau_s)+\kappa_k}\,, \tag{4.4}$$

compared to the rate for the current single-phase process:

$$\sigma_{\text{1-phase}} = \frac{1}{\kappa_{\text{LU}}+1}\,. \tag{4.5}$$

The addition of SDLU as phase 1 always improves the simulation rate of the solver because it makes a fraction of the time steps faster. For Krylov, however, we are not assured of a speed gain larger than one on the time-steps. In fact, the Krylov phase is expected to be slower than the regular simplified process. Recall that the gain from appending Krylov stems from the fact that fewer matrix factorisations will be necessary if the refresh cycles are longer. This means that some work needs to be done (preferably during run-time, see below in *Section* 4.3) to verify whether or not appending a Krylov phase is indeed a good idea. See *Figure* 4.6 for time/effort sketches

that clarify the efficiency of Krylov addition. Notice that the case in which it is favourable, a relatively long time is spent in the factorisation of the matrix. As observed in the previous chapter, Krylov indeed seems favourable for models that have short refresh cycles.

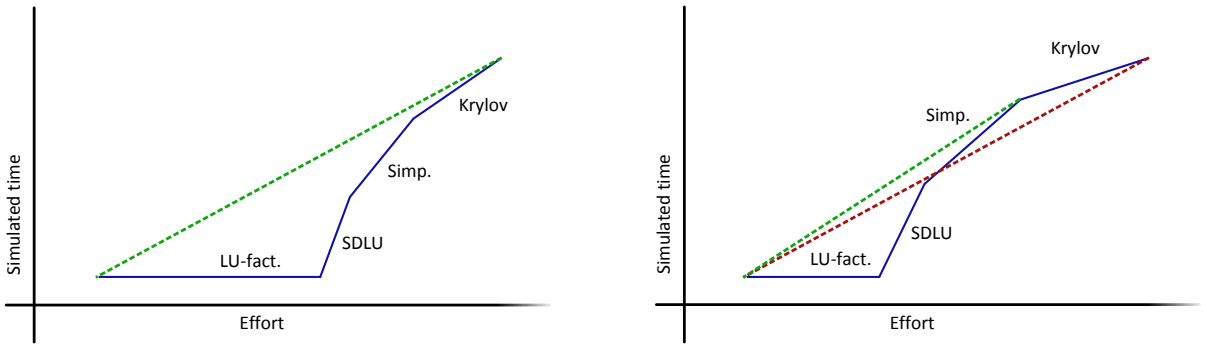


Figure 4.6: T/E-diagrams of a situation in which it is favourable to use the third phase (on the left) and one in which it is not. The overall speed of a cycle with its factorisation is indicated by the slope of the dashed line.

4.2.1 – Results for SDLU

We begin the analysis by only considering the first two phases in the three-phase cycle. The third phase, Krylov iterations, will be discussed shortly after in *Section* 4.2.2. The speed gain on the first two phases due to the addition of SDLU is calculated by finding the reduction of the effort that is needed for these phases via

$$\kappa_{\text{phase 1,2}} = \kappa_s + (1 - \tau_s) \tag{4.6}$$

$$= \frac{\tau_s}{\sigma_s} + (1 - \tau_s)\,, \tag{4.7}$$

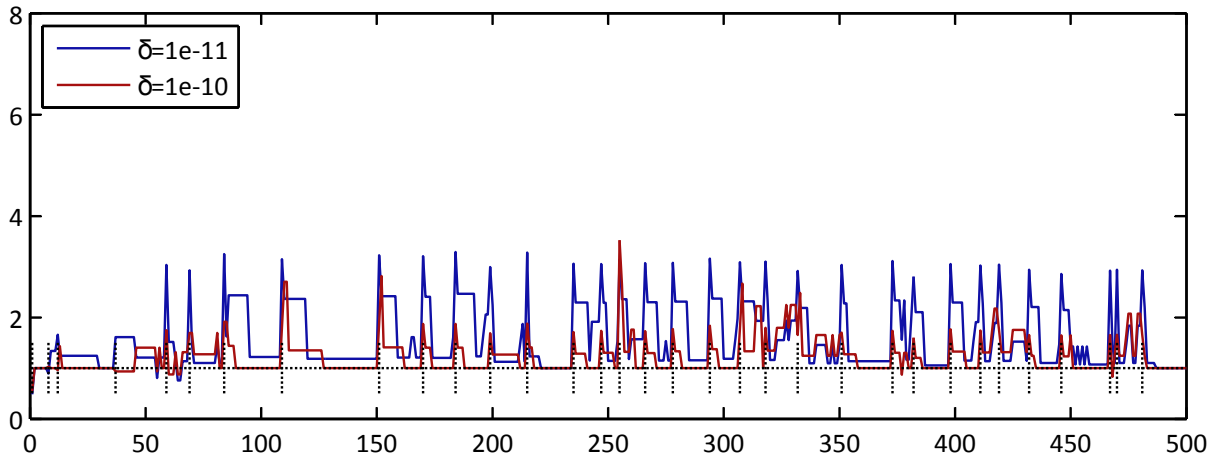
which is a weighted average of the effort of SDLU and simplified.

The speed τ of the Newton steps in phase 1 and 2 is indicated in *Figure* 4.7. This plot shows the speed gain per Newton step. Note that the speed gain in cycles (whose bounds are given by the dashed vertical lines) reduces to one. This is where the first phase ends. Moreover, a larger value of δ does not necessarily respond with a greater speed gain. Often the effect is diminished by the increase in Newton steps (calculated via the worst-observed-case bound).

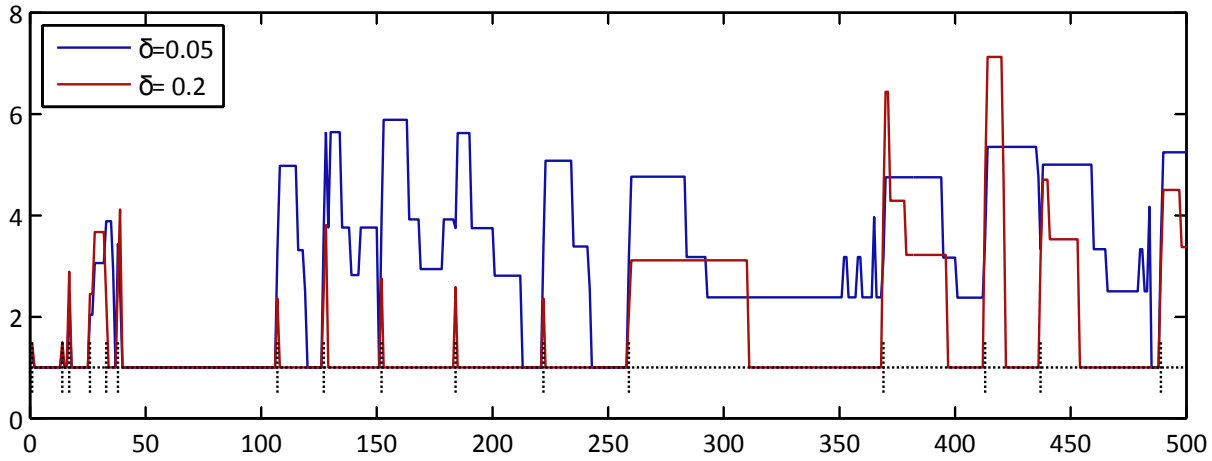
The drop tolerance needs to be chosen such that this speed gain is maximal. A higher drop tolerance results in faster backward substitution but also in a shorter length of the SDLU-phase and in an increase in Newton iterations. These three effects are mostly monotone. Therefore, the speed gain as function of the drop tolerance is roughly unimodal[1] and thus has a single global maximum. See *Figure* 4.8 for the speed gain plotted as a function of the drop-tolerance[2]. There is an optimum for δ between 0.05 and 0.15. In this case, the graph is not very steep in the neighbourhood of this optimum. That suggests that not too much effort needs to be put in fine-tuning δ.

Down below, in *Section* 4.3, it is discussed how this optimum can be found automatically during execution of the solver. For the results presented here, it is found a-posteriori by a simple interval search.

[1] A function is unimodal if it is increasing on the left of the global maximum and decreasing on the right.
[2] This plot was only made for the small gimbal model due to the extensive calculations that are needed to obtain it.



(a) ATV



(b) Gimbal

Figure 4.7: The speed gain on cycles 1 and 2. The dashed vertical lines bound the refresh cycles. The chain model is omitted since the assumption of cyclically increasing error does not hold.

4.2.2 – Results for Krylov

Since the Krylov solver is not yet implemented, as opposed to the LU backward substitution, we cannot properly address the speed of it compared to the backward substitution. In this preliminary analysis, we use the fair assumption that one Krylov iteration is as expensive as a backward substitution. That means that each Newton step is half as fast with Krylov as it is with full-LU. Given that the quantities are scaled such that the speed of LU is equal to 1. We have for the Krylov process a Newton-step speed of $\frac{1}{2}$. To obtain the speed per time-step, this value is multiplied by the ratio of Newton steps from the worst-observed-case bound. We consider the time lost on assembling the matrix at every time-step negligible (see *Section* 3.2.2). This is a fair assumption in our case, but in general it needs to be checked.

The remainder of the analysis of the Krylov-phase largely follows the format that was used for SDLU: First estimate the length of the phase by searching for the moment after the cycle that the spectral radius for Krylov exceeds the threshold; then use the worst case bound to obtain a ratio in Newton steps in phase three; finally the speed of phase 3 is obtained by the multiplication of the two.

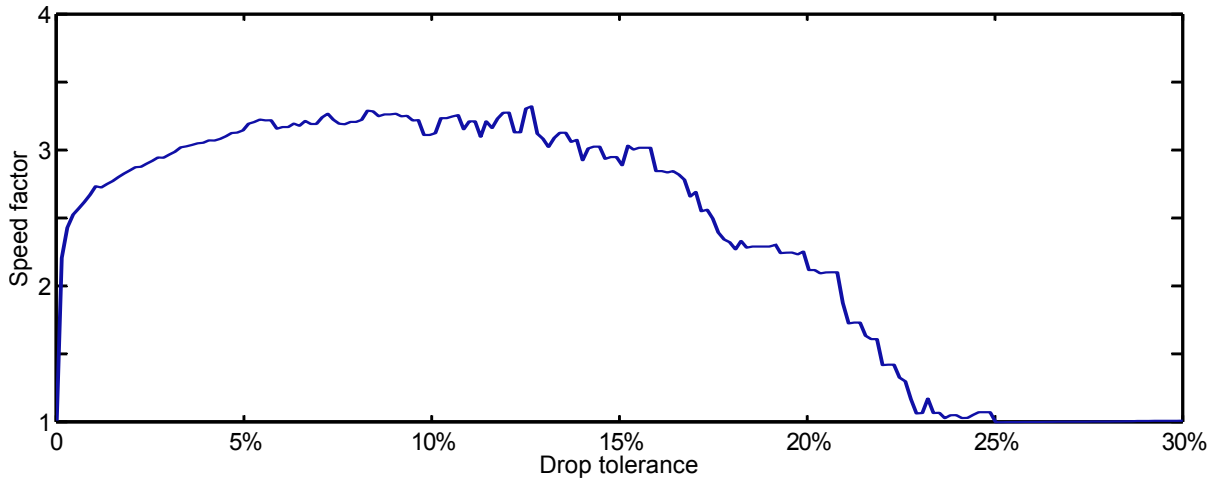


Figure 4.8: The average speed gain on phase 1 and 2 for various drop tolerances on the gimbal model.

The krylov phase appears after the simplified-phase ends. That is, when the regular single-phase implementation decides to start a new cycle. To obtain a measure of when the Krylov-phase is expected to end, we compare once again the error of this phase with the threshold value that is obtained from the very last error for the simplified method (recall *Figure* 4.3). To get a sense of the behaviour after the the simplified phase, the error of the Krylov approximation is plot in *Figure* 4.10 well beyond the cycle into the next cycle. Again, this analysis is omitted for the chain model since it hinges on the assumption of monotone error. Let it be stressed, though, that the fact that the estimates break down for this model does not imply that the model will not benefit from the additions. In fact, we have already seen in *Section* 3.3.2 that the Krylov process is appealing for models like these.

In *Figure* 4.9 the distribution of the length of the third phase is given. In accordance with the scaling, the length is expressed as a fraction of the original single-phase cycle. Note that the Krylov phase often succeeds better in adding more cycle length for the ATV model than for the Gimbal model. This once again reflects the fact that Krylov is more successful if the problem is more challenging. On average for both models, the Krylov phase is expected to last almost as long as the original cycle. This means that the number of LU-factorisations would be reduced by almost half.

The expected length of the Krylov phase is one part of the speed analysis. The other is to compensate for the difference in the number of Newton steps. For this the worst-observed-case bound is used once again. Similar to the plots of *Figure* 4.7 the plots for the Krylov-phase are shown in *Figure* 4.11. Note that now the speed gain drops below one whenever the Krylov phase is active. On average, the speed of the Krylov iterations is about 0.28 for the ATV model and 0.38 for the gimbal.

4.2.3 – Results for the complete three-phase cycle

As mentioned, due to insufficient data on the cost of the LU-factorisation, we are unable to give a single figure that indicates the speed gain on the three phase cycle. Nonetheless, the results that were obtained in *Section* 4.2.1 and *Section* 4.2.2 can be collected to obtain an overview in *Table* 4.1. This figures indicate an average speed gain on the various phases of the three-phase cycle. The first column contains the speed gain that can be expected when only phase 1 and 2 are implemented. This option does not affect the cycle length, so the data on the cost of the LU-factorisation is unnecessary there. When the third phase is implemented as well, the cycle length is extended by the factor indicated in the second column. During this extended period, however,

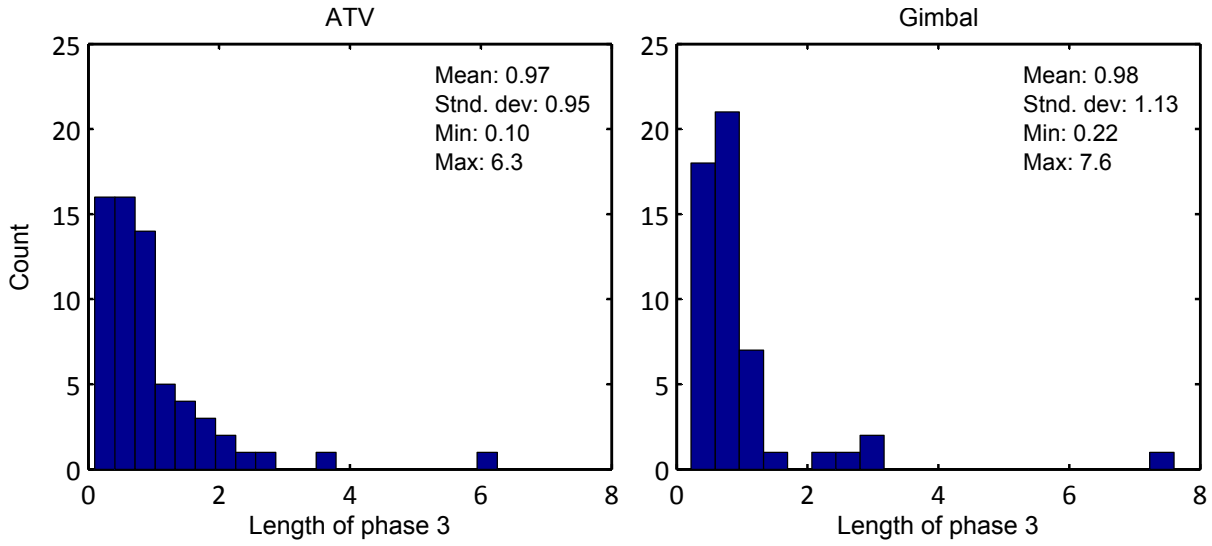


Figure 4.9: A histogram showing the distribution of the additional cycle length that is gained by the Krylov phase. The length is expressed as a fraction of the length of the original cycle.

the speed of the Newton iterations drops by the factor indicated in the third column. On average, during the whole three-phase cycle, the speed of the Newton iterations is indicated by the total speed gain indicated in the fourth column.

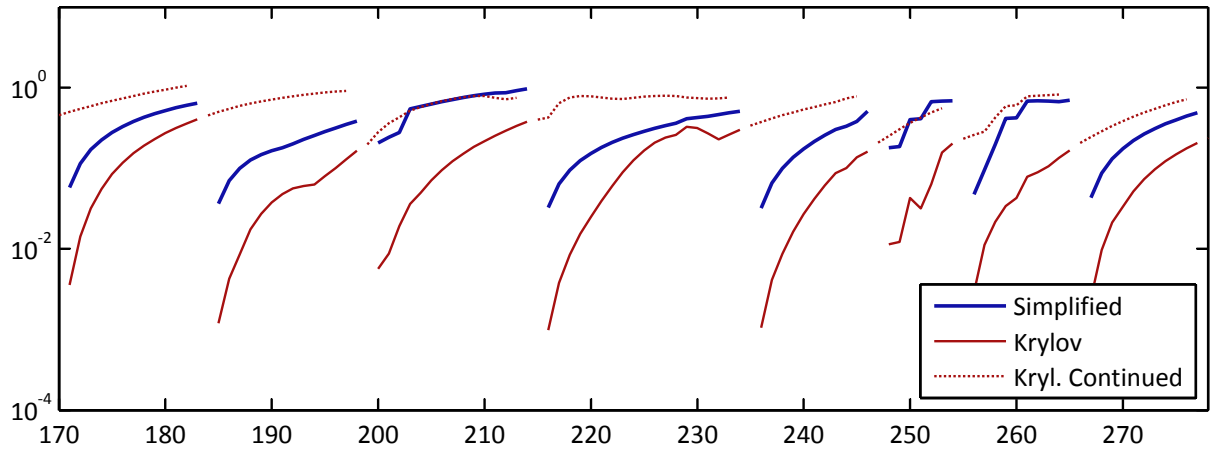
	Gain on phase 1&2: $\kappa_s + (1-\tau_s)$	Length of phase 1,2,3: τ	Gain on phase 3: σ_k	**Total speed gain:** σ
ATV	1.54	1.97	0.28	**0.92**
Gimbal	3.51	1.98	0.38	**1.95**

Table 4.1: Averages of all the results from this chapter. The total speed gain is only on the Newton iterations. It does not incorporate the decrease in required LU-factorisations.

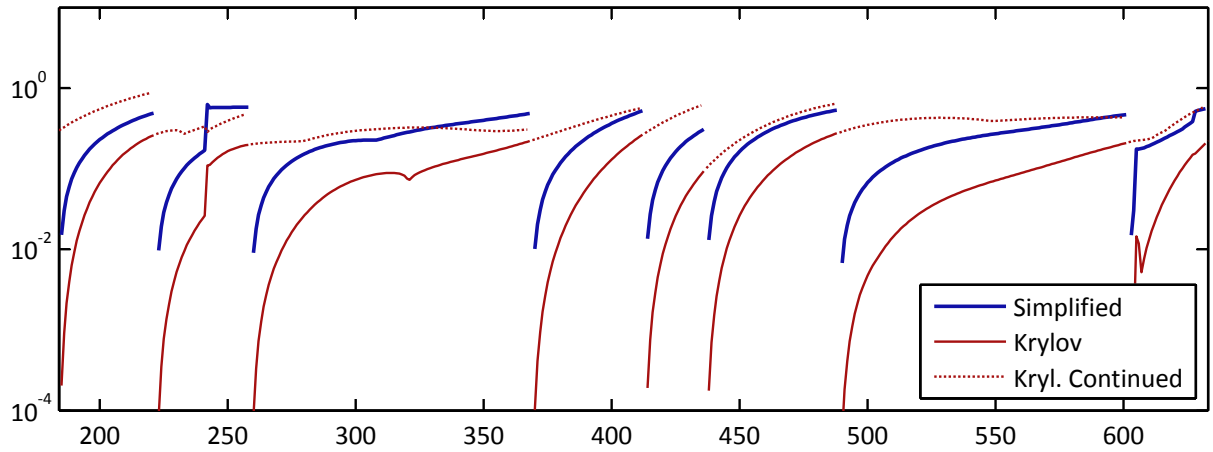
It is probably fair to assume that the gimbal model is too easy, and thus the speed gain of 1.95 can not be relied upon to mirror any real world models. The speed gain of 0.92 on the ATV model is very promising: we trade off 8% of speed on the Newton iterations for a reduction of almost 50% (that is 1/1.98) of LU-factorisations. Again, for deciding whether or not this is favourable, data on the cost of the LU factorisation is needed. But for this model, if the integrator spends more than about 7% of the time on factorising, then it is already beneficial to include phase 3.

If phase three proves not to be beneficial, we are assured of a speed gain of 54% on the newton iterations of ATV model (and 251% on the gimbal model) if just phase 1 is implemented before the current single-phase method.

Finally we remark that the analysis to decide on whether or not to include Newton iterations might just as well be applied to decide whether or not to include phase 2 after phase 1. It might, after all, very well be the case that just phase 1 provides the biggest speed up and that even the inclusion of phase 2 slows the process down. At this moment, we leave this just as a remark for two reasons: firstly, there are again no sufficient data on the cost of the LU-factorisation to asses this, and secondly, discarding the second phase, which is the original method of DASSL, does not adhere to the intention of keeping the program such that it can always fall back on



(a) ATV



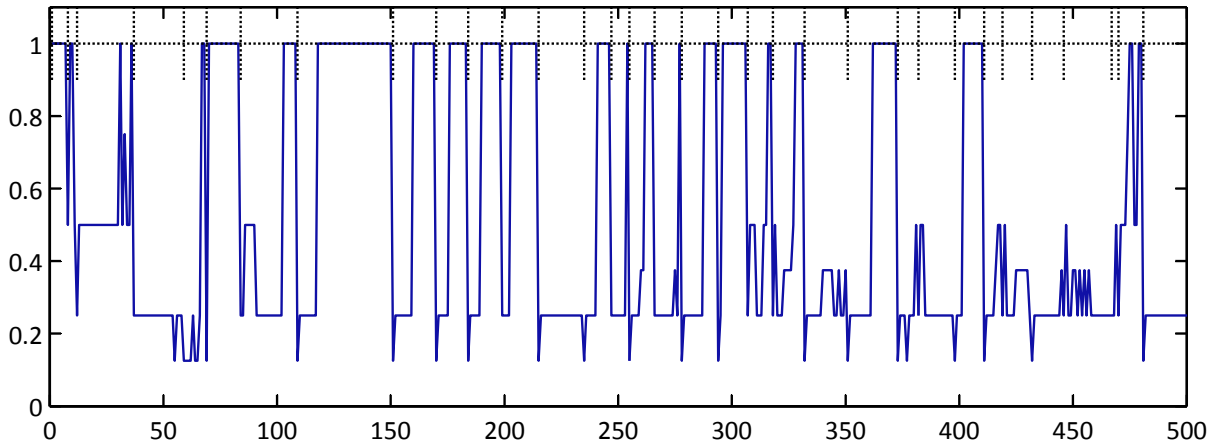
(b) Gimbal

Figure 4.10: The error of the Krylov process (in red). It is extended until after the original cycle (in dashed red).

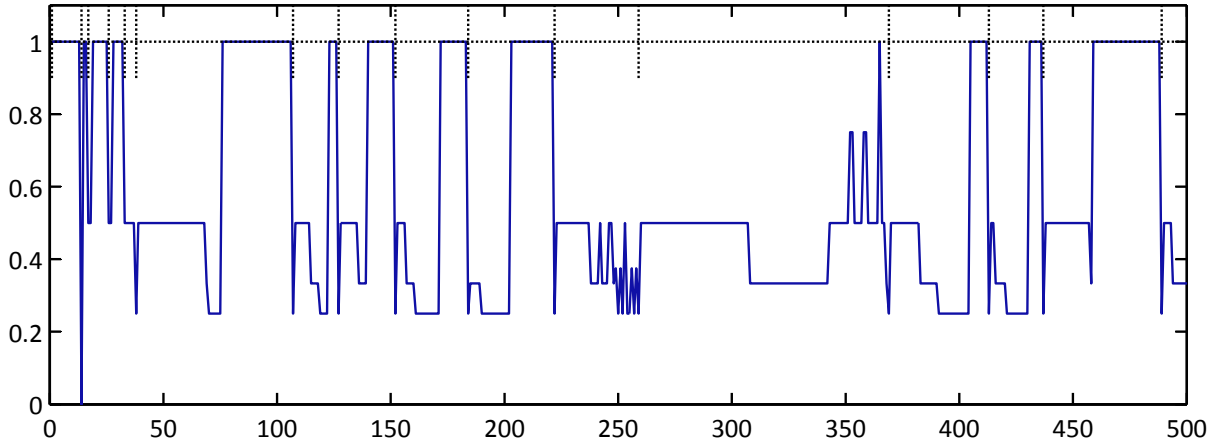
the current solution mentioned in the goals (*Section* 1.2).

4.3 – Parameter decisions

With the three-phase cycle two new parameters are introduced: the drop tolerance for SDLU and the decision whether or not to include the Krylov phase. We could opt to have the end-user submit these choices manually, but a preferably solution is one in which the integrator itself decides on them. There are two main ways to consider: on the fly during runtime and a-posteriori after a complete integration. The first option, on the fly, is the quicker one of the two. The idea for this optimisation is to constantly monitor the simulation rate σ for different parts of the refresh cycle and to adjust the new parameters such that σ is maximised. The second option would be a automated variant of the work above in *Section* 4.2. That means that a simulation is ran once without the additional solvers and then some post-processing is performed to set the parameters. As mentioned in the introduction (*Section* 1.2) some models need to be evaluated many times. This is a viable option for them.



(a) ATV



(b) Gimbal

Figure 4.11: The speed gain on the third phase.

4.3.1 – On the fly

To optimise δ we assume that the speed gain is roughly a unimodal function of δ like in *Figure* 4.8. Then the optimal value is quickly enough approximated by an interval search. For the optimal δ we only need to monitor the speed of the first and second phase of the cycle, that is:

$$\sigma_{12} = \frac{1}{\kappa_s + (1 - \tau_s)} \,. \tag{4.8}$$

There are still two concerns, however, the first is that we have seen in *Figure* 3.7 that the optimal value for δ can range from the order of 10^{-10} to 10^{-1}. This makes it hard to define an initial interval in which the optimum can be expected. To remedy this some additional work in the characteristics of the LU-factors is required. The second concern is the fact that the optimal delta is likely not constant during the simulation, this means that an algorithm that assumes a static value for the optimum will always be lagging behind on the optimal value. On the other hand, we have already observed that we may not need to be very accurate in finding the optimum due to the speed gain function not being very steep

4.3.2 – A-posteriori

The a-posteriori optimisation would be an automated repetition of the analysis in *Section* 4.2. That means that user first runs the simulation with the default settings and then runs an analysis program on the data that are acquired from this first run. The analysis program then sets the parameters to their optimal values.

Many times a simulation is used to test various events to which a device can be subjected. For example, a car accelerating to constant speed, then hitting a bump, then suddenly swerving and finally breaking to a standstill. For a model like this, we expect the parameters to reflect the change in behaviour. When the analysis is done a-posteriori, the integrator can pick up on these changes and adjust the parameters accordingly.

4.4 – Observations and conclusions

The previous chapter instigated the concept of the three-phase cycle consisting of SDLU first, then the Simplified method and then the Krylov method. The expected speed gain is estimated by assuming a monotone increasing spectral radius and a threshold for it above which we expect the method to fail. The speed of the Newton steps is estimated by the combination of the speed of the linear solver and the expected number of Newton steps. This number is obtained from the worst observed case. These estimates are not suitable for the chain model since this does not show the monotone increasing spectral radius.

The estimated speed gain on phase 1 and 2 is about 50% for the ATV model. The Krylov phase allows an extension of the cycle of almost 100%. The time steps of this phase are calculated with a almost 40% of the speed of the original process. That means that the Krylov-phase is beneficial if the LU-factorisation takes up more than 7% of the calculation.

For the actual implementation of this 3-phase cycle, two additional parameters are introduced: The drop parameter δ and the decision whether or not to include phase 3. A short outline is provided on how these parameters can be set automatically either on the fly or a-posteriori.

Chapter 5 – The Jacobian

"Our ancestors had no great tolerance for anachronisms."

H.G. WELLS
THE TIME MACHINE (1895)

Overview

This chapter contains three additional methods to improve the time-integrator. The first method is an exploit of the structure of the Jacobian, the second and third are a means to update the dated Jacobian matrix such that it more closely resembles the current one without the need to refactor.

All the methods provided are designed such that they can be 'wrapped around' the linear system solver. That means that the only change in the program is some pre-processing before and some post-processing after the call to the solver. No global changes need to be made.

5.1 – Structure and ordering

In the corrector loop a root of the main function is aimed to be found by means of the Newton methods described in *Chapter* 3. For this section we only need the global structure of the system. This is as follows

$$\begin{bmatrix} D_1 & D_2 & 0 \\ | & | & | \\ B_1 & B_2 & B_3 \\ | & | & 0 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} , \tag{5.1}$$

with D diagonal matrices and B general blocks.

In the current code this matrix is solved as is, but the two diagonals in the uppermost row can be exploited by an elimination

$$D_1 x_1 + D_2 x_2 = b_1 \tag{5.2}$$

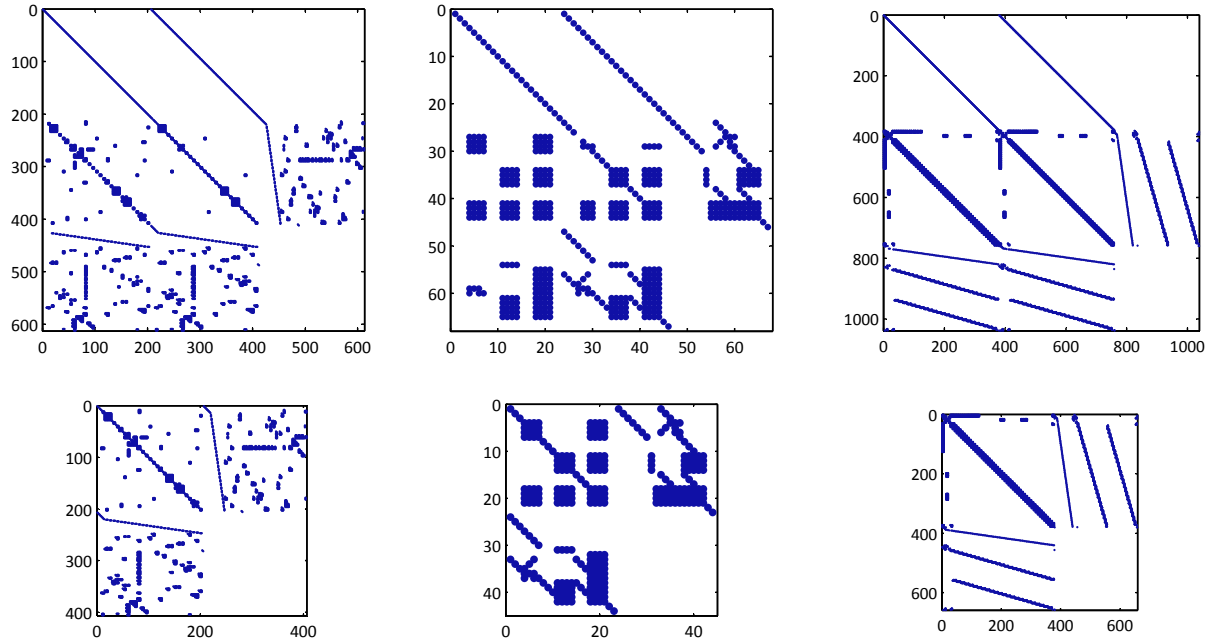
$$\Rightarrow x_1 = -D_1^{-1} D_2 x_2 + D_1^{-1} b_1 , \tag{5.3}$$

and a substitution

$$\begin{bmatrix} | & | & | \\ B_1 & B_2 & B_3 \\ | & | & 0 \end{bmatrix} \left(\begin{bmatrix} -D_1^{-1} D_2 & 0 \\ I & 0 \\ 0 & I \end{bmatrix} \begin{bmatrix} x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} D_1^{-1} b_1 \\ 0 \\ 0 \end{bmatrix} \right) = b_2 \tag{5.4}$$

$$\Rightarrow \begin{bmatrix} -B_1 D_1^{-1} D_2 + B_2 & B_3 \end{bmatrix} \begin{bmatrix} x_2 \\ x_3 \end{bmatrix} = b_2 - B_1 D_1^{-1} b_1 . \tag{5.5}$$

Thus solving for $[x_1, x_2, x_3]^\mathsf{T}$ involves only solving for $[x_2, x_3]^\mathsf{T}$ in (5.5) and then simply setting x_1 according to (5.3). The system in (5.5) has size $(n+k)^2$ as opposed to $(2n+k)^2$ for the original system, where n is the number of differential equations and k is the number of constraint equations in the original system (2.14 - 2.15). The difference in the original system and the reduced system is indicated in *Figure* 5.1 and *Table* 5.1. Note that the reduced system is smaller and denser.



(a) ATV (b) Gimbal (c) Chain

Figure 5.1: The original and reduced Jacobian. A quantification of these plots is provided in *Table* 5.1.

		Equations	Non-zeros	Density
ATV	Original	612	8216	2.19%
	Reduced	406	5268	3.20%
Gimbal	Original	67	522	11.6%
	Reduced	44	317	16.4%
Chain	Original	1038	11309	1.05%
	Reduced	658	6702	1.55%

Table 5.1: Key statistics for the original and reduced Jacobian.

Since LU-factorisation takes $\mathcal{O}(N^2)$ time, with N the number of equations, the time gain that can be expected is a factor in the order of $\left(\frac{2n+k}{n+k}\right)^2 = \left(1+\frac{n}{n+k}\right)^2$, for $2n$ differential equations and k constraint equations. This reduction is not tested in the code of *VL Motion*, but profiling in Matlab revealed that the reduced system solves about 20% to 40% faster for all the test models. In addition to a gain in speed, we remark that this method will also reduce the memory that the solver requires.

Finally we remark that a structural change of the Jacobian matrix affects the other methods: SDLU might

need a different drop-tolerance (likely smaller) and Krylov will perform relatively worse on a smaller system. We therefore recommend to implement this method before SDLU or Krylov (see *Section* 6.2.4).

5.2 – Life-time and updating

The reason for the corrector loop of DASSL to fail is the fact that the dated Jacobian $\tilde{G}$ is no longer recent enough. That is, the current situation has changed so much that the Jacobian in use carries not enough valuable information for the Newton process to converge. This is, of course, remedied by assembling and factoring a new recent Jacobian. This is what we call refreshing. The length of the refresh cycle is the life-time of a Jacobian.

In this section we introduce the idea of improving the life-time of a Jacobian by updating it. This update should resemble the current Jacobian G closer but still be easily invertible without having to refactor. Hereby we aim to reduce the overall number of factorisations that are needed. In the most general sense, we are looking for an update-mapping Ud on matrices such that

1. $\mathrm{Ud}(\tilde{G})$ is closer to G than $\tilde{G}$ itself (in some sense),
2. given $\tilde{G}^{-1}$, then $\left[\mathrm{Ud}(\tilde{G})\right]^{-1}$ is cheaply obtainable.

Like for preconditioners in Krylov methods, these requirements are contradictory. Even more so, the similarity between updating and a preconditioning is not coincidental and many ideas can be interchanged amongst them.

For the remainder of this chapter, we will use the spectral radius-error as before to decide whether or not a matrix is closer to the current one.

5.2.1 – Multiplicative update

Falling back on the remarks about scale, we can choose to update in such a way that the scale of the old Jacobian more closely resembles that of the current one. This adheres to the requirement of the result being easily invertible because scaling corresponds to multiplication with diagonal matrices. So we have

$$\mathrm{Ud}(\tilde{G}) = D_1 \tilde{G} D_2 \,, \tag{5.6}$$

with D_1 a full-rank diagonal matrix to scale the equations and D_2 to scale the unknowns. Note that the second requirement is satisfied due to trivial inversion of diagonal matrices, this gives

$$\mathrm{Ud}(\tilde{G})^{-1} = D_2^{-1} \tilde{G}^{-1} D_1^{-1} \,. \tag{5.7}$$

There are many ways to obtain sensible matrices D_1 and D_2, many of which follow a black box approach of only considering the numbers in the matrix. We propose so called **step size matching**. This is an attempt to mitigate the difference in the matrix due to the fact that the time step changes. The Jacobian is dependent on the time-step size. One of the dependencies shows up in the scaling with $\alpha = \frac{c}{h}$ as mentioned in *Section* 2.2.3. Such a scaling can be expressed by left-multiplication of a diagonal matrix D_h

For this part we use an subscript h to stress the dependence on the time-step size. Note that the dependence on the time step is 'almost linear' due to aforementioned scaling. By this we mean that many of the occurrences of h are a row-factor in the matrix.

$$G_h \approx D_h G_1 \,, \tag{5.8}$$

With D_h a diagonal matrix containing factors of h (or α) on the corresponding columns. This scaling has actually been performed with the dated time step size $\tilde{h}$. That means that these factors in front of the blocks consist of dated information. Now if the current time-step h is different from the one at the beginning of the refresh cycle, then these factors can easily be updated as follows

$$\mathrm{Ud}\tilde{G} = D_h \left(D_{\tilde{h}}\right)^{-1} \tilde{G} \tag{5.9}$$
$$= D_{h/\tilde{h}} \tilde{G}\,. \tag{5.10}$$

The motivation to arrive here is purely heuristic. The term 'almost linear' is used loosely and it provides no formal justification. Nonetheless, the intuition can be easily checked in practice by comparing the spectral radius-error of $\mathrm{Ud}(\tilde{G})$ with that of $\tilde{G}$. The results are shown in *Figure* 5.2 and *Figure* 5.3. These plots indicate the ratio between $\rho(I - \mathrm{Ud}(\tilde{G})^{-1}G)$ and $\rho(I - \tilde{G}^{-1}G)$. Note that quite often the updated version is exactly as good as the original. This is not a surprising result, it occurs when the step size has not changed since the refresh. When it does change, however, the result is mainly positive. Moreover note that the majority of the negative results are in the initialisation phase for the ATV model. This is not surprising either since then the time-step changes many orders of magnitude. A scaling as in (5.10) is likely to go awry. Finally, the scaling is a tremendous improvement for the chain model. More often than not it halves the spectral radius.

To implement this update, it is not even needed to multiply the rows of the matrix (taking $\mathcal{O}(n^2)$ multiplications). Instead one can multiply the right hand side with the inverse of D like

$$D_{h/\tilde{h}} \tilde{G} z = y \tag{5.11}$$
$$\Rightarrow \tilde{G} z = D_{\tilde{h}/h} y\,, \tag{5.12}$$

which takes only $\mathcal{O}(n)$ multiplications. In some cases, though, there is reason to perform the $\mathcal{O}(n^2)$-multiplication on the matrix after all. This is for models where the time step size does not change very often. For example, the gimbal model changes step size just 3% of the time (versus 12% for the ATV and 60% for the chain). In this case, the sporadic matrix multiplication might still be cheaper than the recurring multiplication on the right hand side.

5.2.2 – Additive update

For some specific cases, we can cheaply compute the inverse of $\tilde{G} + U$ given the inverse of $\tilde{G}$. This is done by a technique due to Sherman, Morrison and Woodbury for matrices U of small rank. The SMW-formula is found in an article by Hager on updating the inverse of a matrix [10].

Lemma 5.1 (Sherman-Morrison-Woodbury formula)**.** *Let A an invertible square matrix in $\mathbb{R}^{n\times n}$, let V and W be rectangular matrices in $\mathbb{R}^{n\times k}$ with $k < n$. Then*

$$\left[A + VW^{\mathsf{T}}\right]^{-1} = A^{-1} - A^{-1}V\left(I + W^{\mathsf{T}}A^{-1}V\right)^{-1} W^{\mathsf{T}}A^{-1} \tag{5.13}$$

The matrix VW^{T} is known as a rank-k update to A. In the case where k equals one, some terms reduce to scalars and the equation is simplified to.

$$\left[A + vw^{\mathsf{T}}\right]^{-1} = A^{-1}\left(I - \frac{vw^{\mathsf{T}}A^{-1}}{1 + w^{\mathsf{T}}A^{-1}v}\right) \tag{5.14}$$

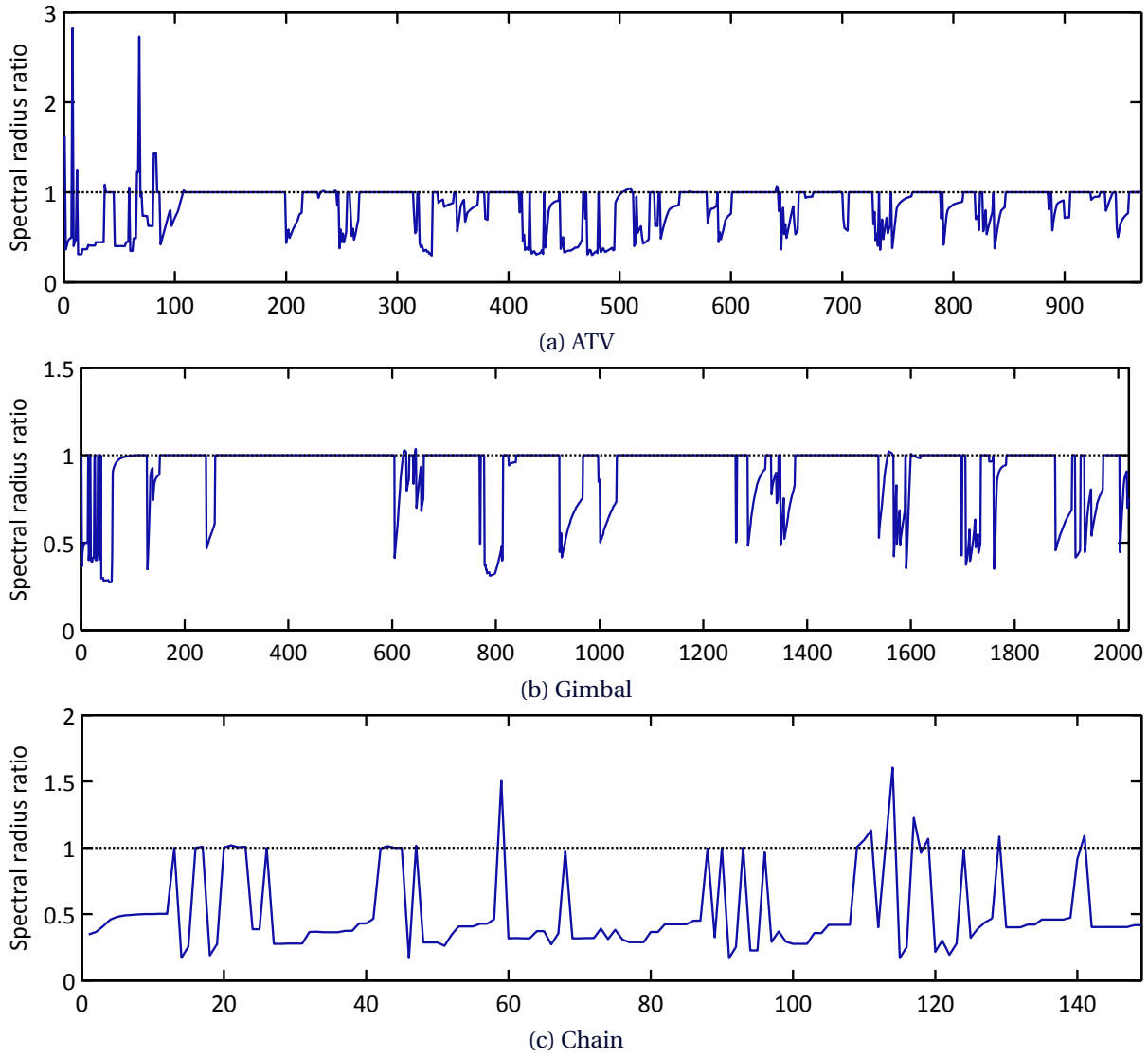


Figure 5.2: The spectral radius of the time-step matching update from (5.10) compared to the original as the ratio $\frac{\rho(I-\mathrm{Ud}(\tilde{G})^{-1}G)}{\rho(I-\tilde{G}^{-1}G)}$.

To put this technique to use, we look for a low rank matrix VW^{T} such that $\tilde{G}+VW^{\mathsf{T}}$ is closer to G. The first that comes to mind is picking VW^{T} equal to D_k a rank-k approximation of $D = G - \tilde{G}$. The obvious way is by the eigenvalue decomposition

$$D = E\Lambda E^{-1} \tag{5.15}$$

$$D_k = \begin{bmatrix} | & | & & | \\ e_1 & e_2 & \dots & e_k \\ | & | & & | \end{bmatrix} \begin{bmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_k \end{bmatrix} \begin{bmatrix} | & | & & | \\ e_1 & e_2 & \dots & e_k \\ | & | & & | \end{bmatrix}^{-1}. \tag{5.16}$$

The downside of methods like these is that they are expensive. Calculating the eigenvalue-decomposition

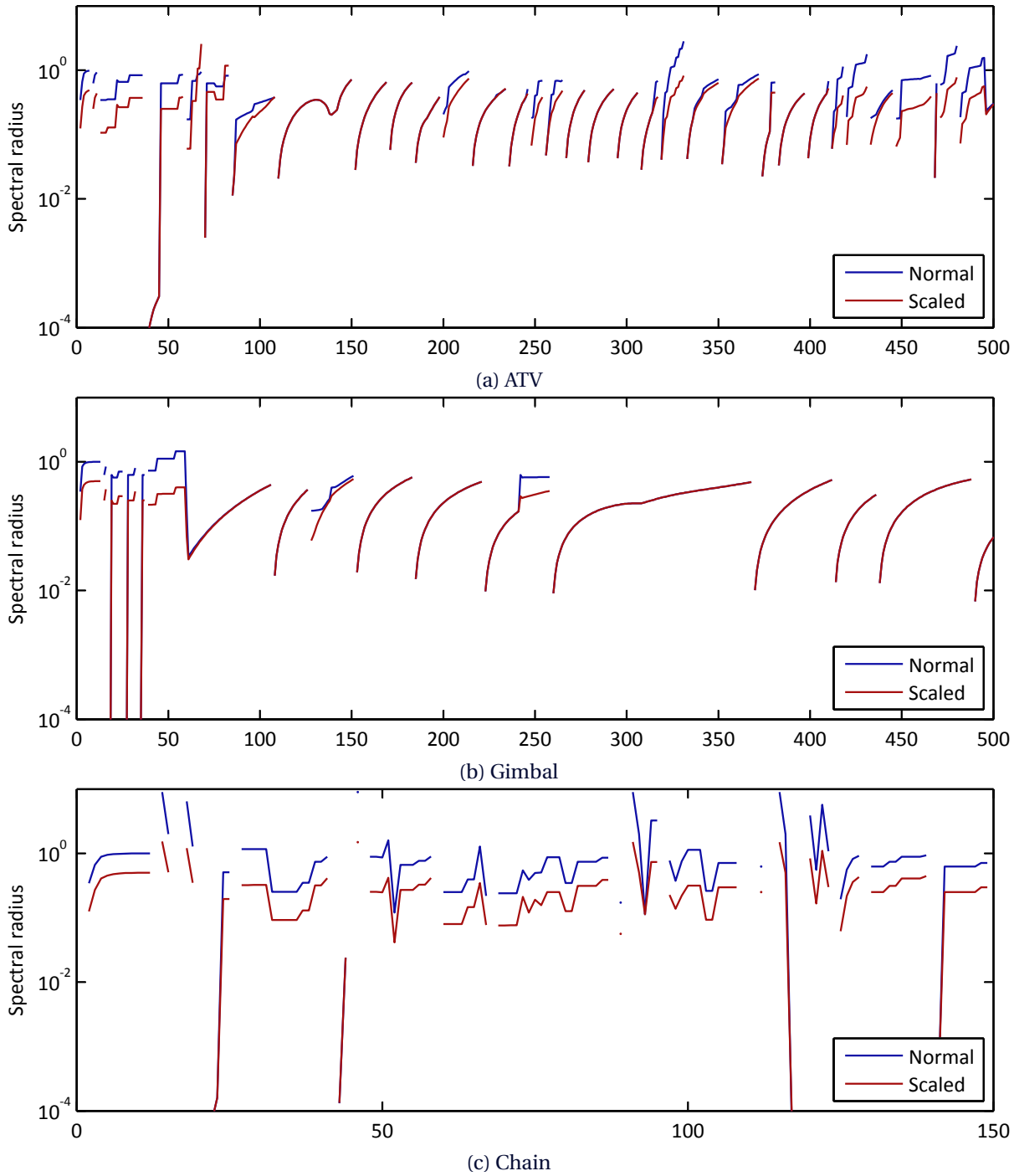


Figure 5.3: Both the spectral radius of the original Jacobian and the time-step update from (5.10).

is an irreconcilable expense. In the philosophy of this thesis we propose a way to approximate this low rank approximation by making use of the data we already have. The following is an extension to Broyden's method as described and motivated in his paper [5] and in [10].

Not only is the low-rank update a means to make the matrix more recent, it also provides a means to speed

up Krylov iterations in particular. The details of this are explained in a conference paper by Giraud & Gratton [9]. For our application the usefulness of this technique is questionable since we only use a single Krylov iteration.

Low-rank updates as a byproduct of the Newton-process

For the Newton process a series of values for the function f is calculated at points $\{y_0, y_2, \ldots\}$. These values contain recent information about the function. Hence it stands to reason that they can be used to renew the data in $\tilde{G}$. This is done by a multidimensional finite difference (the secant equation)

$$G(y_i)(y_{i+1} - y_i) \approx f(y_{i+1}) - f(y_i) \tag{5.17}$$

$$G_i z_i = \Delta f_i \,, \tag{5.18}$$

which gives approximate information about the unknown matrix G. Indeed, the fact that the action of G upon a single vector is given, restricts G to the following form

$$G \approx v w^{\mathsf{T}} + G_\perp \,, \tag{5.19}$$

with

$$w^{\mathsf{T}} = z_i^{\mathsf{T}} \tag{5.20}$$

$$v = \frac{\Delta f_i}{z_i^{\mathsf{T}} z_i} \tag{5.21}$$

$$G_\perp \text{ such that } G_\perp z_i = 0 \,. \tag{5.22}$$

Now assume that the pairs $\big(y_i, f(y_i)\big)$ and $\big(y_{i+1}, f(y_{i+1})\big)$ are obtained from an approximate Newton process with $\tilde{G}$. Define the vectors v and w^{T} as above. Then $\tilde{G}$ can be updated by subtracting the current information in the direction of w and adding the new result as follows

$$\mathrm{Ud}(\tilde{G}) = \tilde{G} + (-\tilde{G}w + v)\, w^{\mathsf{T}} \tag{5.23}$$

$$= \tilde{G} + (f_i + v)\, w^{\mathsf{T}} \,, \tag{5.24}$$

in which we used that $-\tilde{G} z_i = f_i$ by definition of the Newton process. This gives the required result

$$\mathrm{Ud}(\tilde{G}) z = \Delta f \,. \tag{5.25}$$

The addition of the rank-1 matrix $(f_i + v)\, w^{\mathsf{T}}$ in (5.23) can be seen as removing the directional part $\tilde{G} z_i$ in favour of Δf_i. We will refer to this rank-1 matrix as the Broyden-update. Once more we stress the fact that it is still an approximation due to the fact that we use a finite difference rather than the actual derivative. This calls for some caution when deciding to use the update. It might be that matrix error in the dated Jacobian is still less than the discretisation error in the finite difference. In this case the Broyden update would actually be detrimental to the overall matrix. This extra caution can be taken by only using the Broyden-update when the matrix is sufficiently dated, when $\|z_i\|$ is sufficiently small (say, at the very last Newton iteration of each time-step) or by damping the effect with a parameter $\beta \in (0, 1)$ as follows

$$\mathrm{Ud}(\tilde{G}) = \tilde{G} + (\beta f_i + (1 - \beta) v)\, w^{\mathsf{T}} \,. \tag{5.26}$$

Note that the Broyden update is a backward approach to the Newton process. Instead of finding a sequence of points from the derivative of a function we infer the derivative (or rather the finite difference) by a sequence of points that is given by an inferior Newton process. As a final remark we stress that Broyden's method from this viewpoint is, as a Newton method, not distinct from the simplified method. Rather it is the simplified method with a specific choice of additional matrix updating in-between Newton steps. Therefore, Broyden's method is not mentioned in *Section* 3.2.2.

5.3 – Observations and conclusions

In this section, we have provided three additional improvements on the linear solver. They are designed such that they can be 'wrapped around' the current solver. That means that they can be described in terms of pre- and post-processing the Jacobian and/or the residual vector.

The first method is a reduction of the size of the Jacobian. This is an exploit of the two diagonals that appear on top of the matrix. This results in a reduction from 612 to 406 equations for the ATV model and a reduction from 1038 to 658 for the chain model.

The second method is a correction for the time step size on the scaling factors of the matrix. This correction is a cheap way to update the matrix such that it contains more current information. This information is in the form of the step size h. For the chain model, which often changes step size, this correction shows to be most beneficial.

The third method is Broyden's approximate Newton method. In this thesis, it is redefined as a way of matrix updating. The update consists of a rank-1 matrix that is obtained from the Newton steps themselves. The method is extended with a parameter β which allows us to set how influential this update is.

Chapter 6 – Conclusion

"We have so much time and so little to do. Strike that, reverse it."

R. DAHL
CHARLIE AND THE GREAT GLASS ELEVATOR (1972)

6.1 – Summary

The time integrator DASSL is used to numerically integrate the equations of motion for constraint multibody systems. The integrator uses an approximate Newton method called the Simplified method. The approximation consists of the fact that an old Jacobian is kept in use to solve the subsequent Newton steps as long as this works. This leads to the definition of refresh-cycle: the part in between renewals of the Jacobian.

In this thesis we investigated the use of two alternative approximate Newton methods: Simplified with dropped-LU (SDLU) and a truncated Krylov solver. These methods are respectively less accurate but faster and more accurate but slower than the original simplified method. The observation that the error of the dated Jacobian is increasing leads to the idea of the three-phase refresh-cycle as follows: Start with SDLU, then regular simplified and append with Krylov. The three methods share many characteristics which makes using them in tandem appealing. We have estimated the speed gain that can be obtained when this idea is practically implemented.

In addition to the revisions on *refresh cycle-scale* we also provided methods to improve the efficiency *Newton step-scale*. This is done by the concepts of matrix-reduction and matrix-updating. Matrix-reduction is done by eliminating equations to obtain a smaller, but denser, matrix. Matrix-updating is a means to introduce more recent information into the system without the need to refactorise.

6.2 – Conclusions and recommendations

6.2.1 – Approximate Newton methods

The approximate methods SDLU and truncated Krylov show to be useful additions to the original algorithm. The fact that SDLU is faster but less accurate than the original and that the Krylov approximation is more accurate but slower makes the two complementary. This results in the introduction of the multiphase cycle consisting of SDLU, simplified and Krylov in that order. We also observed that Krylov seems to perform well on models that have sudden changes.

6.2.2 – Multi-phase cycle

The multiphase cycle consists of SDLU, simplified and Krylov. This order is based on the assumption that the error in a dated Jacobian matrix is increasing in a refresh cycle. In this way, we benefit from the merits of

the approximate Newton methods at the appropriate points in the cycle. The calculations to find the expected speed gain and cycle length gain are dependent of the assumption that the error is indeed monotone. This makes the analysis unsuitable for the chain model.

The addition of phase one, SDLU, makes the Newton iterations faster by a factor up to 2. The addition of the Krylov-phase is not guaranteed to improve the efficiency. This depends on the effort that is spent in factoring the matrix. No data were obtained in this project to compare the factorisation cost with the iteration cost.

6.2.3 – Jacobian optimisation

Independent of the used Newton method, there are three more options to improve the efficiency of the integrator: Size reduction by elimination, multiplicative updating and additive updating. All these methods are designed such that they can be 'wrapped around' the matrix-solver. From the outside, there is no change in the behaviour of the solver, but the process is performed faster or more accurate.

Jacobian size reduction

The size reduction is a means to exploit the Jacobian's structure for multibody dynamics. Due to the two diagonals in the uppermost row-blocks of the matrix, the system can be reduced from size $(2n+k)^2$ to $(n+k)^2$. This gives a quadratic increase in the speed of the LU-factorisation. For our test models in Matlab, this improvement was 20% to 40%. It also reduces the memory that the solver needs.

Multiplicative update: Step size matching

The step size matching is a simple diagonal scaling to adapt the scaling with α to the new time-step size. For such a simple idea, the results are very impressive.

Additive update: Broyden

The low-rank or Broyden-update is a means to update the Jacobian without having the need to refresh. The update consists of adding a low-rank matrix with more recent information. The low-rank matrix is obtained as a by-product of the Newton iterations. We do not have sufficient data to qualify the efficiency that can be gained by this update.

6.2.4 – Prioritised list of recommendations

Since the purpose of this project was to explore various practical improvements on the solver, an important part of the conclusion is a list of recommendations on which addition to implement. These recommendations will follow from a combination of the expected gain in efficiency and the expected effort that is needed to implement the method.

In *Chapter* 4 two possible major additions to the solver were provided: Adding SDLU as phase 1 and adding Krylov as phase 3. In *Chapter* 5 three smaller ideas were given: dimensional reduction, multiplicative time-step updating and additive Broyden-like updating. For each of these five methods the author has a sense of the efficiency the solver will gain from it and of the effort that is needed to implement it. These are charted in *Figure* 6.1.

This chart is backed by the the following prioritisation and motivation of recommendations:

1. **Multiplicative update**: Rescaling the equations according to the changing time-step should be very easy. The result is a spectral radius which is almost always smaller.

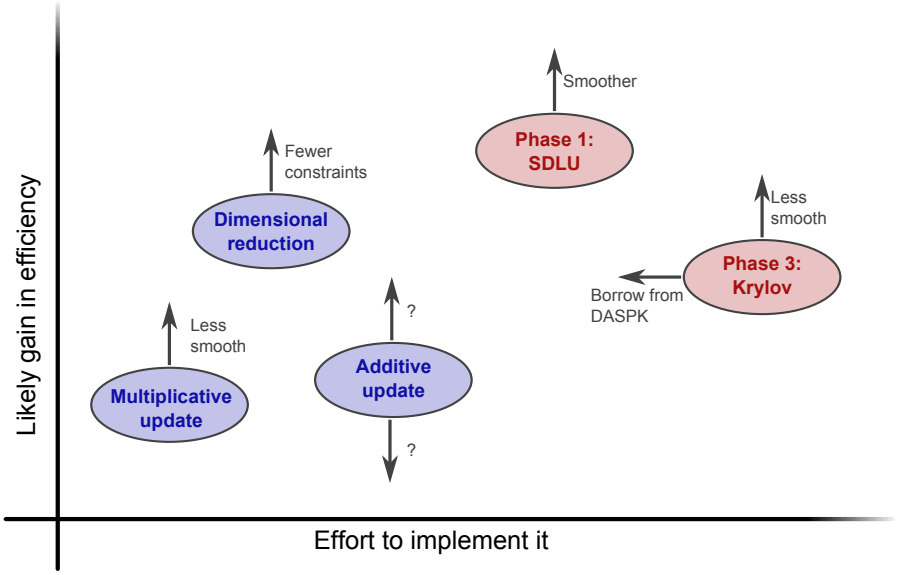


Figure 6.1: The five ideas that are proposed in this thesis charted to compare the likelihood that they increase efficiency and the effort needed to implement it. The major methods are red, the minor ones blue. The arrows indicate the most important characteristics that influence the position on the chart.

2. **Dimensional reduction**: Wrapping the matrix solver inside a routine which first reduces the matrix and afterwards expands the solution is a clean and elegant way to implement this. The speed gain on the factorisation step increases quadratically for the number of bodies. This method should be implemented first since greatly changing the structure of the matrix affects the other methods.

3. SDLU: Phase 1 and 2 are guaranteed to speed up the simulator. Implementing the dropped-LU alongside the regular LU and deciding on when to use it does require some bookkeeping.

4. **Krylov**: Krylov approximations show rather impressive results for the model that lacks smoothness. However, before the time spent on the LU-factorisation is analysed, implementing this option might not be worth the effort. Moreover, the assumption that the cost of Jacobian-assembly is negligible is easily challenged.

5. **Additive update**: This Broyden-like update is completely untested since there are no obvious ways on how to estimate the speed gain. The only real way to assess its usefulness is likely just to implement it. This is not worth the risk of it being a failure.

6.3 – Further research

This project begs for a follow-up due to its exploratory nature. Moreover, some issues were merely remarked upon and left open. A comprehensive list of directions to take for future research is the following

- Implementing one or more of the five options provided. Not only does this improve the integrator but it can also serve as a validation of the work in this thesis.

- Analyse the cost of the LU-factorisation to complete the speed gain estimates in *Chapter* 4.

- Understand the characteristics of the matrices L and U. How do they drive the optimal value for the drop tolerance?
- How much time do we loose on the additional bookkeeping that is required for the three-phase cycle? Is it indeed negligible?
- Theoretically formalise the estimates and provide proper bounds. Most of the estimates are rather heuristic and applied. A more theoretical follow-up could be made to explore some of the inequalities in more detail.
- Other parts of DASSL: Not only on the Newton iterations is a potential for speed-up. Especially when using a general purpose integrator for a specific set of equations there are many more possibilities to fine-tune the design of DASSL. Think of:
  - The various units of the problem (like position, speed and force) are completely disregarded. With a proper handling of them, we might benefit from additional physical knowledge.
  - The Jacobian rejection policy might need to be revisited. The shortcomings of the hard-coded threshold were already remarked upon in *Section* 3.3.2.
- Many models, or parts thereof, show periodic behaviour. For example, the chain model has a long period equal to the complete rotation of the chain and a short period equal to the displacement of link to the position of its neighbour. Can we make DASSL recognise these to obtain a better initial guess, for example? Is a spectral analysis worthwile, for example to synchronise the refresh cycles with the natural frequencies of the model?
- The demand for real-time solvers is growing. How does this work fit in that branch? Is the parameter optimisation from *Section* 4.3 an option? Do we have time for the bookkeeping in between cycle phases?
- How does the idea of the multi-phase cycle translate to other similar time-integrators?
- The simplified method lends itself nicely to computational parallelisation. One core can be working on the Newton iterations with the dated Jacobian whilst another is factorising a more recent Jacobian. How does the three-phase cycle fit in a parallel solution? Is it still worthwhile?

Appendix A – Contraction mappings

Definition A.1. *A vector valued function $\mu : \mathbb{R}^n \to \mathbb{R}^n$ is called a **contraction** in an open subset $\Omega \subset \mathbb{R}^n$ if*

$$\exists \tilde{\kappa} \in [0,1) \; \forall x, y \in \Omega : \|\mu(x) - \mu(y)\| \leq \tilde{\kappa} \|x - y\| . \tag{A.1}$$

*The **contraction factor** κ is the smallest of all such $\tilde{\kappa}$.*

Theorem A.1. *Let $\mu : \mathbb{R}^n \to \mathbb{R}^n$ be differentiable. μ is a contraction in the neighbourhood of x_0 if and only if*

$$\rho\left(\partial\mu(x_0)\right) < 1 , \tag{A.2}$$

where $\rho(\cdot)$ indicates the spectral radius of a square matrix and $\partial\mu$ the total derivative of μ. This inequality is sharp.

Proof. ($\Rightarrow$) μ is a contraction around x_0 so we have, for every unit vector u and for h sufficiently small

$$\|\mu(x_0) - \mu(x_0 + hu)\| \leq \kappa |h| < |h| . \tag{A.3}$$

Now pick u the eigenvector of $\partial\mu$ corresponding to the biggest eigenvalue. Then

$$1 > \frac{\|\mu(x_0) - \mu(x_0 + hu)\|}{h} = \frac{\|h\partial\mu u + \mathcal{O}(h^2)\|}{h} = \|\partial\mu u\| + \mathcal{O}(h) = \rho(\partial\mu) \text{ as } h \to 0 , \tag{A.4}$$

($\Leftarrow$) Consider a vector $x_0 + y$ near x_0. Write y as a linear combination of eigenvectors e_i and a kernel-vector d of $\partial\mu$, so $y = \sum \beta_i e_i + d$. Then

$$\|\mu(x_0) - \mu(x_0 + y)\| = \left\|\partial\mu y + \mathcal{O}(\|y\|^2)\right\| = \left\|\partial\mu\left(\sum \beta_i e_i + d\right) + \mathcal{O}\right\| \tag{A.5}$$

$$= \left\|\sum \beta_i \lambda_i e_i + \mathcal{O}(\|y\|^2)\right\| \leq \sum \left|\beta_i \lambda_i\right| + \mathcal{O}(\|y\|^2) \tag{A.6}$$

$$\leq \rho(\partial\mu)\|y\| + \mathcal{O}(\|y\|^2) , \tag{A.7}$$

which, given that $\rho(\partial\mu) < 1$, implies that μ is a contraction directly around x_0 (i.e. $\|y\| \to 0$). When choosing y equal to the eigenvector with the biggest eigenvalue all inequalities are sharp and the contraction factor κ will equal the spectral radius. □

Appendix B – Manifolds

The following material is more extensively treated by Hairer and Wanner in [12] for our application. A truly fundamental treatise is found in the book by Lee[16].

Given a smooth function $\Phi : \mathbb{R}^n \to \mathbb{R}^m$ with $m < n$, we can have it generate a differentiable manifold by defining the set $\mathcal{M} = \{y \in \mathbb{R}^n \mid \Phi(y) = 0\}$. This is a surface in n-dimensional Euclidean space. To define a differential equation on this manifold, a proper definition for derivatives is needed on this curved space. This is accomplished by the concepts of tangent space and charts.

Definition B.1. *The **tangent space** or **linearisation** of a manifold $\mathcal{M} = \{y \in \mathbb{R}^n \mid \Phi(y) = 0\}$ at a given point $y \in \mathcal{M}$ is denoted as and given by*

$$T_y\mathcal{M} = \{v \in \mathbb{R}^n \mid \Phi_x \cdot v = 0 \in \mathbb{R}^m\} . \tag{B.1}$$

*The **perpendicular space** of $\mathcal{M}$ at y is the space perpendicular to the tangent space. It is given by*

$$\left(T_y\mathcal{M}\right)^{\perp} = \{\Phi_x^{\mathsf{T}} \cdot w \in \mathbb{R}^n \mid w \in \mathbb{R}^m\} . \tag{B.2}$$

Definition B.2. *A **chart** for a manifold $\mathcal{M}$ is a smooth, bijective mapping $\tau_i : U_i \to E_i$ where $U_i \in \mathcal{M}$ and $E_i \subset \mathbb{R}^{n-m}$. A collection of charts $\{\tau_i\}$ such that $\bigcup U_i = \mathcal{M}$ is called an **atlas**.*

Consider a regular evolution equation in Euclidean space for the vector $y(t)$. Without loss of generality, the equation can be assumed to be first order by virtue of the order reduction technique from *Section* 2.1.2. So

$$\dot{y} = f(y, t) \quad \text{for } t \in [0, T] \tag{B.3}$$

$$y(0) = y_0 . \tag{B.4}$$

It is useful to think of the curve $y(t)$ as the trajectory of a single particle in space.
Assume now that the solution is known to belong a certain manifold $\mathcal{M}$, generated by Φ. This information can be used to specify the equation further as

$$\dot{y} = f_{\parallel}(y, t) + f_{\perp}(y, t) \tag{B.5}$$

$$\Phi(y, t) = 0 \tag{B.6}$$

$$y(0) = y_0 , \tag{B.7}$$

with $f_{\parallel} : \mathcal{M} \to T_y\mathcal{M}$ and $f_{\perp} : \mathcal{M} \to \left(T_y\mathcal{M}\right)^{\perp}$. That is, the original driving function is decomposed in two parts according to the direct sum $\mathbb{R}^n = T_y\mathcal{M} \oplus \left(T_y\mathcal{M}\right)^{\perp}$. Now $f_{\parallel}$ is responsible for the movement of y over the manifold and $f_{\perp}$ is solely responsible for keeping y on the manifold. Given this meaning of $f_{\perp}$, the second equation might seem redundant. However, the additional modelling assumption that the solution is constraint to the manifold results in an additional unknown. This unknown is the magnitude of $f_{\perp}$.

Since the direction of $f_\perp$ is known to be perpendicular to the manifold, but the magnitude is unknown, we can write

$$f_\perp(y,t) = \Phi_x^\mathsf{T} \cdot \lambda \,, \tag{B.8}$$

with λ the magnitude. Hence the full system can be written as

$$\dot{y} - \Phi_x^\mathsf{T} \cdot \lambda = f_\perp(y,t) \tag{B.9}$$

$$\Phi(y,t) = 0 \tag{B.10}$$

$$y(0) = y_0 \,. \tag{B.11}$$

Now in addition to $y(t)$, an extra unknown $\lambda(t)$ is added. Note, however, that no initial conditions for λ are known. These need to be chosen such that the initial system is consistent. This problem is addressed from both a theoretical and a numerical angle in [4].

Bibliography

[1] C.L. Botasso, D. Dopico, L. Trainelli
On the optimal scaling of index three differential-algebraic equations in multibody dynamics
(Springer, 2007)

[2] K.E. Brenan, S.L. Campbell & L.R. Petzold
Numerical solutions of initial value problems in differential-algebraic equations
(siam, 1996)

[3] P.N. Brown, A.C. Hindmarsh, L.R. Petzold
Using Krylov methods in the solution of large-scale differential-algebraic systems
(siam, 1994)

[4] P.N. Brown, A.C. Hindmarsh, L.R. Petzold
Consistent initial condition calculation for differential-algebraic systems
(siam, 1998)

[5] C.G. Broyden
On the discovery of the “good Broyden” method
(Springer, 2000)

[6] P. Deuflhard
Newton methods for nonlinear problems
(Springer, 2006)

[7] E. Eich-Soellner & C. Führer
Numerical Methods in multibody dynamics
(Lund, 2008)

[8] M. Gérardin & A. Cardona
Flexible multibody dynamics
(Wiley, 2001)

[9] L.Giraud & S.Gratton
Accelerating krylov solvers with low rank corrections
(European conference on CFD, 2006)

[10] W.W. Hager
Updating the inverse of a matrix
(siam, 1989)

[11] E. HAIRER, S. NØRSETT & G. WANNER
Solving Ordinary Differential Equations Vol. I
(Springer, 2000)

[12] E. HAIRER & G. WANNER
Solving Ordinary Differential Equations Vol. II
(Springer, 2002)

[13] E.J. HAUG
Computer Aided kinematics and dynamics of mechanical systems
(Allyn & Bacon, 1989)

[14] E.M. JONES & P. FJELD
Gimbal Angles, Gimbal Lock, and a Fourth Gimbal for Christmas
(retrieved from: `http://www.hq.nasa.gov/alsj/gimbals.html` in July 2013)

[15] H. KARABULUT
The physical meaning of Lagrange multipliers
(European journal of physics, 2006)

[16] J.M. LEE
Introduction to smooth manifolds
(Springer, 2002)

[17] L.R. PETZOLD
A description of DASSL: A differential/algebraic system solver
(IMACS World Congress, Montreal Canada, 1982)

[18] L.R. PETZOLD
Source code of DASSL
(last update: 2000, from: `http://www.netlib.org/ode/ddassl.f`)

[19] Y. SAAD
Iterative methods for sparse linear systems
(SIAM, 2003)

[20] Y. SAAD
ILUT: A dual threshold incomplete LU-factorization
(Numerical linear algebra with applications, Vol. 1, Issue 4, 1994)

[21] N. V.D. WOUW
Lecture notes multibody dynamics (4J400)
(Eindhoven University for Technology, 2012)