NUMERICAL INTEGRATION OF STIFF DIFFERENTIAL-ALGEBRAIC EQUATIONS

NUMERICAL INTEGRATION OF STIFF DIFFERENTIAL-ALGEBRAIC EQUATIONS

By

REZA ZOLFAGHARI

A Thesis Submitted to the School of Graduate Studies

in Partial Fulfillment of the Requirements for the Degree

Doctor of Philosophy

McMaster University (School of Computational Science and Engineering)

Doctor of Philosophy (2020), Hamilton, Ontario

TITLE:	Numerical Integration of Stiff Differential-Algebraic Equations
AUTHOR:	Reza Zolfaghari
SUPERVISOR:	Professor Nedialko. S. Nedialkov Department of Computing and Software McMaster University
NUMBER OF PAGES:	ix, 268

Abstract

Systems of differential-algebraic equations (DAEs) arise in many areas including electrical circuit simulation, chemical engineering, and robotics. The difficulty of solving a DAE is characterized by its index. For index-1 DAEs, there are several solvers, while a high-index DAE is numerically much more difficult to solve. The DAETS solver by Nedialkov and Pryce integrates numerically high-index DAEs. This solver is based on explicit Taylor series method and is efficient on non-stiff to mildly stiff problems, but can have severe stepsize restrictions on highly stiff problems.

Hermite-Obreschkoff (HO) methods can be viewed as a generalization of Taylor series methods. The former have smaller error than the latter and can be A- or L- stable. In this thesis, we develop an implicit HO method for numerical solution of stiff high-index DAEs. Our method reduces a given DAE to a system of generally nonlinear equations and a constrained optimization problem. We employ Pryce's structural analysis to determine the constraints of the problem and to organize the computations of higher-order Taylor coefficients (TCs) and their gradients. Then, we use automatic differentiation to compute these TCs and gradients, which are needed for evaluating the resulting system and its Jacobian.

We design an adaptive variable-stepsize and variable-order algorithm and implement it in C++ using literate programming. The theory and implementation are interwoven in this thesis, which can be verified for correctness by a human expert. We report numerical results on stiff DAEs illustrating the accuracy and performance of our method, and in particular, its ability to take large steps on stiff problems.

Acknowledgements

This thesis would not have been possible without the expertise and guidance of my supervisor Dr. Ned Nedialkov. He shared with me his profound knowledge about design and evaluation of a numerical software for differential equations. He offered me invaluable suggestions and resources to improve my programming skills and technical writing and spent incredibly many hours on reading and commenting on my codes, writing and slides. I am and have been extremely grateful for his constant support.

I would like to thank my supervisory committee members, Dr. Bartosz Protas (Mathematics & Statistics) and Dr. Tim Field (Electrical & Computer Engineering), for their continuous advice and help. I thank Dr. Wayne Enright and Dr. Kenneth R. Jackson (Department of Computer Science, University of Toronto) for giving me the opportunity to present my work at University of Toronto and for valuable hints and discussions. I wish to thank Dr. Andreas Griewank (Institute of Mathematics, Humboldt-Universität zu Berlin) for being my external examiner. His constructive comments greatly improved the thesis.

To my parents, thank you for inspiring me and loving me without ceasing. To my wife, thank you for standing by me through thick and thin. To my daughter, you have been a source of joy and encouragement, thank you.

Finally, I gratefully acknowledge the support of the Ontario Trillium Scholarship.

Contents

List of Figures

Chapter 1

Introduction

Many dynamical systems are modelled as differential-algebraic equations (DAEs). In this work, we consider the general formulation[1]

$$f_i(t, \text{ the } x_j \text{ and derivatives of them}) = 0, \qquad i = 0, 1, \ldots, n-1, \tag{1.1}$$

where $x_j(t)$, $j = 0, 1, \ldots, n-1$, are state variables, and t is the time variable. The formulation (1.1) may include high-order derivatives and equations that are jointly nonlinear in leading derivatives. Assuming that the functions f_i and x_j are sufficiently differentiable, we develop a numerical method for (1.1).

The difficulty of solving a DAE is characterized by its *index*. Numerical methods for DAEs have been studied by different groups in mathematics, computer science and engineering. There are various definitions of the index in the literature: differentiation index [11], perturbation index [28], tractability index [24], strangeness index [34], structural index

[1]In order to match the theory and implementation that follows, we start the indexing from 0 as in C/C++.

[17], and geometric index [56]. A DAE of index ≥ 2 is considered *high-index* DAE. For analysis and comparison of various index concepts see [12, 22, 23, 35, 39, 54, 55] .

The difficulty of solving a DAE is mainly due to *hidden constraints* that are not explicitly given in the system. If these constraints are not forced during the integration, the computed solution may drift off from the true solution due to numerical errors [35, p. 217].

Numerical methods for DAEs can be divided into two classes [4, p. 261]:

1. methods based on direct discretizations of the given DAE, e.g., backward differentiation formula (BDF) for index-1 DAEs [7, 21], and Radau methods for some special index ≤ 3 DAEs [28].

2. methods that involve an index reduction prior to a discretization, e.g., stabilization techniques [19, 28, 61], projection methods [2, 3], and the differential-geometric approach [53].

When a dynamical system is modelled as (1.1), the state variables usually have a physical significance. Changing the system may produce less meaningful variables. Using a solver based on a direct discretization of the original DAE enables a scientist or engineer to explore easier the effect of modelling changes and parameter variation [7, p. 2]. Furthermore, an index reduction method may be costly and involve user intervention [4, p. 261]. It also can destroy sparsity and prevent the exploitation of the system's structure [7, p. 2].

1.1 Motivation

For index-1 DAEs, there are several solvers, e.g., DASSL [7], IDA of SUNDIALS [31], and MATLAB's ode15s and ode23t. However, a high-index DAE is numerically much more difficult to solve. Basically, solving a high-index problem needs differentiation, instead of integration only. High-index DAE solvers such as RADAU5 [28], MEBDFDAE [13], BIMD [8] and PSIDE [65] solve some DAEs of index ≤ 3.

DAETS by Nedialkov and Pryce [43, 45–47] is a powerful tool for non-stiff high-index DAEs. It uses the Pryce's structural analysis (Σ-method) [54] to analyze a DAE and solve it numerically by expanding the solution in Taylor series (TS) at each integration step. DAETS is suitable for non-stiff to mildly stiff problems, and is not efficient on very stiff systems. This thesis focuses on developing a method suitable for solving stiff DAEs.

Stiff problems are problems for which certain implicit numerical integration methods perform tremendously better than explicit ones [28]. Such problems arise in the study of atmospheric phenomena, chemical kinetics, chemical reactions occurring in living species, electronic circuits, mechanics, and molecular dynamics [1]. Using an explicit Taylor series method, DAETS cannot be very efficient for highly stiff DAEs. A promising approach is to develop an implicit Hermite-Obreschkoff (HO) method, which is known to have much better stability, in the ODE sense, than Taylor series methods.

Derivation and properties of HO methods for first-order ODEs are discussed in [14, 25] and [36, p. 199]. These methods have recently been applied to systems arising from electrical circuits [20, 67] and the forward-dynamics problem of a single-loop kinematic chain [40].

1.2 Contributions

The following are the main contributions of this thesis.

- Using the HO formula for some derivatives of state variables x_j in (1.1), determined by the Pryce's Σ-method, we develop an HO method which reduces the DAE (1.1) to a system of equations and a constrained optimization problem. The proposed method can be A- or L- stable in ODE sense.

- We employ the Σ-method to organize the computations of gradients of higher-order Taylor coefficients in terms of independent ones. Then, we use automatic differentiation to compute these gradients for constructing the Jacobian of the reduced system of equations required by Newton's method.

- We define a specially tailored vector for a function at a point. Constructing this vector for solution components enables us to find an initial guess for the solution and to estimate the discretization error of the HO method with different orders. As a result, we design an adaptive variable-stepsize and variable-order algorithm for integrating the problem.

- We implement our algorithm in C++ using literate programming. The theory, documentation, and source code are interwoven in this thesis, which can be verified for correctness by a human expert, like in a peer-review process.

1.3 Thesis organization

The rest of this thesis is organized as follows.

Chapter 2 gives an overview of relevant concepts and background.

In Chapter 3, we derive the HO formula and employ it for the numerical solution of an ODE of a general form.

In Chapter 4, we develop an HO method for the DAE (1.1) using the HO formula and Pryce's structural analysis.

Chapter 5 describes and implements the computations of higher-order TCs as solution components are given.

Chapter 6 derives and implements the computations of gradients of higher-order TCs in terms of solution components.

In Chapter 7, we solve the resulting nonlinear system using a modified Newton iteration and implement the HO method for one step.

In Chapter 8, we first define and construct Hermite-Nordsieck vectors for solution components. Then, we show how these vectors are employed to predict a solution and to estimate the discretization error. Finally, stepsize and order selection strategies are derived.

Chapter 9 describes the components of the algorithm and implements the integrator function.

In Chapter 10, we first give an example of coding a function defining a DAE, and a main program to solve the problem with DAETS. Then, we report numerical results on stiff ODEs and DAEs.

Chapter 11 gives concluding remarks.

Chapter 2

Background

In this chapter, we present the background needed for the remaining of this thesis. §2.1 gives a brief overview of stability and stiffness concepts. §2.2 summarizes the main steps of Pryce's structural analysis. §2.3 presents some basic concepts of automatic differentiation. §2.4 discusses literate programming. Finally, §2.5 gives an overview of DAETS.

2.1 Stability and stiffness

The stability of an integration method for initial-value problems in ordinary differential equations (ODEs) is typically studied on the test problem

$$y' = \lambda y, \qquad y(0) = y_0,$$

where $\text{Re}(\lambda) < 0$. Since the exact solution $y(t) = y_0 e^{\lambda t}$ decays exponentially, a stepsize h must be selected such that the sequence of numerical approximations $y_i \approx y(ih)$, $i = 1, 2, \dots$

is monotonically decreasing. A region in the complex z-plane (with $z = \lambda h$) for which a numerical method preserves this property is called the *stability domain* of the method [28, p. 16].

If we use an integration method with a small stability domain, we may need to choose a very small stepsize h to maintain stability. Therefore, a desirable property for a numerical integrator is to have a large stability domain.

An integration method is *A-stable* if its stability domain contains the entire left half z-plane [16]. A weakness of the A-stability definition is that it does not distinguish [4, p. 56] between

$$\text{Re}(\lambda) \to -\infty,$$

and

$$-1 \ll \text{Re}(\lambda) \leq 0, \quad |\text{Im}(\lambda)| \to \infty.$$

Another weakness of A-stability definition is that if $\text{Re}(\lambda) \to -\infty$ and h is not small, the numerical solutions y_i may decay to zero very slowly [36, p. 236]. This leads us to a stronger concept of stability. A method is said to be *L-stable* [18] if it is A-stable and

$$\frac{y_{i+1}}{y_i} \to 0 \text{ as } \lambda h \to -\infty.$$

Consider a system of ordinary differential equations of the form

$$\mathbf{y}' = \mathbf{f}(t, \mathbf{y}). \tag{2.1}$$

Suppose that all eigenvalues λ_i of the Jacobian matrix $\partial \mathbf{f} / \partial \mathbf{y}$ are in the left half z-plane so that the ratio $\max_i |\lambda_i| \, / \min_i |\lambda_i|$ is large. Hence, the solution of the system (2.1) contains very fast components as well as very slow components [15]. If an integration method with small stability domain, as in many explicit methods, is used for solving such problem, it might become very inefficient. In this case, the system (2.1) is called *stiff* [5, p. 130]. Although the eigenvalues of $\partial \mathbf{f}/\partial \mathbf{y}$ play central role in the stiffness of (2.1), the size of the system, the smoothness of the solution and the integration interval are also important [28, p. 1].

Ideally, one would like to use an A-stable (or L-stable) method to solve stiff problems. However, it is known that no linear multistep formula of order greater than 2 can be A-stable [16]. As a result of this barrier, one could use so-called $A(\alpha)$-*stable* methods, $\alpha \in (0, \pi/2)$, for which the sector

$$\mathbb{C}_\alpha = \{\lambda h \in \mathbb{C} : |\arg(-\lambda)| < \alpha,\ \ \lambda \neq 0\},$$

is contained in the stability domain [66]. Prime example of these methods are BDF methods of orders 1 to 5.

Explicit Taylor series methods of order κ for $\kappa \to \infty$ are A-stable, i.e., the size of the stability domain grows linearly with κ in its radius [6]. Hence, a high-order method can be used on moderately stiff problems [6, 45].

2.2 Pryce's structural analysis

Before a numerical method is applied to a DAE, some kind of structural analysis is necessary to determine DAE's structure and index. Among structural analysis methods, Pantelides's algorithm [51] is widely used. Pryce's Σ-method [54] is becoming increasingly popular [38,45,48,57,58,68] due to its capability of analyzing high-order systems. In this section, we review this method for a DAE in the form of (1.1).

The Σ-method for (1.1) consists of the following steps.

1. Build the signature matrix $\mathbf{\Sigma} = (\sigma_{ij})$, where

$$\sigma_{ij} = \begin{cases} \text{order of the highest order derivative to which } x_j \text{ occurs in } f_i; \text{ or} \\ -\infty \qquad \text{if } x_j \text{ does not occur in } f_i. \end{cases}$$

2. Find a highest value transversal (HVT) of $\mathbf{\Sigma}$. A transversal T of $\mathbf{\Sigma}$ is a set of n positions (i, j) with one entry in each row and each column. We seek a transversal with the maximal value

$$\text{Val}(\mathbf{\Sigma}) = \sum_{(i,j)\in T} \sigma_{ij}.$$

3. Compute n-dimensional non-negative integer vectors $\mathbf{c}$ and $\mathbf{d}$ that satisfy

$$\begin{aligned} d_j - c_i &\geq \sigma_{ij}, && \text{for all } i, j = 0, 1, ..., n-1, \text{ and} \\ d_j - c_i &= \sigma_{ij}, && \text{for all } (i, j) \in \text{HVT}. \end{aligned}$$

   Vectors $\mathbf{c}$ and $\mathbf{d}$ are referred to as the offsets of the problem. They are not unique, but we choose the smallest or canonical offsets; smallest being in the sense of $a \leq b$ if $a_i \leq b_i$ for all i.

4. Form the system Jacobian matrix $\mathbf{J}$ with

$$(\mathbf{J})_{ij} = \frac{\partial f_i^{(c_i)}}{\partial x_j^{(d_j)}} = \begin{cases} \dfrac{\partial f_i}{\partial x_j^{(\sigma_{ij})}}, & \text{if } d_j - c_i = \sigma_{ij}, \text{ and} \\ 0 & \text{otherwise.} \end{cases} \tag{2.3}$$

5. Seek values for the x_j and for appropriate derivatives, consistent with the DAE, and at which $\mathbf{J}$ is nonsingular. If such values are found, we say the method "succeeds" and there is locally a unique solution of the DAE.

When the method succeeds:

- Val$(\boldsymbol{\Sigma})$ equals the number of degrees of freedom (DOF) of the DAE, that is the number of independent initial conditions required.
- The *structural index* is defined by

$$\nu_s = \max_i c_i + \begin{cases} 1 & \text{if some } d_j \text{ is } 0, \\ 0 & \text{otherwise.} \end{cases}$$

  The structural index is an upper bound for differentiation index, which is the minimum number of differentiations needed to reduce a DAE to a system of ODEs.

We illustrate the above concepts using the following example.

Example 2.1. The simple pendulum DAE in Cartesian coordinates is

$$\begin{aligned} 0 &= f = x'' + \lambda x, \\ 0 &= g = y'' + \lambda y + G, \\ 0 &= h = x^2 + y^2 - L^2. \end{aligned} \tag{2.4}$$

Here the state variables are x, y, λ; G is gravity, and L is the length of the pendulum.

The signature matrix of this DAE is

$$\mathbf{\Sigma} = \begin{array}{c|ccc|c} & x & y & \lambda & c_i \\ \hline f & 2^\bullet & -\infty & 0 & 0 \\ g & -\infty & 2 & 0^\bullet & 0 \\ h & 0 & 0^\bullet & -\infty & 2 \\ \hline d_j & 2 & 2 & 0 & \end{array},$$

where an HVT is marked by $\bullet$. The system Jacobian matrix (2.3) is

$$\mathbf{J} = \begin{bmatrix} \partial f/\partial x'' & 0 & \partial f/\partial\lambda \\ 0 & \partial g/\partial y'' & \partial g/\partial\lambda \\ \partial h/\partial x & \partial h/\partial y & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 2x & 2y & 0 \end{bmatrix}, \tag{2.5}$$

which is nonsingular $(\det(\mathbf{J}) = -2(x^2+y^2) = -2L^2 \neq 0)$ and the method succeeds. The structural index is

$$\nu_s = \max_i c_i + 1 = c_2 + 1 = 3,$$

which is the same as the differentiation index.

Definition 2.1. The DAE (1.1) is called quasilinear if $x_j^{(d_j)}$, $j = 0, 1, \ldots, n-1$, occur in a jointly linear way in the f_i, $i = 0, 1, \ldots, n-1$, and non-quasilinear otherwise [47].

Example 2.2. For (2.4), the relevant derivatives $x_j^{(d_j)}$ are x'', y'' and λ. Their occurrence is jointly linear in f, g and h, so the system is quasilinear. It would still be quasilinear, if f were changed to $x''x + x\lambda$, $x''x' + x\lambda$, $x''y + x\lambda$, or to $x''y' + x\lambda$. However, it would be non-quasilinear if it were changed to $(x'')^2 + x\lambda$, $x''y'' + x\lambda$, or to $x''\lambda + x\lambda$.

2.3 Automatic differentiation

Automatic differentiation (AD) is the process of differentiating a computer program based on the chain rule [29]. Suppose that a computer program $\mathcal{P}$ computes a differentiable function $\boldsymbol{\mathcal{F}} : \mathbb{R}^n \to \mathbb{R}^m$ with runtime T. This program can be implemented as a sequence of instructions $\{I_1; I_2; \ldots; I_r\}$ computing the elementary differentiable functions $\mathbf{f}_i : \mathbb{R}^{n_{i-1}} \to \mathbb{R}^{n_i}$, $i = 1, \ldots, r$, with $n_0 = n$ and $n_r = m$, such that

$$\mathbf{y} = \boldsymbol{\mathcal{F}}(\mathbf{x}) = \mathbf{f}_r(\ldots(\mathbf{f}_2(\mathbf{f}_1(\mathbf{x})))\ldots), \quad \mathbf{x} \in \mathbb{R}^n. \tag{2.6}$$

That is,

$$I_1 \text{ computes } \mathbf{y}_1 = \mathbf{f}_1(\mathbf{x}),$$

$$I_i \text{ computes } \mathbf{y}_i = \mathbf{f}_i(\mathbf{y}_{i-1}), \ \text{ for } \ i = 2, \ldots, r,$$

and we obtain $\mathbf{y} = \mathbf{y}_r \in \mathbb{R}^m$.

Applying the chain rule to (2.6) gives

$$\boldsymbol{\mathcal{F}}'(\mathbf{x}) = \mathbf{f}_r'(\mathbf{y}_{r-1}) \cdots \mathbf{f}_2'(\mathbf{y}_1)\mathbf{f}_1'(\mathbf{x}).$$

To avoid the above matrix-matrix products, we may create a new program with runtime T that computes one of the following matrix-vector products

$$\boldsymbol{\mathcal{F}}'(\mathbf{x})\,\dot{\mathbf{x}}, \quad \text{or} \quad \overline{\mathbf{y}}\boldsymbol{\mathcal{F}}'(\mathbf{x}),$$

with some *seed* vectors $\dot{\mathbf{x}} \in \mathbb{R}^{n\times 1}$ and $\overline{\mathbf{y}} \in \mathbb{R}^{1\times m}$.

2.3.1 Forward mode

In the forward or tangent mode, we compute

$$\dot{\mathbf{y}} = \boldsymbol{\mathcal{F}}'(\mathbf{x})\dot{\mathbf{x}} = \mathbf{f}_r'(\mathbf{y}_{r-1}) \cdots \mathbf{f}_2'(\mathbf{y}_1)\mathbf{f}_1'(\mathbf{x})\dot{\mathbf{x}}.$$

The program $\mathcal{P}'$, referred to as *tangent program*, is created as the sequence of instructions $\{I_1'; I_1; I_2'; I_2; \dots; I_r'\}$ where

I_1' computes $\dot{\mathbf{y}}_1 = \mathbf{f}_1'(\mathbf{x})\dot{\mathbf{x}}$,

I_i' computes $\dot{\mathbf{y}}_i = \mathbf{f}_i'(\mathbf{y}_{i-1})\dot{\mathbf{y}}_{i-1}$, for $i = 2, \dots, r$.

The variables $\dot{\mathbf{y}}_i \in \mathbb{R}^{n_i \times 1}$, $i = 1, \dots, r$, are called *tangent variables*, and we obtain $\dot{\mathbf{y}} = \dot{\mathbf{y}}_r \in \mathbb{R}^{m \times 1}$.

Since the $\mathbf{f}_i$, $i = 1, \dots, r$, in (2.6) are elementary functions, the Jacobian matrices $\mathbf{f}_i'$, $i = 1, \dots, r$, are very sparse and differ from the identity only in a few positions. To obtain $\boldsymbol{\mathcal{F}}'(\mathbf{x})$, we can repeatedly call the tangent program using the Cartesian basis vectors in $\mathbb{R}^n$ as seeds. This yields the complete Jacobian in a runtime of $\mathcal{O}(nT)$.

2.3.2 Reverse mode

In the reverse or adjoint mode, we compute

$$\overline{\mathbf{x}} = \overline{\mathbf{y}}\boldsymbol{\mathcal{F}}'(\mathbf{x}) = \overline{\mathbf{y}}\mathbf{f}_r'(\mathbf{y}_{r-1}) \cdots \mathbf{f}_2'(\mathbf{y}_1)\mathbf{f}_1'(\mathbf{x}).$$

The program $\overline{\mathcal{P}}$, referred to as *adjoint program*, is created as the sequence of instructions $\{I_1; I_2; \dots; I_r; \bar{I}_r; \bar{I}_{r-1}; \dots; \bar{I}_1\}$, where

$\bar{I}_r$ computes $\overline{\mathbf{x}}_r = \overline{\mathbf{y}}\mathbf{f}_r'(\mathbf{y}_{r-1})$,

$\bar{I}_i$ computes $\overline{\mathbf{x}}_i = \overline{\mathbf{x}}_{i+1}\mathbf{f}_i'(\mathbf{y}_{i-1})$, for $i = r-1, r-2, \dots, 2$,

$\bar{I}_1$ computes $\overline{\mathbf{x}}_1 = \overline{\mathbf{x}}_2\mathbf{f}_i'(\mathbf{x})$.

The variables $\overline{\mathbf{x}}_i \in \mathbb{R}^{1\times(n_{i-1})}$, $i = 1, \dots, r$, are called *adjoint variables* and we obtain $\overline{\mathbf{x}} = \overline{\mathbf{x}}_1 \in \mathbb{R}^{1\times n}$. By calling the adjoint program repeatedly with all Cartesian basis vectors in $\mathbb{R}^m$, the Jacobian $\boldsymbol{\mathcal{F}}'(\mathbf{x})$ can be computed with runtime $\mathcal{O}(mT)$. Therefore, reverse methods can greatly reduce the computational cost if $m \ll n$.

2.3.3 Taylor coefficients

TCs of a sufficiently differentiable function can be generated automatically by formulas developed by Moore [42, p.107–130]. Denote the kth TC of a function u at a point a by

$$u_k = \frac{u^{(k)}(a)}{k!}.$$

If sufficient TCs of functions u and v at a are given, we can compute the kth TC of a function $w(t) = f\Big(u(t), v(t)\Big)$ at a using classical rules for automatic differentiation of arithmetic operations and elementary functions. For example,

$$w = u + cv \quad \Rightarrow \quad w_k = u_k + cv_k, \quad c \text{ is a constant,} \tag{2.7}$$

$$w = uv \quad \Rightarrow \quad w_k = \sum_{r=0}^{k} u_r v_{k-r}, \tag{2.8}$$

$$w = u/v \quad \Rightarrow \quad w_k = \frac{1}{v_0}\left[u_k - \sum_{r=0}^{k-1} w_r v_{k-r}\right], \quad v_0 \neq 0,$$

$$w = \sqrt{u} \quad \Rightarrow \quad w_0 = \sqrt{u_0},\ w_k = \frac{1}{2w_0}\left[u_k - \sum_{r=1}^{k-1} w_r w_{k-r}\right],\ k \geq 1,$$

$$w = \exp(u) \quad \Rightarrow \quad w_0 = \exp(u_0),\ w_k = \frac{1}{k}\sum_{r=0}^{k-1}(k-r)w_r u_{k-r},\ k \geq 1.$$

Similar formulas can be derived for other elementary functions, e.g., $\sin$, $\cos$, $\log, \ldots$ [26, Chapter 10]. We also use

$$w = u^{(m)} \quad \Rightarrow \quad w_k = (k+1)(k+2)\cdots(k+m)u_{k+m}. \tag{2.9}$$

Consider a function $w(t) = f\big(\mathbf{u}(t)\big)$, with $\mathbf{u} : \mathbb{R} \to \mathbb{R}^n$. The computational complexity of evaluating $w_0, w_1, \ldots, w_k$ is [42]

- $\mathcal{O}(k)$, if f is linear, or
- $\mathcal{O}(sk^2)$, if f involves s multiplications, divisions, and/or elementary functions.

Packages for generating TCs include ADOL-C [27] and FADBAD++ [63].

2.3.4 The FADBAD++ package

We compute required TCs and their gradients through operator overloading. This approach is carried out using FADBAD++ developed by Stauning and Bendtsen [63]. This package contains C++ templates and works by overloading arithmetic operations and elementary functions to include calculation of derivatives. To enable these overloaded operations the arithmetic type (normally double) is changed to the appropriate AD-type. The AD-types are defined by three templates F<> for forward mode, B<> for reverse mode and T<> for Taylor coefficients. A unique feature of FADBAD++ is the ability to compute high order derivatives in a flexible way by combining the methods of automatic differentiation. These combinations are produced by applying the templates on themselves. For example the combination T< F<double> > can be used to compute gradients of Taylor coefficients [46].

2.4 Literate programming

Literate programming was introduced by Donald Knuth [32] in the early 1980s based on the idea that [52] "programs should be written more for people's consumption than for computers' consumption". In a literate program, documentation and code are in one source. Then literate programming tools either *tangle* the program to produce a source code suitable for compilation, or *weave* it to produce a document suitable for typesetting.

A literate program is presented in a form that enhances the readability of code [62]. An algorithm is decomposed into smaller parts and explained in that order is most appropriate to aid comprehension. Each named block of code is called a *chunk* or *section*, and each chunk can refer to other chunks by name. The description of a chunk is as important as its code, encouraging careful design and documentation [52].

This thesis is a literate program using CWEB [33] and its *ctangle* and *cweave* utilities. This literate programming tool enables the inclusion of documentation and C++ code in a

CWEB file, which is a LATEX file with additional statements for dealing with C++ code [44].

2.5 The DAETS solver

The DAETS (DAE by Taylor Series) solver [43, 45–47] accepts a DAE of the general form (1.1) where the functions f_i are sufficiently differentiable. It first employs the Σ-method to analyze the DAE and prescribe a stage by stage solution scheme. This scheme indicates at each stage which equations need to be solved and for which Taylor coefficients (TCs) for the solution. Then, it computes these TCs up to some order using automatic differentiation and expands the solution in a Taylor series. This solver is implemented as a collection of C++ classes in the variable-stepsize and fixed-order mode where a typical order is in the range $12 - 20$.

Chapter 3

An Hermite-Obreschkoff method for ODEs

Obreschkoff [50] developed in 1940 a quadrature formula that utilizes derivatives of the integrand up to any order at two points. It appears that Milne [41] was the first to advocate the use of this formula for the numerical solution of ordinary differential equations [36]. Quadrature formulae involving higher-order derivatives go back to Hermite in 1912 [30]. Obreschkoff's formula can be derived using an Hermite polynomial that interpolates the integrand and a certain number of its derivatives at two points. For these reasons, we refer to methods based on such formulae as Hermite-Obreschkoff (HO) methods.

We derive the HO formula that determines the relation between Taylor coefficients of the kth derivative of a sufficiently differentiable function at two points, §3.1. Then, we develop a numerical method for an ODE of the general form (implicit and any order), §3.2.

3.1 Hermite-Obreschkoff formula

In this section, we derive a relation between Taylor coefficients of a sufficiently differentiable function at two points. For arbitrary non-negative integers p and q, and for a scalar function $y \in C^{p+q+1}[a, b]$, consider the identity

$$y(b) - y(a) = \int_a^b y'(t)dt. \tag{3.1}$$

Denoting $h = b - a$ and

$$g(s) = y'(a + sh), \qquad 0 \le s \le 1,$$

we have

$$\int_a^b y'(t)dt = h \int_0^1 g(s)ds. \tag{3.2}$$

Following the idea in [14], we approximate $g(s)$ with an Hermite interpolating polynomial that interpolates $g(s)$ and a certain number of its derivatives at the two endpoints a and b. Let Π_n be the set of all polynomials whose degrees do not exceed n. There is a unique interpolating polynomial $P(s) \in \Pi_{p+q-1}$ [64, p. 52], such that

$$\begin{aligned} P^{(j)}(0) &= g^{(j)}(0) = h^j y^{(j+1)}(a), && j = 0, 1, \ldots, p-1, \quad \text{and} \\ P^{(j)}(1) &= g^{(j)}(1) = h^j y^{(j+1)}(b), && j = 0, 1, \ldots, q-1. \end{aligned} \tag{3.3}$$

The Lagrangian representation of P has the form [64, p. 52]

$$P(s) = \sum_{j=0}^{p-1} h^j y^{(j+1)}(a) L_j^{pq}(s) + \sum_{j=0}^{q-1} h^j y^{(j+1)}(b) L_j^{qp}(s), \tag{3.4}$$

with generalized Lagrange polynomials $L_j^{pq}, L_j^{qp} \in \Pi_{p+q-1}$ defined as follows. Denote

$$l_j^{pq}(s) = \frac{s^j(1-s)^q}{j!}, \qquad j = 0, 1, \dots, p-1, \quad \text{and}$$
$$l_j^{qp}(s) = \frac{s^p(s-1)^j}{j!}, \qquad j = 0, 1, \dots, q-1.$$

Then

$$L_{p-1}^{pq}(s) = l_{p-1}^{pq}(s), \qquad L_{q-1}^{qp}(s) = l_{q-1}^{qp}(s),$$
$$L_j^{pq}(s) = l_j^{pq}(s) - \sum_{i=j+1}^{p-1} \frac{d^i}{ds^i} l_j^{pq}(s)\Big|_{s=0} L_i^{pq}(s), \qquad j = p-2, p-3, \dots, 0, \quad \text{and}$$
$$L_j^{qp}(s) = l_j^{qp}(s) - \sum_{i=j+1}^{q-1} \frac{d^i}{ds^i} l_j^{qp}(s)\Big|_{s=1} L_i^{qp}(s), \qquad j = q-2, q-3, \dots, 0.$$

By induction

$$\frac{d^m}{ds^m} L_j^{pq}(s)|_{s=0} = \begin{cases} 1 & \text{if } j = m, \\ 0 & \text{otherwise,} \end{cases}$$

and

$$\frac{d^m}{ds^m} L_j^{qp}(s)|_{s=1} = \begin{cases} 1 & \text{if } j = m, \\ 0 & \text{otherwise,} \end{cases}$$

which show that (3.4) satisfies in (3.3).

Using the convergence theorem of Hermite interpolation [64, p. 57], there exists $\zeta(s) \in [0, 1]$ such that

$$g(s) = P(s) + \frac{s^p(s-1)^q}{(p+q)!} g^{(p+q)}\big(\zeta(s)\big). \tag{3.5}$$

Denote

$$Q(s) = (p+q)!\, \frac{g(s) - P(s)}{s^p(s-1)^q},$$

which is a continuous function on the interval $(0,1)$. By (3.3) and (3.5) we can assume, without loss of generality, that

$$g^{(p+q)}\big(\zeta(s)\big)=\begin{cases} Q(s), & s\in(0,1),\\ \lim\limits_{s\to 0^+} Q(s), & s=0,\\ \lim\limits_{s\to 1^-} Q(s), & s=1. \end{cases}$$

Clearly $g^{(p+q)}\big(\zeta(s)\big)$ is continuous on $[0,1]$.

Integrating (3.5), we write

$$\int_0^1 g(s)ds=\int_0^1 P(s)ds+\int_0^1 \frac{s^p(s-1)^q}{(p+q)!}g^{(p+q)}\big(\zeta(s)\big)ds. \tag{3.6}$$

Denote

$$c_{j+1}^{pq}=\int_0^1 L_j^{pq}(s)ds, \quad \text{and}$$
$$c_{j+1}^{qp}=\int_0^1 L_j^{qp}(s)ds.$$

Integrating the interpolating polynomial (3.4), we obtain

$$\int_0^1 P(s)ds=\sum_{j=0}^{p-1}h^j y^{(j+1)}(a)c_{j+1}^{pq}+\sum_{j=0}^{q-1}h^j y^{(j+1)}(b)c_{j+1}^{qp}. \tag{3.7}$$

By the mean-value theorem for integrals

$$\begin{aligned}\int_0^1 \frac{s^p(s-1)^q}{(p+q)!}g^{(p+q)}(\zeta(s))ds &= \frac{g^{(p+q)}(\delta)}{(p+q)!}\int_0^1 s^p(s-1)^q ds\\ &=(-1)^q\frac{p!q!}{(p+q)!}\frac{y^{(p+q+1)}(\eta)}{(p+q+1)!}h^{p+q}, \end{aligned}\tag{3.8}$$

for some $\delta\in(0,1)$ and $\eta\in(a,b)$. Substituting (3.2) in (3.1), and by (3.6), (3.7) and (3.8), we obtain

$$\begin{aligned}
y(b) - y(a) &= h\int_0^1 g(s)ds \\
&= h\int_0^1 P(s)ds + h\int_0^1 \frac{s^p(s-1)^q}{(p+q)!} g^{(p+q)}\big(\zeta(s)\big)ds \\
&= \sum_{j=0}^{p-1} h^{j+1}y^{(j+1)}(a)c_{j+1}^{pq} + \sum_{j=0}^{q-1} h^{j+1}y^{(j+1)}(b)c_{j+1}^{qp} \\
&\quad + (-1)^q \frac{p!q!}{(p+q)!}\frac{y^{(p+q+1)}(\eta)}{(p+q+1)!}h^{p+q+1},
\end{aligned}$$

which we write it as

$$\sum_{j=0}^{q} c_j^{qp} y^{(j)}(b)h^j - \sum_{j=0}^{p} c_j^{pq} y^{(j)}(a)h^j = (-1)^q \frac{p!q!}{(p+q)!}\frac{y^{(p+q+1)}(\eta)}{(p+q+1)!}h^{p+q+1}. \tag{3.9}$$

The coefficients c_j^{pq} and c_j^{qp} are known to be [14, 25]

$$c_j^{pq} = \frac{p!(p+q-j)!}{j!(p+q)!(p-j)!}, \qquad j = 0, 1, \ldots, p, \tag{3.10}$$

$$c_j^{qp} = (-1)^j \frac{q!(p+q-j)!}{j!(p+q)!(q-j)!}, \qquad j = 0, 1, \ldots, q. \tag{3.11}$$

There is an important fact about these coefficients stated as the following theorem.

Theorem 3.1. [14, 49] Let $R^{pq}(z)$ be the rational (p, q) Pade approximation of $\exp(z)$. The coefficients (3.10) and (3.11) are identical with the coefficients of $R^{pq}(z)$. That is,

$$R^{pq}(z) = \frac{\sum_{j=0}^{p} c_j^{pq} z^j}{\sum_{j=0}^{q} c_j^{qp} z^j}.$$

Consider a function $x \in C^{p+q+k+1}[a, b]$ with $k \geq 0$. Replacing y by $x^{(k)}$ in (3.9), we obtain

$$\sum_{j=0}^{q} c_j^{qp} x^{(k+j)}(b)h^j - \sum_{j=0}^{p} c_j^{pq} x^{(k+j)}(a)h^j = (-1)^q \frac{p!q!}{(p+q)!}\frac{x^{(k+p+q+1)}(\eta)}{(p+q+1)!}h^{p+q+1},$$

which we write as

$$\sum_{j=0}^{q} c_j^{qp}(k+j)!\frac{x^{(k+j)}(b)}{(k+j)!}h^j - \sum_{j=0}^{p} c_j^{pq}(k+j)!\frac{x^{(k+j)}(a)}{(k+j)!}h^j \\ = (-1)^q \frac{p!q!}{(p+q)!}\frac{x^{(k+p+q+1)}(\eta)}{(p+q+1)!}h^{p+q+1}. \tag{3.12}$$

Denoting the lth Taylor coefficient of x at a τ by $\big(x(\tau)\big)_l$, and denoting

$$\alpha_{kj} = c_j^{pq}(k+j)!, \quad \text{and} \tag{3.13}$$

$$\beta_{kj} = c_j^{qp}(k+j)!, \tag{3.14}$$

we write (3.12) as

$$\sum_{j=0}^{q} \beta_{kj}\big(x(b)\big)_{k+j}h^j - \sum_{j=0}^{p} \alpha_{kj}\big(x(a)\big)_{k+j}h^j = (-1)^q \frac{p!q!}{(p+q)!}\frac{x^{(k+p+q+1)}(\eta)}{(p+q+1)!}h^{p+q+1}. \tag{3.15}$$

We call (3.15) the (p,q) Hermite-Obreschkoff (HO) formula for $x^{(k)}(t)$ at a and b. This formula shows the relation between TCs of $x^{(k)}(t)$ at a and b up to orders p and q, respectively.

3.2 Proposed method

Consider an ODE of the form

$$f\big(t, y, y', \dots, y^{(d)}\big) = 0,$$

where $f : \mathbb{R}^{d+2} \to \mathbb{R}$ is sufficiently differentiable, $\partial f/\partial y^{(d)} \neq 0$, and the initial values

$$y(t^*), y'(t^*), \dots, y^{(d-1)}(t^*),$$

are given. Although this equation can be converted into a d-dimensional system of first-order equations, we are interested in integrating it directly.

Denote

$$\mathbf{y}^*_{<d} = \Big[\big(y(t^*)\big)_0, \big(y(t^*)\big)_1, \dots, \big(y(t^*)\big)_{d-1}\Big].$$

Generating TCs of f at t^* using automatic differentiation (see §2.3.3) and equating them to zero, we can compute the higher-order TCs of y at t^*. That is, for each $j = d, d+1, \dots$, a function T_j is defined such that

$$\big(y(t^*)\big)_j = T_j\big(t^*, \mathbf{y}^*_{<d}\big).$$

Example 3.1. The Van der Pol oscillator evolves in time according to the following second-order ODE

$$f(t, y, y', y'') = y'' - \mu(1-y^2)y' + y = 0. \tag{3.16}$$

Here μ is a scalar parameter indicating the strength of the damping.

For brevity, we also write TCs without parentheses; y_i rather that $(y)_i$. Given $y_0 = y(t)$ and $y_1 = y'(t)$, we can compute higher-order TCs in terms of y_0 and y_1. Applying (2.7), (2.8) and (2.9) to (3.16) at t, we write

$$\begin{aligned}
f_0 &= \big(y'' - \mu(1-y^2)y' + y\big)_0 \\
&= \big(y''\big)_0 - \big(\mu(1-y^2)y'\big)_0 + y_0 \\
&= 2y_2 - \mu\big(1-y^2\big)_0\big(y'\big)_0 + y_0 \\
&= 2y_2 - \mu\big(1-y_0^2\big)y_1 + y_0,
\end{aligned}$$

$$\begin{aligned}
f_1 &= \Big(y'' - \mu(1-y^2)y' + y\Big)_1 \\
&= \Big(y''\Big)_1 - \Big(\mu(1-y^2)y'\Big)_1 + y_1 \\
&= 6y_3 - \mu\Big(\Big(1-y^2\Big)_0\Big(y'\Big)_1 + \Big(1-y^2\Big)_1\Big(y'\Big)_0\Big) + y_1, \\
&= 6y_3 - \mu\Big(1-y_0^2\Big)2y_2 - \mu(-2y_0y_1)y_1 + y_1, \\
&\vdots
\end{aligned}$$

Equating the above TCs to zero, we obtain

$$\begin{aligned}
y_2 &= T_2(y_0,y_1) = \frac{1}{2}\Big(\mu(1-y_0^2)y_1 - y_0\Big), \\
y_3 &= T_3(y_0,y_1) = \frac{1}{6}\Big(\mu\Big(1-y_0^2\Big)2y_2 + \mu(-2y_0y_1)y_1 - y_1\Big), \\
&\vdots \\
y_{p+1} &= T_{p+1}(y_0,y_1).
\end{aligned} \tag{3.17}$$

Assume that we have computed at t^* the TCs

$$\Big(y(t^*)\Big)_0, \Big(y(t^*)\Big)_1, \dots, \Big(y(t^*)\Big)_{p+d-1}.$$

The objective is to find

$$\mathbf{y}_{<d} = \Big[\Big(y(t^*+h)\Big)_0, \Big(y(t^*+h)\Big)_1, \dots, \Big(y(t^*+h)\Big)_{d-1}\Big],$$

for a given stepsize h. Using the (p,q) HO formula (3.15) for $y^{(k)}(t)$ with $k = 0, 1, \dots, d-1$

at t^* and $t^* + h$ yields to

$$\begin{aligned}
&\sum_{j=0}^{q} \beta_{kj} \Big(y(t^* + h) \Big)_{k+j} h^j - \sum_{j=0}^{p} \alpha_{kj} \Big(y(t^*) \Big)_{k+j} h^j \\
&\quad = \sum_{j=0}^{q} \beta_{kj} T_{k+j} \Big(t^* + h, \mathbf{y}_{<d} \Big) h^j - \sum_{j=0}^{p} \alpha_{kj} T_{k+j} \Big(t^*, \mathbf{y}^*_{<d} \Big) h^j \\
&\quad = (-1)^q \frac{p!q!}{(p+q)!} \frac{y^{(p+q+k+1)}(\eta_k)}{(p+q+1)!} h^{p+q+1}.
\end{aligned}$$

Hence, we can solve the nonlinear (in general) system of equations

$$\sum_{j=0}^{q} \beta_{kj} T_{k+j} \Big(t^* + h, \widetilde{\mathbf{y}}_{<d} \Big) h^j - \sum_{j=0}^{p} \alpha_{kj} T_{k+j} \Big(t^*, \mathbf{y}^*_{<d} \Big) h^j = 0, \qquad k = 0, 1, \dots, d-1, \quad (3.18)$$

to find $\widetilde{\mathbf{y}}_{<d}$ as an approximation to $\mathbf{y}_{<d}$.

We call this method the (p, q) Hermite-Obreschkoff (HO) method. Clearly, the choices $(1, 0)$, $(0, 1)$ and $(1, 1)$ yield the explicit Euler method, the implicit Euler method and the trapezoidal scheme, respectively. Moreover, the choices $(p, 0)$ and $(0, q)$ yield the explicit and implicit Taylor series methods, respectively. Hence, we can consider the HO methods as a generalization of Taylor series methods.

Example 3.2. Consider the Van der Pol equation in Example 3.1. Assume that we have computed y^*_j, for $j = 0, \dots, p+1$ by (3.17). To find $\widetilde{y}_0$ and $\widetilde{y}_1$ as approximations to $y(t^* + h)$ and $y'(t^* + h)$, respectively, using the (p, q) HO method (3.18), we can solve the following system of two nonlinear equations

$$\begin{aligned}
\sum_{j=0}^{q} \beta_{0j} T_j \Big(\widetilde{y}_0, \widetilde{y}_1 \Big) h^j - \sum_{j=0}^{p} \alpha_{0j} y^*_j h^j = 0, \\
\sum_{j=0}^{q} \beta_{1j} T_{j+1} \Big(\widetilde{y}_0, \widetilde{y}_1 \Big) h^j - \sum_{j=0}^{p} \alpha_{1j} y^*_{j+1} h^j = 0,
\end{aligned}$$

where α_{0j} and α_{1j} are given by (3.13), and β_{0j} and β_{1j} are given by (3.14).

Theorem 3.2. [14] For arbitrary non-negative integers p and q and for $y \in C^{p+q+d}[t^*, t^* + h]$, the (p, q) HO method (3.18) has the order of consistency $p + q$; the order of the local error is $p + q + 1$.

By Theorem 3.1

$$\widetilde{y}(t^* + h) = R^{pq}(\lambda h) y(t^*),$$

when the (p, q) HO method (3.18) is used for the test problem $y' = \lambda y$. This leads to the following theorem.

Theorem 3.3. [14] For $y \in C^{p+q+d}[t^*, t^* + h]$, the (p, q) HO method (3.18) is

- A-stable if $q \in \{p, p + 1, p + 2\}$, and
- L-stable if $q \in \{p + 1, p + 2\}$.

Thus, an HO method can be made to yield an approximation of arbitrary order without conflicting with the A-stability (or L-stability) requirement.

Chapter 4

An Hermite-Obreschkoff method for DAEs

In this chapter, we develop a numerical method based on Pryce's structural analysis and the HO formula (3.15) for a DAE of the general form (1.1). The overall solution process goes in steps over the t range. TCs are computed up to a chosen order at the current t^*. Using the HO formula (3.15) for a certain number of derivatives of the state variables at t^* and $t^* + h$, we compute approximate values for solution components at $t^* + h$, and this process repeats. Here, we describe one step of this method. First, we explain how the computational scheme for TCs is guided by the two non-negative integer vectors $\mathbf{c}$ and $\mathbf{d}$ (see §2.2) found by Pryce's structural analysis, §4.1. Then, we describe the proposed HO method in §4.2. Finally, we present the implementation of the method in §4.3.

4.1 Computational scheme for Taylor coefficients

In general, it is not obvious how the various equations obtained by differentiating the original DAE equations f_i can be organized to solve for the TCs of the state variables x_j. This is made clear by the offsets $\mathbf{c}$ and $\mathbf{d}$, which tell us that the computation forms a sequence of stages [45, 54]. The process starts at stage $s_d = -\max_j d_j$ and is performed for stages $s = s_d, s_d + 1, \ldots$, as follows.

At stage s, we consider the set of equations

$$(f_i)_{s+c_i} = 0, \qquad \text{for all } i \text{ such that } s + c_i \geq 0, \tag{4.1}$$

to determine values for

$$(x_j)_{s+d_j}, \qquad \text{for all } j \text{ such that } s + d_j \geq 0, \tag{4.2}$$

using previously found

$$(x_j)_l, \qquad \text{for all } j \text{ such that } 0 \leq l < s + d_j. \tag{4.3}$$

Example 4.1. For the pendulum, we obtained $\mathbf{c} = [0, 0, 2]$ and $\mathbf{d} = [2, 2, 0]$ in Example 2.1. The process implied by (4.1) and (4.2) is illustrated in Table 4.1. For brevity, the TCs will be written without parentheses: x_l rather than $(x)_l$, etc.

To express (4.1) and (4.2) in a more compact form, let

$$\mathcal{Z}_n = \{(z, l) : z = 0, 1, \ldots, n-1, \;\; l = 0, 1, \ldots\},$$

stage	uses equations	to obtain	previously found
-2	$0 = h_0 = x_0^2 + y_0^2 + L^2$	x_0, y_0	
-1	$0 = h_1 = 2x_0x_1 + 2y_0y_1$	x_1, y_1	x_0, y_0
0	$\begin{cases} 0 = f_0 = 2x_2 + \lambda_0 x_0 \\ 0 = g_0 = 2y_2 + \lambda_0 y_0 + G \\ 0 = h_2 = 2x_0x_2 + x_1^2 + 2y_0y_2 + y_1^2 \end{cases}$	x_2, y_2, λ_0	x_0, x_1, y_0, y_1
$\vdots$	$\vdots$	$\vdots$	$\vdots$

Table 4.1: Computational scheme for TCs of the pendulum.

and for a $s \in \mathbb{Z}$ define

$$I_s = \{(i,l) \in \mathcal{Z}_n : l = s + c_i\}, \quad \text{and}$$
$$J_s = \{(j,l) \in \mathcal{Z}_n : l = s + d_j\}.$$

Denote by $\mathbf{f}_{I_s}$ the vector of $(f_i)_l$ for $(i,l) \in I_s$, and by $\mathbf{x}_{J_s}$ the vector of $(x_j)_l$ for $(j,l) \in J_s$. The order of entries does not matter, but assume fixed. Also, define $I_{\leq s}$ to be the union of I_l for all $l \leq s$, and similarly for $I_{<s}$, $J_{\leq s}$, $J_{<s}$. With the above notation, we write (4.1) as

$$\mathbf{f}_{I_s}\Big(t, \mathbf{x}_{J_{<s}}, \mathbf{x}_{J_s}\Big) = 0. \tag{4.4}$$

In the solution process, $\mathbf{x}_{J_{<s}}$ is known (as (4.3)), and we solve for $\mathbf{x}_{J_s}$ (as (4.2)). We will describe this process in Chapter 5.

Example 4.2. For the pendulum, let

$$\mathbf{x}_{J_{<0}} = [x_0, x_1, y_0, y_1],$$

be known at t. Using (4.4), we have the scheme in (Table 4.2).

stage	solve	to obtain
0	$0 = f_{I_0}\big(t, \mathbf{x}_{J_{<0}}, \mathbf{x}_{J_0}\big)$	$\mathbf{x}_{J_0} = [x_2, y_2, \lambda_0]$
1	$0 = f_{I_1}\big(t, \mathbf{x}_{J_{<1}}, \mathbf{x}_{J_1}\big)$	$\mathbf{x}_{J_1} = [x_3, y_3, \lambda_1]$
2	$0 = f_{I_2}\big(t, \mathbf{x}_{J_{<2}}, \mathbf{x}_{J_2}\big)$	$\mathbf{x}_{J_2} = [x_4, y_4, \lambda_2]$
$\vdots$	$\vdots$	$\vdots$
$p-1$	$0 = f_{I_{p-1}}\big(t, \mathbf{x}_{J_{<p-1}}, \mathbf{x}_{J_{p-1}}\big)$	$\mathbf{x}_{J_{p-1}} = [x_{p+1}, y_{p+1}, \lambda_{p-1}]$

Table 4.2: Scheme to compute TCs for pendulum in a compact form.

Before we propose a method which prescribes how to compute $\mathbf{x}_{J_{<0}}$ at $t^* + h$ for a given stepsize h, we define an *irregular matrix*.

Definition 4.1. Let $\mathbf{v}$ be a vector of m positive integers. An irregular matrix of size $\mathbf{v}$, is a matrix with m rows and v_i entries in its ith row.

Example 4.3. $\mathbf{A} = \begin{bmatrix} 3 & -1 & \\ 0 & & \\ 5 & 1 & 2 \end{bmatrix}$ is an irregular matrix of size $\mathbf{v} = [2, 1, 3]$.

Example 4.4. Consider the following irregular matrix of size $\mathbf{d}$,

$$\begin{bmatrix} (x_0)_0 & (x_0)_1 & \dots & (x_0)_{d_0-1} \\ (x_1)_0 & (x_1)_1 & \dots & (x_1)_{d_1-1} \\ \vdots & & & \\ (x_{n-1})_0 & (x_{n-1})_1 & \dots & (x_{n-1})_{d_{n-1}-1} \end{bmatrix}. \tag{4.5}$$

The vector $\mathbf{x}_{J_{<0}}$ is created by concatenating rows of (4.5).

4.2 Proposed method

Consider the DAE (1.1) with canonical offsets $\mathbf{c}$ and $\mathbf{d}$ found by the Σ-method (see §2.2). Given $\mathbf{x}_{J_{<0}}$ at t^*, denote it by $\mathbf{x}^*_{J_{<0}}$, assume that we have solved (4.4) for $s = 0, 1, \dots, p-1$ and computed at this point the TCs

$$\begin{array}{llll} (x^*_0)_0, & (x^*_0)_1, & \dots & (x^*_0)_{p+d_0-1}, \\ (x^*_1)_0, & (x^*_1)_1, & \dots & (x^*_1)_{p+d_1-1}, \\ \vdots & & & \\ (x^*_{n-1})_0, & (x^*_{n-1})_1, & \dots & (x^*_{n-1})_{p+d_{n-1}-1}. \end{array}$$

Our goal is to find values for $\mathbf{x}_{J_{<0}}$ at $t^* + h$. We refer to them as *independent TCs*. By (4.5), the number of these TCs is

$$N = \sum_{j=0}^{n-1} d_j. \tag{4.6}$$

Using the (p, q) HO formula (3.15) for $x_j^{(k)}(t)$, for $j = 0, \dots, n-1$, and $k = 0, \dots, d_j - 1$, at t^* and $t^* + h$, we obtain

$$\sum_{r=0}^{q} \beta_{kr} (x_j)_{k+r} h^r - \sum_{r=0}^{p} \alpha_{kr} (x^*_j)_{k+r} h^r = e_{pq} \xi_{jk} h^{p+q+1}, \tag{4.7}$$

where

$$e_{pq} = \frac{(-1)^q p! q!}{(p+q)!}, \quad \text{and} \tag{4.8}$$

$$\xi_{jk} = \frac{x_j^{(p+q+1+k)}(\eta_{jk})}{(p+q+1)!}, \qquad \text{with } \eta_{jk} \in (t^*, t^*+h). \tag{4.9}$$

Using the computational scheme for TCs in §4.1, we can compute higher-order TCs in terms of independent ones. That is, there is a function $T_{j,k+r}$ such that

$$(x_j^*)_{k+r} = T_{j,k+r}\Big(t^*, \mathbf{x}^*_{J_{<0}}\Big), \tag{4.10}$$

$$(x_j)_{k+r} = T_{j,k+r}\Big(t^*+h, \mathbf{x}_{J_{<0}}\Big). \tag{4.11}$$

Substituting (4.10) and (4.11) in (4.7), we obtain

$$\sum_{r=0}^{q} \beta_{kr} T_{j,k+r}\Big(t^*+h, \mathbf{x}_{J_{<0}}\Big) h^r - \sum_{r=0}^{p} \alpha_{kr} T_{j,k+r}\Big(t^*, \mathbf{x}^*_{J_{<0}}\Big) h^r = e_{pq}\xi_{jk} h^{p+q+1}. \tag{4.12}$$

Hence, we can solve the nonlinear (in general) system of N equations

$$\sum_{r=0}^{q} \beta_{kr} T_{j,k+r}\Big(t^*+h, \widetilde{\mathbf{x}}_{J_{<0}}\Big) h^r - \sum_{r=0}^{p} \alpha_{kr} T_{j,k+r}\Big(t^*, \mathbf{x}^*_{J_{<0}}\Big) h^r = 0, \tag{4.13}$$

for $j = 0, \dots, n-1$ and $k = 0, \dots, d_j - 1$, to find $\widetilde{\mathbf{x}}_{J_{<0}}$ as an approximation to $\mathbf{x}_{J_{<0}}$.

To write the system (4.13) in a matrix form, addition and subtraction are defined as element-wise operations on elements of irregular matrix operands of same sizes. Multiplication of an irregular matrix by a scalar is defined as multiplication of every entry of the irregular matrix by the scalar. Also, we define a product between an irregular matrix and a vector.

Definition 4.2. For an irregular matrix $\mathbf{A}$ of size $\mathbf{v}$ and a vector $\mathbf{z}$ of size $\max_i v_i$, we define the product

$$\mathbf{B} = \mathbf{A} \otimes \mathbf{z},$$

such that $\mathbf{B}$ is an irregular matrix of size $\mathbf{v}$ with entries $(\mathbf{B})_{ij} = (\mathbf{A})_{ij} z_j$.

Example 4.5. Consider the pendulum with given $\mathbf{x}^*_{J_{<0}}$. As in Table 4.2, we can compute at t^* the TCs

$$\begin{array}{llll} x_0^*, & x_1^*, & \dots & x_{p+1}^*, \\ y_0^*, & y_1^*, & \dots & y_{p+1}^*, \\ \lambda_0^*, & \lambda_1^*, & \dots & \lambda_{p-1}^*. \end{array}$$

To find

$$\widetilde{\mathbf{x}}_{J_{<0}} = [\widetilde{x}_0, \widetilde{x}_1, \widetilde{y}_0, \widetilde{y}_1],$$

as an approximation to $\mathbf{x}_{J_{<0}}$ at $t^* + h$, we solve the nonlinear system

$$\begin{aligned} \sum_{r=0}^{q} \beta_{0r} T_{0r}\Big(t^* + h, \widetilde{\mathbf{x}}_{J_{<0}}\Big) h^r - \sum_{r=0}^{p} \alpha_{0r} x_r^* h^r &= 0, \\ \sum_{r=0}^{q} \beta_{1r} T_{0,r+1}\Big(t^* + h, \widetilde{\mathbf{x}}_{J_{<0}}\Big) h^r - \sum_{r=0}^{p} \alpha_{1r} x_{r+1}^* h^r &= 0, \\ \sum_{r=0}^{q} \beta_{0r} T_{1r}\Big(t^* + h, \widetilde{\mathbf{x}}_{J_{<0}}\Big) h^r - \sum_{r=0}^{p} \alpha_{0r} y_r^* h^r &= 0, \\ \sum_{r=0}^{q} \beta_{1r} T_{1,r+1}\Big(t^* + h, \widetilde{\mathbf{x}}_{J_{<0}}\Big) h^r - \sum_{r=0}^{p} \alpha_{1r} y_{r+1}^* h^r &= 0. \end{aligned}$$

Here α_{0r} and α_{1r} are given by (3.13), and β_{0r} and β_{1r} are given by (3.14). Also, for $l = r, r+1$, we have

$$\begin{aligned} \widetilde{x}_l &= T_{0l}\Big(t^* + h, \widetilde{\mathbf{x}}_{J_{<0}}\Big), \\ \widetilde{y}_l &= T_{1l}\Big(t^* + h, \widetilde{\mathbf{x}}_{J_{<0}}\Big). \end{aligned}$$

Using the product in Definition 4.2, we can write the above system in the following compact form

$$\begin{bmatrix}\widetilde{x}_0 & \widetilde{x}_1\\ \\ \widetilde{y}_0 & \widetilde{y}_1\end{bmatrix}\otimes\begin{bmatrix}\beta_{00}\\ \\ \beta_{10}\end{bmatrix}+\begin{bmatrix}\widetilde{x}_1 & \widetilde{x}_2\\ \\ \widetilde{y}_1 & \widetilde{y}_2\end{bmatrix}h\otimes\begin{bmatrix}\beta_{01}\\ \\ \beta_{11}\end{bmatrix}+\ldots+\begin{bmatrix}\widetilde{x}_q & \widetilde{x}_{q+1}\\ \\ \widetilde{y}_q & \widetilde{y}_{q+1}\end{bmatrix}h^q\otimes\begin{bmatrix}\beta_{0q}\\ \\ \beta_{1q}\end{bmatrix}$$

$$=\begin{bmatrix}x^*_0 & x^*_1\\ \\ y^*_0 & y^*_1\end{bmatrix}\otimes\begin{bmatrix}\alpha_{00}\\ \\ \alpha_{10}\end{bmatrix}+\begin{bmatrix}x^*_1 & x^*_2\\ \\ y^*_1 & y^*_2\end{bmatrix}h\otimes\begin{bmatrix}\alpha_{01}\\ \\ \alpha_{11}\end{bmatrix}+\ldots+\begin{bmatrix}x^*_p & x^*_{p+1}\\ \\ x^*_p & y^*_{p+1}\end{bmatrix}h^p\otimes\begin{bmatrix}\alpha_{0p}\\ \\ \alpha_{1p}\end{bmatrix}.$$

That is,

$$\sum_{r=0}^{q}\begin{bmatrix}\widetilde{x}_r & \widetilde{x}_{r+1}\\ \\ \widetilde{y}_r & \widetilde{y}_{r+1}\end{bmatrix}h^r\otimes\begin{bmatrix}\beta_{0r}\\ \\ \beta_{1r}\end{bmatrix}=\sum_{r=0}^{p}\begin{bmatrix}x^*_r & x^*_{r+1}\\ \\ y^*_r & y^*_{r+1}\end{bmatrix}h^r\otimes\begin{bmatrix}\alpha_{0r}\\ \\ \alpha_{1r}\end{bmatrix}.$$

Let $d = \max_j d_j$. Denote

$$\mathbf{a}_r = [\alpha_{0r}, \alpha_{1r}, \ldots, \alpha_{d-1,r}], \tag{4.14}$$

$$\mathbf{b}_r = [\beta_{0r}, \beta_{1r}, \ldots, \beta_{d-1,r}], \tag{4.15}$$

and the irregular matrices

$$\mathbf{F}^{[r]}\big(t,\mathbf{x}_{J_{<0}}\big) = \begin{bmatrix} T_{0,r}(t,\mathbf{x}_{J_{<0}}) & T_{0,r+1}(t,\mathbf{x}_{J_{<0}}) & \dots & T_{0,r+d_0-1}(t,\mathbf{x}_{J_{<0}}) \\ T_{1,r}(t,\mathbf{x}_{J_{<0}}) & T_{1,r+1}(t,\mathbf{x}_{J_{<0}}) & \dots & T_{1,r+d_1-1}(t,\mathbf{x}_{J_{<0}}) \\ \vdots & & & \\ T_{n-1,r}(t,\mathbf{x}_{J_{<0}}) & T_{n-1,r+1}(t,\mathbf{x}_{J_{<0}}) & \dots & T_{n-1,r+d_{n-1}-1}(t,\mathbf{x}_{J_{<0}}) \end{bmatrix}$$

$$= \begin{bmatrix} (x_0)_r & (x_0)_{r+1} & \dots & (x_0)_{r+d_0-1} \\ (x_1)_r & (x_1)_{r+1} & \dots & (x_1)_{r+d_1-1} \\ \vdots & & & \\ (x_{n-1})_r & (x_{n-1})_{r+1} & \dots & (x_{n-1})_{r+d_{n-1}-1} \end{bmatrix}, \tag{4.16}$$

and $\mathbf{E}$ whose the (j,k)th entry is ξ_{jk}, for $j = 0,\dots,n-1$, $k = 0,\dots,d_j-1$. Then, (4.12) can be written in the following form

$$\sum_{r=0}^{q} \mathbf{F}^{[r]}\big(t^*+h,\mathbf{x}_{J_{<0}}\big)h^r \otimes \mathbf{b}_r - \sum_{r=0}^{p} \mathbf{F}^{[r]}\big(t^*,\mathbf{x}^*_{J_{<0}}\big)h^r \otimes \mathbf{a}_r = e_{pq}h^{p+q+1}\mathbf{E}. \tag{4.17}$$

Denoting

$$\mathbf{F}(\mathbf{x}_{J_{<0}}) = \sum_{r=0}^{q} \mathbf{F}^{[r]}\big(t^*+h,\mathbf{x}_{J_{<0}}\big)h^r \otimes \mathbf{b}_r - \sum_{r=0}^{p} \mathbf{F}^{[r]}\big(t^*,\mathbf{x}^*_{J_{<0}}\big)h^r \otimes \mathbf{a}_r, \tag{4.18}$$

(4.17) is

$$\mathbf{F}(\mathbf{x}_{J_{<0}}) = e_{pq}h^{p+q+1}\mathbf{E}. \tag{4.19}$$

To find $\widetilde{\mathbf{x}}_{J_{<0}}$ as an approximation to $\mathbf{x}_{J_{<0}}$, we solve

$$\mathbf{F}(\widetilde{\mathbf{x}}_{J_{<0}}) = 0. \tag{4.20}$$

We refer to (4.20) as *Hermite-Obreschkoff system*. Since our interest in this method is for stiff DAEs, we must solve (4.20) using a Newton's method which will be described in Chapter 7. By (4.17) and (4.16), we need to solve (4.4) for $s = 0, 1, \ldots, q-1$ at each iteration of Newton's method. Hence, the constraints $\mathbf{f}_{I_{<0}} = 0$ are not enforced during the above discretization. Therefore, the system (4.20) may lead to a numerical solution, say $\mathbf{x}^{\text{HO}}_{J_{<0}}$, that violates the constraints $\mathbf{f}_{I_{<0}} = 0$. To avoid this a likely *drift-off phenomenon*, we perform a projection step onto these algebraic constraints after each successful integration step. That is, the constrained optimization problem

$$\min_{\mathbf{x}_{J_{<0}}} \|\mathbf{x}_{J_{<0}} - \mathbf{x}^{\text{HO}}_{J_{<0}}\|_2 \quad \text{subject to} \quad \mathbf{f}_{I_{<0}}(t, \mathbf{x}_{J_{<0}}) = 0, \tag{4.21}$$

is solved.

4.3 Implementation

DAETS is implemented as a collection of C++ classes. We manage the implementation of the proposed HO method by adding four classes and some functions to this solver. First we list the classes in DAETS along with brief descriptions. Then, we introduce the new ones. All classes are depicted in Figure 4.1.

4.3.1 Classes in DAETS

SAdata computes equation and variable offsets.

TaylorSeries is a pure virtual class for computing and accessing TCs. Class **FadbadTS** implements the functionality of **TaylorSeries** using the FADBAD++ package.

Jacobian is a pure virtual class for computing the system Jacobian. Class **FadbadJac**

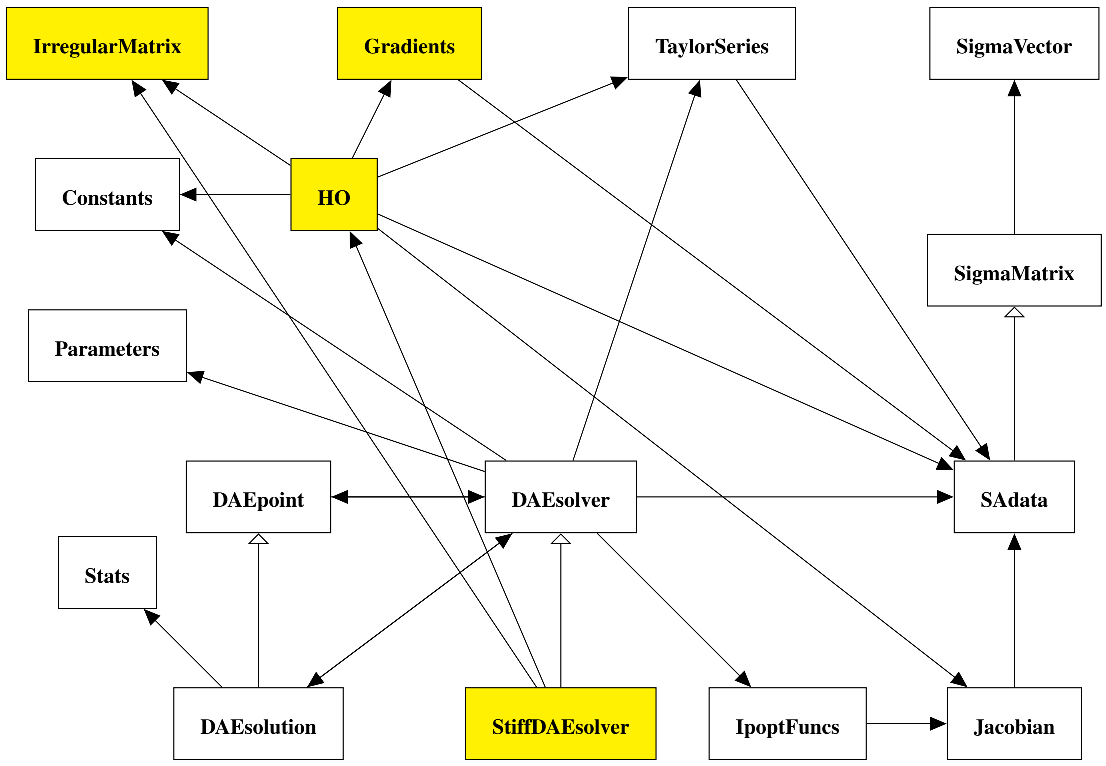

Figure 4.1: Solver class diagram. The arrows with the triangle denote inheritance; a normal arrow from class A to class B means A uses B.

implements the functionality of **Jacobian** using the FADBAD++ package.

DAEsolver implements the integration process and contains policy data about the integration, such as Taylor series order, accuracy tolerance and type of error test.

DAEsolution implements the moving point. This includes the numerical solution and the current value of t; also data describing the current state of the solution.

DAEpoint is a base class for **DAEsolution**. An object of this class stores an irregular matrix whose entries representing derivatives $x_j^{(k)}$.

Constants stores various constants, such as default values for order, tolerance, etc. needed during integration.

IpoptFuncs provides the functions needed by IPOPT in (4.21).

Parameters encapsulates various parameters that can be set before integration, such as order, tolerance, smallest allowed stepsize, etc.

Stats collects various statistics during an integration, such as CPU time, number of steps, percentage rejected steps, etc.

SigmaMatrix provides functions for computing the signature matrix of a DAE. These computations are performed by propagating **SigmaVector** objects through operator overloading.

4.3.2 The HO class

This class constructs and solves the HO system (4.20).

```
23 ⟨HO Declarations 23⟩ ≡
   class HO {
   public:
     ⟨HO Public Functions 344⟩;
   private:
     ⟨HO Private Functions 35⟩;
     ⟨HO Data Members 28⟩;
   };
```

This code is used in chunk 343.

The constructor and destructor of the **HO** class are in Appendix D.1. We will declare the data members and implement the functions in next chapters. We use the following

24 ⟨enumeration type for HO method 24⟩ ≡

```
typedef enum {
    HO_JAC_SINGULAR = −3,    /* if Jacobian of (4.20) is singular */
    SYS_JAC_SINGULAR,    /* if system Jacobian is singular */
    STAGE0_FAIL,    /* if computing TCs at stage zero fails */
    HO_SUCCESS,    /* if evaluating required functions in HO method succeeds */
    HO_CONVERGENT    /* if iteration method for HO system is convergent */
} HoFlag;
```

This code is used in chunk 369.

4.3.3 The Gradients class

It employs the FADBAD++ package to compute gradients of TCs needed for the Jacobian of (4.20).

25 ⟨Gradients Declarations 25⟩ ≡

```
class Gradients {
public:
    ⟨Gradients Public Functions 82⟩;
private:
    ⟨Gradients Data Members 80⟩;
};
```

This code is used in chunk 350.

The constructor and destructor of the **Gradients** class are in Appendix D.2. We will declare the data members and implement the functions in Chapter 6.

4.3.4 The StiffDAEsolver class

This class implements the integration process.

27 ⟨ StiffDAEsolver Declarations 27 ⟩ ≡

```
class StiffDAEsolver : public daets::DAEsolver {
public:
  ⟨ StiffDAEsolver Public Functions 357 ⟩;
private:
  ⟨ StiffDAEsolver Private Functions 360 ⟩;
  ⟨ StiffDAEsolver Data Members 192 ⟩;
};
```

This code is used in chunk 356.

28 ⟨ HO Data Members 28 ⟩ ≡

```
friend class StiffDAEsolver;
```

See also chunks 34, 38, 39, 41, 42, 43, 46, 47, 50, 52, 53, 74, 78, 85, 119, 120, 123, 126, 128, 129, 132, 133, 139, 144, 150, 153, 155, 157, 158, 167, and 174

This code is used in chunk 23.

The constructor and destructor of the **StiffDAEsolver** class are in Appendix D.3. We will declare the data members and implement the functions in Chapter 8 and Chapter 9.

4.3.5 The IrregularMatrix class

It is a template class that overloads the arithmetic operations and required functions for irregular matrices. The implementation is in Appendix B.

Chapter 5

Computing Taylor coefficients

To compute a numerical solution of a DAE by the proposed HO method in §4.2, we should find higher-order TCs as independent ones are given. That is, given $\mathbf{x}_{J_{<s}}$ at t, we solve the system (4.4) for $s \geq 0$ to find $\mathbf{x}_{J_s}$ at this point. This system is square for $s \geq 0$, where it can be nonlinear when $s = 0$, but always linear for $s > 0$ [45, 54]. We first consider the case of linear square systems in §5.1 and then describe the nonlinear case in §5.2.

5.1 Solving linear systems

When (4.4) is linear at $s = 0$ and for $s > 0$, we apply one iteration of Newton's method to (4.4) to solve for $\mathbf{x}_{J_s} \in \mathbb{R}^n$. Given $\mathbf{x}_{J_{<s}}$ as constants and $\widetilde{\mathbf{x}}_{J_s}$ as an initial guess for $\mathbf{x}_{J_s}$,

denote

$$\begin{aligned} \mathbf{b}_s &= \mathbf{f}_{I_s}\Big(t, \mathbf{x}_{J_{<s}}, \widetilde{\mathbf{x}}_{J_s}\Big), & \mathbf{b}_s &\in \mathbb{R}^n, \\ \mathbf{A}_s &= \frac{\partial \mathbf{f}_{I_s}}{\partial \mathbf{x}_{J_s}}\Big(t, \mathbf{x}_{J_{<s}}, \widetilde{\mathbf{x}}_{J_s}\Big), & \mathbf{A}_s &\in \mathbb{R}^{n\times n}. \end{aligned} \tag{5.1}$$

Then (4.4) is

$$0 = \mathbf{b}_s + \mathbf{A}_s\Big(\mathbf{x}_{J_s} - \widetilde{\mathbf{x}}_{J_s}\Big). \tag{5.2}$$

Here $\mathbf{A}_s$ does not depend on $\widetilde{\mathbf{x}}_{J_s}$. Before we describe the solution process of (5.2) for $\mathbf{x}_{J_s}$, we show how $\mathbf{A}_s$ is obtained.

5.1.1 Forming the matrix

The (i,j)th entry of $\mathbf{A}_s$ is [46]

$$\Big(\mathbf{A}_s\Big)_{ij} = \frac{\partial (f_i)_{s+c_i}}{\partial (x_j)_{s+d_j}} = \frac{\partial f_i^{(s+c_i)}/(s+c_i)!}{\partial x_j^{(s+d_j)}/(s+d_j)!} = \frac{(s+d_j)!}{(s+c_i)!}\cdot\frac{\partial f_i^{(s+c_i)}}{\partial x_j^{(s+d_j)}}, \tag{5.3}$$

where $s+c_i \geq 0$ and $s+d_j \geq 0$. Using Griewank's Lemma [46]

$$\frac{\partial f_i^{(s+c_i)}}{\partial x_j^{(s+d_j)}} = \frac{\partial f_i^{(c_i)}}{\partial x_j^{(d_j)}}$$

in (5.3), we write it as

$$\begin{aligned} \Big(\mathbf{A}_s\Big)_{ij} &= \frac{(s+d_j)!}{(s+c_i)!}\cdot\frac{\partial f_i^{(c_i)}}{\partial x_j^{(d_j)}} = \frac{(s+d_j)!}{(s+c_i)!}\cdot\frac{c_i!\ \partial(f_i)_{c_i}}{d_j!\ \partial(x_j)_{d_j}} \\ &= \frac{c_i!}{(s+c_i)!}\cdot\frac{\partial(f_i)_{c_i}}{\partial(x_j)_{d_j}}\cdot\frac{(s+d_j)!}{d_j!}. \end{aligned} \tag{5.4}$$

Let $\mathbf{C}_s$ and $\mathbf{D}_s$ be diagonal matrices such that the ith entry on their main diagonals are $(s+c_i)!/c_i!$ and $(s+d_i)!/d_i!$, respectively. Then, from (5.4)

$$\mathbf{A}_s = \mathbf{C}_s^{-1}\mathbf{A}_0\mathbf{D}_s \quad \text{for } s \geq 0, \tag{5.5}$$

where

$$\left(\mathbf{A}_0\right)_{ij} = \frac{\partial (f_i)_{c_i}}{\partial (x_j)_{d_j}} = \frac{d_j!}{c_i!}\frac{\partial f_i^{(c_i)}}{\partial x_j^{d_j}} = \frac{1}{c_i!}\,(\mathbf{J})_{ij}\ d_j!. \tag{5.6}$$

That is, $\mathbf{A}_0 = \partial \mathbf{f}_{I_0}/\partial \mathbf{x}_{J_0}$ can be written as a diagonally-scaled version of the system Jacobian $\mathbf{J}$ given in (2.3).

Example 5.1. For the pendulum in Example 2.1, the elements of $\mathbf{f}_{I_0}$ are

$$\begin{aligned} f_0 &= 2x_2 + \lambda_0 x_0,\\ g_0 &= 2y_2 + \lambda_0 y_0 + G,\\ h_2 &= 2x_0x_2 + x_1^2 + 2y_0y_2 + y_1^2. \end{aligned}$$

By (5.6), we obtain

$$\begin{aligned}\mathbf{A}_0 = \begin{bmatrix} \partial f_0/\partial x_2 & \partial f_0/\partial y_2 & \partial f_0/\partial \lambda_0 \\ \partial g_0/\partial x_2 & \partial g_0/\partial y_2 & \partial g_0/\partial \lambda_0 \\ \partial h_2/\partial x_2 & \partial h_2/\partial y_2 & \partial h_2/\partial \lambda_0 \end{bmatrix} &= \begin{bmatrix} 2 & 0 & x_0 \\ 0 & 2 & y_0 \\ 2x_0 & 2y_0 & 0 \end{bmatrix}\\ &= \text{diag}[1,1,2]^{-1}\ \mathbf{J}\ \text{diag}[2,2,1],\end{aligned}$$

with $\mathbf{J}$ given in (2.5).

At stage $s = 0$, given $\mathbf{x}_{J_{<0}} = [x_0, x_1, y_0, y_1]$, we compute $\mathbf{x}_{J_0} = [x_2, y_2, \lambda_0]$, by solving

$$\begin{aligned} 0 &= 2x_2 + \lambda_0 x_0,\\ 0 &= 2y_2 + \lambda_0 y_0 + G,\\ 0 &= 2x_0x_2 + x_1^2 + 2y_0y_2 + y_1^2. \end{aligned}$$

It can be written as

$$0 = \underbrace{\begin{bmatrix} 2 & 0 & x_0 \\ 0 & 2 & y_0 \\ 2x_0 & 2y_0 & 0 \end{bmatrix}}_{\mathbf{A}_0} \underbrace{\begin{bmatrix} x_2 \\ y_2 \\ \lambda_0 \end{bmatrix}}_{\mathbf{x}_{J_0}} + \begin{bmatrix} 0 \\ G \\ x_1^2 + y_1^2 \end{bmatrix}.$$

At stage $s = 1$, taking $\mathbf{x}_{J_{<0}}$ and $\mathbf{x}_{J_0}$ as known, we compute $\mathbf{x}_{J_1} = [x_3, y_3, \lambda_1]$ by solving

$$0 = 6x_3 + \lambda_0 x_1 + \lambda_1 x_0,$$

$$0 = 6y_3 + \lambda_0 y_1 + \lambda_1 y_0,$$

$$0 = 2x_0x_3 + 2x_1x_2 + 2y_0y_3 + 2y_1y_2.$$

It can be written as

$$0 = \underbrace{\begin{bmatrix} 6 & 0 & x_0 \\ 0 & 6 & y_0 \\ 2x_0 & 2y_0 & 0 \end{bmatrix}}_{\mathbf{A}_1} \underbrace{\begin{bmatrix} x_3 \\ y_3 \\ \lambda_1 \end{bmatrix}}_{\mathbf{x}_{J_1}} + \begin{bmatrix} \lambda_0 x_1 \\ \lambda_0 y_1 \\ 2x_1x_2 + 2y_1y_2 \end{bmatrix},$$

with

$$\mathbf{A}_1 = \text{diag}[1, 1, 3]^{-1}\ \mathbf{A}_0\ \text{diag}[3, 3, 1].$$

5.1.2 Implementation

To compute $\mathbf{C}_s$ and $\mathbf{D}_s$, we precompute factorials in the constructor of the **HO** class and store them in

34 ⟨ HO Data Members 28 ⟩ +≡

```
std::vector<double> factorial_;
```

The function *comp_cs_ds* is implemented to compute the entries of $\mathbf{C}_s$ and $\mathbf{D}_s$.

35 ⟨ HO Private Functions 35 ⟩ ≡

```
double comp_cs_ds(int $s$, int offset)
{      /* offset is either $c_i$ or $d_j$. If $c_i$ e.g., returns $(s + c_i)!/c_i!$ */
   return factorial_[$s$ + offset]/factorial_[offset];
}
```

See also chunk 347.

This code is used in chunk 23.

We can set the initial guess $\widetilde{\mathbf{x}}_{J_s}$ to zero. However, using a good approximation can lead to better accuracy, as the solution process (5.8) can be regarded as one step of an iterative refinement of an already reasonable solution. After the first step, we use the (currently) computed TCs as an initial guess for the TCs for the next step. This happens automatically in DAETS since TCs are stored in an array: the values in this array are used as an initial guess and overwritten with the new TCs [45].

From (5.2), we solve the following system of linear equations

$$\mathbf{A}_s \boldsymbol{\delta}_s = \mathbf{b}_s, \tag{5.7}$$

and compute

$$\mathbf{x}_{J_s} = \widetilde{\mathbf{x}}_{J_s} - \boldsymbol{\delta}_s. \tag{5.8}$$

By (5.5), (5.7) is

$$\mathbf{C}_s^{-1} \mathbf{A}_0 \mathbf{D}_s \boldsymbol{\delta}_s = \mathbf{b}_s,$$

which leads to

$$\mathbf{A}_0\mathbf{D}_s\boldsymbol{\delta}_s = \mathbf{C}_s\mathbf{b}_s. \tag{5.9}$$

We find $\mathbf{x}_{J_s}$ by the following steps:

- compute $\mathbf{b}_s$,
- compute $\boldsymbol{\beta}_s = \mathbf{C}_s\mathbf{b}_s$,
- solve $\mathbf{A}_0\mathbf{y}_s = \boldsymbol{\beta}_s$ for $\mathbf{y}_s$, and
- set $\mathbf{x}_{J_s} = \widetilde{\mathbf{x}}_{J_s} - \mathbf{D}_s^{-1}\mathbf{y}_s$.

They are implemented in the function *CompTCsLinear*.

37 ⟨Definitions of HO Private Functions 37⟩ ≡

```
void HO :: CompTCsLinear(int s)
{
    ⟨compute $\mathbf{b}_s$ 40⟩;
    ⟨compute $\boldsymbol{\beta}_s = \mathbf{C}_s\mathbf{b}_s$ 44⟩;
    ⟨solve $\mathbf{A}_0\mathbf{y}_s = \boldsymbol{\beta}_s$ 54⟩;
    ⟨compute and set $\mathbf{x}_{J_s} = \widetilde{\mathbf{x}}_{J_s} - \mathbf{D}_s^{-1}\mathbf{y}_s$ 56⟩;
}
```

See also chunks 49, 51, 57, 62, 69, 79, 121, 124, 127, 130, 135, 137, 145, 146, 147, 151, 154, 156, 161, 168, 170, 221, 262, 263, and 280

This code is used in chunk 348.

5.1.2.1 Computing $\mathbf{b}_s$

The **HO** class maintains a pointer to an object of the **TaylorSeries** class from DAETS.

38 ⟨HO Data Members 28⟩ +≡

daets::**TaylorSeries** $*$*ts_*; /$*$ the source code of DAETS is in namespace **daets** $*$/

To evaluate the $\mathbf{b}_s$ in (5.1), we call *EvalEqnsAtStage* from the **TaylorSeries** class. This function stores $\mathbf{b}_s$ at

39 ⟨HO Data Members 28⟩ +≡

double $*$*rhs_*;

40 ⟨compute $\mathbf{b}_s$ 40⟩ ≡

*ts_*→*EvalEqnsAtStage*(s, *rhs_*);

This code is used in chunk 37.

5.1.2.2 Computing β_s

An **HO** object obtains all structural data about the given DAE through a pointer to an object of the **SAdata** class.

41 ⟨HO Data Members 28⟩ +≡

const daets::**SAdata** $*$*sadata_*;

For convenience, we can get the size of the DAE through *sadata_* and store it in *n_*.

42 ⟨HO Data Members 28⟩ +≡

int *n_*;

The offsets are computed and stored during the structural analysis in a **SAdata** object. Calling the functions *get_c* and *get_d* from the **SAdata** class, we can access the equation and variable offsets, respectively. In the constructor of the **HO** class, we store the offsets in *c_* and *d_*.

43 $\langle$HO Data Members 28$\rangle$ +≡

```
std::vector<size_t> c_, d_;
```

After the $\mathbf{b}_s$ is computed and stored at *rhs_*, we compute $\boldsymbol{\beta}_s$.

44 $\langle$compute $\boldsymbol{\beta}_s = \mathbf{C}_s\mathbf{b}_s$ 44$\rangle$ ≡

```
for (int i = 0; i < n_; i++)
    rhs_[i] *= comp_cs_ds(s, c_[i]);
```

This code is used in chunk 37.

5.1.2.3 Solving $\mathbf{A}_0\mathbf{y}_s = \beta_s$

In the **HO** class, we store the current time in

46 $\langle$HO Data Members 28$\rangle$ +≡

```
double t_;
```

The **HO** class maintains a pointer to an object of the **Jacobian** class.

47 $\langle$HO Data Members 28$\rangle$ +≡

```
daets::Jacobian *jac_;
```

We implement the function *CompA0* that computes $\mathbf{A}_0$ by calling *computeJacobian* from the **Jacobian** class and returns it at *jac* column-wise using *getScaledDenseJacobian*.

49 ⟨ Definitions of HO Private Functions 37 ⟩ +≡

```
void HO::CompA0(double *jac)
{
  SetIndepTCsJac();    /* sets $\mathbf{x}_{J_{<0}}$ in the Jacobian object */
  jac_→setT(t_);
  for (int $s = -$sadata_→get_max_d(); $s \le 0$; $s$++)
  {
    jac_→computeJacobian($s$);
    jac_→getScaledDenseJacobian($s$, jac);
  }
  ts_→set_time_coeffs(t_, 1);
  jac_→resetAll();
}
```

Since $\mathbf{A}_0$ depends on $\mathbf{x}_{J_{<0}}$, stored in *indep_tcs_*, we first set these TCs in the **Jacobian** object.

50 ⟨ HO Data Members 28 ⟩ +≡

```
IrregularMatrix⟨double⟩ indep_tcs_;
```

This is done by the function *SetIndepTCsJac*, which calls *set_indep_var_coeff* from the **Jacobian** class. *set_indep_var_coeff* (j, k, tc) sets *tc* to be the kth TC for jth variable.

51 ⟨ Definitions of HO Private Functions 37 ⟩ +≡

```
void HO::SetIndepTCsJac()
```

```
{
  for (int $j = 0$; $j < n\_$; $j$++)
    for (int $k = 0$; $k < d\_[j]$; $k$++)
      jac_→set_indep_var_coeff($j, k$, indep_tcs_($j, k$));
}
```

After computing $\mathbf{A}_0$, we find the LU factorization of $\mathbf{A}_0$ and store it at

52 ⟨HO Data Members 28⟩ +≡

```
double *sys_jac_;
```

The pivot vector that defines the permutation matrix is stored at

53 ⟨HO Data Members 28⟩ +≡

```
int *ipiv_;
```

Using this LU factorization, *LSolve* computes a solution and stores it at *rhs_*.

54 ⟨solve $\mathbf{A}_0\mathbf{y}_s = \boldsymbol{\beta}_s$ 54⟩ ≡

```
daets::LSolve(n_, sys_jac_, ipiv_, rhs_);
```

This code is used in chunk 37.

5.1.2.4 Correcting the initial guess

The initial guess $\widetilde{\mathbf{x}}_{J_s}$ is corrected by (5.8), which is equivalent to

$$\mathbf{x}_{J_s} = \widetilde{\mathbf{x}}_{J_s} - \mathbf{D}_s^{-1}\mathbf{y}_s.$$

We call *get_var_coeff*(j, l) from the **TaylorSeries** class to return the previously stored TC as an initial guess for $(x_j)_l$. Then *set_var_coeff*(j, l, tc) sets *tc* to be the lth TC for jth variable.

56 $\langle$compute and set $\mathbf{x}_{J_s} = \widetilde{\mathbf{x}}_{J_s} - \mathbf{D}_s^{-1}\mathbf{y}_s$ 56$\rangle \equiv$

```
for (int $j = 0$; $j < n\_$; $j$++)
{
    $rhs\_[j]$ /= $comp\_cs\_ds(s, d\_[j])$;     /* $\mathbf{D}_s^{-1}\mathbf{y}_s$ */
    int $l = s + d\_[j]$;
    double $tc = ts\_\to get\_var\_coeff(j, l) - rhs\_[j]$;     /* $\widetilde{\mathbf{x}}_{J_s} - \mathbf{D}_s^{-1}\mathbf{y}_s$ */
    $ts\_\to set\_var\_coeff(j, l, tc)$;     /* $\mathbf{x}_{J_s} = \widetilde{\mathbf{x}}_{J_s} - \mathbf{D}_s^{-1}\mathbf{y}_s$ */
}
```

This code is used in chunk 37.

5.2 Solving nonlinear systems

In non-quasilinear DAEs, (4.4) is nonlinear when $s = 0$. That is, we need to solve the nonlinear system

$$\mathbf{f}_{I_0}\Big(t, \mathbf{x}_{J_{<0}}, \mathbf{x}_{J_0}\Big) = 0, \tag{5.10}$$

whose unknown is $\mathbf{x}_{J_0} \in \mathbb{R}^n$.

The function *CompTCsNonlinear* is implemented to solve the system (5.10) by calling routines in the KINSOL software package [31]. In this function, x stores an initial guess for $\mathbf{x}_{J_0}$. After solving the system successfully, x is updated with the computed solution. The return value of type **HoFlag** is one of the following:

- `HO_SUCCESS`, if $\mathbf{x}_{J_0}$ is computed successfully, or
- `STAGE0_FAIL`, if solving (5.10) fails.

57 $\langle$ Definitions of HO Private Functions 37 $\rangle$ $+\equiv$

```
HoFlag HO::CompTCsNonlinear(double *x)
{
  HoFlag flag;
  ⟨ solve $\mathbf{f}_{I_0} = 0$ 75 ⟩;
  if (flag ≡ HO_CONVERGENT)
    return HO_SUCCESS;
  else
    return STAGE0_FAIL;
}
```

5.2.1 Solving $\mathbf{f}_{I_0} = 0$ by KINSOL

KINSOL is part of the SUNDIALS suite [31]. KINSOL is a general-purpose nonlinear system solver based on Newton-Krylov, Picard, and fixed point solvers. KINSOL's Newton solver employs the Modified or Inexact Newton method and the resulting linear systems can be solved by direct (dense, sparse, or banded) or iterative methods.

To solve (5.10) by KINSOL, we should implement the required functions that evaluate $\mathbf{f}_{I_0}$ and $\partial\mathbf{f}_{I_0}/\partial\mathbf{x}_{J_0}$.

5.2.1.1 Evaluating $\mathbf{f}_{I_0}$

We provide a function *FcnKinsol* of type **KINSysFn** that evaluates $\mathbf{f}_{I_0}$. In this function, x is the given variable vector, and f is the output vector. They are of type **N_Vector** which is a generic vector in SUNDIALS.

59 ⟨ Nonlinear Solver Functions 59 ⟩ ≡

```
int FcnKinsol(N_Vector x, N_Vector f, void *user_data)
{
    ⟨ set parameters to evaluate f_{I_0} 60 ⟩;
    ⟨ evaluate f_{I_0} 64 ⟩;
    return 0;
}
```

See also chunks 65, 101, 177, 178, and 340

This code is used in chunk 364.

In all SUNDIALS solvers, the type **realtype** is used for all floating-point data, with the default being **double**. Calling the function *N_VGetArrayPointer_Serial* from **NVECTOR** operations, we create pointers to **realtype** arrays *xdata* and *fdata* from **N_Vector**s x and f, respectively.

60 ⟨ set parameters to evaluate $\mathbf{f}_{I_0}$ 60 ⟩ ≡

```
realtype *xdata = N_VGetArrayPointer_Serial(x);
realtype *fdata = N_VGetArrayPointer_Serial(f);
```

See also chunk 63.

This code is used in chunk 59.

Given x as a guess for $\mathbf{x}_{J_0}$, the function *EvalEqnsStageZero* below evaluates $\mathbf{f}_{I_0}$ by calling *EvalEqnsAtStage* from the **TaylorSeries** class and stores it at f.

61 ⟨Definitions of HO Public Functions 61⟩ ≡

```
void HO::EvalEqnsStageZero(const double *x, double *f)
{
   SetStageZeroTCs(x);
   ts_→EvalEqnsAtStage(0, f);
}
```

See also chunk 148.

This code is used in chunk 348.

The function *SetStageZeroTCs* sets x to be TCs of variables at stage zero. That is, by calling *set_var_coeff* $(j, d\_[j], x[j])$, $x[j]$ will be the $d_[j]$th TC of the jth variable.

62 ⟨Definitions of HO Private Functions 37⟩ +≡

```
void HO::SetStageZeroTCs(const double *x)
{
   for (int j = 0; j < n_; j++)
      ts_→set_var_coeff(j, d_[j], x[j]);
}
```

Assume that an object of the **HO** class is passed through the *user_data* parameter. We cast a pointer to the **HO** class from *user_data*.

63 ⟨ set parameters to evaluate $\mathbf{f}_{I_0}$ 60 ⟩ +≡

```
HO *ho = (HO *) user_data;
```

Now, we call *EvalEqnsStageZero* to

64 ⟨ evaluate $\mathbf{f}_{I_0}$ 64 ⟩ ≡

```
ho→EvalEqnsStageZero(xdata, fdata);
```

This code is used in chunk 59.

5.2.1.2 Computing $\partial\mathbf{f}_{I_0}/\partial\mathbf{x}_{J_0}$

We provide a function *JacKinsol* of type **KINLsJacFn** that evaluates $\partial\mathbf{f}_{I_0}/\partial\mathbf{x}_{J_0}$. In this function, x is the given variable vector, and f is the output vector. The output Jacobian matrix is stored at *sun_jac* of type **SUNMatrix** which is a generic matrix in SUNDIALS.

65 ⟨ Nonlinear Solver Functions 59 ⟩ +≡

```
int JacKinsol(N_Vector x, N_Vector f, SUNMatrix sun_jac, void *user_data, N_Vector
        tmp1, N_Vector tmp2)
{
  ⟨ set parameters to evaluate $\frac{\partial\mathbf{f}_{I_0}}{\partial\mathbf{x}_{J_0}}$ 66 ⟩;
  ⟨ compute $\frac{\partial\mathbf{f}_{I_0}}{\partial\mathbf{x}_{J_0}}$ 71 ⟩;
  ⟨ store the computed $\frac{\partial\mathbf{f}_{I_0}}{\partial\mathbf{x}_{J_0}}$ at sun_jac 73 ⟩;
  return 0;
}
```

Calling the function *SUNDenseMatrix_Columns*, we obtain the size of the system and store it in n. In SUNDIALS solvers the type **sunindextype** is used for integer data.

66 $\langle$ set parameters to evaluate $\dfrac{\partial \mathbf{f}_{I_0}}{\partial \mathbf{x}_{J_0}}$ 66 $\rangle \equiv$

sunindextype $n = \mathit{SUNDenseMatrix\_Columns}(\mathit{sun\_jac});$

See also chunks 67, 68, and 70

This code is used in chunk 65.

Calling the function *N_VGetArrayPointer_Serial*, we create a pointer to **realtype** array *xdata* from x.

67 $\langle$ set parameters to evaluate $\dfrac{\partial \mathbf{f}_{I_0}}{\partial \mathbf{x}_{J_0}}$ 66 $\rangle +\!\equiv$

realtype $*\mathit{xdata} = \mathit{N\_VGetArrayPointer\_Serial}(x);$

The function *SUNDenseMatrix_Data* is called to return the pointer *jacobian* to the data array for *sun_jac*.

68 $\langle$ set parameters to evaluate $\dfrac{\partial \mathbf{f}_{I_0}}{\partial \mathbf{x}_{J_0}}$ 66 $\rangle +\!\equiv$

realtype $*\mathit{jacobian} = \mathit{SUNDenseMatrix\_Data}(\mathit{sun\_jac});$

To compute $\mathbf{A}_0$ for given $\mathbf{x}_{J_0}$, we first set these TCs in the **Jacobian** object. This is done by the function *SetStageZeroTCsJac*, which calls *set_indep_var_coeff* from the **Jacobian** class.

69 $\langle$ Definitions of HO Private Functions 37 $\rangle +\!\equiv$

void HO :: *SetStageZeroTCsJac*(**const double** $*x$)

{

```
    for (int $j = 0$; $j < n\_$; $j{+}{+}$)
        jac_→set_indep_var_coeff($j, d\_[j], x[j]$);
}
```

Then, we cast a pointer to the **HO** class from *user_data*.

70 ⟨ set parameters to evaluate $\frac{\partial \mathbf{f}_{I_0}}{\partial \mathbf{x}_{J_0}}$ 66 ⟩ +≡

```
HO *ho = (HO *) user_data;
```

71 ⟨ compute $\frac{\partial \mathbf{f}_{I_0}}{\partial \mathbf{x}_{J_0}}$ 71 ⟩ ≡

```
ho→SetStageZeroTCsJac(xdata);
```

See also chunk 72.

This code is used in chunk 65.

Now, we call *CompA0* from the **HO** class to compute $\partial \mathbf{f}_{I_0}/\partial \mathbf{x}_{J_0}$ and store it column-wise at *jacobian*.

72 ⟨ compute $\frac{\partial \mathbf{f}_{I_0}}{\partial \mathbf{x}_{J_0}}$ 71 ⟩ +≡

```
ho→CompA0(jacobian);
```

In SUNDIALS, `SM_COLUMN_D`(*sun_jac*, j) returns a pointer to the first element of the jth column of *sun_jac*.

73 ⟨ store the computed $\frac{\partial \mathbf{f}_{I_0}}{\partial \mathbf{x}_{J_0}}$ at *sun_jac* 73 ⟩ ≡

```
for (int col = 0; col < n; col++)
    SM_COLUMN_D(sun_jac, col) = jacobian + col * n;
```

This code is used in chunk 65.

5.2.1.3 Calling KINSOL

After providing the functions *FcnKinsol* and *JacKinsol* for evaluating $\mathbf{f}_{I_0}$ and $\partial \mathbf{f}_{I_0}/\partial \mathbf{x}_{J_0}$, respectively, we pass them as arguments of the function *NSolveKin* (implemented in Appendix C). Here, we set the tolerance to be $\max\{10\epsilon, \mathrm{tol}/100\}$, where ϵ is the machine precision, and tol is the integration tolerance sotred in

74 ⟨HO Data Members 28⟩ +≡

double *tol_*;

75 ⟨solve $\mathbf{f}_{I_0} = 0$ 75⟩ ≡

double *min_tol* = 10 ∗ **std**::*numeric_limits* < **double** > ::*epsilon*( );

double *tol* = **std**::*max*(*min_tol*, *tol_*/100);

int *max_num_iter* = 4;

flag = *NSolveKin*(*n_*, *tol*, *max_num_iter*, *x*, *FcnKinsol*, *JacKinsol*, (**void** ∗) **this**);

This code is used in chunk 57.

Chapter 6

Computing gradients of Taylor coefficients

To solve the HO system (4.20) using Newton's method, we need to compute the gradients

$$\frac{\partial (x_j)_{k+r}}{\partial \mathbf{x}_{J_{<0}}},$$

for $j = 0, \ldots, n-1$, $k = 0, \ldots, d_j - 1$, and $r = 0, \ldots, q$. Let $\nabla = \partial/\partial \mathbf{x}_{J_{<0}}$ and denote for a row or column vector $\mathbf{u}$ with elements $u_1, u_2, \ldots, u_m$,

$$\boldsymbol{\nabla}\mathbf{u} = \begin{bmatrix} \nabla u_1 \\ \nabla u_2 \\ \vdots \\ \nabla u_m \end{bmatrix}.$$

By (4.10) and the computational scheme for TCs in §4.1, we can obtain the required gradients if we compute the Jacobian matrices $\boldsymbol{\nabla}\mathbf{x}_{J_s}$ for $0 \le s < q$.

Recall that $\mathbf{x}_{J_{<0}}$ is a vector with N independent TCs, and $\mathbf{x}_{J_s}$ is a vector of size n containing TCs of state variables at stage $s \geq 0$. Hence, $\boldsymbol{\nabla}\mathbf{x}_{J_s}$ is a Jacobian matrix of size $n \times N$. In this chapter, we show how $\boldsymbol{\nabla}\mathbf{x}_{J_s}$ can be computed using automatic differentiation. First, we derive a method for their computations in §6.1. Then, we implement the required functions to compute them in §6.2.

6.1 Computational scheme for gradients

Differentiating (4.4) with respect to $\mathbf{x}_{J_{<0}}$, we obtain

$$\frac{\partial \mathbf{f}_{I_s}}{\partial \mathbf{x}_{J_{<s}}} \boldsymbol{\nabla}\mathbf{x}_{J_{<s}} + \frac{\partial \mathbf{f}_{I_s}}{\partial \mathbf{x}_{J_s}} \boldsymbol{\nabla}\mathbf{x}_{J_s} = 0. \tag{6.1}$$

Denote the left hand side of equation (6.1) by

$$\mathbf{g}_{I_s}\Big(t, \mathbf{x}_{J_{\leq s}}, \boldsymbol{\nabla}\mathbf{x}_{J_{<s}}, \boldsymbol{\nabla}\mathbf{x}_{J_s}\Big).$$

Given $\mathbf{x}_{J_{\leq s}}$ and $\boldsymbol{\nabla}\mathbf{x}_{J_{<s}}$ at t, we can find $\boldsymbol{\nabla}\mathbf{x}_{J_s}$ by solving the system

$$\mathbf{g}_{I_s}\Big(t, \mathbf{x}_{J_{\leq s}}, \boldsymbol{\nabla}\mathbf{x}_{J_{<s}}, \boldsymbol{\nabla}\mathbf{x}_{J_s}\Big) = 0, \tag{6.2}$$

which is linear in $\boldsymbol{\nabla}\mathbf{x}_{J_s}$, and $\partial \mathbf{g}_{I_s} / \partial\Big(\boldsymbol{\nabla}\mathbf{x}_{J_s}\Big)$ is $\mathbf{A}_s$ in (5.5). As in §5.1, we can apply one iteration of Newton's method to (6.2) to find $\boldsymbol{\nabla}\mathbf{x}_{J_s} \in \mathbb{R}^{n \times N}$. That is, given $\mathbf{x}_{J_{\leq s}}$ and $\boldsymbol{\nabla}\mathbf{x}_{J_{<s}}$ as constants, and $\widetilde{\boldsymbol{\nabla}\mathbf{x}}_{J_s}$ as a an initial guess for $\boldsymbol{\nabla}\mathbf{x}_{J_s}$,

$$\boldsymbol{\nabla}\mathbf{x}_{J_s} = \widetilde{\boldsymbol{\nabla}\mathbf{x}}_{J_s} - \mathbf{A}_s^{-1}\mathbf{B}_s, \tag{6.3}$$

where

$$\mathbf{B}_s = \mathbf{g}_{I_s}\Big(t, \mathbf{x}_{J_{\le s}}, \boldsymbol{\nabla}\mathbf{x}_{J_{<s}}, \widetilde{\boldsymbol{\nabla}\mathbf{x}}_{J_s}\Big) \tag{6.4}$$

is a Jacobian matrix of size $n \times N$ containing the gradients of TCs of equations at stage s.

Example 6.1. For the pendulum in Example 2.1, we have $\mathbf{x}_{J_{<0}} = [x_0, x_1, y_0, y_1]$. Clearly,

$$\nabla x_0 = [1, 0, 0, 0],$$

$$\nabla x_1 = [0, 1, 0, 0],$$

$$\nabla y_0 = [0, 0, 1, 0],$$

$$\nabla y_1 = [0, 0, 0, 1].$$

The computational schemes for $\boldsymbol{\nabla}\mathbf{x}_{J_0}$ and $\boldsymbol{\nabla}\mathbf{x}_{J_1}$ are illustrated as follows.

At stage $s = 0$, (4.4) is

$$0 = 2x_2 + \lambda_0 x_0,$$

$$0 = 2y_2 + \lambda_0 y_0 + G,$$

$$0 = 2x_0x_2 + x_1^2 + 2y_0y_2 + y_1^2.$$

Differentiating with respect to $\mathbf{x}_{J_{<0}}$, we obtain

$$0 = 2\nabla x_2 + \lambda_0 \nabla x_0 + x_0 \nabla \lambda_0,$$

$$0 = 2\nabla y_2 + \lambda_0 \nabla y_0 + y_0 \nabla \lambda_0,$$

$$0 = 2x_2\nabla x_0 + 2x_0\nabla x_2 + 2x_1\nabla x_1 + 2y_2\nabla y_0 + 2y_0\nabla y_2 + 2y_1\nabla y_1.$$

It can be written as

$$0 = \underbrace{\begin{bmatrix} 2 & 0 & x_0 \\ 0 & 2 & y_0 \\ 2x_0 & 2y_0 & 0 \end{bmatrix}}_{\mathbf{A}_0} \underbrace{\begin{bmatrix} \nabla x_2 \\ \nabla y_2 \\ \nabla \lambda_0 \end{bmatrix}}_{\boldsymbol{\nabla} \mathbf{x}_{J_0}} + \underbrace{\begin{bmatrix} \lambda_0 \nabla x_0 \\ \lambda_0 \nabla y_0 \\ 2x_2 \nabla x_0 + 2x_1 \nabla x_1 + 2y_2 \nabla y_0 + 2y_1 \nabla y_1 \end{bmatrix}}_{\frac{\partial \mathbf{f}_{I_0}}{\partial \mathbf{x}_{J_{<0}}} \boldsymbol{\nabla} \mathbf{x}_{J_{<0}}}.$$

At stage $s = 1$, (4.4) is

$$0 = 6x_3 + \lambda_0 x_1 + \lambda_1 x_0,$$

$$0 = 6y_3 + \lambda_0 y_1 + \lambda_1 y_0,$$

$$0 = 2x_0 x_3 + 2x_1 x_2 + 2y_0 y_3 + 2y_1 y_2.$$

Differentiating with respect to $\mathbf{x}_{J_{<0}}$, we obtain

$$\begin{aligned} 0 &= 6\nabla x_3 + \lambda_0 \nabla x_1 + x_1 \nabla \lambda_0 + x_0 \nabla \lambda_1 + \lambda_1 \nabla x_0, \\ 0 &= 6\nabla y_3 + \lambda_0 \nabla y_1 + y_1 \nabla \lambda_0 + y_0 \nabla \lambda_1 + \lambda_1 \nabla y_0, \\ 0 &= 2x_3 \nabla x_0 + 2x_0 \nabla x_3 + 2x_1 \nabla x_2 + 2x_2 \nabla x_1 + 2y_3 \nabla y_0 \\ &\quad + 2y_0 \nabla y_3 + 2y_1 \nabla y_2 + 2y_2 \nabla y_1. \end{aligned}$$

It can be written as

$$0 = \underbrace{\begin{bmatrix} 6 & 0 & x_0 \\ 0 & 6 & y_0 \\ 2x_0 & 2y_0 & 0 \end{bmatrix}}_{\mathbf{A}_1} \underbrace{\begin{bmatrix} \nabla x_3 \\ \nabla y_3 \\ \nabla \lambda_1 \end{bmatrix}}_{\boldsymbol{\nabla}\mathbf{x}_{J_1}}$$

$$+ \underbrace{\begin{bmatrix} \lambda_0 \nabla x_1 + x_1 \nabla \lambda_0 + x_0 \nabla \lambda_1 + \lambda_1 \nabla x_0 \\ \lambda_0 \nabla y_1 + y_1 \nabla \lambda_0 + y_0 \nabla \lambda_1 + \lambda_1 \nabla y_0 \\ 2x_3 \nabla x_0 + 2x_1 \nabla x_2 + 2x_2 \nabla x_1 + 2y_3 \nabla y_0 + 2y_1 \nabla y_2 + 2y_2 \nabla y_1 \end{bmatrix}}_{\frac{\partial \mathbf{f}_{I_1}}{\partial \mathbf{x}_{J_{<1}}} \boldsymbol{\nabla}\mathbf{x}_{J_{<1}}}.$$

6.2 Implementation

The number of independent TCs, N (4.6), is stored in

78 $\langle$ HO Data Members 28 $\rangle$ +≡

```
int num_indep_tcs_;
```

$\mathbf{B}_s$ is a matrix of size $n \times N$, and the kth column of this matrix contains the gradients of the TCs of equations at stage s in terms of kth independent TC. Let $[\mathbf{A}]_k$ denote the kth column of a matrix $\mathbf{A}$. We need to solve the system of linear equations

$$\mathbf{A}_s[\mathbf{G}_s]_k = [\mathbf{B}_s]_k, \quad k = 0, 1, \ldots, N-1$$

to find the kth column of the matrix $\mathbf{G}_s$. By (5.5), we write

$$\mathbf{C}_s^{-1}\mathbf{A}_0\mathbf{D}_s[\mathbf{G}_s]_k = [\mathbf{B}_s]_k,$$

which leads to

$$\mathbf{A}_0\mathbf{D}_s[\mathbf{G}_s]_k = \mathbf{C}_s[\mathbf{B}_s]_k. \tag{6.5}$$

That is, we need to

- compute $\mathbf{B}_s$,
- compute $[\mathcal{B}_s]_k = \mathbf{C}_s[\mathbf{B}_s]_k$,
- solve $\mathbf{A}_0[\mathbf{Y}_s]_k = [\mathcal{B}_s]_k$ to find $[\mathbf{Y}_s]_k$, and
- set $[\boldsymbol{\nabla}\mathbf{x}_{J_s}]_k = [\widetilde{\boldsymbol{\nabla}\mathbf{x}}_{J_s}]_k - \mathbf{D}_s^{-1}[\mathbf{Y}_s]_k$.

This is implemented by the function *CompGradients* which computes $\boldsymbol{\nabla}\mathbf{x}_{J_s}$ at stages $s = 0, 1, \ldots, q-1$ where $\mathbf{x}_{J_{<s}}$ is given.

79 ⟨Definitions of HO Private Functions 37⟩ +≡

```
void HO::CompGradients(int q)
{
  ⟨initialize $\boldsymbol{\nabla}\mathbf{x}_{J_{<q}}$ 86⟩;
  ⟨set $\mathbf{x}_{J_{<q}}$ 88⟩;
  for (int $s = 0$; $s < q$; $s$++)
  {
    ⟨compute $\mathbf{B}_s$ 91⟩;
    for (int $k = 0$; $k <$ num_indep_tcs_; $k$++)
    {
      ⟨compute $[\mathcal{B}_s]_k = \mathbf{C}_s[\mathbf{B}_s]_k$ 93⟩;
```

```
        ⟨ solve $\mathbf{A}_0[\mathbf{Y}_s]_k = [\mathcal{B}_s]_k$ 94 ⟩;
        ⟨ compute and set $[\nabla\mathbf{x}_{J_s}]_k = [\widetilde{\nabla\mathbf{x}}_{J_s}]_k - \mathbf{D}_s^{-1}[\mathbf{Y}_s]_k$ 96 ⟩;
      }
    }
  }
```

6.2.1 Initializing gradients

A **Gradients** object obtains all structural data of the given DAE through a pointer to a **SAdata** object.

80 ⟨ Gradients Data Members 80 ⟩ ≡

```
  const daets::SAdata *sadata_;
```

See also chunks 81, 89, 164, and 352

This code is used in chunk 25.

We use the forward mode of FADBAD++ to differentiate TCs. The input variables are in

81 ⟨ Gradients Data Members 80 ⟩ +≡

```
  fadbad::T⟨fadbad::F⟨double⟩⟩ *grad_in_;
```

First, we need to initialize $\nabla\mathbf{x}_{J_{<q}}$. We define the function *set_var_grad_component* which sets value for the derivative of $(x_j)_l$ with respect to the kth independent TC.

82 ⟨ Gradients Public Functions 82 ⟩ ≡

```
  void set_var_grad_component(int j, int l, int k, double der)
  {
    grad_in_[j][l].d(k) = der;
  }
```

See also chunks 84, 87, 90, 92, 95, 165, 353, and 354

This code is used in chunk 25.

The $\nabla \mathbf{x}_{J_{<0}}$ is the $N \times N$ identity matrix and we initialize gradients of higher-order TCs with zeros. This is implemented by the function *initialize_gradients*. In this function, we call **fadbad**::*diff* to indicate the independent TCs that we want to differentiate with respect to. Calling *diff* (k, m), m becomes the number of independent TCs and k denotes the index of the independent TC.

84 ⟨ Gradients Public Functions 82 ⟩ +≡

```
  void initialize_gradients(int q, int num_indep_tcs)
  {
    int k = 0;      /* kth independent TC */
    for (int j = 0; j < sadata_→get_size( ); j++)
    {
      int dj = sadata_→get_d(j);
      for (int l = 0; l < dj; l++)
        grad_in_[j][l].diff(k++, num_indep_tcs);      /* ∇x_{J<0} identity matrix */
      for (int l = dj; l < q + dj; l++)
      {
        grad_in_[j][l].diff(0, num_indep_tcs);
        $set\_var\_grad\_component(j, l, 0, 0);$
      }
    }
  }
```

The **HO** class maintains a pointer to a **Gradients** object.

85 ⟨ HO Data Members 28 ⟩ +≡

```
**Gradients** $*grads\_;$
```

Now, we can call *initialize_gradients*.

86 ⟨ initialize $\nabla \mathbf{x}_{J_{<q}}$ 86 ⟩ ≡

```
$grads\_\rightarrow initialize\_gradients(q, num\_indep\_tcs\_);$
```

This code is used in chunk 79.

Since we always compute $\mathbf{x}_{J_{<q}}$ before computing gradients, we copy it's entries to $\nabla \mathbf{x}_{J_{<q}}$. The function *set_var_coeff* is defined to set the lth TC of x_j in *grad_in_*.

87 ⟨ Gradients Public Functions 82 ⟩ +≡

```
**void** $set\_var\_coeff$(**int** $j$, **int** $l$, **double** $tc$)
{
  $grad\_in\_[j][l].x() = tc;$
}
```

88 ⟨ set $\mathbf{x}_{J_{<q}}$ 88 ⟩ ≡

```
**for** (**int** $j = 0;\ j < n\_;\ j{+}{+}$)
```

```
    for (int $l = 0$; $l < d\_[j] + q$; $l$++)
    {
        double $tc$ = $ts\_ \rightarrow get\_var\_coeff(j, l)$;
        $grads\_ \rightarrow set\_var\_coeff(j, l, tc)$;
    }
```

This code is used in chunk 79.

6.2.2 Computing $\mathbf{B}_s$

We first define the function *EvalEqnCoeffGrad* which computes an entry of $\mathbf{B}_s$ in (6.4) and stores it in *grad_out_*. This contains the output variables in the differentiation process.

89 ⟨ Gradients Data Members 80 ⟩ +≡

```
  fadbad::T⟨fadbad::F⟨double⟩⟩ *grad_out_;
```

EvalEqnCoeffGrad(i, l) computes $\nabla(f_i)_l$. This is done by calling the function *eval* from FADBAD++.

90 ⟨ Gradients Public Functions 82 ⟩ +≡

```
  void EvalEqnCoeffGrad(int i, int l)
  {
      grad_out_[i].reset( );
      grad_out_[i].eval(l);
  }
```

Then by calling *EvalEqnCoeffGrad* all entries of $\mathbf{B}_s$ are computed.

91 $\langle$ compute $\mathbf{B}_s$ 91 $\rangle \equiv$

```
for (int $i = 0$; $i <$ n_; $i$++)
{
    int $l$ = c_[$i$] + $s$;
    grads_→EvalEqnCoeffGrad($i, l$);
}
```

This code is used in chunk 79.

6.2.3 Computing $[\mathcal{B}_s]_k$

To access the computed gradients in $\mathbf{B}_s$, we define the function *get_eqn_grad_component*. Denote by y_k the kth component of $\mathbf{x}_{J_{<0}}$. Then, *get_eqn_grad_component*(i, l, k) returns $\partial(f_i)_l/\partial y_k$ which is the ith element of the column vector $[\mathbf{B}_s]_k$ where $l = s + c_i$.

92 $\langle$ Gradients Public Functions 82 $\rangle$ +$\equiv$

```
double get_eqn_grad_component(int $i$, int $l$, int $k$) const
{
    return grad_out_[$i$][$l$].$d$($k$);
}
```

Here, we multiply these elements by the corresponding entries on the main diagonal of matrix $\mathbf{C}_s$. The resulting column vector is stored at *rhs_*.

93 $\langle$ compute $[\mathcal{B}_s]_k = \mathbf{C}_s[\mathbf{B}_s]_k$ 93 $\rangle \equiv$

```
for (int $i = 0$; $i <$ n_; $i$++)
{
    int $ci = c\_[i]$;
    int $l = ci + s$;
    $rhs\_[i] = grads\_\rightarrow get\_eqn\_grad\_component(i, l, k) * comp\_cs\_ds(s, ci)$;
}
```

This code is used in chunk 79.

6.2.4 Solving $\mathbf{A}_0[\mathbf{Y}_s]_k = [\mathcal{B}_s]_k$

We have the LU factorization of $\mathbf{A}_0$ stored at *sys_jac_* with *ipiv_* that contains the pivot indices that define the required permutation matrix. Using this LU factorization, *LSolve* computes a solution of $\mathbf{A}_0[\mathbf{Y}_s]_k = [\mathcal{B}_s]_k$ and stores it in *rhs_*.

94 ⟨ solve $\mathbf{A}_0[\mathbf{Y}_s]_k = [\mathcal{B}_s]_k$ 94 ⟩ ≡

```
  daets::LSolve(n_, sys_jac_, ipiv_, rhs_);
```

This code is used in chunk 79.

6.2.5 Correcting initial guess

The kth column of the initial guess $\widetilde{\nabla \mathbf{x}}_{J_s}$ is corrected by (6.3) which is equivalent to

$$[\nabla \mathbf{x}_{J_s}]_k = [\widetilde{\nabla \mathbf{x}}_{J_s}]_k - \mathbf{D}_s^{-1}[\mathbf{Y}_s]_k.$$

To access the computed gradients in $\nabla \mathbf{x}_{J_s}$, we define the function *get_var_grad_component*. Denote by y_k the kth component of $\mathbf{x}_{J_{<0}}$. Then, *get_var_grad_component*(j, l, k) returns $\partial(x_j)_l/\partial y_k$ which is the jth element of the column vector $[\nabla \mathbf{x}_{J_s}]_k$ where $l = s + d_j$.

95 ⟨ Gradients Public Functions 82 ⟩ +≡

```
  double get_var_grad_component(int j, int l, int k) const
```

```
{
    return grad_in_[j][l].d(k);
}
```

Also, the function *set_var_grad_component* is employed to set the corrected gradient.

96 ⟨compute and set $[\boldsymbol{\nabla}\mathbf{x}_{J_s}]_k = [\widetilde{\boldsymbol{\nabla}\mathbf{x}}_{J_s}]_k - \mathbf{D}_s^{-1}[\mathbf{Y}_s]_k$ 96⟩ ≡

```
for (int j = 0; j < n_; j++)
{
    int l = d_[j] + s;
    double der = grads_→get_var_grad_component(j, l, k)
            − rhs_[j]/comp_cs_ds(s, d_[j]);
    grads_→set_var_grad_component(j, l, k, der);
}
```

This code is used in chunk 79.

Chapter 7

Solving the Hermite-Obreschkoff system

In this chapter, we solve the Hermite-Obreschkoff system (4.20) using Newton's method. Let $\mathbf{f}_{\text{HO}}(\mathbf{x}_{J_{<0}})$ be the vector of size N created by concatenating rows of the irregular matrix $\mathbf{F}(\mathbf{x}_{J_{<0}})$ in (4.18). We compute

$$\mathbf{x}_{J_{<0}}^{m} = \mathbf{x}_{J_{<0}}^{m-1} - \mathbf{J}_{\text{HO}}^{-1}\,\mathbf{f}_{\text{HO}}(\mathbf{x}_{J_{<0}}^{m-1}), \quad m = 1, 2, \ldots, \tag{7.1}$$

where an initial guess $\mathbf{x}_{J_{<0}}^{0} \in \mathbb{R}^N$ is constructed from the past behaviour of the solution, and $\mathbf{J}_{\text{HO}} = \partial \mathbf{f}_{\text{HO}} / \partial \mathbf{x}_{J_{<0}}$ at $\mathbf{x}_{J_{<0}}^{0}$. The computation of $\mathbf{x}_{J_{<0}}^{0}$ is discussed in §8.2. We describe how we test the convergence of the iteration (7.1) and improve the reliability of this test in §7.1. Then, this iteration is implemented in §7.2. Finally, in §7.3 we implement a function that performs one step of the HO method.

7.1 Convergence of the iteration

In general, (4.20) may have multiple solutions. Provided that $\mathbf{x}^0_{J_{<0}}$ is a good guess, we wish to find the closest solution to it. The iterative procedure (7.1) should be a contraction mapping in a ball about $\mathbf{x}^0_{J_{<0}}$. Contraction from $\mathbf{x}^0_{J_{<0}}$ is a hypothesis guaranteeing local existence and uniqueness of the solution [60]. By observing the iterates, we would like to gain some confidence that the process is converging reasonably fast so as to obtain a solution economically. Our experience suggests that it is better to reduce the step size in order to improve the accuracy of $\mathbf{x}^0_{J_{<0}}$ and the rate of convergence of the iteration than to iterate many times [60]. Here, we iterate (7.1) at most 4 times.

To obtain $\mathbf{x}^m_{J_{<0}}$, we first solve the linear system

$$\mathbf{J}_{\text{HO}}\,\boldsymbol{\delta}^m = \mathbf{f}_{\text{HO}}(\mathbf{x}^{m-1}_{J_{<0}}), \tag{7.2}$$

whose unknown is $\boldsymbol{\delta}^m \in \mathbb{R}^N$. Then we write

$$\mathbf{x}^m_{J_{<0}} = \mathbf{x}^{m-1}_{J_{<0}} - \boldsymbol{\delta}^m.$$

The matrix $\mathbf{J}_{\text{HO}}$ is nonsingular for sufficiently small stepsize h [25]. This matrix is usually rather ill-conditioned with the consequence that solving a linear system involving it may not be very accurate. Assuming $\mathbf{x}^0_{J_{<0}}$ is close to the solution, the difference $\|\boldsymbol{\delta}^m\| = \|\mathbf{x}^{m-1}_{J_{<0}} - \mathbf{x}^m_{J_{<0}}\|$ is rather small. Computing $\boldsymbol{\delta}^m$ from (7.2), we can usually get an accurate $\mathbf{x}^m_{J_{<0}}$ even when $\boldsymbol{\delta}^m$ has only a few digits correct [59]. To solve (7.2), the matrix $\mathbf{J}_{\text{HO}}$ is factored into a product of an upper and lower triangular matrix. $\mathbf{J}_{\text{HO}}$ is usually dense and we perform the factorization and the solution of the system (7.2) by routines in the LAPACK software package. When the system to be solved is large, the costs of computing and factoring $\mathbf{J}_{\text{HO}}$ dominate the cost of the integration.

We use a weighted root mean square (WRMS) norm, denoted $\|.\|_{\text{wrms}}$, for all error-like quantities [31]

$$\|\mathbf{v}\|_{\text{wrms}} = \sqrt{\frac{1}{N}\sum_{k=0}^{N-1}\left(\frac{v_k}{\text{atol} + \text{rtol}\cdot|y_k|}\right)^2}, \tag{7.3}$$

where rtol and atol are relative and absolute error tolerances, N is defined in (4.6), and y_k for $k = 0, \dots, N-1$, are components of the solution at the beginning of the step. For brevity, we drop the subscript wrms on norms in what follows.

If $\widetilde{\mathbf{x}}_{J<0}$ is the solution of (4.20), when will the iteration error $\|\widetilde{\mathbf{x}}_{J<0} - \mathbf{x}^m_{J<0}\|$ be sufficiently small? It is well-known that [59]

$$\|\widetilde{\mathbf{x}}_{J<0} - \mathbf{x}^m_{J<0}\| \leq \frac{\rho}{1-\rho}\|\boldsymbol{\delta}^m\|,$$

where ρ is an estimate of the rate of convergence of the iteration (7.1). Following Shampine [59], we continue the iteration until

$$\frac{\rho}{1-\rho}\|\boldsymbol{\delta}^m\| < 0.1, \tag{7.4}$$

so that the iteration error $\|\widetilde{\mathbf{x}}_{J<0} - \mathbf{x}^m_{J<0}\|$ will be sufficiently small. After evaluating the solution of (4.20) by (7.1), we may not accept the result because the discretization error $\|e_{pq}h^{p+q+1}\mathbf{E}\|$ in (4.19) is large. The constant 0.1 in (7.4) was chosen so that the errors due to terminating the iteration (7.1) would not adversely affect the discretization error estimates.

The rate of convergence is estimated, whenever two or more iterations have been taken, by [59]

$$\rho \approx \frac{\|\boldsymbol{\delta}^m\|}{\|\boldsymbol{\delta}^{m-1}\|}. \tag{7.5}$$

If $\rho > 0.9$, or $m > 4$, and the iteration has not yet converged, then the iteration (7.1) is considered to have failed at solving (4.20).

7.2 Implementation

We implement the function *NSolve* based on the considerations in §7.1. In this function, *size* is the size of the system, and x contains an initial guess for the solution. After solving the system successfully, x will be updated with the computed solution. *weight* is used to compute the required WRMS norms. *Fcn* and *Jac* are functions which compute $\mathbf{f}_{\text{HO}}(\mathbf{u})$ and $\mathbf{J}_{\text{HO}}$ for a given value of a vector $\mathbf{u}$, respectively. *user_data* is a pointer to user data, which here is a pointer to the **HO** class.

NSolve returns one of the following values of type **HoFlag**:

- `SYS_JAC_SINGULAR`, if evaluating $\mathbf{f}_{\text{HO}}(\mathbf{x}_{J_{<0}}^{m-1})$ fails as the system Jacobian is singular,
- `STAGE0_FAIL`, if evaluating $\mathbf{f}_{\text{HO}}(\mathbf{x}_{J_{<0}}^{m-1})$ fails as solving $\mathbf{f}_{I_0} = 0$ in (5.10) fails,
- `HO_SUCCESS`, if $\mathbf{f}_{\text{HO}}(\mathbf{x}_{J_{<0}}^{m-1})$ is computed successfully,
- `HO_JAC_SINGULAR`, if $\mathbf{J}_{\text{HO}}$ is singular, or
- `HO_CONVERGENT`, if the iteration (7.2) for solving $\mathbf{f}_{\text{HO}}(\mathbf{x}_{J_{<0}}) = 0$ is convergent.

```
101 ⟨Nonlinear Solver Functions 59⟩ +≡
  HoFlag NSolve(int size, const IrregularMatrix⟨double⟩ &weight, double *x, EvalF
        Fcn, EvalJ Jac, double *residual, double *jacobian, int *ipiv, void
        *user_data)
  {
    ⟨declare variables for nonlinear solver 113⟩;
    ⟨compute $\mathbf{f}_{\text{HO}}(\mathbf{x}^0_{J<0})$ 102⟩;
    ⟨compute $\mathbf{J}_{\text{HO}}$ 103⟩;
    ⟨find LU factorization of $\mathbf{J}_{\text{HO}}$ 104⟩;
    for (int $m = 1$; $m \le 4$; $m$++)
    {
      ⟨solve $\mathbf{J}_{\text{HO}}\boldsymbol{\delta}^m = \mathbf{f}_{\text{HO}}(\mathbf{x}^{m-1}_{J<0})$ 106⟩;
      ⟨evaluate $\|\boldsymbol{\delta}^m\|$ 112⟩;
      ⟨compute iteration_error $= \frac{\rho}{1-\rho}\|\boldsymbol{\delta}^m\|$ 115⟩;
      ⟨compute $\mathbf{x}^m_{J<0} = \mathbf{x}^{m-1}_{J<0} - \boldsymbol{\delta}^m$ 108⟩;
      ⟨compute $\mathbf{f}_{\text{HO}}(\mathbf{x}^m_{J<0})$ 109⟩;
      if (iteration_error < 0.1)
        return HO_CONVERGENT;
      ⟨save $\|\boldsymbol{\delta}^m\|$ for computing $\rho$ in next iteration 116⟩;
    }
    return HO_SUCCESS;
```

}

We call a function *Fcn* which computes $\mathbf{f}_{\mathrm{HO}}(\mathbf{x}^0_{J_{<0}})$ and stores it at *residual* (see §7.3.3.2).

102 $\langle$ compute $\mathbf{f}_{\mathrm{HO}}(\mathbf{x}^0_{J_{<0}})$ 102 $\rangle \equiv$

```
HoFlag fcn_flag = Fcn(size, x, residual, user_data);
if (fcn_flag ≡ SYS_JAC_SINGULAR ∨ fcn_flag ≡ STAGE0_FAIL)
   return fcn_flag;
```

This code is used in chunk 101.

Also, we call a function *Jac* which computes and stores $\mathbf{J}_{\mathrm{HO}}$ at *jacobian* (see §7.3.3.2).

103 $\langle$ compute $\mathbf{J}_{\mathrm{HO}}$ 103 $\rangle \equiv$

```
Jac(size, x, jacobian, user_data);
```

This code is used in chunk 101.

To solve (7.2), the matrix $\mathbf{J}_{\mathrm{HO}}$ is factored into a product of an upper and lower triangular matrix.

104 $\langle$ find LU factorization of $\mathbf{J}_{\mathrm{HO}}$ 104 $\rangle \equiv$

```
int jac_flag;
daets::LU(size, jacobian, ipiv, &jac_flag);
```

See also chunk 105.

This code is used in chunk 101.

Now, *jacobian* contains both lower and upper triangular matrices. The pivot vector that defines the permutation matrix is stored at *ipiv*. If the Jacobian is singular we terminate with the return value HO_JAC_SINGULAR.

105 $\langle$ find LU factorization of $\mathbf{J}_{\text{HO}}$ 104 $\rangle +\equiv$

```
if ($jac\_flag \neq 0$)
    return HO_JAC_SINGULAR;
```

Using the above factorization, *LSolve* computes $\boldsymbol{\delta}^m$ and stores it at *residual*.

106 $\langle$ solve $\mathbf{J}_{\text{HO}}\,\boldsymbol{\delta}^m = \mathbf{f}_{\text{HO}}(\mathbf{x}^{m-1}_{J<0})$ 106 $\rangle \equiv$

```
daets::LSolve(size, jacobian, ipiv, residual);
```

This code is used in chunk 101.

Now, we can call the function *subtract*.

107 $\langle$ Auxiliary functions 107 $\rangle \equiv$

```
void subtract(int size, const double *x, double *y)
{
    for (int i = 0; i < size; i++)
        y[i] -= x[i];
}
```

See also chunks 111, 125, 160, 172, 184, 185, 188, 189, 205, 208, 210, 226, 230, 281, 285, 289, 291, and 367

This code is used in chunk 366.

to

108 $\langle$ compute $\mathbf{x}^{m}_{J<0} = \mathbf{x}^{m-1}_{J<0} - \boldsymbol{\delta}^m$ 108 $\rangle \equiv$

```
subtract(size, residual, x);
```

This code is used in chunk 101.

Similarly, at each iteration m, we compute $\mathbf{f}_{\text{HO}}(\mathbf{x}^m_{J_{<0}})$ and store it at *residual*.

109 $\langle$ compute $\mathbf{f}_{\text{HO}}(\mathbf{x}^m_{J_{<0}})$ 109 $\rangle \equiv$

```
$fcn\_flag = Fcn(size, x, residual, user\_data);$
if $(fcn\_flag \equiv$ SYS_JAC_SINGULAR $\vee fcn\_flag \equiv$ STAGE0_FAIL)
    return $fcn\_flag;$
```

This code is used in chunk 101.

To compute the rate of convergence and the iteration error, we need to evaluate $\|\boldsymbol{\delta}^m\|$. This is done by calling the following function which computes $\|v\|$ in (7.3) for a given vector v of size $size$. Here the (j,k)th entry of the irregular matrix $\mathbf{W}$ is

$$w_{jk} = \frac{1}{\text{atol} + \text{rtol} \cdot |x_j^{(k)}|} \quad \text{for } (j,k) \in J_{<0}. \tag{7.6}$$

111 $\langle$ Auxiliary functions 107 $\rangle +\equiv$

```
double CompWRMSnorm(const double $*v$, const IrregularMatrix$\langle$double$\rangle$ $\&w$)
{
    int $i = 0;$
    double $s = 0;$
    for (size_t $j = 0;\ j < w.num\_rows(\,);\ j{+}{+}$)
        for (size_t $k = 0;\ k < w.num\_cols(j);\ k{+}{+}$)
        {
            double $z = v[i{+}{+}] * w(j,k);$
            $s \mathrel{+}= z * z;$
        }
```

$s \mathrel{/}= w.\mathit{num\_entries}(\,);$

$\textbf{return std}::\mathit{sqrt}(s);$

$\}$

112 $\langle$ evaluate $\|\boldsymbol{\delta}^m\|$ 112 $\rangle \equiv$

$\textbf{double}\ \mathit{norm\_delta} = \mathit{CompWRMSnorm}(\mathit{residual}, \mathit{weight});$

This code is used in chunk 101.

When $\|\boldsymbol{\delta}^m\|$ is smaller than some multiple of the machine precision, we are not able to use (7.5) to estimate ρ. In such a case, we accept the computed $\mathbf{x}^m_{J<0}$ that it is as accurate as possible for the machine being used. This situation happens either if the prediction is extremely good or when the tolerance is very small [59]. On the first iteration, if $\|\boldsymbol{\delta}^1\|$ is very small, the iteration is terminated and we accept $\mathbf{x}^1_{J<0}$ as a solution. When $m > 1$ we can get an estimate of the rate of convergence by (7.5).

113 $\langle$ declare variables for nonlinear solver 113 $\rangle \equiv$

$\textbf{double}\ \mathit{iteration\_error} = 1.0;$

See also chunk 114.

This code is used in chunk 101.

114 $\langle$ declare variables for nonlinear solver 113 $\rangle \mathrel{+}\equiv$

$\textbf{double}\ \mathit{rate\_convergence},\ \mathit{norm\_delta\_prev} = 1;$

115 $\langle$ compute $\mathit{iteration\_error} = \dfrac{\rho}{1-\rho}\|\boldsymbol{\delta}^m\|$ 115 $\rangle \equiv$

$\textbf{if}\ (\ \mathit{norm\_delta} < 10 * \textbf{std}::\mathit{numeric\_limits} < \textbf{double} > ::\mathit{epsilon}(\,)\ )$

```
$\textit{iteration\_error} = \textit{norm\_delta};$
else
  if $(m \equiv 1)$
    $\textit{iteration\_error} = 10 * \textit{norm\_delta};$
  else
  {
    $\textit{rate\_convergence} = \textit{norm\_delta} / \textit{norm\_delta\_prev};$
    if $(\textit{rate\_convergence} \geq 0.9)$
      return HO_SUCCESS;
    $\textit{iteration\_error} = (\textit{rate\_convergence} / (1 - \textit{rate\_convergence})) * \textit{norm\_delta};$
  }
```

This code is used in chunk 101.

Finally we

116 ⟨ save $\|\boldsymbol{\delta}^m\|$ for computing ρ in next iteration 116 ⟩ ≡

```
$\textit{norm\_delta\_prev} = \textit{norm\_delta};$
```

This code is used in chunk 101.

7.2.1 Evaluating residual

Denote

$$\boldsymbol{\psi} = \sum_{r=0}^{p} \mathbf{F}^{[r]}\Big(t^*, \mathbf{x}^*_{J_{<0}}\Big) h^r \otimes \mathbf{a}_r \quad \text{and} \tag{7.7}$$

$$\boldsymbol{\varphi}(\mathbf{x}_{J_{<0}}) = \sum_{r=0}^{q} \mathbf{F}^{[r]}\Big(t^* + h, \mathbf{x}_{J_{<0}}\Big) h^r \otimes \mathbf{b}_r, \tag{7.8}$$

where $\mathbf{F}^{[r]}$, $\mathbf{a}_r$, and $\mathbf{b}_r$ are defined in (4.16), (4.14), and (4.15), respectively. Given $\mathbf{x}_{J_{<0}}$, we need to evaluate

$$\mathbf{F}(\mathbf{x}_{J_{<0}}) = \boldsymbol{\varphi}(\mathbf{x}_{J_{<0}}) - \boldsymbol{\psi}. \tag{7.9}$$

We first compute $\boldsymbol{\psi}$ which does not depend on $\mathbf{x}_{J_{<0}}$. Then we evaluate $\boldsymbol{\varphi}(\mathbf{x}_{J_{<0}})$. Finally, we implement the function *CompF* to compute (7.9).

7.2.1.1 Computing ψ

By (7.7), we first set $\boldsymbol{\psi} = 0$, and then

for $r = 0, 1, \ldots, p$

- compute $\mathbf{a}_r$,
- form $\mathbf{F}^{[r]}\left(t^*, \mathbf{x}^*_{J_{<0}}\right)h^r$, and
- accumulate $\boldsymbol{\psi} \mathrel{+}= \mathbf{F}^{[r]}\left(t^*, \mathbf{x}^*_{J_{<0}}\right)h^r \otimes \mathbf{a}_r$.

We store p in

119 $\langle$ HO Data Members 28 $\rangle$ $+\equiv$

int *p_*;

and implement the function *CompPsi* to compute $\boldsymbol{\psi}$ and store it in

120 $\langle$ HO Data Members 28 $\rangle$ $+\equiv$

IrregularMatrix$\langle$**double**$\rangle$ *psi_*;

In this function, *tcs* contains TCs $\mathbf{x}^*_{J_{<p}}$ at t^*.

121 ⟨ Definitions of HO Private Functions 37 ⟩ +≡

```
void HO :: CompPsi(const std :: vector⟨vector⟨double⟩⟩ &tcs)
{
  psi_.set_to_zero( );      /* $\psi = 0$ */
  for (int $r = 0$; $r \le p\_$; $r$++)
  {
    comp_a($r$);      /* computes $\mathbf{a}_r$ */
    FormFr($r$, tcs);      /* forms $\mathbf{F}^{[r]}\left(t^*, \mathbf{x}^*_{J<0}\right)h^r$ */
    multiply_add(f_, coef_, psi_);      /* computes $\psi$ += $\mathbf{F}^{[r]}\left(t^*, \mathbf{x}^*_{J<0}\right)h^r \otimes \mathbf{a}_r$ */
  }
}
```

Computing $\mathbf{a}_r$

By (3.13) and (4.14), the kth element of the vector $\mathbf{a}_r$ is

$$c_r^{pq}(k+r)!.$$

The coefficients c_r^{pq} are precomputed and stored in

123 ⟨ HO Data Members 28 ⟩ +≡

```
std :: vector⟨double⟩ cpq_;
```

By (3.10), $c_0^{pq} = 1$, and

$$c_r^{pq} = \frac{p-r+1}{r(p+q-r+1)}\, c_{r-1}^{pq}, \qquad \text{for } r = 1, \dots, p.$$

The function *CompCpq* computes these coefficients.

124 ⟨ Definitions of HO Private Functions 37 ⟩ +≡

```
void HO :: CompCpq( )
{
  cpq_.resize(p_ + 1);
  cpq_[0] = 1.0;
  for (int r = 1; r ≤ p_; r++)
    cpq_[r] = (cpq_[r − 1] ∗ (p_ − r + 1))/double(r ∗ (p_ + q_ − r + 1));
}
```

For a scalar z and a vector $\mathbf{u}$, the following function computes $\mathbf{v} = z * \mathbf{u}$.

125 ⟨ Auxiliary functions 107 ⟩ +≡

```
void scalar_times_vector(double z, int size, double ∗u, double ∗v)
{
  for (int l = 0; l < size; l++)
    v[l] = z ∗ u[l];
}
```

The function *comp_a* is implemented to compute the elements of $\mathbf{a}_r$ and store them in

126 ⟨ HO Data Members 28 ⟩ +≡

```
std :: vector⟨double⟩ coef_;
```

127 ⟨ Definitions of HO Private Functions 37 ⟩ +≡

```
void HO :: comp_a(int r)
```

```
{
  scalar_times_vector(cpq_[r], coef_.size( ), factorial_.data( ) + r, coef_.data( ));
}
```

Forming $\mathbf{F}^{[r]}\left(t^*, \mathbf{x}^*_{J_{<0}}\right)h^r$

We precompute powers of h on each step and store them in

128 ⟨ HO Data Members 28 ⟩ +≡

```
std::vector⟨double⟩ h_pow_;
```

By (4.10) and (4.16), the (j, k)th entry of the irregular matrix $\mathbf{F}^{[r]}\left(t^*, \mathbf{x}^*_{J_{<0}}\right)h^r$ is $(x^*_j)_{k+r}h^r$, and $tcs[j][k+r]$ contains the TC $(x^*_j)_{k+r}$. Hence,

$$(x^*_j)_{k+r}h^r = tcs[j][k+r] \cdot h^r.$$

The function *FormFr* is implemented to compute these entries and store them in

129 ⟨ HO Data Members 28 ⟩ +≡

```
IrregularMatrix⟨double⟩ f_;
```

130 ⟨ Definitions of HO Private Functions 37 ⟩ +≡

```
void HO::FormFr(int r, const std::vector⟨vector⟨double⟩⟩ &tcs)
{
  for (size_t j = 0; j < n_; j++)
    for (size_t k = 0; k < d_[j]; k++)
      f_(j, k) = tcs[j][k + r] * h_pow_[r];
}
```

7.2.1.2 Evaluating $\varphi(\mathbf{x}_{J_{<0}})$

Given $\mathbf{x}_{J_{<0}}$, we need to evaluate $\boldsymbol{\varphi}(\mathbf{x}_{J_{<0}})$. After computing higher-order TCs, we first set $\boldsymbol{\varphi}(\mathbf{x}_{J_{<0}}) = 0$. Then, by (7.8), for each $r = 0, 1, \ldots, q$, we

- compute $\mathbf{b}_r$,
- form $\mathbf{F}^{[r]}\Big(t^* + h, \mathbf{x}_{J_{<0}}\Big)h^r$, and
- compute $\boldsymbol{\varphi}(\mathbf{x}_{J_{<0}}) \mathrel{+}= \mathbf{F}^{[r]}\Big(t^* + h, \mathbf{x}_{J_{<0}}\Big)h^r \otimes \mathbf{b}_r$.

We store q in

132 $\langle$ HO Data Members 28 $\rangle \mathrel{+}\equiv$

int *q_*;

and implement the function *CompPhi* to evaluate $\boldsymbol{\varphi}(\mathbf{x}_{J_{<0}})$ and store it in

133 $\langle$ HO Data Members 28 $\rangle \mathrel{+}\equiv$

IrregularMatrix⟨**double**⟩ *phi_*;

This function returns one of the following values of type **HoFlag**

- `SYS_JAC_SINGULAR`, if system Jacobian is singular,
- `STAGE0_FAIL`, if solving the nonlinear system for computing TCs at stage zero in non-quasilinear DAEs fails, or
- `HO_SUCCESS`, if $\boldsymbol{\varphi}(\mathbf{x}_{J_{<0}})$ is computed successfully.

135 ⟨Definitions of HO Private Functions 37⟩ +≡

```
**HoFlag HO**::*CompPhi*(**const double** $*x$)
{
  ⟨set $\mathbf{x}_{J_{<0}}$ 136⟩;
  ⟨compute $\mathbf{x}_{J_s}$, for $s = 0, 1, \ldots, q-1$ 141⟩;
  phi_.set_to_zero();      /* $\boldsymbol{\varphi}(\mathbf{x}_{J_{<0}}) = 0$ */
  **for** (**int** $r = 0$; $r \le$ q_; $r$++)
  {
    comp_b($r$);      /* computes $\mathbf{b}_r$ */
    FormFr($r$);      /* forms $\mathbf{F}^{[r]}\left(t^* + h, \mathbf{x}_{J_{<0}}\right)h^r$ */
    multiply_add(f_, coef_, phi_);      /* $\boldsymbol{\varphi}(\mathbf{x}_{J_{<0}})$ += $\mathbf{F}^{[r]}\left(t^* + h, \mathbf{x}_{J_{<0}}\right)h^r \otimes \mathbf{b}_r$ */
  }
  **return** HO_SUCCESS;
}
```

Setting $\mathbf{x}_{J_{<0}}$

We first copy the entries of x to *indep_tcs_*.

136 ⟨set $\mathbf{x}_{J_{<0}}$ 136⟩ ≡

```
indep_tcs_.set($x$);
```

See also chunk 138.

This code is used in chunk 135.

Then, the function *SetIndepTCs* sets entries of *indep_tcs_* in the object of the **TaylorSeries** class for computing TCs by calling *set_var_coeff*.

137 ⟨Definitions of HO Private Functions 37⟩ +≡

```
void HO::SetIndepTCs()
{
  for (int $j = 0$; $j <$ n_; $j$++)
    for (int $k = 0$; $k <$ d_[$j$]; $k$++)
      ts_→set_var_coeff($j, k$, indep_tcs_($j, k$));
}
```

138 ⟨set $\mathbf{x}_{J_{<0}}$ 136⟩ +≡

```
SetIndepTCs();
```

Computing $\mathbf{x}_{J_{<q}}$

Given $\mathbf{x}_{J_{<0}}$, we can compute $\mathbf{x}_{J_s}$, for $s = 0, 1, \ldots, q-1$ (see Chapter 5). Recall that (4.4) for computing $\mathbf{x}_{J_0}$ is

- linear in quasilinear DAEs and the matrix $\mathbf{A}_0$ is in terms of $\mathbf{x}_{J_{<0}}$. In this case, after computing $\mathbf{A}_0$ and finding it's LU decomposition, we call the function *CompTCsLinear*(0).

- nonlinear in non-quasilinear DAEs and the matrix $\mathbf{A}_0$ is in terms of $\mathbf{x}_{J_{\leq 0}}$. Hence, we first call the function *CompTCsNonlinear*( ) to compute $\mathbf{x}_{J_0}$ and store it in *tcs_stage0_*. Then, we compute $\mathbf{A}_0$ and find it's LU decomposition required later for computing $\mathbf{x}_{J_s}$, $s = 1, 2, \ldots, q-1$.

139 $\langle$ HO Data Members 28 $\rangle$ $+\equiv$

```
**double** $*tcs\_stage0\_$;
```

To check if the given DAE is quasilinear, we call the function *isLinear* from the **SAdata** class.

141 $\langle$ compute $\mathbf{x}_{J_s}$, for $s = 0, 1, \ldots, q-1$ 141 $\rangle$ $\equiv$

```
if (sadata_→isLinear( ))
{
    CompA0(sys_jac_);
    ⟨ find LU factorization of $\mathbf{A}_0$ 142 ⟩;
    CompTCsLinear(0);
}
else
{
    jac_→resetAll( );
    HoFlag info = CompTCsNonlinear(tcs_stage0_);
    if (info ≡ STAGE0_FAIL)
        return info;
    SetStageZeroTCs(tcs_stage0_);
    SetStageZeroTCsJac(tcs_stage0_);
    CompA0(sys_jac_);
    ⟨ find LU factorization of $\mathbf{A}_0$ 142 ⟩;
}
```

See also chunk 143.

This code is used in chunk 135.

We use the LU decomposition of $\mathbf{A}_0$ for solving systems of linear equations (5.9) and (6.5).

142 $\langle$ find LU factorization of $\mathbf{A}_0$ 142 $\rangle \equiv$

```
int sys_info;
daets::LU(n_, sys_jac_, ipiv_, &sys_info);
if (sys_info ≠ 0)
    return SYS_JAC_SINGULAR;
```

This code is used in chunk 141.

The system (4.4) for computing $\mathbf{x}_{J_s}$, $s = 1, 2, \dots, q-1$ is linear. In addition, we have the LU decomposition of $\mathbf{A}_0$. Hence, *CompTCsLinear*(s) computes $\mathbf{x}_{J_s}$.

143 $\langle$ compute $\mathbf{x}_{J_s}$, for $s = 0, 1, \dots, q-1$ 141 $\rangle +\equiv$

```
for (int s = 1; s < q_; s++)
    CompTCsLinear(s);
```

Computing b_r

By (3.14) and (4.15), the kth element of the vector $\mathbf{b}_r$ is

$$c_r^{qp}(r+k)!.$$

The coefficients c_r^{qp} are precomputed and stored in

144 ⟨ HO Data Members 28 ⟩ +≡

```
std::vector⟨double⟩ cqp_;
```

By (3.11), $c_0^{qp} = 1$, and

$$c_r^{qp} = -c_{r-1}^{qp} \, \frac{q-r+1}{r(p+q-r+1)} \,, \qquad \text{for } r = 1, \ldots, q.$$

The function *CompCqp* computes these coefficients.

145 ⟨ Definitions of HO Private Functions 37 ⟩ +≡

```
void HO::CompCqp( )
{
  cqp_.resize(q_ + 1);
  cqp_[0] = 1.0;
  for (int r = 1; r ≤ q_; r++)
    cqp_[r] = (−cqp_[r − 1] ∗ (q_ − r + 1))/double(r ∗ (p_ + q_ − r + 1));
}
```

The function *comp_b* is implemented to compute the elements of b_r and store them in *coef_*.

146 ⟨ Definitions of HO Private Functions 37 ⟩ +≡

```
void HO::comp_b(int r)
{
  scalar_times_vector(cqp_[r], coef_.size( ), factorial_.data( ) + r, coef_.data( ));
}
```

Forming $\mathbf{F}^{[r]}\left(t^* + h, \mathbf{x}_{J_{<0}}\right)h^r$

By (4.11) and (4.16), the (j, k)th entry of the irregular matrix $\mathbf{F}^{[r]}\left(t^* + h, \mathbf{x}_{J_{<0}}\right)h^r$ is $(x_j)_{r+k}h^r$. We implement the function *FormFr* to compute these entries and store them in *f_*.

147 ⟨ Definitions of HO Private Functions 37 ⟩ +≡

```
void HO::FormFr(int r)
{
  for (size_t j = 0; j < n_; j++)
    for (size_t k = 0; k < d_[j]; k++)
      f_(j, k) = ts_→get_var_coeff(j, r + k) * h_pow_[r];
}
```

7.2.1.3 The function *CompF*

The function *CompF* computes $\mathbf{F}(\mathbf{x}_{J_{<0}})$ in (7.9) and stores it in *f_*. In this function, the input array x contains $\mathbf{x}_{J_{<0}}$, the output f contains $\mathbf{f}_{\text{HO}}(\mathbf{x}_{J_{<0}})$ by concatenating the rows of $\mathbf{F}(\mathbf{x}_{J_{<0}})$.

148 ⟨ Definitions of HO Public Functions 61 ⟩ +≡

```
HoFlag HO::CompF(const double *x, double *f)
{
  HoFlag info = CompPhi(x);      /* evaluates $\varphi(\mathbf{x}_{J_{<0}})$ */
  if (info ≠ HO_SUCCESS)
    return info;
    f_ = phi_ − psi_;    /* $\mathbf{F}(\mathbf{x}_{J_{<0}}) = \boldsymbol{\varphi}(\mathbf{x}_{J_{<0}}) - \boldsymbol{\psi}$ */
    f_.to_vector(f);    /* copies entries of $\mathbf{F}(\mathbf{x}_{J_{<0}})$ to $f$ */
    return info;
  }
```

7.2.2 Computing Jacobian

For an irregular matrix $\mathbf{M}$, denote by $\boldsymbol{\nabla}\mathbf{M}$ the irregular matrix whose (j,k)th entry is

$$\Big(\boldsymbol{\nabla}\mathbf{M}\Big)_{jk} = \nabla\Big(\mathbf{M}\Big)_{jk}.$$

Let $\nabla = \partial/\partial\mathbf{x}_{J_{<0}}$. Recall that $\mathbf{F}^{[r]}\Big(t^* + h, \mathbf{x}_{J_{<0}}\Big)$ given by (4.16) is an irregular matrix whose (j,k)th entry is $(x_j)_{k+r} = T_{j,k+r}(t^* + h, \mathbf{x}_{J_{<0}})$. Hence, $\boldsymbol{\nabla}\mathbf{F}^{[r]}$ is an irregular matrix with

$$\Big(\boldsymbol{\nabla}\mathbf{F}^{[r]}\Big)_{jk} = \nabla(x_j)_{k+r}. \tag{7.10}$$

By (4.18), we can write

$$\boldsymbol{\nabla}\mathbf{F} = \sum_{r=0}^{q} \boldsymbol{\nabla}\mathbf{F}^{[r]} h^r \otimes \mathbf{b}_r. \tag{7.11}$$

To compute (7.11), we first set $\boldsymbol{\nabla}\mathbf{F} = 0$, and then

for $r = 0, 1, \dots, q$,

- compute $\mathbf{b}_r$,
- form $\boldsymbol{\nabla}\mathbf{F}^{[r]} h^r$, and
- accumulate $\boldsymbol{\nabla}\mathbf{F} \mathrel{+}= \boldsymbol{\nabla}\mathbf{F}^{[r]} h^r \otimes \mathbf{b}_r$.

This is implemented by the function *CompHoJac* which stores $\mathbf{\nabla F}$ in

150 ⟨HO Data Members 28⟩ +≡

```
IrregularMatrix⟨std::vector⟨double⟩⟩ f_prime_;
```

In this function, the output *ho_jac* contains the Jacobian $\mathbf{J}_{\mathrm{HO}}$ column-wise, as we later pass it to the linear solver.

151 ⟨Definitions of HO Private Functions 37⟩ +≡

```
void HO::CompHoJac(double *ho_jac)
{
    CompGradients(q_);
    f_prime_.set_to_zero( );    /* sets $\mathbf{\nabla F} = 0$ */
    for (int $r = 0$; $r \le q\_$; $r$++)
    {
        comp_b($r$);    /* computes $\mathbf{b}_r$ */
        FormFrPrime($r$);    /* forms $\mathbf{\nabla F}^{[r]}h^r$ */
        multiply_add(fr_prime_, coef_, f_prime_);    /* $\mathbf{\nabla F}$ += $\mathbf{\nabla F}^{[r]}h^r \otimes \mathbf{b}_r$ */
    }
    f_prime_.to_vector(ho_jac);
    ⟨compute $\|\mathbf{J}_{\mathrm{HO}}\|_\infty$, if requested 159⟩;
}
```

7.2.2.1 Forming $\nabla\mathbf{F}^{[r]}h^r$

From (7.10), the (j,k)th entry of $\nabla\mathbf{F}^{[r]}h^r$ is $\nabla(x_j)_{k+r}h^r$. Calling the function *CompGradients*, gradients of all TCs are computed (see Chapter 6). To access their components, we call the function *get_var_grad_component* from the **Gradients** class. Denote by y_l the lth component of $\mathbf{x}_{J_{<0}}$. Then, *get_var_grad_component*$(j, k+r, l)$ returns $\dfrac{\partial(x_j)_{k+r}}{\partial y_l}$. The function *form_grad* stores $\dfrac{\partial(x_j)_{k+r}}{\partial y_l}h^r$ for $l = 0,\dots,N-1$, in *tc_grad*.

153 ⟨HO Data Members 28⟩ +≡

std :: **vector**⟨**double**⟩ *tc_grad_*;

154 ⟨Definitions of HO Private Functions 37⟩ +≡

```
void HO::form_grad(int j, int k, int r, std::vector⟨double⟩ &tc_grad)
{
  for (size_t l = 0; l < num_indep_tcs_; l++)
    tc_grad[l] = grads_→get_var_grad_component(j, k + r, l) ∗ h_pow_[r];
}
```

Now, we can compute all entries of and store them in

155 ⟨HO Data Members 28⟩ +≡

IrregularMatrix⟨**std** :: **vector**⟨**double**⟩⟩ *fr_prime_*;

This is implemented by the function *FormFrPrime*.

156 ⟨Definitions of HO Private Functions 37⟩ +≡

void HO :: *FormFrPrime*(**int** r)

```
{
  for (size_t $j = 0$; $j < n\_$; $j$++)
    for (size_t $k = 0$; $k < d\_[j]$; $k$++)
      $form\_grad(j, k, r, fr\_prime\_(j, k))$;
}
```

For some experiments, we may need the condition number of the Jacobian matrix $\mathbf{J}_{\text{HO}}$.

157 ⟨HO Data Members 28⟩ +≡

```
bool $need\_cond\_jac\_ = false$;
```

Here, we compute $\|\mathbf{J}_{\text{HO}}\|_\infty$ by routines in the LAPACK software package and store it in *norm_jac_*, as we later use it to compute the condition number.

158 ⟨HO Data Members 28⟩ +≡

```
double $norm\_jac\_$;
```

159 ⟨compute $\|\mathbf{J}_{\text{HO}}\|_\infty$, if requested 159⟩ ≡

```
if ($need\_cond\_jac\_$)
  $norm\_jac\_ = MNorm(num\_indep\_tcs\_, ho\_jac)$;
```

This code is used in chunk 151.

160 ⟨Auxiliary functions 107⟩ +≡

```
double $MNorm$(int $n$, double $*mat$)
{
  int $lda = n$;
```

```
    char norm = 'I';

    double *work = new double[n];

    double mat_norm = dlange_(&norm, &n, &n, mat, &lda, work);

    delete[ ] work;

    return mat_norm;

  }
```

7.3 Implementation of HO method for one step

To implement one step of the proposed HO method in §4.2, we

- set parameters to form the HO system,
- compute powers of the stepsize h needed in computing $\boldsymbol{\psi}$ in (7.7) and $\boldsymbol{\varphi}$ in (7.8),
- compute $\boldsymbol{\psi}$ which does not depend on $\mathbf{x}_{J_{<0}}$, and
- form and solve the HO system (4.20).

The above tasks are implemented in the function *CompHoSolution*. In this function, x contains an initial guess for the solution. After solving the system successfully, x will be updated with the computed solution. *weight* is used to compute the required WRMS norms. *tcs_prev* contains TCs at the previous t.

161 ⟨ Definitions of HO Private Functions 37 ⟩ +≡

```
HoFlag HO::CompHoSolution(double t, double h, double tol, const
        IrregularMatrix⟨double⟩ &weight, const std::vector⟨vector⟨double⟩⟩
        &tcs_prev, daets::DAEpoint &x)
{
  ⟨ set parameters to form the HO system 163 ⟩;
  ⟨ compute powers of h 173 ⟩;
  CompPsi(tcs_prev);      /* computes ψ */
  ⟨ solve the HO system 179 ⟩;      /* returns the HoFlag info and the solution */
  if (info ≡ HO_CONVERGENT)
    ⟨ update x 180 ⟩;
  ⟨ reset parameters 181 ⟩;
  return info;
}
```

7.3.1 Setting parameters

Calling the function *set_order*, we set the order for the **TaylorSeries** object.

163 ⟨ set parameters to form the HO system 163 ⟩ ≡

```
ts_→set_order(q_);
ts_→set_h(q_);
```

See also chunks 166, 169, and 171

This code is used in chunk 161.

We set TCs and gradients of t and store them in

164 ⟨ Gradients Data Members 80 ⟩ +≡

fadbad :: **T**⟨**fadbad** :: **F**⟨**double**⟩⟩ *t_tfdouble_*;

All these coefficients and gradients are zero unless the first two TCs which are t and dt. The function *set_time_coeffs* is implemented to set them.

165 ⟨ Gradients Public Functions 82 ⟩ +≡

void *set_time_coeffs*(**double** t, **double** *dt*)

{

  *t_tfdouble_*[0] = t;

  *t_tfdouble_*[1] = *dt*;

}

166 ⟨ set parameters to form the HO system 163 ⟩ +≡

grads_→set_time_coeffs(*t_*, 1);

ts_→set_time_coeffs(*t_*, 1);

In the **HO** class, we store the current stepsize in

167 ⟨ HO Data Members 28 ⟩ +≡

double *h_*;

The function *set_h* sets the stepsize in this class.

168 ⟨ Definitions of HO Private Functions 37 ⟩ +≡

void HO :: *set_h*(**double** h) { *h_* = h; }

169 ⟨ set parameters to form the HO system 163 ⟩ +≡

```
set_h(h);
```

The function *set_t* sets the current t in the **HO** class.

170 ⟨ Definitions of HO Private Functions 37 ⟩ +≡

```
void HO::set_t(double t) { t_ = t; }
```

171 ⟨ set parameters to form the HO system 163 ⟩ +≡

```
set_t(t);
```

7.3.2 Powers of the stepsize

Given h, the function *CompPowersH* computes *h_pow* so that *h_pow*$[i]$ contains h^i.

172 ⟨ Auxiliary functions 107 ⟩ +≡

```
void CompPowersH(int size, double h, std::vector<double> &h_pow)
{
  if (h_pow.size() ≠ size)
    h_pow.resize(size);
  h_pow[0] = 1;
  for (size_t i = 1; i < size; i++)
    h_pow[i] = h * h_pow[i − 1];
}
```

Calling this function, we

173 ⟨ compute powers of h 173 ⟩ ≡

```
int size_h_pow = sadata_→get_max_d() + p_ + q_ + 1;
```

CompPowersH(*size_h_pow*, h, *h_pow_*);

This code is used in chunk 161.

7.3.3 Solving the system

To solve the HO system (4.20) using the function *NSolve* implemented in §7.2, we require an initial guess for the solution and two functions, which evaluate $\mathbf{f}_{\mathrm{HO}}(\mathbf{u})$ and $\mathbf{J}_{\mathrm{HO}}(\mathbf{u})$ for a given vector $\mathbf{u}$. In addition, we need

174 ⟨ HO Data Members 28 ⟩ +≡

double $*$*indep_tcs_flat_*, /$*$ to store the input vector $\mathbf{u}$ $*$/
$*$*residual_flat_*, /$*$ to store the vector $\mathbf{f}_{\mathrm{HO}}$ $*$/
$*$*ho_jacobian_*; /$*$ to store $\mathbf{J}_{\mathrm{HO}}$ and $*$/
int $*$*ho_ipiv_*; /$*$ for the pivot vector that defines the permutation matrix of the LU factorization of $\mathbf{J}_{\mathrm{HO}}$. $*$/

7.3.3.1 The initial guess

$x(j,k)$ contains an initial guess for $x_j^{(k)}$. Hence, $x(j,k)/k!$ gives an initial guess for the TC $x_j^{(k)}/k!$, which is stored as the (j,k)th entry of the irregular matrix *indep_tcs_*.

175 ⟨ get $\mathbf{x}^0_{J_{\le\alpha}}$ from x 175 ⟩ ≡

indep_tcs_.set(x); /$*$ the irregular matrix *indep_tcs_* contains derivatives $*$/
*indep_tcs_* /= *factorial_*; /$*$ *indep_tcs_* contains TCs $*$/
indep_tcs_.to_vector(*indep_tcs_flat_*); /$*$ the array *indep_tcs_flat* contains TCs $*$/

See also chunk 176.

This code is used in chunk 179.

For non-quasilinear problems, we also require an initial guess for TCs at stage zero.

176 ⟨ get $\mathbf{x}^0_{J_{\leq\alpha}}$ from x 175 ⟩ +≡

```
if (¬sadata_→isLinear( ))
{
   for (int j = 0; j < n_; j++)
   {
      int dj = d_[j];
      tcs_stage0_[j] = x(j, dj)/factorial_[dj];
   }
   tol_ = tol;      /* see §5.2.1.3 */
}
```

7.3.3.2 Required functions to call NSolve

The function *Fcn* evaluates $\mathbf{f}_{\text{HO}}(\mathbf{u})$ by calling the function *CompF* from the **HO** class. In this function, n is the size of the input vector x and the evaluated $\mathbf{f}_{\text{HO}}(\mathbf{u})$ is stored at f.

177 ⟨ Nonlinear Solver Functions 59 ⟩ +≡

```
HoFlag Fcn(int n, double *x, double *f, void *user_data)
{
   HO *ho = (HO *) user_data;
   HoFlag info = ho→CompF(x, f);
   return info;
}
```

We implement the function *Jac* to compute $\mathbf{J}_{\text{HO}}$ by calling the function *CompHoJac* from the **HO** class. In this function, n is the size of the input vector x and the computed $\mathbf{J}_{\text{HO}}(\mathbf{u})$ is stored at *jac*.

178 ⟨Nonlinear Solver Functions 59⟩ +≡

```
void Jac(int n, double *x, double *jac, void *user_data)
{
  HO *ho = (HO *) user_data;
  ho→CompHoJac(jac);
}
```

We can now call the function *NSolve*.

179 ⟨solve the HO system 179⟩ ≡

```
⟨get $\mathbf{x}^0_{J_{\le\alpha}}$ from $x$ 175⟩;     /* $\alpha$ given by (8.9) */
HoFlag info = NSolve(num_indep_tcs_, weight, indep_tcs_flat_, Fcn, Jac,
    residual_flat_, ho_jacobian_, ho_ipiv_, (void *) this);
```

This code is used in chunk 161.

7.3.3.3 Updating x

After solving the system, we extract the derivatives from TCs to update x.

180 ⟨update x 180⟩ ≡

```
for (int j = 0; j < n_; j++)
  for (int k = 0; k ≤ d_[j]; k++)
```

$x(j,k) =$ *ts_→get_var_coeff* $(j,k) *$ *factorial_*$[k]$;

This code is used in chunk 161.

Finally, we

181 ⟨ reset parameters 181 ⟩ ≡

ts_→resetAll( );

This code is used in chunk 161.

Chapter 8

Integration strategies

An important part in the design of a numerical algorithm is to estimate the error of a numerical solution. Here, we need to estimate $\mathbf{E}$ in (4.19), so that $e_{pq}h^{p+q+1}\mathbf{E}$ can be calculated as an approximation to the discretization error of (4.20). Since we wish to implement our HO method in a variable-order mode, we also need to estimate the error for each alternative order that is under consideration.

In the present chapter, we first create a Nordsieck vector for an Hermite interpolating polynomial in §8.1. Then, we show how this vector is used to predict a solution, §8.2, and to estimate the discretization error, §8.3. Finally, the stepsize and order selection strategies are derived in §8.4.

8.1 Hermite-Nordsieck vector

For a scalar function $y \in C^m[a,b]$, the vector

$$\left[y(t), y'(t), \frac{y''(t)}{2!}, \dots, \frac{y^{(m)}(t)}{m!}\right],$$

is referred to as a *Nordsieck vector* for y at t [10]. In this section, we create a Nordsieck vector for an Hermite interpolating polynomial. We employ this vector to predict a solution, §8.2, and to estimate the discretization error, §8.3.

Given

$$y(a), y'(a), \dots, y^{(p)}(a) \quad \text{and} \quad y(b), y'(b), \dots, y^{(q)}(b), \tag{8.1}$$

there is a unique interpolating polynomial $P(t) \in \Pi_{p+q+1}$ (see e.g., [64, p. 52], such that

$$\begin{aligned} P^{(j)}(a) &= y^{(j)}(a), \quad \text{for} \quad j = 0, 1, \dots, p \quad \text{and} \\ P^{(i)}(b) &= y^{(i)}(b), \quad \text{for} \quad i = 0, 1, \dots, q. \end{aligned} \tag{8.2}$$

Using the *generalized divided differences* (see e.g. [64, p. 56])

$$\begin{aligned} y\Big[\underbrace{b, \dots, b}_{i}\Big] &= \frac{y^{(i-1)}(b)}{(i-1)!}, \quad i = 0, 1, \dots, q+1, \\ y\Big[\underbrace{a, \dots, a}_{j}\Big] &= \frac{y^{(j-1)}(a)}{(j-1)!}, \quad j = 0, 1, \dots, p+1, \\ y\Big[\underbrace{b, \dots, b}_{i}, \underbrace{a, \dots, a}_{j}\Big] &= \frac{y\Big[\underbrace{b, \dots, b}_{i-1}, \underbrace{a, \dots, a}_{j}\Big] - y\Big[\underbrace{b, \dots, b}_{i}, \underbrace{a, \dots, a}_{j-1}\Big]}{a-b}, \end{aligned}$$

$i = 1, \dots, q+1$ and $j = 1, \dots, p+1$, and denoting

$$\gamma_j = y\Big[\underbrace{b, \dots, b}_{q+1}, \underbrace{a, \dots, a}_{j}\Big], \tag{8.3}$$

then

$$P(t) = \sum_{j=0}^{q} \frac{y^{(j)}(b)}{j!}(t-b)^j + (t-b)^{q+1}\sum_{j=1}^{p+1}\gamma_j (t-a)^{j-1}, \tag{8.4}$$

is the Hermite interpolating polynomial satisfying (8.2).

Consider the following Nordsieck vector for P at b

$$\left[P(b), P'(b), \frac{P''(b)}{2!}, \dots, \frac{P^{(p+q+1)}(b)}{(p+q+1)!}\right]. \tag{8.5}$$

Differentiating (8.4) and setting $t=b$, we obtain

$$P^{(q+j)}(b) = (q+j)!\,\gamma_j, \quad \text{for} \quad j = 1,\dots,p+1. \tag{8.6}$$

By (8.3) and (8.6), we write

$$\frac{P^{(q+j)}(b)}{(q+j)!} = y\big[\underbrace{b,\dots,b}_{q+1},\underbrace{a,\dots,a}_{j}\big]. \tag{8.7}$$

Using (8.2) and (8.7), the Nordsieck vector (8.5) is

$$\left[y(b), y'(b), \dots, \frac{y^{(q)}(b)}{q!}, y\big[\underbrace{b,\dots,b}_{q+1},a\big], y\big[\underbrace{b,\dots,b}_{q+1},a,a\big], y\big[\underbrace{b,\dots,b}_{q+1},\underbrace{a,\dots,a}_{p+1}\big]\right]. \tag{8.8}$$

We refer to (8.8) as the (p,q) *Hermite-Nordsieck* vector for y at b.

Let

$$\alpha = \begin{cases} -1 & \text{if the DAE is quasilinear,} \\ 0 & \text{otherwise.} \end{cases} \tag{8.9}$$

Assume that values for $x_j(a), x_j'(a), \dots, x_j^{(p+d_j+\alpha)}(a)$ and $x_j(b), x_j'(b), \dots, x_j^{(q+d_j+\alpha)}(b)$ ($j = 0,\dots,n-1$) are given. For a $(j,k) \in J_{\le\alpha}$, we use

$$\left[x_j^{(k)}(a), x_j^{(k+1)}(a), \dots, x_j^{(k+p)}(a)\right] \quad \text{and} \tag{8.10}$$

$$\left[x_j^{(k)}(b), x_j^{(k+1)}(b), \dots, x_j^{(k+q)}(b)\right], \tag{8.11}$$

and construct the (p,q) Hermite-Nordsieck vector for $x_j^{(k)}$ at b.

8.1.1 Implementation

To construct the (p, q) Hermite-Nordsieck vector for $x_j^{(k)}$ at b, we first merge the vectors (8.10) and (8.11) to form

$$\mathbf{y} = \left[x_j^{(k)}(b), \dots, x_j^{(k+q)}(b), x_j^{(k)}(a), \dots, x_j^{(k+p)}(a)\right]. \tag{8.12}$$

Then, we use

$$[\underbrace{b, b, \dots, b}_{q+1}, \underbrace{a, a, \dots, a}_{p+1}], \tag{8.13}$$

and (8.12) to form a tableau and compute the generalized divided differences (8.7) (see Appendix D.5.1). We implement the function *CompNordsieck* to construct the (p, q) Hermite-Nordsieck vectors for all $x_j^{(k)}$ at b and store them in *nordsieck*. In this function, *t_vec* is (8.13). *ders_a* and *ders_b* contain $x_j(a), x_j'(a), \dots, x_j^{(p+d_j+\alpha)}(a)$ and $x_j(b), x_j'(b), \dots, x_j^{(q+d_j+\alpha)}(b)$ $(j = 0, \dots, n-1)$, respectively.

184 ⟨ Auxiliary functions 107 ⟩ +≡

```
void CompNordsieck(size_t p, size_t q,
        const std::vector⟨double⟩ &t_vec, const std::vector⟨std::vector⟨double⟩⟩
        &ders_a, const std::vector⟨std::vector⟨double⟩⟩ &ders_b,
        IrregularMatrix⟨std::vector⟨double⟩⟩ &nordsieck)
{
  size_t nord_size = p + q + 2;
  std::vector⟨double⟩ y(nord_size);
  for (int j = 0; j < nordsieck.num_rows( ); j++)
```

```
        for (int $k = 0$; $k < nordsieck.num\_cols(j)$; $k$++)
        {
          $merge\_ders(k, p, q, ders\_a[j], ders\_b[j], y)$;
          $comp\_gen\_divdif(t\_vec, y, nordsieck(j, k))$;      /* see Appendix D.5.1 */
        }
    }
```

The function *merge_ders* is implemented to merge the vectors (8.10) and (8.11) and get (8.12).

185 ⟨ Auxiliary functions 107 ⟩ +≡

```
    void merge_ders(int $k$, int $p$, int $q$, const std::vector⟨double⟩ &$v\_a$, const
            std::vector⟨double⟩ &$v\_b$, std::vector⟨double⟩ &$y$)
    {
      for (size_t $r = 0$; $r \le q$; $r$++)
        $y[r] = v\_b[k + r]$;
      for (size_t $r = 0$; $r \le p$; $r$++)
        $y[q + 1 + r] = v\_a[k + r]$;
    }
```

8.2 Prediction

To solve the system (4.20) using the iteration (7.1), we need an initial guess for $\mathbf{x}_{J_{<0}}$ at the next point. If the DAE (1.1) is non-quasilinear, we also need an initial guess for $\mathbf{x}_{J_0}$ to solve (5.10).

For an $x_j^{(k)}$ with $(j,k) \in J_{\leq\alpha}$, where α is given by (8.9), suppose that the values

$$\begin{aligned} &x_j^{(k)}(a), \ldots, x_j^{(k+p-1)}(a) \quad \text{and} \\ &x_j^{(k)}(b), \ldots, x_j^{(k+q)}(b), \end{aligned} \tag{8.14}$$

are given. Denoting by P_{jk} the polynomial interpolating (8.14), we approximate

$$x_j^{(k)}(b+h) \approx P_{jk}(b+h), \quad \text{for a given } h. \tag{8.15}$$

The (p,q) Hermite-Nordsieck vector for $x_j^{(k)}$ at b contains the coefficients of the polynomial $P_{jk} \in \Pi_{p+q}$. Let v_l be the $(l+1)$th element of this vector. Then

$$P_{jk}(t) = \sum_{l=0}^{q} v_l (t-b)^l + (t-b)^{q+1} \sum_{l=1}^{p} v_{q+l}(t-a)^{l-1}. \tag{8.16}$$

Hence, given the Hermite-Nordsieck vector we just need to evaluate the polynomial at $b+h$.

The error of the approximation (8.15) is (see e.g., Theorem 2.1.5.9 in [64])

$$x_j^{(k)}(b+h) - P_{jk}(b+h) = \frac{x_j^{(k+p+q+1)}(\eta_{jk})}{(p+q+1)!} h^{q+1}(h+b-a)^p \quad \text{for some } \eta_{jk} \in [a, b+h].$$

8.2.1 Implementation

We implement the function *PredictSolution* to evaluate $P_{jk}(b+h)$ for all $(j,k) \in J_{\leq\alpha}$. In this function, t is $b+h$, and *nordsieck*(j,k) contains the (p,q) Hermite-Nordsieck vector for $x_j^{(k)}$ at b. The predicted values are stored in the **DAEpoint** *prediction*.

188 ⟨ Auxiliary functions 107 ⟩ +≡

```
void PredictSolution(int p, int q, double a, double b, double t, const
        IrregularMatrix⟨std::vector⟨double⟩⟩ &nordsieck, daets::DAEpoint
        &prediction)
{
  for (size_t j = 0; j < nordsieck.num_rows(); j++)
    for (size_t k = 0; k < nordsieck.num_cols(j); k++)
      prediction(j, k) = eval_hermite(p, q, a, b, t, nordsieck(j, k));      /* P_jk(t) */
}
```

8.2.1.1 Evaluating the polynomial

Denoting $x = t - b$, $y = t - a$, and $s = P_{jk}(t)$, (8.16) is

$$s = v_0 + v_1 x + v_2 x^2 + \ldots + v_{q+1} x^{q+1} + v_{q+2} x^{q+1} y + \ldots + v_{q+p} x^{q+1} y^{p-1}. \qquad (8.17)$$

We can rewrite (8.17) in the form

$$s = v_0 + x\left[v_1 + x\left[v_2 + \ldots + x\Big[v_{q+1} + y\Big[v_{q+2} + y[v_{q+3} + \ldots + y[v_{q+p-1} + yv_{q+p}]]\Big]\Big]\right]\right],$$

and compute it by the following function.

189 ⟨ Auxiliary functions 107 ⟩ +≡

```
double eval_hermite(int p, int q, double a, double b, double t, std::vector⟨double⟩ v)
{
  double x = t − b;
  double y = t − a;
```

```
    double $s = v[q + p]$;
    for (int $l = q + p - 1;\ l \geq q + 1;\ l--$)
        $s = s * y + v[l]$;
    for (int $l = q;\ l \geq 0;\ l--$)
        $s = s * x + v[l]$;
    return $s$;
}
```

8.3 Error estimation

To provide an estimate for the discretization error (4.19), we need to estimate (4.9), namely,

$$\frac{x_j^{(p+q+1+k)}(\eta_{jk})}{(p+q+1)!} \quad \text{where } \eta_{jk} \in (t^*, t^* + h), \tag{8.18}$$

for each $(j, k) \in J_{<0}$.

Denote by u_{jk} and v_{jk} the $(p+q+1)$th elements of the (p, q) Hermite-Nordsieck vectors for $x_j^{(k)}$ at t^* and $t^* + h$, respectively. From (8.5), we approximate

$$\begin{aligned} \frac{x_j^{(k+p+q)}(t^*)}{(p+q)!} &\approx u_{jk} \quad \text{and} \\ \frac{x_j^{(k+p+q)}(t^* + h)}{(p+q)!} &\approx v_{jk}. \end{aligned} \tag{8.19}$$

By the mean-value theorem, there is an $\widetilde{\eta}_{jk} \in (t^*, t^* + h)$ such that

$$x_j^{(k+p+q+1)}(\widetilde{\eta}_{jk}) = \frac{x_j^{(k+p+q)}(t^* + h) - x_j^{(k+p+q)}(t^*)}{h}.$$

Using the approximations (8.19), we write

$$x_j^{(k+p+q+1)}(\widetilde{\eta}_{jk}) \approx \frac{v_{jk}(p+q)! - u_{jk}(p+q)!}{h}.$$

Then we use

$$\widetilde{\xi}_{jk} = \frac{v_{jk} - u_{jk}}{h(p+q+1)} \approx \frac{x_j^{(k+p+q+1)}(\widetilde{\eta}_{jk})}{(p+q+1)!}, \tag{8.20}$$

as an estimation for (8.18).

Theorem 8.1. For a scalar function $y \in C^{p+q+1}[a,b]$ and the polynomial (8.4) obtained from data (8.1) we have

$$y^{(m)}(b) - P^{(m)}(b) = \mathcal{O}\left((b-a)^{p+q+2-m}\right), \quad \text{for } m = q+1, q+2, \ldots, p+q+2.$$

Proof. Using the convergence theorem of Hermite interpolation (see e.g., Theorem 2.1.5.9 in [64]), we can write

$$y(t) - P(t) = \frac{y^{(p+q+2)}(\eta_t)}{(p+q+2)!}(t-a)^{p+1}(t-b)^{q+1}, \quad \text{for some} \quad \eta_t \in [a,b].$$

with $t \in [a,b]$. Denote

$$w(t) = (t-a)^{p+1}(t-b)^{q+1}.$$

Differentiating $w(t)$, and setting $t = b$, the proof follows from

$$w^{(m)}(b) = \mathcal{O}\left((b-a)^{p+q+2-m}\right).$$

The proof of this, although elementary, is tedious and we omit it. □

Here, u_{jk} and v_{jk} are the $(p+q)$th derivatives of Hermite interpolating polynomials of the form (8.4). Computing the (p,q) Hermite-Nordsieck vectors for $x_j^{(k)}$ at t^* and t^*+h with accurate data, we obtain approximations (8.19) with the error of order $\mathcal{O}(h^2)$ by Theorem 8.1. However, in practice, we compute Hermite-Nordsieck vectors using the approximate data with errors depending on the user specified tolerance.

If approximate data of a function y contain errors of order $\mathcal{O}(h^r)$, then an mth order differentiation of an interpolating polynomial of any degree $\geq (r-1)$ obtained from this data yields approximations of $y^{(m)}$ with reduced order of accuracy $\mathcal{O}(h^{r-m})$ [9]. Therefore, if we apply a (p,q) HO method when $p+q$ and user specified tolerances are large, then (8.20) is not a good estimation for (8.18).

Let the irregular matrix $\widetilde{\mathbf{E}}$ contain the values $\widetilde{\xi}_{jk}$ for $(j,k) \in J_{<0}$. We compute

$$e_{pq}h^{p+q+1}\widetilde{\mathbf{E}}, \tag{8.21}$$

where e_{pq} is the error constant given by (4.8). We refer to (8.21) as the *estimated discretization error* (EDE).

8.3.1 Implementation

The **StiffDAEsolver** class maintains a pointer to an object of the **HO** class.

192 ⟨StiffDAEsolver Data Members 192⟩ ≡

HO $*$*ho_*;

See also chunks 195, 196, 197, 212, 229, 233, 235, 253, 295, and 361

This code is used in chunk 27.

The constructor and destructor of the **StiffDAEsolver** class are given in Appendix D.3. We declare the data members and implement the functions in this chapter and Chapter 9.

The function *EstErrHO* is implemented to compute the norm of (8.21). In this function, *order* is $p+q$ and *weight* is used to compute the scaled WRMS norm in (8.21).

194 ⟨Definitions of StiffDAEsolver Private Functions 194⟩ ≡

double StiffDAEsolver :: *EstErrHO*(**int** *order*, **double** *e_pq*,

const IrregularMatrix⟨**double**⟩ &*weight*)

{

⟨ compute $e_{pq}h^{p+q+1}\widetilde{\mathbf{E}}$ 199 ⟩;

⟨ compute $\|e_{pq}h^{p+q+1}\widetilde{\mathbf{E}}\|$ 200 ⟩;

}

See also chunks 203, 215, 220, 222, 223, 234, 238, 258, 297, and 336

This code is used in chunk 362.

Assume that for all $(j,k) \in J_{<0}$ the (p,q) Hermite-Nordsieck vectors for $x_j^{(k)}$ at t^* have been stored in

195 ⟨ StiffDAEsolver Data Members 192 ⟩ +≡

IrregularMatrix⟨**std** :: **vector**⟨**double**⟩⟩ *nordsieck_cur_*;

and at $t^* + h$ they have been stored in

196 ⟨ StiffDAEsolver Data Members 192 ⟩ +≡

IrregularMatrix⟨**std** :: **vector**⟨**double**⟩⟩ *nordsieck_trial_*;

That is, the *nordsieck_trial_*$(j,k)[p+q]$ contains v_{jk}, and *nordsieck_cur_*$(j,k)[p+q]$ contains u_{jk}. By (8.20),

$$h\widetilde{\xi}_{jk} = \frac{v_{jk} - u_{jk}}{p+q+1}.$$

For each $(j,k) \in J_{<0}$ we compute $h\widetilde{\xi}_{jk}$ and store it in

197 ⟨ StiffDAEsolver Data Members 192 ⟩ +≡

```
IrregularMatrix⟨double⟩ ede_;
```

198 ⟨compute $h\widetilde{\xi}_{jk}$ 198⟩ ≡

```
ede_(j, k) = (nordsieck_trial_(j, k)[order] − nordsieck_cur_(j, k)[order])/(order + 1);
```

This code is used in chunk 199.

Now, we estimate the error as

199 ⟨compute $e_{pq}h^{p+q+1}\widetilde{\mathbf{E}}$ 199⟩ ≡

```
for (int j = 0; j < ede_.num_rows( ); j++)
  for (int k = 0; k < ede_.num_cols(j); k++)
    ⟨compute hξ̃_jk 198⟩;
ede_ *= ho_→h_pow_[order] * e_pq;
```

This code is used in chunk 194.

Finally, by calling *wrms_norm* function from the **IrregularMatrix** class, we compute the norm of the error.

200 ⟨compute $\|e_{pq}h^{p+q+1}\widetilde{\mathbf{E}}\|$ 200⟩ ≡

```
return ede_.wrms_norm(weight);
```

This code is used in chunk 194.

8.4 Stepsize and order selection

Suppose that we intend to apply the (p, q) HO method to solve a DAE. Since our interest in this method is for stiff problems, we choose the parameters p and q such that the method is at least A-stable. For given integer $\kappa > 0$, by Theorem 3.3, we choose

$$p = \left\lceil \frac{\kappa}{2} \right\rceil \quad \text{and} \quad q = \kappa - p. \tag{8.22}$$

That is, if κ is odd, then $q = p + 1$, and the method is L-stable. When κ is even, then $q = p$, and the method is A-stable.

We provide the HO method with a variable-order formulation. After an accepted step, we consider keeping the same order, κ, or $\kappa - 1$ or $\kappa + 1$. We consider the HO parameters p and q for the next step as in Table 8.1.

last step	considered for next step		
κ	$\kappa - 1$	κ	$\kappa + 1$
(p, p)	$(p-1, p)$	(p, p)	$(p, p+1)$
$(p, p+1)$	(p, p)	$(p, p+1)$	$(p+1, p+1)$

Table 8.1: Considered (p, q) for next step.

After applying the method of order κ with stepsize h_κ for the last step, we consider stepsizes $\widehat{h}_i$ to continue the integration with orders $i \in \{\kappa - 1, \kappa, \kappa + 1\}$ determined by

$$\widehat{h}_i = \sigma_i h_\kappa,$$

where

$$\sigma_i = \left(\frac{s_i}{\|\mathrm{EDE}_i\|} \right)^{1/(i+1)}. \tag{8.23}$$

Here s_i is a safety factor and EDE_i is the EDE given by (8.21) at order i.

In a variable-order formulation, we need to know a priori an estimation of the computational cost per step for each order that is under consideration for the next step. Ignoring the cost of computing $\mathbf{A}_0$ which is strongly problem dependent, the computational cost of the (p, q) HO method per step is $\mathcal{O}\Big(n^3 + Nn^2q + Nq^2 + N^3\Big)$ (see Table 8.2). We define the following *cost per step* function for the (p, q) HO method

$$\mathcal{C}(n, N, q, h) = \frac{1}{h}\Big(n^3 + Nn^2q + Nq^2 + N^3\Big). \tag{8.24}$$

We determine the order $i \in \{\kappa - 1, \kappa, \kappa + 1\}$ for the next step that minimizes (8.24).

task	computational complexity
LU factorization of $\mathbf{A}_0$	$\mathcal{O}(n^3)$
computing $\mathbf{x}_{J_s}$ for $s = 0, \dots, q-1$	$\mathcal{O}(n^2q + q^2)$
evaluating $\mathbf{f}_{\text{HO}}(\mathbf{x}_{J_{<0}})$	$\mathcal{O}(N(p+q))$
computing $\nabla\mathbf{x}_{J_s}$ for $s = 0, \dots, q-1$	$\mathcal{O}(Nn^2q + Nq^2)$
evaluating $\mathbf{J}_{\text{HO}}$	$\mathcal{O}(N^2q)$
LU factorization of $\mathbf{J}_{\text{HO}}$	$\mathcal{O}(N^3)$
substitutions for solving $\mathbf{J}_{\text{HO}}\boldsymbol{\delta} = -\mathbf{f}_{\text{HO}}(\mathbf{x}_{J_{<0}})$	$\mathcal{O}(N^2)$
total	$\mathcal{O}(n^3 + Nn^2q + Nq^2 + N^3)$

Table 8.2: Cost of the (p, q) HO method per step.

8.4.1 Implementation

To determine the order $i \in \{\kappa - 1, \kappa, \kappa + 1\}$ for the next step that minimizes (8.24), we

- compute σ_i (8.23),
- compute

$$\mathcal{C}_i = \frac{1}{\sigma_i}\Big(n^3 + Nn^2q_i + Nq_i^2 + N^3\Big), \tag{8.25}$$

  for the (p_i, q_i) HO method, and
- find i that minimizes (8.25).

We implement the function *SelectOrder* to select the order for the next step. In this function, *sigma_k* is σ_κ, *weight* is used to compute the scaled WRMS norm, and the vector *epq* contains the constants $|e_{pq}|$ in (4.8) for the p and q under consideration. This function returns an **OrderFlag**.

202 ⟨ enumeration type for order selection 202 ⟩ ≡

```
typedef enum {
  DECREASE_ORDER,      /* if the order for the next step should be κ − 1, */
  DONT_CHANGE_ORDER,   /* if the integration could continue with order κ, */
  INCREASE_ORDER       /* if the order for the next step should be κ + 1 */
} OrderFlag;
```

This code is used in chunk 369.

203 ⟨ Definitions of StiffDAEsolver Private Functions 194 ⟩ +≡

```
OrderFlag StiffDAEsolver::SelectOrder(int k, double sigma_k, const
        IrregularMatrix⟨double⟩ &weight, const std::vector⟨double⟩ &epq)
{
    ⟨ compute $\sigma_{\kappa-1}$ and $\sigma_{\kappa+1}$ 204 ⟩;
    ⟨ compute $\mathcal{C}_{\kappa-1}, \mathcal{C}_\kappa$ and $\mathcal{C}_{\kappa+1}$ 207 ⟩;
    ⟨ find $\min\{\mathcal{C}_{\kappa-1}, \mathcal{C}_\kappa, \mathcal{C}_{\kappa+1}\}$ to determine the possible order change 211 ⟩;
    ⟨ check not to exceed maximum or fall minimum order 213 ⟩;
    return flag;
}
```

Calling the function *EstErrHO*, we estimate $\|\mathrm{EDE}_{\kappa-1}\|$ and $\|\mathrm{EDE}_{\kappa+1}\|$ and store them in *ede_low* and *ede_high*, respectively. $epq[0]$ contains the $|e_{pq}|$ corresponding to order $\kappa - 1$ and $epq[2]$ contains this term for order $\kappa + 1$ (see Table 8.1).

204 ⟨ compute $\sigma_{\kappa-1}$ and $\sigma_{\kappa+1}$ 204 ⟩ ≡

```
double ede_low = EstErrHO(k − 1, epq[0], weight);
double ede_high = EstErrHO(k + 1, epq[2], weight);
```

See also chunk 206.

This code is used in chunk 203.

We implement the following auxiliary function to compute (8.23).

205 ⟨ Auxiliary functions 107 ⟩ +≡

```
double comp_sigma(int order, double lte, double safety)
{
    double power = 1.0/(order + 1);
    double sigma = std::pow(safety/lte, power);
    return sigma;
}
```

Calling this function, we compute $\sigma_{\kappa-1}$ and $\sigma_{\kappa+1}$ and store them in *sigma_low* and *sigma_high*, respectively.

206 $\langle$ compute $\sigma_{\kappa-1}$ and $\sigma_{\kappa+1}$ 204 $\rangle$ $+\equiv$

```
double $sigma\_low = comp\_sigma(k-1, ede\_low, 0.1)$;
double $sigma\_high = comp\_sigma(k+1, ede\_high, 0.05)$;
```

To compute $\mathcal{C}_{\kappa-1}$ and $\mathcal{C}_{\kappa+1}$ we first select the HO parameters $(p_{\kappa-1}, q_{\kappa-1})$ and $(p_{\kappa+1}, q_{\kappa+1})$ as in Table 8.1.

207 $\langle$ compute $\mathcal{C}_{\kappa-1}, \mathcal{C}_{\kappa}$ and $\mathcal{C}_{\kappa+1}$ 207 $\rangle$ $\equiv$

```
int $p = ho\_\rightarrow p\_$,
$q = ho\_\rightarrow q\_$,
$n = ho\_\rightarrow n\_$,
$nn = ho\_\rightarrow num\_indep\_tcs\_$,
$p\_high, p\_low, q\_high, q\_low$;
if $(p \equiv q)$
{
   $p\_high = p$;
   $q\_high = p + 1$;
   $p\_low = p - 1$;
   $q\_low = p$;
}
```

```
else
{
    $p\_high = p + 1;$
    $q\_high = p + 1;$
    $p\_low = p;$
    $q\_low = p;$
}
```

See also chunk 209.

This code is used in chunk 203.

We implement the following function to compute (8.25).

208 ⟨ Auxiliary functions 107 ⟩ +≡

```
double $cost\_per\_step$(int $n$, int $nn$, int $p$, int $q$, double $h$)
{
    int $n2 = n * n;$
    int $n3 = n * n2;$
    int $nn3 = nn * nn * nn;$
    int $q2 = q * q;$
    double $cost = (n3 + nn * n2 * q + nn * q2 + nn3)/h;$
    return $cost;$
}
```

Calling the function *cost_per_step*, we compute $\mathcal{C}_{\kappa-1}, \mathcal{C}_{\kappa}$ and $\mathcal{C}_{\kappa+1}$ and store them in *cost_low*, *cost_k*, and *cost_high*, respectively.

209 ⟨ compute $\mathcal{C}_{\kappa-1}, \mathcal{C}_{\kappa}$ and $\mathcal{C}_{\kappa+1}$ 207 ⟩ +≡

```
double cost_low = cost_per_step(n, nn, p_low, q_low, sigma_low);
double cost_k = cost_per_step(n, nn, p, q, sigma_k);
double cost_high = cost_per_step(n, nn, p_high, q_high, sigma_high);
```

The new order is then chosen to minimize $\{\mathcal{C}_{\kappa-1}, \mathcal{C}_{\kappa}, \mathcal{C}_{\kappa+1}\}$. This is done by the following function.

210 ⟨ Auxiliary functions 107 ⟩ +≡

```
OrderFlag min_cost(double cost1, double cost2, double cost3)
{
  if ((cost1 < cost2) ∧ (cost1 < cost3))
    return DECREASE_ORDER;
  if ((cost3 < cost2) ∧ (cost3 < cost1))
    return INCREASE_ORDER;
  return DONT_CHANGE_ORDER;
}
```

211 ⟨ find $\min\{\mathcal{C}_{\kappa-1}, \mathcal{C}_{\kappa}, \mathcal{C}_{\kappa+1}\}$ to determine the possible order change 211 ⟩ ≡

```
OrderFlag flag = min_cost(cost_low, cost_k, cost_high);
```

This code is used in chunk 203.

Finally, we check if the new order is in the range of specified minimum and maximum orders stored in

212 ⟨StiffDAEsolver Data Members 192⟩ +≡

```
int min_order_, max_order_;
```

213 ⟨check not to exceed maximum or fall minimum order 213⟩ ≡

```
if ((k ≡ max_order_ ∧ flag ≡ INCREASE_ORDER) ∨
      (k ≡ min_order_ ∧ flag ≡ DECREASE_ORDER))
  return DONT_CHANGE_ORDER;
```

This code is used in chunk 203.

Chapter 9

The integrator function

To start an integration and to obtain the required data for computing the Hermite-Nordsieck vectors (§8.1), we use the explicit Taylor series method on the first step. This is carried out by the *IntegrateByExplicitTS* function, Appendix A. Then, we use our HO method, step by step, from the found solution by *IntegrateByExplicitTS* up to a final time.

In this chapter, we first describe the components of our overall algorithm and implement the function *IntegrateByHO* for integrating the given problem using the HO method in §9.1. Then, in §9.2 we implement the function *integrate* which advances the solution from the initial time up to the final time by calling the functions *IntegrateByExplicitTS* and *IntegrateByHO*.

9.1 Integration by HO method

Consider the DAE (1.1) and assume that the following are given

- absolute error tolerance atol, and relative error tolerance rtol,
- current order κ, minimum order κ_{min} and maximum order κ_{max},
- previous time t_{prev}, current time t_{cur} and final time t_{end},
- TCs $\mathbf{x}_{J_{<p+\alpha}}$ at t_{prev} and $\mathbf{x}_{J_{<q+\alpha}}$ at t_{cur} with $p = \lceil \kappa/2 \rceil$, $q = \kappa - p$, and α in (8.9), and
- a trial stepsize h.

Our algorithm for integrating (1.1) using the HO method in §4.2 to find a numerical solution $\mathbf{x}_{J_{<0}}$ at t_{end} consists of the following steps (see Figure 9.1 for an illustration).

1. Prepare for integration, that is,
   - compute $p = \lceil \kappa/2 \rceil$, $q = \kappa - p$, coefficients c_r^{pq} in (3.10) for $r = 0, \ldots, p$, and c_r^{qp} in (3.11) for $r = 0, \ldots, q$, and the error constants e_{pq} in (4.8) for orders $\kappa - 1$, κ and $\kappa + 1$,
   - compute the weights (7.6) required for the WRMS norm (7.3),
   - construct an irregular matrix $\mathbf{HN}_{\text{cur}}$ whose (j, k)th entry is the (p, q) Hermite-Nordsieck vector for $x_j^{(k)}$, $(j, k) \in J_{\leq\alpha}$, at t_{cur} (see §8.1), and
   - the minimum allowed stepsize h_{min}.

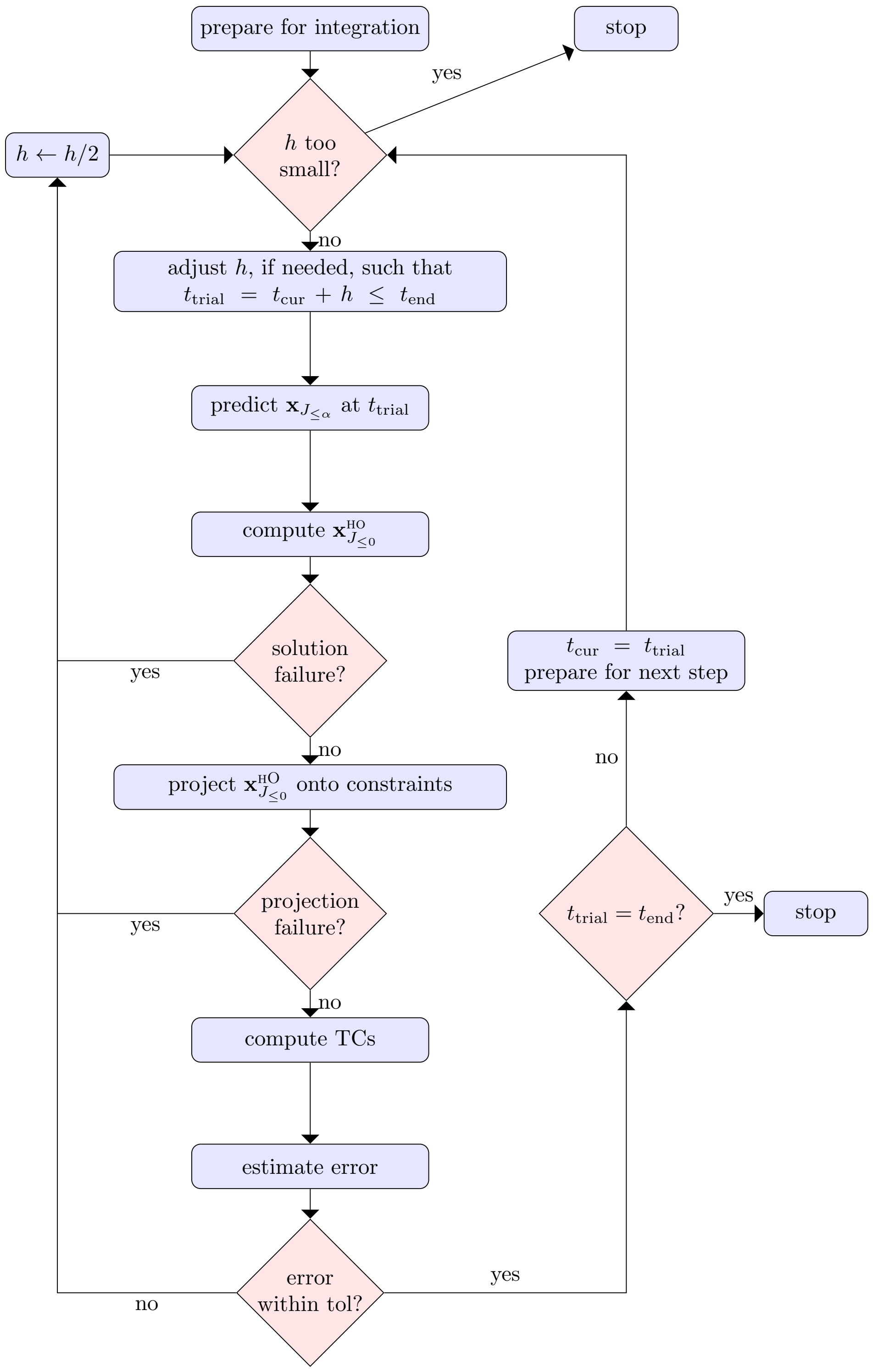
prepare for integration
stop
yes
h too small?
$h \leftarrow h/2$
no
adjust h, if needed, such that $t_{\text{trial}} = t_{\text{cur}} + h \leq t_{\text{end}}$
predict $\mathbf{x}_{J_{\leq \alpha}}$ at t_{trial}
compute $\mathbf{x}^{\text{HO}}_{J_{\leq 0}}$
solution failure?
yes
no
$t_{\text{cur}} = t_{\text{trial}}$ prepare for next step
project $\mathbf{x}^{\text{HO}}_{J_{\leq 0}}$ onto constraints
no
projection failure?
yes
no
$t_{\text{trial}} = t_{\text{end}}$?
yes
stop
compute TCs
estimate error
error within tol?
no
yes

Figure 9.1: Algorithm overview.

2. Check and adjust the stepsize h, that is, if $h < h_{\text{min}}$ terminate the integration and return the solution at t_{cur}. Otherwise, adjust h, if needed, such that $t_{\text{trial}} = t_{\text{cur}} + h \le t_{\text{end}}$.

3. Find an initial guess for $\mathbf{x}_{J_{\le\alpha}}$ at t_{trial} using $\mathbf{HN}_{\text{cur}}$ (see §8.2).

4. Compute $\mathbf{x}^{\text{HO}}_{J_{\le 0}}$ at t_{trial}, that is, construct and solve the HO system $\mathbf{F}(\mathbf{x}_{J_{<0}}) = 0$ to obtain a solution $\mathbf{x}^{\text{HO}}_{J_{<0}}$ at t_{trial} (see §7.3). In addition, compute TCs $\mathbf{x}^{\text{HO}}_{J_0}$ at t_{trial} by solving (see Chapter 5)

$$\mathbf{f}_{I_0}\left(t_{\text{trial}}, \mathbf{x}^{\text{HO}}_{J_{<0}}, \mathbf{x}^{\text{HO}}_{J_0}\right) = 0.$$

If solving one of the above systems fails, then $h \leftarrow h/2$ and go to 2.

5. Solve the constrained optimization problem

$$\min_{\mathbf{x}_{J_{\le 0}}} \|\mathbf{x}_{J_{\le 0}} - \mathbf{x}^{\text{HO}}_{J_{\le 0}}\|_2 \quad \text{subject to} \quad \mathbf{f}_{I_{\le 0}}(t_{\text{trial}}, \mathbf{x}_{J_{\le 0}}) = 0,$$

to obtain the projected solution $\mathbf{x}^{\text{PR}}_{J_{\le 0}}$ at t_{trial}. If the projection fails, then $h \leftarrow h/2$ and go to 2.

6. Compute TCs $\mathbf{x}^{\text{PR}}_{J_{<q+\alpha}}$ at t_{trial} by solving (see Chapter 5)

$$\mathbf{f}_{I_s}\left(t_{\text{trial}}, \mathbf{x}^{\text{PR}}_{J_{<s}}, \mathbf{x}^{\text{PR}}_{J_s}\right) = 0, \quad s = 1, \dots, q+\alpha.$$

7. Compute EDE, that is,

   - use $\mathbf{x}_{J_{<p+\alpha}}$ at t_{cur} and $\mathbf{x}^{\text{PR}}_{J_{<q+\alpha}}$ at t_{trial} to construct an irregular matrix $\mathbf{HN}_{\text{trial}}$ whose (j,k)th entry is the (p,q) Hermite-Nordsieck vector for $x_j^{(k)}$, $(j,k) \in J_{\le\alpha}$, at t_{trial}, and

- use $\mathbf{HN}_{\text{cur}}$ and $\mathbf{HN}_{\text{trial}}$ to compute EDE (see §8.3).

8. If $\|\text{EDE}\| > 1$, then $h \leftarrow h/2$ and go to 2. If $\|\text{EDE}\| \leq 1$ and $t_{\text{trial}} = t_{\text{end}}$, then terminate the integration with $\mathbf{x}^{\text{PR}}_{J_{<0}}$ as our solution at t_{end}. Otherwise, prepare to perform the next step, that is,

   - $t_{\text{cur}} \leftarrow t_{\text{trial}}$,
   - $\mathbf{HN}_{\text{cur}} \leftarrow \mathbf{HN}_{\text{trial}}$,
   - update the weights using entries of $\mathbf{x}^{\text{PR}}_{J_{<0}}$,
   - update $h_{\min}$,
   - predict a stepsize h for next step,
   - determine an order κ with $\kappa_{\min} \leq \kappa \leq \kappa_{\max}$ for next step (see §8.4). If change in order, compute $p = \lceil \kappa/2 \rceil$, $q = \kappa - p$, coefficients c_r^{pq} for $r = 0, \ldots, p$, and c_r^{qp} for $r = 0, \ldots, q$, and the error constants e_{pq} for orders $\kappa - 1$, κ and $\kappa + 1$, and
   - go to 2.

This algorithm is carried out by the function *IntegrateByHO*. In this function, x contains an initial point and will be updated by the solution at t_{end}. If the function cannot reach t_{end}, the solution at some $t < t_{\text{end}}$ is returned.

215 ⟨ Definitions of StiffDAEsolver Private Functions 194 ⟩ +≡

void StiffDAEsolver :: *IntegrateByHO*(**daets** :: **DAEsolution** &x, **double**

t_end, **daets** :: **SolverExitFlag** &*state*)

{

```
    ⟨ declare variables for integration 218 ⟩;
    ⟨ prepare for integration 216 ⟩;
    while ($x.t\_ \neq t\_end$)
    {
      ⟨ check and adjust the stepsize $h$ 247 ⟩;
      ⟨ find an initial guess for $\mathbf{x}_{J_{\leq\alpha}}$ 250 ⟩;
      ⟨ compute $\mathbf{x}^{\mathrm{HO}}_{J_{\leq 0}}$ 252 ⟩;
      ⟨ project $\mathbf{x}^{\mathrm{HO}}_{J_{\leq 0}}$ onto constraints 260 ⟩;
      ⟨ compute higher-order TCs 266 ⟩;
      ⟨ estimate the error 270 ⟩;
      ⟨ prepare for next step 275 ⟩;
      ⟨ optional output 292 ⟩;
    }
  }
```

9.1.1 Preparation for integration

The **DAEsolver** class has the protected member *params_* which is a pointer to an object of the **Parameters** class. We obtain absolute and relative tolerances by calling *getAtol* and *getRtol* from **Parameters** class and store them in *atol* and *rtol*, respectively.

```
216 ⟨ prepare for integration 216 ⟩ ≡
      double atol = params_→getAtol( );
      double rtol = params_→getRtol( );
```

x.*stats_*→*setTols*(*atol*, *rtol*);

See also chunks 219, 225, 228, 232, 237, 240, 242, 244, 246, 255, and 268

This code is used in chunk 215.

9.1.1.1 Parameters and coefficients of the method

Given the order, we compute parameters p and q in the (p, q) HO method by (8.22).

218 ⟨ declare variables for integration 218 ⟩ ≡

int p, q;

See also chunks 224, 227, 231, 236, 239, 241, 243, 249, 251, 259, 267, 269, 277, 283, 287, and 288

This code is used in chunk 215.

219 ⟨ prepare for integration 216 ⟩ +≡

$p = x.$*order_*$/2$;

$q = x.$*order_* $- p$;

We implement the function *set_pq_comp_coeffs* to

- set p and q in the **HO** object,
- compute coefficients c_r^{pq} in (3.10) for $r = 0, \dots, p$, and c_r^{qp} in (3.11) for $r = 0, \dots, q$, and
- compute $|e_{pq}|$ in (4.8) for all (i, j) HO methods that are under consideration for the next step.

220 ⟨ Definitions of StiffDAEsolver Private Functions 194 ⟩ +≡

void StiffDAEsolver :: *set_pq_comp_coeffs*(**int** p, **int** q, **std** :: **vector**⟨**double**⟩ &*ho_epq*)

{

  *ho_*→*set_pq*(p, q);

  *ho_*→*CompCpq*( );

  *ho_*→*CompCqp*( );

  CompEpq$(p, q,$ *ho_epq*$)$;

}

The function *set_pq* in the **HO** class sets the parameters p and q of the (p, q) HO method.

221 ⟨ Definitions of HO Private Functions 37 ⟩ +≡

void HO :: *set_pq*(**int** p, **int** q)

{

  *p_* $= p$;

  *q_* $= q$;

}

We implement the function *CompErrorCanstant* to compute $|e_{pq}|$ for the (p, q) HO method.

222 ⟨ Definitions of StiffDAEsolver Private Functions 194 ⟩ +≡

inline double StiffDAEsolver :: *CompErrorConstant*(**int** p, **int** q)

{

  return *ho_*→*factorial_*$[p]$ ∗ *ho_*→*factorial_*$[q]$/*ho_*→*factorial_*$[p + q]$;

}

Calling this function, the function *CompEpq* computes e_{ij} for all (i, j) HO methods that are under consideration for the next step as in Table 8.1.

223 ⟨ Definitions of StiffDAEsolver Private Functions 194 ⟩ +≡

```
void StiffDAEsolver::CompEpq(int p, int q, std::vector⟨double⟩ &e_pq)
{
  assert(e_pq.size() ≡ 3);
  if (p ≡ q)
  {
    e_pq[0] = CompErrorConstant(p − 1, p);
    e_pq[1] = CompErrorConstant(p, p);
    e_pq[2] = CompErrorConstant(p, p + 1);
  }
  else
  {
    e_pq[0] = CompErrorConstant(p, p);
    e_pq[1] = CompErrorConstant(p, p + 1);
    e_pq[2] = CompErrorConstant(p + 1, p + 1);
  }
}
```

224 ⟨ declare variables for integration 218 ⟩ +≡

```
std::vector⟨double⟩ epq(3);
```

225 ⟨ prepare for integration 216 ⟩ +≡

```
set_pq_comp_coeffs(p, q, epq);
```

9.1.1.2 The weights for WRMS norm

To compute the WRMS norm given by (7.3) for the error estimation (8.21), we precompute (7.6). The following function performs this task.

226 ⟨ Auxiliary functions 107 ⟩ +≡

```
void CompWeight(const daets::DAEpoint &x, double rtol, double
        atol, IrregularMatrix⟨double⟩ &w)
{
  for (size_t j = 0; j < w.num_rows( ); j++)
    for (size_t k = 0; k < w.num_cols(j); k++)
      w(j, k) = 1.0/(atol + std::fabs(x(j, k)) * rtol);
}
```

Calling *CompWeight*, we store (7.6) in

227 ⟨ declare variables for integration 218 ⟩ +≡

```
IrregularMatrix⟨double⟩ weight;
weight = IrregularMatrix⟨double⟩(ho_¬d_);
```

228 ⟨ prepare for integration 216 ⟩ +≡

```
CompWeight(x, rtol, atol, weight);
```

9.1.1.3 Hermite-Nordsieck vectors

$x.t\_$ contains the current time. To construct the (p, q) Hermite-Nordsieck vectors for $x_j^{(k)}$, $(j, k) \in J_{\leq\alpha}$, at $x.t\_$ by the function *CompNordsieck* implemented in §8.1, we need (8.13), (8.10) and (8.11) , with $a = t\_prev\_$ and $b = x.t\_$.

229 ⟨ StiffDAEsolver Data Members 192 ⟩ +≡

```
double t_prev_;
```

The following function returns the vector (8.13) through *t_vec*.

230 ⟨ Auxiliary functions 107 ⟩ +≡

```
void create_t_vec(double a, double b, int p, int q, std::vector⟨double⟩ &t_vec)
{
  if (t_vec.size() ≠ (p + q + 2))
    t_vec.resize(p + q + 2);
  for (int i = 0; i ≤ q; i++)
    t_vec[i] = b;
  for (int i = 0; i ≤ p; i++)
    t_vec[q + 1 + i] = a;
}
```

We call the function *create_t_vec* to create the vector (8.13) and store it in

231 ⟨ declare variables for integration 218 ⟩ +≡

```
std::vector⟨double⟩ t_vec_cur;
```

t_vec_cur.reserve$($*max_order_* $+\ 2)$;

232 ⟨ prepare for integration 216 ⟩ +≡

create_t_vec(*t_prev_*, x.*t_*, p, q, *t_vec_cur*);

Scaled TCs $\mathbf{x}_{J_{<p}}$ at *t_prev_* are stored in

233 ⟨ StiffDAEsolver Data Members 192 ⟩ +≡

std :: **vector**⟨**std** :: **vector**⟨**double**⟩⟩ *tcs_prev_*;

and x.*savedTCs_* contains scaled TCs $\mathbf{x}_{J_{<q}}$ at x.*t_*. However, in (8.10) and (8.11), we need derivatives $x_j^{(l)}$ at *t_prev_* and x.*t_*, respectively. We implement the function *unscale_tcs* to unscale TCs.

234 ⟨ Definitions of StiffDAEsolver Private Functions 194 ⟩ +≡

```
void StiffDAEsolver :: unscale_tcs(const std :: vector⟨double⟩ &pow_h,
            std :: vector⟨std :: vector⟨double⟩⟩ &tcs)
{
  for (size_t j = 0; j < tcs.size( ); j++)
    for (size_t l = 0; l < tcs[j].size( ); l++)
      tcs[j][l] /= pow_h[l];
}
```

In this function, we need the powers of the stepsize. The x.*h_saved_tcs_* contains the current stepsize, and the previous stepsize is stored in

235 ⟨ StiffDAEsolver Data Members 192 ⟩ +≡

```
double h_prev_;
```

We use the function *CompPowersH* implemented in §7.3.2 to compute the powers of the previous and current stepsizes and store them in

236 ⟨ declare variables for integration 218 ⟩ +≡

```
std::vector⟨double⟩ h_prev_pow_, h_cur_pow_;
```

respectively.

237 ⟨ prepare for integration 216 ⟩ +≡

```
int size_h_pow = sadata_→get_max_d( ) + x.order_ + 1;
CompPowersH(size_h_pow, h_prev_, h_prev_pow_);
CompPowersH(size_h_pow, x.h_saved_tcs_, h_cur_pow_);
```

The following function extracts derivatives from TCs.

238 ⟨ Definitions of StiffDAEsolver Private Functions 194 ⟩ +≡

```
void StiffDAEsolver::tcs_to_ders(const std::vector⟨std::vector⟨double⟩⟩
        &tcs, std::vector⟨std::vector⟨double⟩⟩ &ders)
{
  ders = tcs;
  for (size_t j = 0; j < ders.size( ); j++)
    for (size_t l = 0; l < ders[j].size( ); l++)
      ders[j][l] *= ho_→factorial_[l];
}
```

Calling the functions *unscale_tcs* and *tcs_to_ders*, we store (8.10) in

239 ⟨ declare variables for integration 218 ⟩ +≡

```
std::vector⟨std::vector⟨double⟩⟩ ders_prev;
```

240 ⟨ prepare for integration 216 ⟩ +≡

```
unscale_tcs(h_prev_pow_, tcs_prev_);
tcs_to_ders(tcs_prev_, ders_prev);
```

Analogously, we store (8.11) in

241 ⟨ declare variables for integration 218 ⟩ +≡

```
std::vector⟨std::vector⟨double⟩⟩ ders_cur;
```

242 ⟨ prepare for integration 216 ⟩ +≡

```
unscale_tcs(h_cur_pow_, x.savedTCs_);
tcs_to_ders(x.savedTCs_, ders_cur);
for (size_t j = 0; j < sadata_→get_size(); j++)
  for (size_t l = 0; l ≤ sadata_→get_d(j); l++)
    ho_→ts_→set_var_coeff(j, l, x.savedTCs_[j][l]);
```

Now, we can call the function *CompNordsieck* to obtain the (p, q) Hermite-Nordsieck vectors for $x_j^{(k)}$, $(j, k) \in J_{\leq\alpha}$, at $x.t\_$ and store them in *nordsieck_cur_*.

243 ⟨ declare variables for integration 218 ⟩ +≡

```
std::vector⟨size_t⟩ nord_size(sadata_→get_size());
for (int j = 0; j < sadata_→get_size(); j++)
```

```
    nord_size[j] = x.getNumDerivatives(j);
  std::vector⟨double⟩ nord_vec(max_order_);
  nordsieck_cur_ = IrregularMatrix⟨std::vector⟨double⟩⟩(nord_size, nord_vec);
```

244 ⟨prepare for integration 216⟩ +≡

```
  CompNordsieck(p, q, t_vec_cur, ders_prev, ders_cur, nordsieck_cur_);
```

We call the function *compHmin* to compute the smallest allowed stepsize.

245 ⟨compute the smallest allowed stepsize 245⟩ ≡

```
  double h_smallest = daets::compHmin(x, t_end, params_);
  params_→setHmin(h_smallest);
```

This code is used in chunks 246 and 275

246 ⟨prepare for integration 216⟩ +≡

  ⟨compute the smallest allowed stepsize 245⟩

9.1.2 Checking the stepsize

x.*htrial_* contains the trial stepsize. Comparing x.*htrial_* with the smallest allowed stepsize, we decide to terminate the integration if the stepsize is too small.

247 ⟨check and adjust the stepsize h 247⟩ ≡

```
  if (fabs(x.htrial_) < params_→getHmin())
  {
    state = daets::htoosmall;
    x.state_ = daets::DAEsolution::EndOfPath;
```

x.*stats_*→*stopTimer*( );

return;

}

See also chunk 248.

This code is used in chunk 215.

x.*ttrial_* contains the trial time, namely, x.*ttrial_* = x.*t_* + x.*htrial_*. Calling the function **daets**::*checkIfLastStep*, we adjust the stepsize x.*htrial_* such that x.*ttrial_* $\leq$ *t_end*.

248 ⟨ check and adjust the stepsize h 247 ⟩ +≡

daets::*checkIfLastStep*(x.*ttrial_*, x.*htrial_*, x.*t_*, *t_end*);

9.1.3 Finding an initial guess

We have already obtained the (p, q) Hermite-Nordsieck vectors for $x_j^{(k)}$, $(j, k) \in J_{\leq\alpha}$, at x.*t_* and stored them in *nordsieck_cur_*. Hence, we can call the function *PredictSolution* implemented in §8.2 to find an initial guess for $\mathbf{x}_{J_{\leq\alpha}}$ and store it in

249 ⟨ declare variables for integration 218 ⟩ +≡

daets::**DAEpoint** *ho_solution*(∗**this**);

250 ⟨ find an initial guess for $\mathbf{x}_{J_{\leq\alpha}}$ 250 ⟩ ≡

PredictSolution(p, q, *t_vec_cur*[$q+1$], *t_vec_cur*[0], x.*ttrial_*, *nordsieck_cur_*,
        ho_solution);

This code is used in chunk 215.

9.1.4 Applying the HO method

We call the function *CompHoSolution* implemented in §7.3 to compute the solution of the HO system (4.20). It returns the computed solution $\mathbf{x}^{\text{HO}}_{J_{\leq 0}}$ at x.*ttrial_* through *ho_solution*.

251 ⟨ declare variables for integration 218 ⟩ +≡

HoFlag *ho_flag*;

252 ⟨ compute $\mathbf{x}^{\text{HO}}_{J_{\leq 0}}$ 252 ⟩ ≡

ho_flag = *ho_*→*CompHoSolution*(x.*ttrial_*, x.*htrial_*, x.*tol_*, *weight*, x.*savedTCs_*,
    ho_solution);

assert(*ho_flag* ≠ `SYS_JAC_SINGULAR`);

See also chunk 257.

This code is used in chunk 215.

By default, we disable the computing of the condition number of $\mathbf{J}_{\text{HO}}$.

253 ⟨ StiffDAEsolver Data Members 192 ⟩ +≡

bool *cond_flag_* = *false*;

However, a user can activate it by calling the function *comp_cond*.

254 ⟨ Definitions of StiffDAEsolver Public Functions 254 ⟩ ≡

void StiffDAEsolver :: *comp_cond*( ) { *cond_flag_* = *true*; }

See also chunks 294 and 296

This code is used in chunk 362.

255 ⟨ prepare for integration 216 ⟩ +≡

```
$ho\_\!\!\rightarrow\!need\_cond\_jac(cond\_flag\_);$
```

If the iteration (7.1) for solving the HO system (4.20) is not convergent, we half the stepsize and repeat the step.

257 $\langle$ compute $\mathbf{x}^{\mathrm{HO}}_{J_{\leq 0}}$ 252 $\rangle$ +≡

```
if (ho_flag ≠ HO_CONVERGENT)
{
  x.htrial_ /= 2;
  x.stats_→countSteps(false);      /* counts the number of rejected steps */
  SetSavedTCs(x.savedTCs_, q);      /* see bellow */
  continue;
}
```

We implement the function *SetSavedTCs* to set the TCs at $x.t\_$ in the **TaylorSeries** object. We will use them later as initial guesses for TCs at next point (see §5.1).

258 $\langle$ Definitions of StiffDAEsolver Private Functions 194 $\rangle$ +≡

```
void StiffDAEsolver::SetSavedTCs(const std::vector⟨std::vector⟨double⟩⟩ &tcs, int
        q)
{
  for (int j = 0; j < sadata_→get_size(); j++)
    for (int l = 0; l < q + sadata_→get_d(j); l++)
      ho_→ts_→set_var_coeff(j, l, tcs[j][l]);
```

}

9.1.5 Projection

The projection step is done by calling the function **daets**::*projectTSsolution*. It returns the projected solution $\mathbf{x}^{\text{PR}}_{J_{\leq 0}}$ at x.*ttrial_* through x.*xtrial_*. Since the matrix $\mathbf{A}_0 = \partial \mathbf{f}_{I_0}/\partial \mathbf{x}_{J_0}$ is used for projection, this function computes and returns the updated $\mathbf{A}_0$ as well.

259 ⟨declare variables for integration 218⟩ +≡

bool *projected*;

260 ⟨project $\mathbf{x}^{\text{HO}}_{J_{\leq 0}}$ onto constraints 260⟩ ≡

int *exitflag*;

daets::*projectTSsolution*(x.*ttrial_*, *ho_solution*, *params_*→*get_ts_proj_tol*( ), *jac_*,

&x.*xtrial_*, *ho_*→*sys_jac_*, &*exitflag*);

projected = ¬*exitflag*;

See also chunk 261.

This code is used in chunk 215.

If solving this problem fails, we half the stepsize and repeat the step.

261 ⟨project $\mathbf{x}^{\text{HO}}_{J_{\leq 0}}$ onto constraints 260⟩ +≡

if (¬*projected*)

{

x.*htrial_* /= 2;

x.*stats_*→*countSteps*(*false*); /∗ counts the number of rejected steps ∗/

```
    SetSavedTCs(x.savedTCs_, q);
    continue;
  }
```

9.1.6 Computing higher-order TCs

To compute the (p, q) Hermite-Nordsieck vectors for $x_j^{(k)}$, $(j, k) \in J_{\leq\alpha}$, at t_{trial}, we need $\mathbf{x}^{\text{PR}}_{J_{<q+\alpha}}$ at t_{trial}. Given $\mathbf{x}^{\text{PR}}_{J_{\leq 0}}$ and $\mathbf{A}_0$, the following function computes $\mathbf{x}^{\text{PR}}_{J_s}$, $s = 1, 2, \dots, q+\alpha$ at t_{trial} by calling the function *CompTCsLinear* after finding the LU decomposition of $\mathbf{A}_0$ (see §5.1). Here, x contains $\mathbf{x}^{\text{PR}}_{J_{\leq 0}}$.

262 ⟨Definitions of HO Private Functions 37⟩ +≡

```
  bool HO::CompTCs(daets::DAEpoint &x)
  {
    ⟨set $\mathbf{x}^{\text{PR}}_{J_{\leq 0}}$ 264⟩;
    ⟨find LU decomposition of $\mathbf{A}_0$ 265⟩;
    for (int s = 1; s ≤ q_ − sadata_→isLinear( ); s++)
      CompTCsLinear(s);
    return true;
  }
```

The $x(j, k)$ contains the projected $x_j^{(k)}$ at t_{trial}. Hence, $x(j, k)/k!$ gives the projected TC $x_j^{(k)}/k!$. we implement the function *SetProjected* to compute and set these TCs for $j = 0, 1, \dots, n-1$ and $k = 0, 1, \dots, d_j$.

263 ⟨Definitions of HO Private Functions 37⟩ $+\equiv$

```
void HO::SetProjected(daets::DAEpoint &x)
{
  for (int j = 0; j < n_; j++)
    for (int k = 0; k ≤ d_[j]; k++)
    {
      double tc = x(j, k)/factorial_[k];
      ts_→set_var_coeff(j, k, tc);
    }
}
```

264 ⟨set $\mathbf{x}^{\text{PR}}_{J_{\leq 0}}$ 264⟩ $\equiv$

```
SetProjected(x);
```

This code is used in chunk 262.

The *sys_jac_* contains $\mathbf{A}_0$. We need its LU decomposition.

265 ⟨find LU decomposition of $\mathbf{A}_0$ 265⟩ $\equiv$

```
int sys_info;
daets::LU(n_, sys_jac_, ipiv_, &sys_info);
if (sys_info ≠ 0)
  return false;
```

This code is used in chunk 262.

Calling the function *CompTCs*, we compute $\mathbf{x}_{J_{<q+\alpha}}$.

266 ⟨ compute higher-order TCs 266 ⟩ ≡

$\textbf{bool}\ \textit{computed\_tcs} = \textit{ho\_}{\rightarrow}\textit{CompTCs}(x.\textit{xtrial\_});$

$\textit{assert}(\textit{computed\_tcs});$

This code is used in chunk 215.

9.1.7 Error estimation

Analogously to §9.1.1.3, we compute Hermite-Nordsieck vectors for $x_j^{(k)}$, $(j,k) \in J_{\leq\alpha}$, at $x.\textit{ttrial\_}$ and store them in

267 ⟨ declare variables for integration 218 ⟩ +≡

$\textit{nordsieck\_trial\_} = \textbf{IrregularMatrix}\langle\textbf{std}::\textbf{vector}\langle\textbf{double}\rangle\rangle(\textit{nord\_size}, \textit{nord\_vec});$

such that

268 ⟨ prepare for integration 216 ⟩ +≡

$\textit{nordsieck\_trial\_} = \textit{nordsieck\_cur\_};$ /* for memory allocation. */

Calling the function *create_t_vec*, we create and store the vector (8.13) with $a = x.t\_$ and $b = x.\textit{ttrial\_}$ in

269 ⟨ declare variables for integration 218 ⟩ +≡

$\textbf{std}::\textbf{vector}\langle\textbf{double}\rangle\ \textit{t\_vec\_trial};$

270 ⟨ estimate the error 270 ⟩ ≡

$\textit{create\_t\_vec}(x.t\_, x.\textit{ttrial\_}, p, q, \textit{t\_vec\_trial});$

See also chunks 271, 272, 273, and 274

This code is used in chunk 215.

We need TCs at x.*t_* and x.*ttrial_*.

271 ⟨estimate the error 270⟩ +≡

$tcs\_prev\_ = x.savedTCs\_;$ /* *tcs_prev_* contains TCs at x.*t_* */

$x.saveTCs(ho\_\neg ts\_);$ /* x.*savedTCs_* contains TCs at x.*ttrial_* */

We call the function *tcs_to_ders* to extract derivatives from TCs.

272 ⟨estimate the error 270⟩ +≡

$tcs\_to\_ders(tcs\_prev\_, ders\_prev);$

$tcs\_to\_ders(x.savedTCs\_, ders\_cur);$

Now, we can call the function *CompNordsieck* which returns the (p, q) Hermite-Nordsieck vectors for $x_j^{(k)}$, $(j, k) \in J_{\le\alpha}$, at x.*ttrial_* through *nordsieck_trial_*.

273 ⟨estimate the error 270⟩ +≡

$CompNordsieck(p, q, t\_vec\_trial, ders\_prev, ders\_cur, nordsieck\_trial\_);$

Using *nordsieck_cur_* and *nordsieck_trial_*, we compute ||EDE|| in (8.21) by calling the function *EstErrHO* implemented in §8.3 and store it in x.*e_*.

274 ⟨estimate the error 270⟩ +≡

$x.e\_ = EstErrHO(x.order\_, epq[1], weight);$

9.1.8 Preparation for next step

If $\|\text{EDE}\| \le 1$ and $t_{\text{trial}} < t_{\text{end}}$, we prepare to perform the next step. If $\|\text{EDE}\| > 1$, we half the stepsize and repeat the step.

```
275 ⟨ prepare for next step 275 ⟩ ≡
  if ($x$.e_ ≤ 1)
  {
    ⟨ compare the taken stepsize and order with previous ones 276 ⟩;
    ⟨ accept the solution 279 ⟩;
    ⟨ predict the stepsize for next step 284 ⟩;
    $x$.tol_ = atol + rtol ∗ $x$.max_norm( );
    ⟨ determine the order for next step 290 ⟩;
    ⟨ compute the smallest allowed stepsize 245 ⟩;
    CompWeight($x$.xtrial_, rtol, atol, weight);
    t_vec_cur = t_vec_trial;
    nordsieck_cur_ = nordsieck_trial_;
  }
  else
  {
    $x$.htrial_ /= 2;
    $x$.savedTCs_ = tcs_prev_;
    SetSavedTCs($x$.savedTCs_, $q$);
    $x$.stats_→countSteps(false);    /∗ counts the number of rejected steps ∗/
    continue;
  }
```

This code is used in chunk 215.

We call the function *setHminMax* from the **Stats** class to check if x.*htrial_* is the smallest or largest stepsize.

276 ⟨ compare the taken stepsize and order with previous ones 276 ⟩ ≡

```
$x$.stats_→setHminMax(fabs($x$.htrial_));
```

See also chunk 278.

This code is used in chunk 275.

In addition, we check if x.*order_* is the largest order that has been applied so far.

277 ⟨ declare variables for integration 218 ⟩ +≡

```
int largest_order = $x$.order_;
```

278 ⟨ compare the taken stepsize and order with previous ones 276 ⟩ +≡

```
if (largest_order ≤ $x$.order_)
{
   largest_order = $x$.order_;
   $x$.stats_→setOrder(largest_order);
}
```

If $\|\text{EDE}\| \leq 1$ and $t_{\text{trial}} = t_{\text{end}}$, we terminate the integration and accept $\mathbf{x}^{\text{PR}}_{J<0}$ as our solution at t_{end}.

279 ⟨ accept the solution 279 ⟩ ≡

```
$x$.t_ = $x$.ttrial_;
```

```
(daets::DAEpoint &) $x = x$.xtrial_;
⟨ compute condition number of $\mathbf{J}_{\mathrm{HO}}$, if requested 282 ⟩;
$x$.printData( );
$x$.state_ = daets::DAEsolution::OnPath;
$x$.stats_→countSteps(true);      /* counts the number of accepted steps */
if ($x$.t_ ≡ t_end)
    break;
```

This code is used in chunk 275.

We implement the function *CompCondJac* to compute the condition number of $\mathbf{J}_{\mathrm{HO}}$. In this function, *RCond* computes the reciprocal condition number of a matrix by routines in the LAPACK software package.

280 ⟨ Definitions of HO Private Functions 37 ⟩ +≡

```
double HO::CompCondJac( )
{
    double rcond = RCond(num_indep_tcs_, ho_jacobian_, norm_jac_);
    assert(rcond ≠ 0);
    return 1/rcond;
}
```

In the following function, *mat* is the LU factorized matrix.

281 ⟨ Auxiliary functions 107 ⟩ +≡

```
double RCond(int n, double *mat, double mat_norm)
```

```
{
    int lda = n;
    char norm = 'I';
    double *work = new double[4 * n];
    int *iwork = new int[n];
    double rcond;
    int info;
    dgecon_(&norm, &n, mat, &lda, &mat_norm, &rcond, work, iwork, &info);
    delete[ ] work;
    delete[ ] iwork;
    return rcond;
}
```

Calling the function *CompCondJac*, we store the condition number of $\mathbf{J}_{\mathrm{HO}}$ in x.*cond_jac_*.

282 ⟨ compute condition number of $\mathbf{J}_{\mathrm{HO}}$, if requested 282 ⟩ ≡

```
if (cond_flag_)
    x.cond_jac_ = ho_→CompCondJac( );
```

This code is used in chunk 279.

9.1.8.1 The stepsize selection

To predict a stepsize for the next step, we first compute σ in (8.23) and store it in

283 ⟨ declare variables for integration 218 ⟩ +≡

```
double sigma;
```

284 $\langle$ predict the stepsize for next step 284 $\rangle \equiv$

```
    sigma = comp_sigma(x.order_, x.e_, 0.16);
```

See also chunk 286.

This code is used in chunk 275.

Then, we call the following function.

285 $\langle$ Auxiliary functions 107 $\rangle +\equiv$

```
    double comp_stepsize(double sigma, double max_sigma, double h_old)
    {
        double h_new;
        if (sigma > max_sigma)
            h_new = 0.5 * max_sigma * h_old;
        else
            h_new = 0.5 * sigma * h_old;
        return h_new;
    }
```

286 $\langle$ predict the stepsize for next step 284 $\rangle +\equiv$

```
    x.htrial_ = comp_stepsize(sigma, 2.5, x.htrial_);
```

9.1.8.2 The order selection

We count the number of successful consecutive steps and store it in

287 $\langle$ declare variables for integration 218 $\rangle +\equiv$

int *consecutive_acc_count* $= 0$;

After integrating the problem for at least two successful consecutive steps with the current order, we call the function *SelectOrder* which returns

288 ⟨ declare variables for integration 218 ⟩ +≡

OrderFlag *order_flag*;

If the order needed to be changed, we would update the parameters p and q by Table 8.1. This is done by the following function.

289 ⟨ Auxiliary functions 107 ⟩ +≡

```
void update_pq(OrderFlag flag, int &p, int &q)
{
    if (flag ≡ DECREASE_ORDER)
        $p \equiv q$ ? p−− : q−−;
    if (flag ≡ INCREASE_ORDER)
        $p \equiv q$ ? q++ : p++;
    assert($p \le q$);
}
```

Thus, we

290 ⟨ determine the order for next step 290 ⟩ ≡

```
if (consecutive_acc_count ≥ 1)
{
    order_flag = SelectOrder(x.order_, sigma, weight, epq);
    if (order_flag ≠ DONT_CHANGE_ORDER)
    {
      update_pq(order_flag, p, q);
      x.order_ = p + q;
      set_pq_comp_coeffs(p, q, epq);
      consecutive_acc_count = 0;
    }
  }
  else
    consecutive_acc_count++;
```

This code is used in chunk 275.

9.1.9 Optional output

To display the current t, the number of steps, the stepsize, the error, and the order during the integration, we implement the following function.

291 ⟨ Auxiliary functions 107 ⟩ +≡

```
  void PrintProgress(double t, int no_steps, double h, double err, int order)
  {
    static char *OutputString = (char *) "␣␣␣␣t␣=␣%.4e␣␣␣step\
        s␣=␣%5d␣␣␣h␣=␣%.2e␣␣␣le␣=␣%.2e␣␣␣order␣=␣%2d";
    static char delete_space[80];
    sprintf(delete_space, OutputString, $t$, no_steps, $h$, err, order);
    fprintf(stderr, "%s", delete_space);
    for (unsigned int $i = 0$; $i <$ strlen(delete_space); $i$++)
      fputc($8$, stderr);
  }
```

292 $\langle$ optional output 292 $\rangle \equiv$

```
  if ($x$.print_progress_ $\geq 0$)
  {
    PrintProgress($x$.t_, $x$.getNumAccSteps( ), $x$.htrial_, $x$.e_, $x$.order_);
    sleep($x$.print_progress_);
  }
```

See also chunk 293.

This code is used in chunk 215.

To simplify writing output to a file, we can turn on the *one-step mode* which is an optional output feature in DAETS. In this mode, we return after each successful step and reuse the solution as an input for the integrator function in the next call.

293 $\langle$ optional output 292 $\rangle$ +$\equiv$

```
  if ($x$.onestep_mode_)
  {
    $x$.state_ = daets::DAEsolution::OnPath;
    state = daets::success;
    x.stats_→stopTimer( );
    return;
  }
```

9.2 The function integrate

In this section, we implement the function *integrate* which advances a numerical solution of a DAE of the form (1.1) from an initial time up to a final time. In this function, x contains an initial point and will be updated by the solution at t_{end}. If the function cannot reach t_{end}, the solution at some $t < t_{\text{end}}$ is returned.

294 ⟨ Definitions of StiffDAEsolver Public Functions 254 ⟩ +≡

```
  void StiffDAEsolver::integrate(daets::DAEsolution &x, double t_end,
                daets::SolverExitFlag &state) throw(std::logic_error)
  {
    ⟨ set order 298 ⟩;
    ⟨ integrate by explicit TS method on the first step 300 ⟩;
    ⟨ integrate by HO method up to the final time 301 ⟩;
  }
```

A user can specify the minimum and maximum orders for the HO method by calling the function *SetMinMaxOrder*. This function stores the orders in

295 ⟨ StiffDAEsolver Data Members 192 ⟩ +≡

```
int user_min_order_, user_max_order_;
```

296 ⟨Definitions of StiffDAEsolver Public Functions 254⟩ +≡

```
void StiffDAEsolver::SetMinMaxOrder(int min, int max)
{
  user_min_order_ = min;
  user_max_order_ = max;
}
```

To obtain the user specified orders, we implement the function *GetMinMaxOrder*. If the user does not call *SetMinMaxOrder* function, we will consider the default values, 1 and 20.

297 ⟨Definitions of StiffDAEsolver Private Functions 194⟩ +≡

```
void StiffDAEsolver::GetMinMaxOrder()
{
  if (user_min_order_ ≡ 0)
  {
    min_order_ = 1;
    max_order_ = 20;
  }
  else
  {
    min_order_ = user_min_order_;
    max_order_ = user_max_order_;
    }
  }
```

298 $\langle$ set order 298 $\rangle \equiv$

```
  GetMinMaxOrder( );
```

See also chunk 299.

This code is used in chunk 294.

We start the integration with the minimum order.

299 $\langle$ set order 298 $\rangle +\equiv$

```
  x.order_ = min_order_;
```

We call the function *IntegrateByExplicitTS* to use the explicit Taylor series method on the first step.

300 $\langle$ integrate by explicit TS method on the first step 300 $\rangle \equiv$

```
  IntegrateByExplicitTS(x, t_end, 1, state);      /* terminate if the stepsize is too small */
  if (state ≡ daets::htoosmall)
    return;
  if (x.t_ ≡ t_end)      /* checks if the final time has reached */
  {
    state = daets::success;
    x.stats_→stopTimer( );
    return;
  }
```

This code is used in chunk 294.

Then, we call the function *IntegrateByHO* to continue the integration using the HO method up to the final time.

301 ⟨integrate by HO method up to the final time 301⟩ ≡

```
IntegrateByHO(x, t_end, state);
    /* terminate the integration if the stepsize is too small */
if (state ≡ daets::htoosmall)
  return;
state = daets::success;
x.stats_→stopTimer();
```

This code is used in chunk 294.

Chapter 10

Numerical results

In this chapter, we start with an example showing a basic integration with DAETS, §10.1. Then in §10.2, we show results from solving several test problems.

10.1 Basic usage

To integrate a DAE problem using DAETS, a user should specify the problem and provide a main program.

10.1.1 Problem definition

A DAE must be specified by a template function. As an example consider the simple pendulum in Example 2.1. The following function evaluates (2.4). Here, the *Diff* operator performs the differentiation of a variable with respect to t. That is, *Diff* $(x[j], l)$ results in $x_j^{(l)}$. The input $x[0], x[1], x[2]$ store the state variables x, y, λ, respectively, and the output

$f[0]$, $f[1]$, $f[2]$ store the evaluated f, g, h, respectively.

304 ⟨ Pendulum 304 ⟩ ≡

```
template⟨typename T⟩
void fcn(T t, const T *x, T *f, void *param)
{
    const double G = 9.8, L = 10.0;
    f[0] = Diff(x[0], 2) + x[0] * x[2];
    f[1] = Diff(x[1], 2) + x[1] * x[2] − G;
    f[2] = sqr(x[0]) + sqr(x[1]) − sqr(L);
}
```

10.1.2 Main program

We implement a main program for this problem and explain its parts.

305 ⟨ solve simple pendulum 305 ⟩ ≡

```
int main(int argc, char *argv[ ])
{
    ⟨ set size of DAE and integration interval 306 ⟩;
    ⟨ create a solver 307 ⟩;
    ⟨ set order and tolerance 308 ⟩;
    ⟨ create a DAEsolution object 309 ⟩;
    ⟨ set initial values 310 ⟩;
    ⟨ integrate the problem 311 ⟩;
```

$\langle$ output results 312 $\rangle$;

return 0;

}

First we set the DAE size and integration interval.

306 $\langle$ set size of DAE and integration interval 306 $\rangle \equiv$

int $n = 3$;

double *t0* $= 0$, *tend* $= 100$;

This code is used in chunk 305.

Then, we construct an object *solver* of the class **StiffDAEsolver**, where we pass the size of the problem and the function *fcn*. A predefined macro `STIFF_DAE_FCN` is used to simplify a call to the constructor of the class **StiffDAEsolver** (see Appendix D.3).

307 $\langle$ create a solver 307 $\rangle \equiv$

sdaets::**StiffDAEsolver** *solver*(n, `STIFF_DAE_FCN`(*fcn*));

This code is used in chunks 305, 372, 377, 382, 386, and 390

We intend to investigate the accuracy of the numerical solutions computed by our HO method with different orders over a range of tolerances. We obtain minimum and maximum orders, and the exponent r for tolerance 10^{-r} from the command prompt and set them.

308 $\langle$ set order and tolerance 308 $\rangle \equiv$

int *min_order* $=$ *atoi*(*argv*[1]);

int *max_order* $=$ *atoi*(*argv*[2]);

int *exp* $= atoi(argv[3]);$

double *tol* $=$ **std**$::pow(10, -exp);$

$solver.setTol(tol);$

$solver.SetMinMaxOrder(min\_order, max\_order);$

This code is used in chunks 305, 372, 377, 382, 386, 390, and 394

Another key object for problem solution is a **DAEsolution** object which may be viewed as a point moving along the solution path.

309 ⟨create a **DAEsolution** object 309⟩ ≡

daets::**DAEsolution** $x(solver);$

This code is used in chunks 305, 372, 377, 382, 386, 390, and 394

We set the initial values for the problem by calling the functions *setT* and *setX* from the class **DAEsolution**. The function *setT* initializes the independent variable, and $setX(j, l, a)$ initializes $x_j^{(l)} = a$.

310 ⟨set initial values 310⟩ ≡

$x.setT(t0)$

$.setX(0, 0, -10.0).setX(1, 0, 0.0)$ /* sets x , y */

$.setX(0, 1, 0.0).setX(1, 1, 1.0);$ /* sets x' , y' */

This code is used in chunk 305.

The integration is performed by the call to the function *integrate* implemented in §9.2. If the **SolverExitFlag** is *success*, the **DAEsolution** object x contains solution values at *tend*; otherwise, x contains solution values at the reached t between *t0* and *tend*.

311 ⟨integrate the problem 311⟩ ≡

```
daets::SolverExitFlag flag;
solver.integrate(x, tend, flag);
if (flag ≠ daets::success)
    daets::printSolverExitFlag(flag);
```

This code is used in chunks 305, 372, 377, 382, 386, 390, and 394

Denote the ith component of a reference solution at $tend$ by r_i and the ith component of a computed solution at $tend$ by x_i. We first estimate the relative error in x_i by $|x_i - r_i| / |r_i|$. Then, the minimum number of correct digits in a numerical solution at *tend*, denoted by *significant correct digits* (SCD), is

$$\text{SCD} = -\log_{10} \|\text{relative error at the end of integration interval}\|_\infty.$$

We also need the CPU time, and the number of accepted and rejected steps. These values are obtained by calling the functions *getCPUtime*, *getNumAccSteps* and *getNumRejSteps*, respectively. For all examples, we output the above results in *table.dat* which is in a format suitable for gnuplot to produce plots.

312 ⟨output results 312⟩ ≡

```
ofstream plot_out("table.dat", ios::app);
plot_out ≪ exp ≪ "\t" ≪ min_order ≪ "\t" ≪ max_order ≪
    "\t" ≪ CompSCD(x) ≪ "\t" ≪ x.getCPUtime() ≪ "\t" ≪
    x.getNumAccSteps() + x.getNumRejSteps() ≪ endl;
plot_out.close();
```

This code is used in chunks 305, 372, 377, 382, 386, 390, and 394

10.2 Numerical experiments

In §10.2.1, we consider several test problems to examine the capacity of our code for solving stiff ODEs and DAEs. We perform experiments in which the code is run repeatedly on each of the problems with tolerances

$$\text{atol} = \text{rtol} = 10^{-r}, \quad r = 1, \dots, 13. \tag{10.1}$$

First, we study the accuracy of the computed solutions on test problems in §10.2.2. Then, we examine the efficiency of our code on these problems in §10.2.3. Finally, we compare the performance of the variable-order version of the code against the fixed-order version in §10.2.4.

All numerical results are produced on an Intel(R) Core(TM) i7-6700HQ, CPU 2.60GHz with Ubuntu 18.04.1 LTS, 16 GB RAM, and 64KB L1, 256 KB L2, and 6144 KB L3 cache.

10.2.1 Test problems

In this section, we describe 4 stiff problems from the Test Set for IVP Solvers [37], namely, Van der Pol oscillator, Oregonator, Chemical Akzo Nobel, and Car Axis. In addition, we describe an artificial stiff index-2 DAE and a Multi Pendula problem. We present template functions for evaluating these ODEs or DAEs. The main programs for integrating these problems are in Appendix D.7.

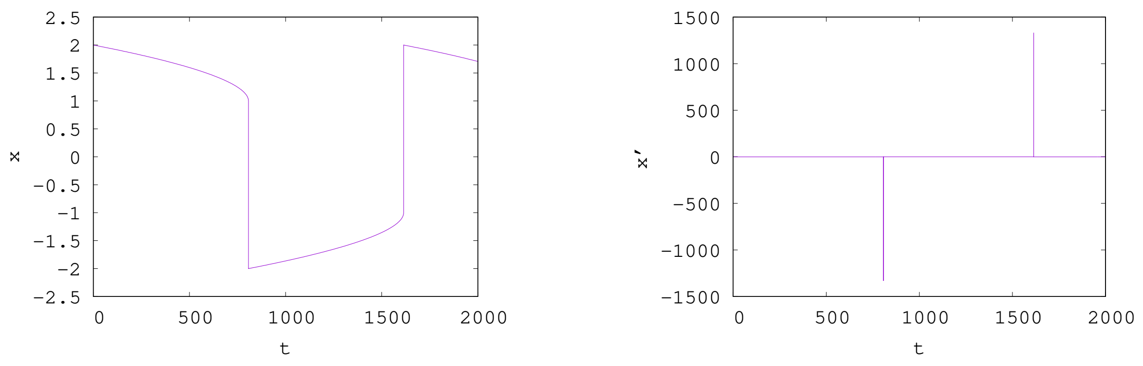

Figure 10.1: Van der Pol, plots of x and x' versus t.

10.2.1.1 Van der Pol oscillator

To illustrate how our solver performs on a problem which repeatedly changes character during the integration interval, we consider the well-known Van der Pol oscillator [37],

$$x'' - \mu(1 - x^2)x' + x = 0, \qquad x(0) = 2,\ x'(0) = 0, \tag{10.2}$$

with parameter $\mu = 1000$ over the interval $[0, 2000]$. Plots of the solution components to this problem are shown in Figure 10.4.

The following template function evaluates (10.2).

316 $\langle$ Van der Pol 316 $\rangle \equiv$

template$\langle$**typename T**$\rangle$

void *fcn*(**T** t, **const T** $*x$, **T** $*f$, **void** $*$*param*)

{

  double *mu* $= 1 \cdot 10^3$;

  $f[0] =$ *Diff*$(x[0], 2) -$ *mu* $* (1 -$ *sqr*$(x[0])) *$ *Diff*$(x[0], 1) + x[0]$;

```
}
```

This code is used in chunk 375.

10.2.1.2 Oregonator

The Oregonator system [37] is a chemical model with a periodic solution describing the Belousov–Zhabotinskii reaction. It is presented in the form of the following ODEs

$$\begin{aligned} x' &= 77.27\Big(y + x(1 - 8.375 \cdot 10^{-6}x - y)\Big), \\ y' &= \Big(z - y(1+x)\Big)/77.27, \\ z' &= 0.161(x - z), \end{aligned} \tag{10.3}$$

over the interval $[0, 360]$. Plots of the solution components to this problem are shown in Figure 10.4.

The following template function evaluates (10.3).

319 ⟨ Oregonator 319 ⟩ ≡

```
template⟨typename T⟩
void fcn(T t, const T *x, T *f, void *param)
{
  double s = 77.27, w = 0.161, q = 8.375 · 10^{-6};
  f[0] = Diff(x[0], 1) − s ∗ (x[1] − x[0] ∗ x[1] + x[0] − q ∗ sqr(x[0]));
  f[1] = Diff(x[1], 1) − (1/s) ∗ (−x[1] − x[0] ∗ x[1] + x[2]);
  f[2] = Diff(x[2], 1) − w ∗ (x[0] − x[2]);
}
```

This code is used in chunk 380.

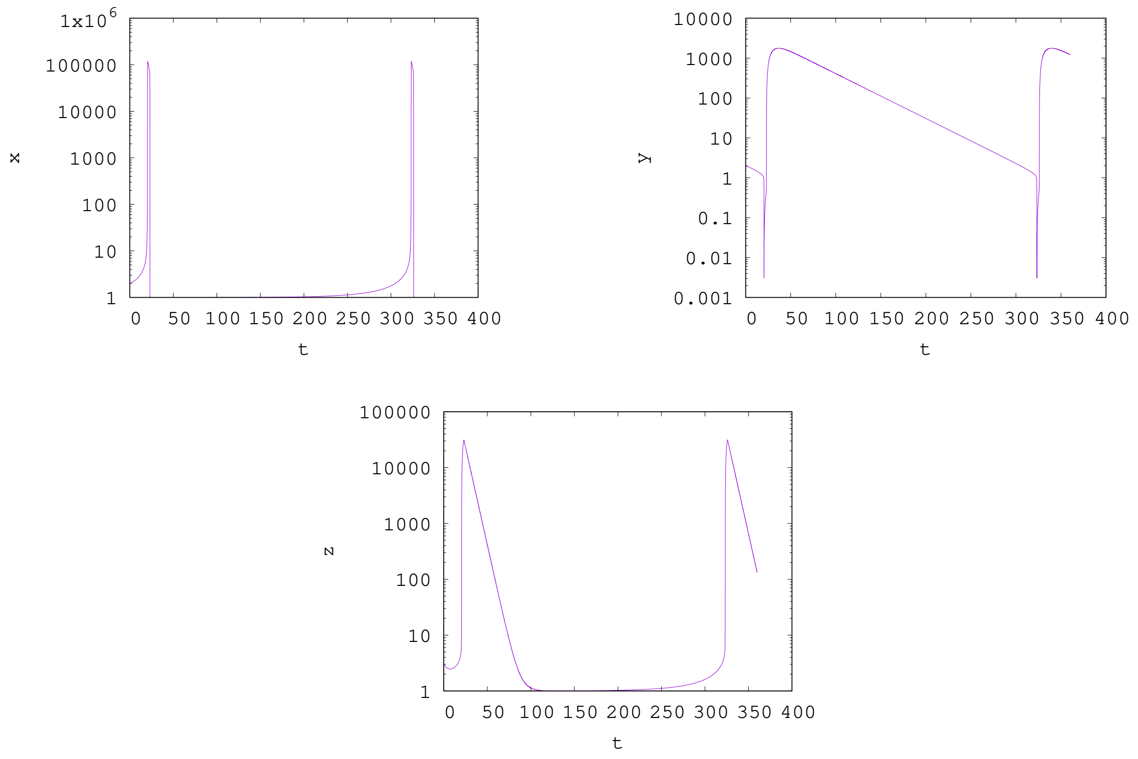

Figure 10.2: Oregonator, plots of x, y and z versus t.

10.2.1.3 Chemical Akzo Nobel

This is a non-quasilinear DAE of index 1 of the form

$$\mathbf{K}\mathbf{x}' = \mathbf{g}(\mathbf{x}), \tag{10.4}$$

with $\mathbf{x} \in \mathbb{R}^6$ over the interval $[0, 180]$. It describes a chemical process, in which two species are mixed, while carbon dioxide is continuously added, to produce a product. The defining equations are in [37]. Plots of the solution components to this problem are shown in Figure 10.3. The following template function defines (10.4).

320 ⟨Chemical Akzo Nobel 320⟩ ≡

```
template⟨typename T⟩
void fcn(T t, const T *x, T *f, void *param)
{
  double k1 = 18.7, k2 = 0.58, k3 = 0.09, k4 = 0.42, kbig = 34.4, kla = 3.3,
      ks = 115.83, po2 = 0.9, hen = 737;
  T r1 = k1 * (pow(x[0], 4)) * sqrt(x[1]),
      r2 = k2 * x[2] * x[3],
      r3 = k2/kbig * x[0] * x[4],
      r4 = k3 * x[0] * (sqr(x[3])),
      r5 = k4 * (sqr(x[5])) * sqrt(x[1]),
      fin = kla * (po2/hen - x[1]);
  f[0] = -Diff(x[0], 1) - 2 * r1 + r2 - r3 - r4;
  f[1] = -Diff(x[1], 1) - 0.5 * r1 - r4 - 0.5 * r5 + fin;
```

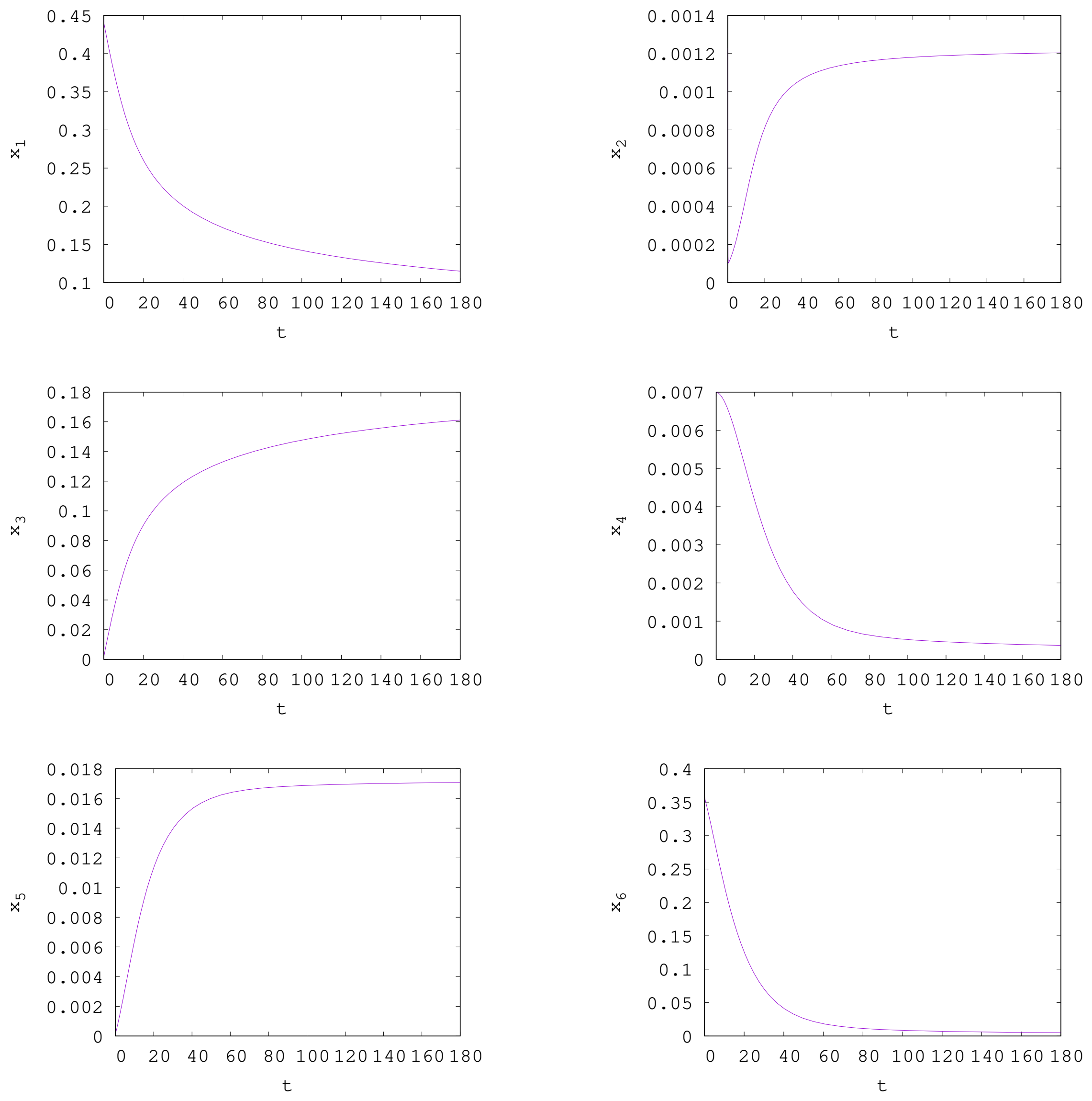

Figure 10.3: Chemical Akzo Nobel, plots of $x_1, \ldots, x_6$ versus t.

$f[2] = -\textit{Diff}\,(x[2], 1) + \textit{r1} - \textit{r2} + \textit{r3};$

$f[3] = -\textit{Diff}\,(x[3], 1) - \textit{r2} + \textit{r3} - 2 * \textit{r4};$

$f[4] = -\textit{Diff}\,(x[4], 1) + \textit{r2} - \textit{r3} + \textit{r5};$

$f[5] = \textit{ks} * x[0] * x[3] - x[5];$

}

This code is used in chunk 385.

10.2.1.4 A highly stiff index-2 DAE

From the Van der Pol problem (10.2), we construct an artificial stiff index-2 DAE,

$$\begin{aligned} x'' - \mu(1 - x^2)x' + x &= 0, \\ xy' - z &= 0, \\ x^2 - y^2 + 5 &= 0, \end{aligned} \tag{10.5}$$

where $\mu = 1000$ and $t \in [0, 2000]$. Plots of the solution components to this problem are shown in Figure 10.4.

The following template function evaluates (10.5).

322 ⟨ Stiff index-2 322 ⟩ ≡

template⟨**typename T**⟩

void *fcn*(**T** t, **const T** $*x$, **T** $*f$, **void** $*$*param*)

{

double *mu* $= 1 \cdot 10^3;$

$f[0] = \textit{Diff}\,(x[0], 2) - \textit{mu} * (1 + (-\textit{sqr}(x[0]))) * \textit{Diff}\,(x[0], 1) + x[0];$

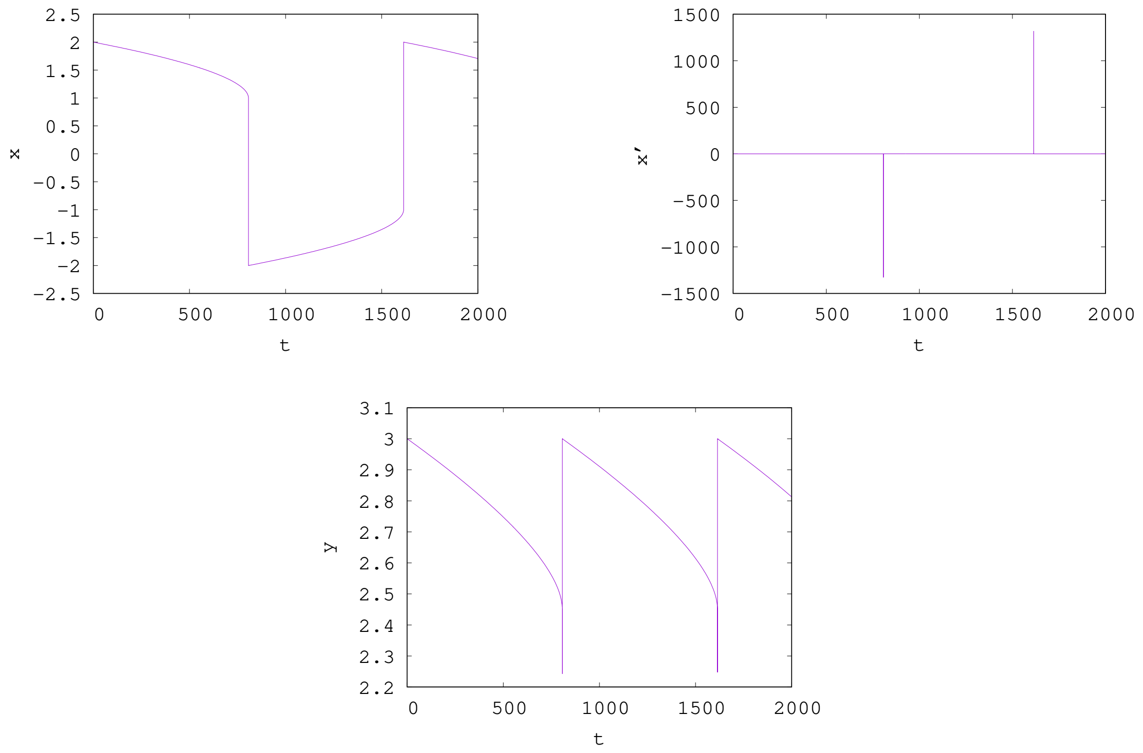

Figure 10.4: Index-2 from Van der Pol, plots of x, x' and y versus t.

```
    f[1] = Diff(x[1], 1) * x[0] - x[2];

    f[2] = sqr(x[0]) - sqr(x[1]) + 5;

  }
```

This code is used in chunk 389.

10.2.1.5 Car Axis

A simple model of a car axis going over a bumpy road is [37]

$$\mathbf{K}\mathbf{p}'' = \mathbf{g}(t, \mathbf{p}, \boldsymbol{\lambda}),$$

$$0 = \boldsymbol{\phi}(t, \mathbf{p}),$$

with $\mathbf{p}, \mathbf{g}$ of dimension 4, and $\boldsymbol{\lambda}, \boldsymbol{\phi}$ of dimension 2. Here, $\mathbf{p} = (x_l, y_l, x_r, y_r)^T$, and (x_l, x_r) and (y_l, y_r) are the coordinates of the left and right wheels, respectively; $\boldsymbol{\lambda} = (\lambda_1, \lambda_2)^T$ are Lagrange multipliers. This problem is a moderately stiff DAE of index 3. Figure 10.5 shows the solutions x_l, y_l, x_r, and y_r.

The following template function defines the Car Axis DAE.

323 ⟨ Car Axis 323 ⟩ ≡

```
#define xl   x[0]

#define yl   x[1]

#define xr   x[2]

#define yr   x[3]

#define lam1   x[4]

#define lam2   x[5]
```

template$\langle$**typename T**$\rangle$

void *fcn*(**T** t, **const T** $*x$, **T** $*f$, **void** $*param$)

{

double $eps = 1 \cdot 10^{-2},\ M = 10.0,\ epsM = sqr(eps) * M/2,\ L = 1.0,\ \texttt{L0} = 0.5,$

$W = 10.0,\ R = 0.1;$

T $yb = R * sin(W * t),$

$xb = sqrt(sqr(L) - sqr(yb)),$

$Ll = sqrt(sqr(xl) + sqr(yl)),$

$Lr = sqrt(sqr(xr - xb) + sqr(yr - yb));$

$f[0] = -epsM * Diff(xl, 2) + (\texttt{L0} - Ll) * xl/Ll + lam1 * xb + 2.0 * lam2 * (xl - xr);$

$f[1] = -epsM * Diff(yl, 2) + (\texttt{L0} - Ll) * yl/Ll + lam1 * yb + 2.0 * lam2 * (yl - yr) - epsM;$

$f[2] = -epsM * Diff(xr, 2) + (\texttt{L0} - Lr) * (xr - xb)/Lr - 2.0 * lam2 * (xl - xr);$

$f[3] = -epsM * Diff(yr, 2) + (\texttt{L0} - Lr) * (yr - yb)/Lr - 2.0 * lam2 * (yl - yr) - epsM;$

$f[4] = xl * xb + yl * yb;$

$f[5] = sqr(xl - xr) + sqr(yl - yr) - sqr(L);$

}

This code is used in chunk 393.

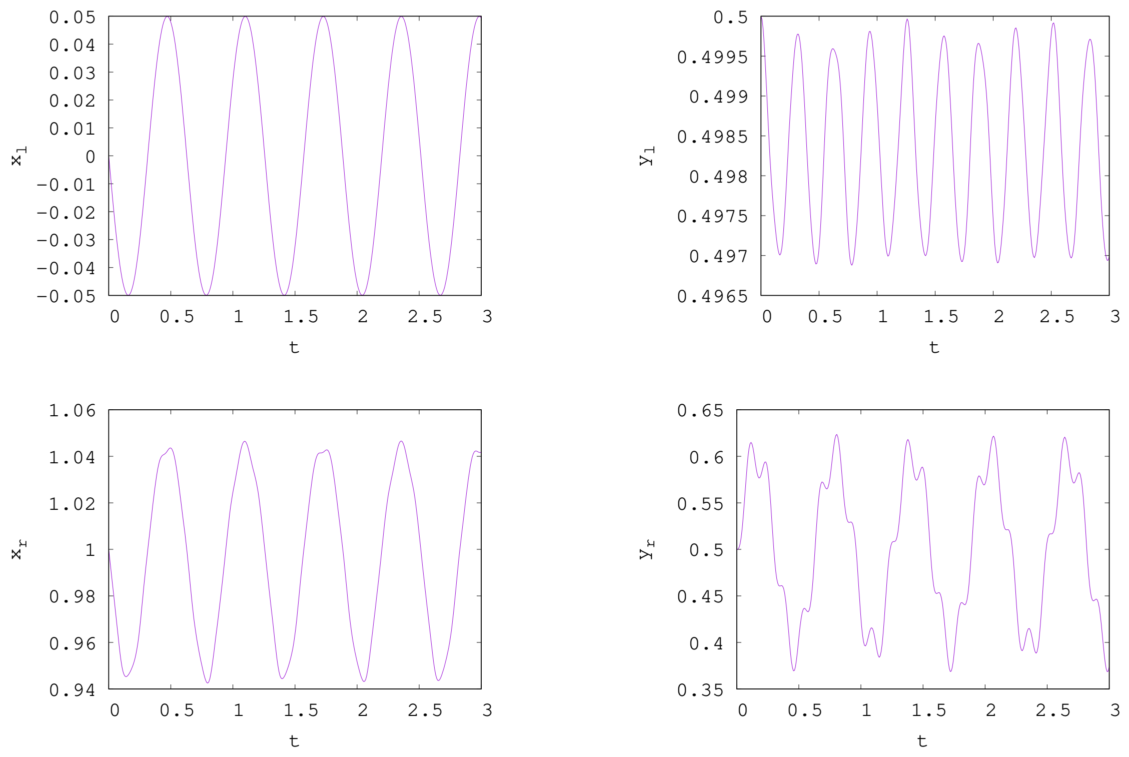

Figure 10.5: Car axis, plots of x_l, y_l, x_r and y_r versus t.

10.2.1.6 Multi Pendula

To illustrate how our code can handle higher index DAEs, we consider the DAE problem consisting of P pendula [47]

$$\begin{aligned}
0 &= x_1'' + \lambda_1 x_1, \\
0 &= y_1'' + \lambda_1 y_1 - G, \\
0 &= x_1^2 + y_1^2 - L^2, \\
0 &= x_i'' + \lambda_i x_i, \\
0 &= y_i'' + \lambda_i y_i - G, \qquad\qquad i = 2, 3, \dots, P, \\
0 &= x_i^2 + y_i^2 - (L + c\lambda_{i-1})^2,
\end{aligned} \tag{10.6}$$

where G, L and c are given constants. Here, the first pendulum is undriven, and pendulum $i-1$ exerts a driving effect on pendulum i for $i = 2, 3, \dots, P$. This DAE is of size $3P$ and index $2P+1$ [47].

The following template function evaluates (10.6).

326 $\langle$ Multi Pendula 326 $\rangle \equiv$

```
template⟨typename T⟩
void fcn(T t, const T *x, T *f, void *parameters)
{
   double *constants = (double *) parameters,
        G = *constants,
        L = *(constants + 1),
        c = *(constants + 2);
```

```
    int $P = *(constants + 3)$;
    $f[0] = Diff(x[0], 2) + x[0] * x[2]$;
    $f[1] = Diff(x[1], 2) + x[1] * x[2] - G$;
    $f[2] = sqr(x[0]) + sqr(x[1]) - sqr(L)$;
    for (int $i = 1$; $i < P$; $i$++)
    {
      $f[3 * i] = Diff(x[3 * i], 2) + x[3 * i] * x[3 * i + 2]$;
      $f[3 * i + 1] = Diff(x[3 * i + 1], 2) + x[3 * i + 1] * x[3 * i + 2] - G$;
      $f[3 * i + 2] = sqr(x[3 * i]) + sqr(x[3 * i + 1]) - sqr(L + c * x[3 * i - 1])$;
    }
  }
```

See also chunk 396.

This code is used in chunk 398.

Here, we consider 8 pendula and set $G = 9.8$, $L = 10$ and $c = 0.1$. The index of the DAE is 17. Plots of the computed $x_1, \dots, x_8$ and $y_1, \dots, y_8$ versus t are shown in Figures 10.6 and 10.7.

10.2.2 Accuracy

We have computed the SCD of numerical solutions with tolerances (10.1). For problems from the Test Set for IVP Solvers, we determine SCD using the reference solutions given in [37]. For other problems, we determine SCD using the reference solutions computed by our code with atol = rtol = 10^{-14}. Plots of SCD versus tolerance for the above test problems

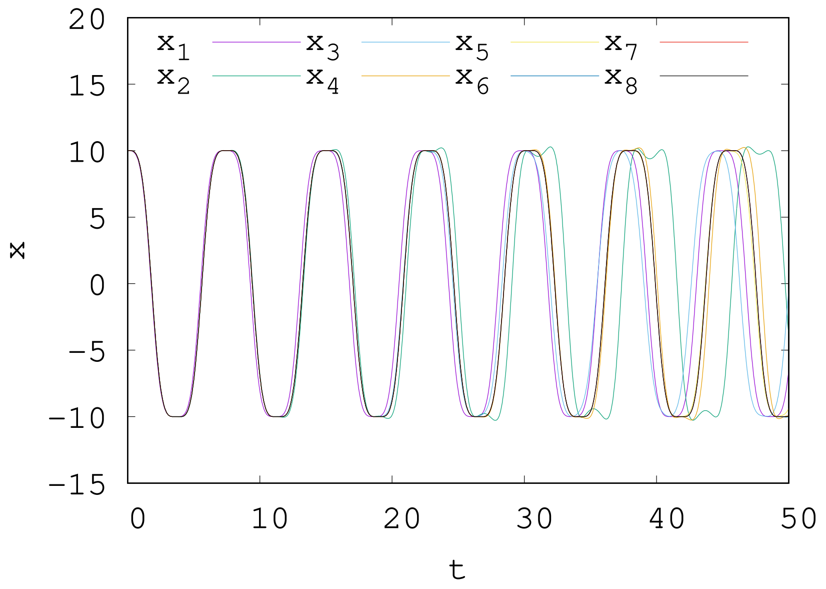

Figure 10.6: Multi Pendula, index-17, plots of $x_1, \ldots, x_8$ versus t.

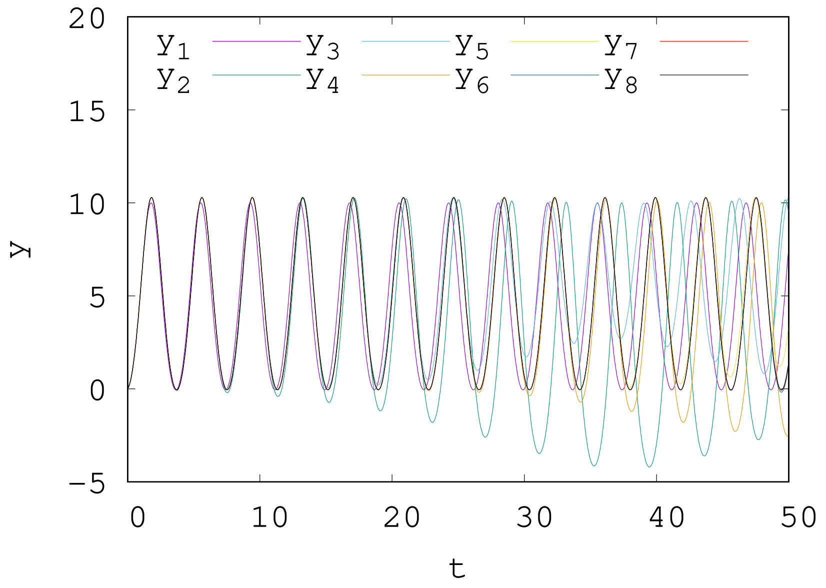

Figure 10.7: Multi Pendula, index-17, plots of $y_1, \ldots, y_8$ versus t.

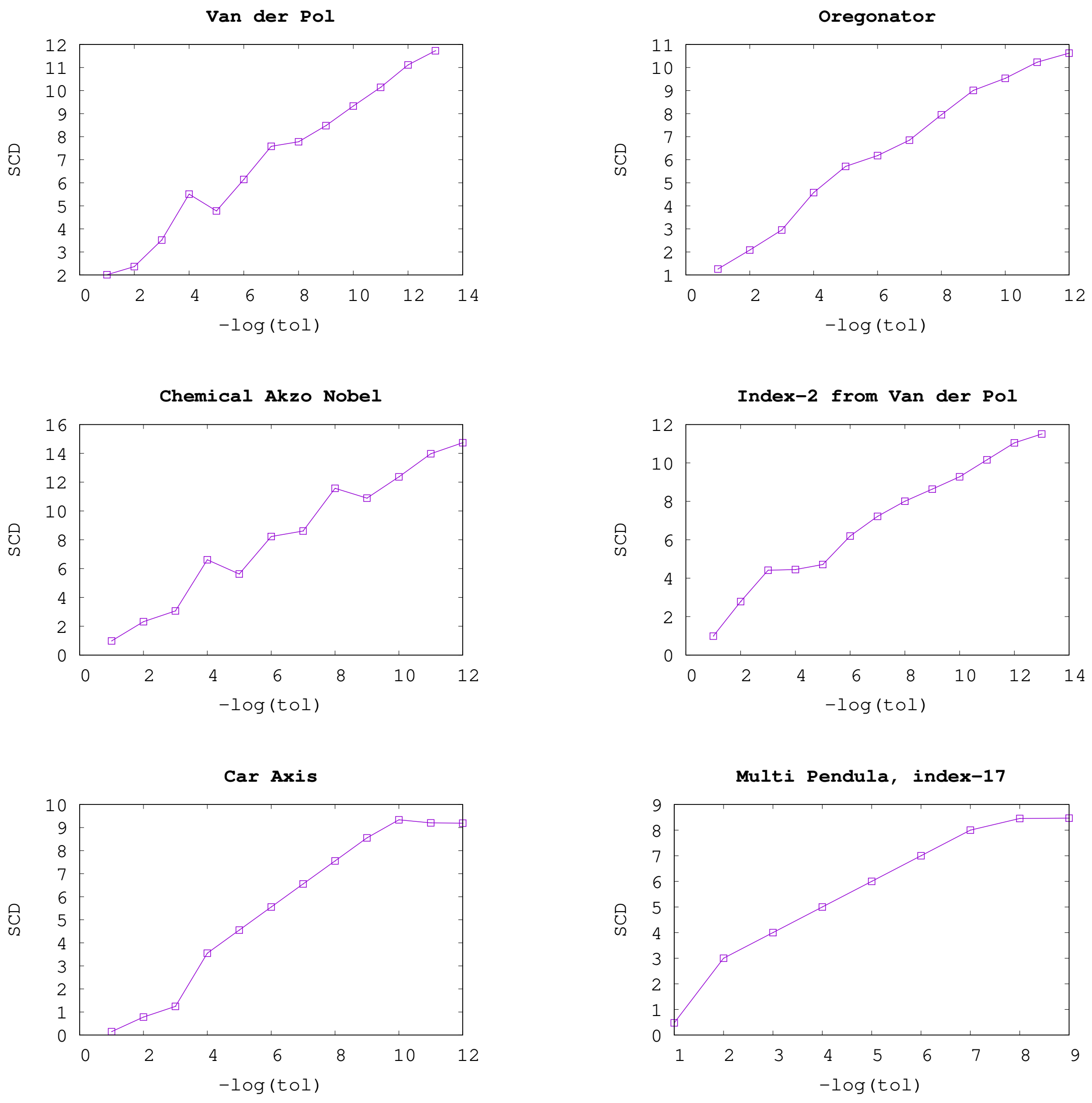

Figure 10.8: Accuracy diagrams.

are displayed in Figure 10.8. The output data confirm the success of the HO method for the accurate numerical solutions of these test problems. Also, our experiments show that the projection (4.21) maintains the accuracy of the HO method.

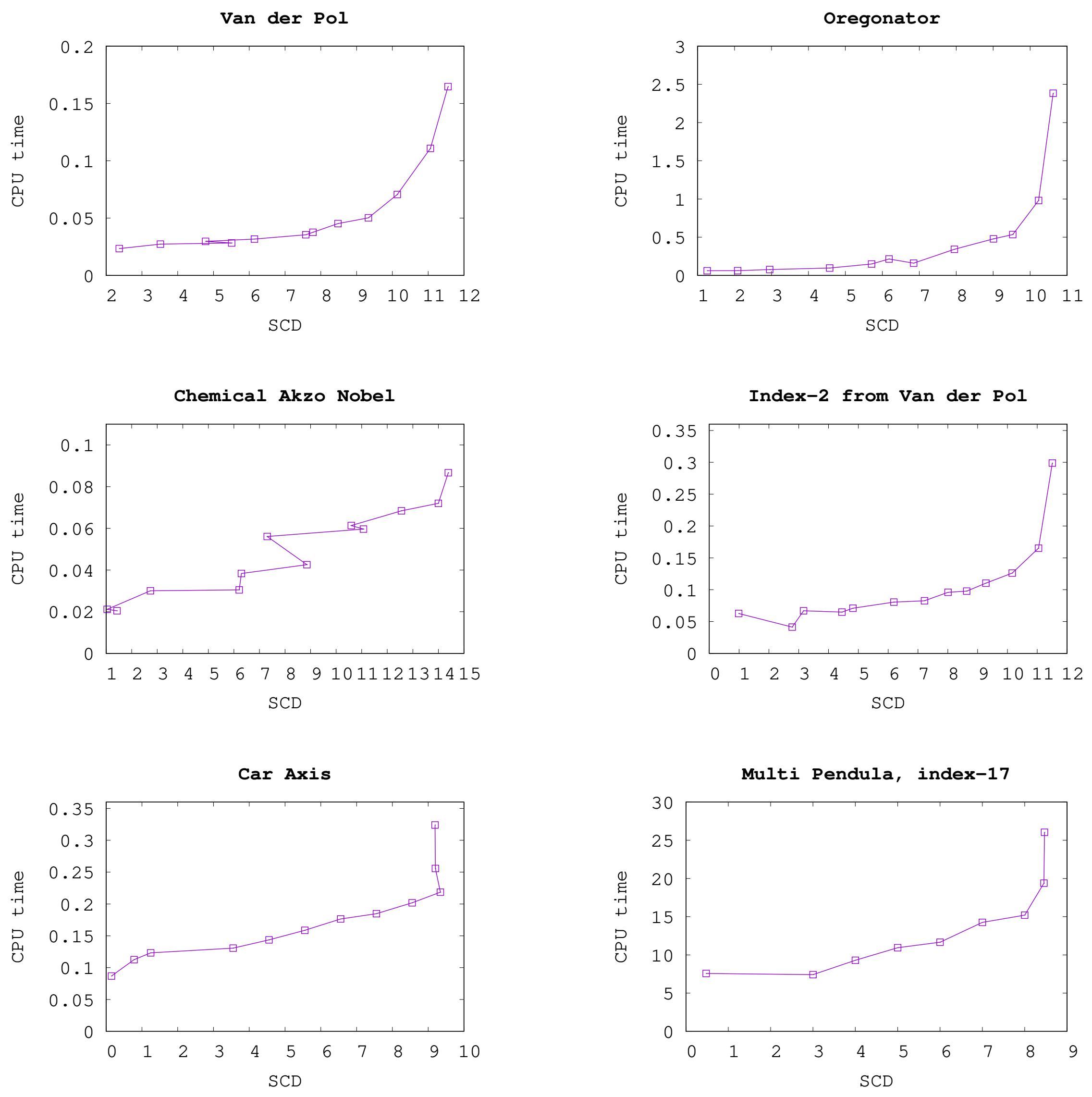

Figure 10.9: Work precision diagrams.

10.2.3 Efficiency

We plot CPU time versus SCD for each problem in Figure 10.9.

10.2.4 Variable-order versus fixed-order

Plots in Figure 10.10 show how $p+q$ in the (p,q) HO method changes during the integration interval for the test problems running the code with atol $=$ rtol $= 10^{-8}$, $min\_order = 1$ and $max\_order = 20$. We also show the work-precision diagrams for the above problems using variable-order and fixed-order strategies in Figures 10.11 to 10.15.

In Figure 10.10 for Van der Pol, Oregonator, and the DAE (10.5), we see that during the times when the solution is changing rapidly, the order is increased, and when it is changing more slowly, the code selects low orders. From the Figures 10.11, 10.12 and 10.14, we observe that the variable-order scheme gives the best performance for these problems.

In Figure 10.10 for Chemical Akzo Nobel and Car Axis, we see that the order is increased up to a high-order and is barely changed after that. Figures 10.13 and 10.15 show that our order selection scheme works reasonably robustly for these problems. A fixed-order strategy is useful when we know in advance a good order for a particular problem and precision, but the variable-order strategy works for all problems.

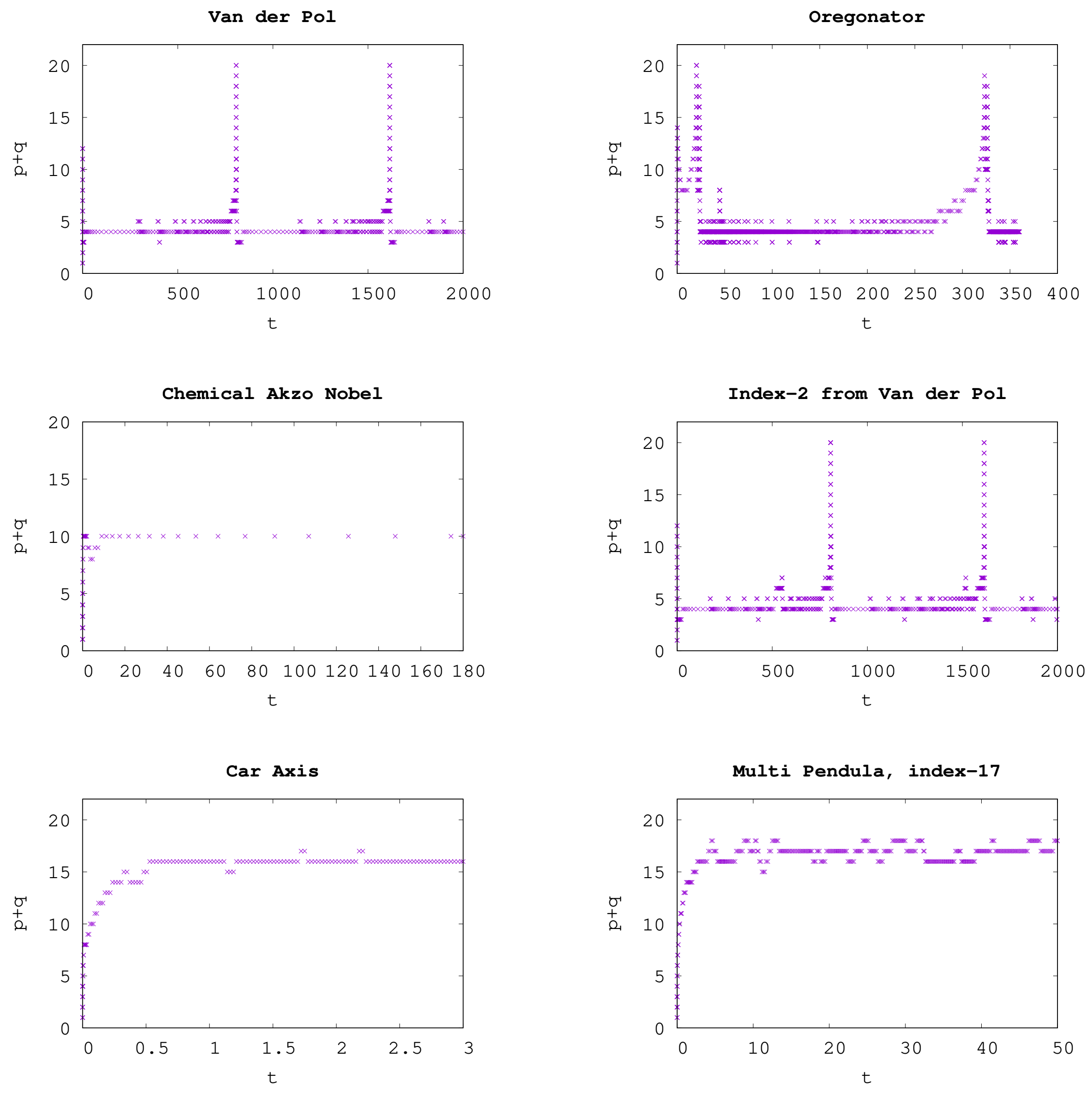

Figure 10.10: $p+q$ during the integration interval with tol $= 10^{-8}$.

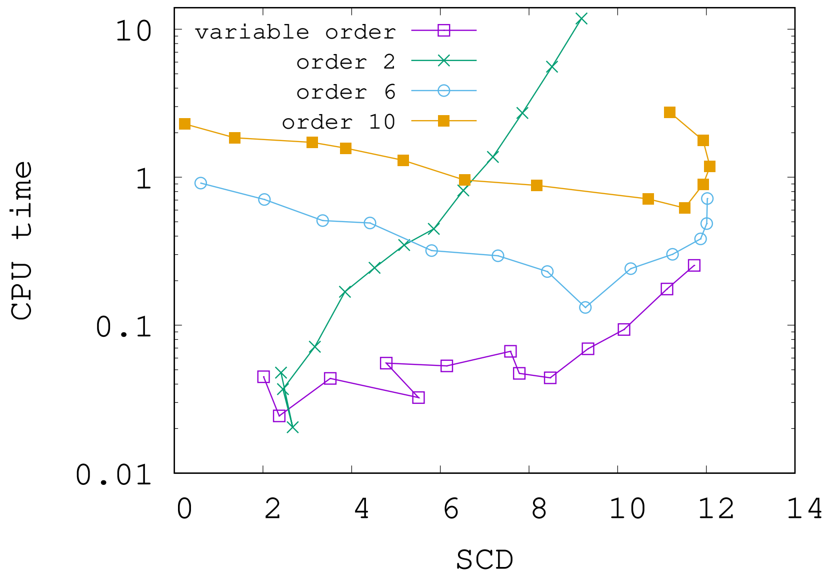

Figure 10.11: Van der Pol, variable-order versus fixed-order.

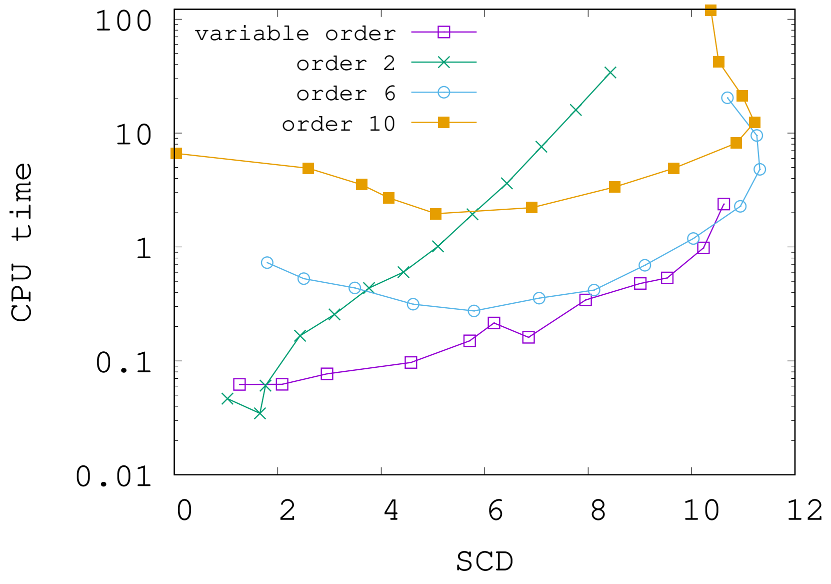

Figure 10.12: Oregonator, variable-order versus fixed-order.

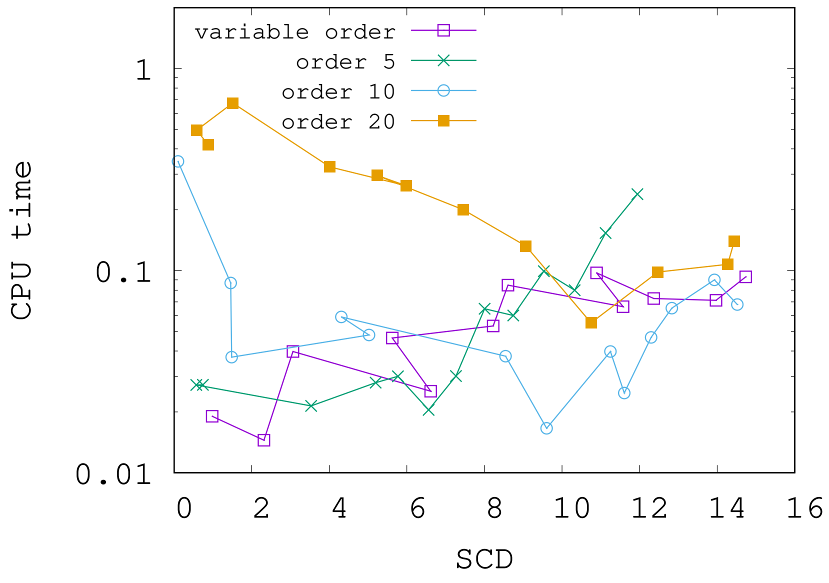

Figure 10.13: Chemical Akzo Nobel, variable-order versus fixed-order.

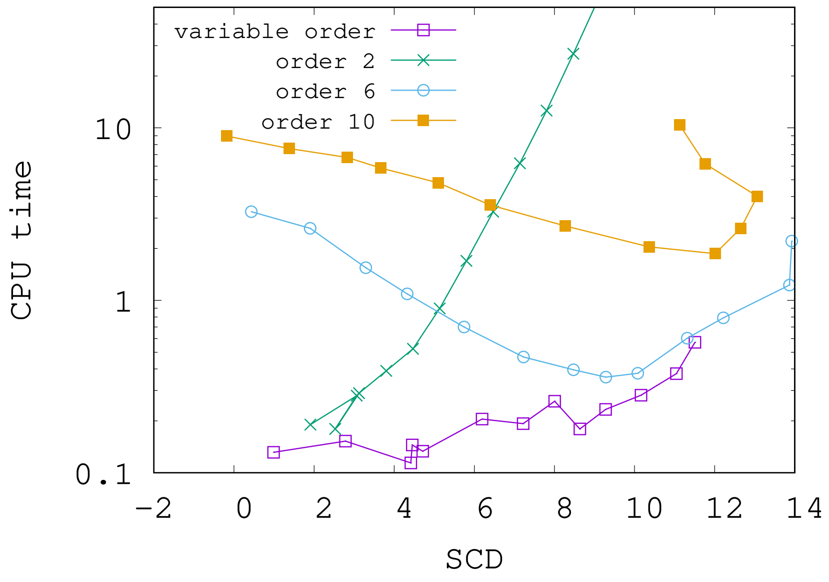

Figure 10.14: Index-2 from Van der Pol, variable-order versus fixed-order.

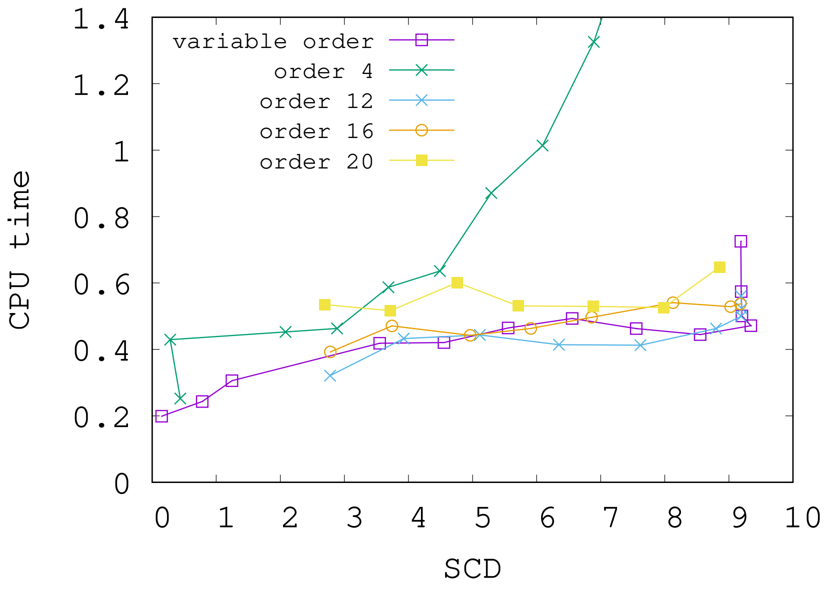

Figure 10.15: Car Axis, variable-order versus fixed-order.

Chapter 11

Conclusions

In this thesis, we have developed and implemented an implicit Hermite-Obreschkoff (HO) method for numerical solution of a stiff DAE. We employ Pryce's structural analysis to determine the constraints of the problem and to organize the computations of required Taylor coefficients (TCs) and their gradients. Then, we use automatic differentiation to compute these TCs and gradients and form the residual vector and Jacobian matrix required for Newton's iteration.

Given a general DAE described by a computer program, the structural analysis data are obtained via operator overloading. Hence, a simulation software that automatically converts a model to a DAE need not to produce it in a particular (first-order or lower-index) form.

The relation between Taylor coefficients of the kth derivative of a sufficiently differentiable function at two points can be determined by the HO formula. We developed our HO method using this formula for some derivatives of state variables of the DAE, determined from the structural analysis data. The method can be A- or L- stable and can handle high-index DAEs.

We defined the Hermite-Nordsieck vector for a sufficiently differentiable function at a point. Constructing Hermite-Nordsieck vectors for solution components at each step, we find an initial guess for the solution, required for Newton's iteration, and estimate the discretization error of the HO method with different orders. As a result, we designed an adaptive variable-stepsize and variable-order algorithm for integrating a DAE problem. We have implemented our algorithm in C++ using literate programming.

We considered several test problems to examine the ability of our code to solve stiff ODEs and DAEs. We performed experiments in which the code was run repeatedly on each of the problems with different tolerances. The output data confirm the success of the HO method for the accurate numerical solutions of the test problems. Also, we compared the performance of the variable-order version of the code against the fixed-order version. We observed that the variable-order scheme gives better performance.

It is clear that our numerical method is expensive in that it requires repeated evaluations of TCs and their gradients. Generally, on problems for which a solver based on a Runge-Kutta or a multistep method is already very efficient, our code may not be competitive. However, it will be competitive on problems that current methods cannot handle because of high index, or if high accuracy is required.

Several related investigations follow naturally from this thesis. A scheme can be developed for automatically determining whether a DAE can be solved more efficiently using the explicit Taylor series method or the HO method. Switching between these methods can be more efficient than using one of the methods alone for problems which are non-stiff in some regions of the integration interval and stiff in other regions. The stiffness detection and switching strategies will be addressed in a future work.

The Jacobian matrix for Newton's iterations is usually dense, and we perform the LU factorization to compute the solution of the corresponding linear systems. Hence, our code is not efficient for large DAE problems. Alternatively, we can use an iterative (Krylov) method to solve the linear systems. Since the Jacobian matrix is ill-conditioned in stiff problems, the convergence of any Krylov-based algorithm is slow. Developing an efficient preconditioner is essential to making the convergence of Newton-Krylov iterations sufficiently fast. Construction of such preconditioner is left for future research.

Appendix A

The integrator function in DAETS

As explained in Chapter 9, we first integrate a given DAE using the explicit Taylor series method on the first step. The following function is an adapted version of the function *integrate* in DAETS implemented by Nedialkov and Pryce. The algorithm of this function is described in [47].

336 ⟨Definitions of StiffDAEsolver Private Functions 194⟩ +≡

```
void StiffDAEsolver::IntegrateByExplicitTS(daets::DAEsolution &x, double
        t_end, unsigned int num_steps, daets::SolverExitFlag &state)
{
  daets::DAEpoint ts(*this);
  daets::DAEpoint ts_saved(ts);
  double errorTS, errorPRJ;
  bool projected, accepted, computed_tcs;
```

```
**unsigned int** *step_count* $= 0$;
**int** *exitflag*;
**double** *direction* $= ($*t_end* $\geq x.$*t_*$)\ ?\ 1 : -1$;
**double** *atol* = *params_*→*getAtol*( );
**double** *rtol* = *params_*→*getRtol*( );
$x$.*stats_*→*setTols*(*atol*, *rtol*);
**if** ($x$.*state_* $\equiv$ **daets**::**DAEsolution**::*Initial*)
{
  $x$.*stats_*→*reset*( );
  $x$.*stats_*→*startTimer*( );
  *state* = *checkInput*($x$);
  **if** (*state* $\neq$ **daets**::*success*)
  {
    $x$.*stats_*→*stopTimer*( );
    **return**;
  }
  **daets**::DAETS_H_SCALE $= 1.0$;
  $x$.*xtrial_* $= x$;
  **daets**::*projectInitPoint*($x$, *params_*→*get_ipopt_proj_tol*( ), *jac_*, *ipopt_funcs_*,
      &$x$.*xtrial_*, $x$.*sysJac_*, &*exitflag*);
  **if** (*exitflag* $\neq 0$)
```

```
{
  if ($exitflag \equiv 5$)
    $state = $ daets::toofewdof;
  else
    $state = $ daets::nonconsistentpt;
  $x$.stats_→stopTimer( );
  return;
}
$x$.setFirstEntry( );
$x$.updatePoint($x$.xtrial_);
$x$.state_ = daets::DAEsolution::InitialConsistent;
$x$.htrial_ = 1.0;
$x$.htrial_ = fabs($x$.htrial_) $*$ direction;
bool computed_tcs = tcs_ad_→compTCsX($x$);
assert(computed_tcs);
$x$.setDerivatives(tcs_ad_);
$x$.printData( );
$x$.e_ = estError(tcs_ad_, $x$.order_, $x$.htrial_);
$x$.tol_ = atol + rtol $*$ std::min($x$.max_norm( ), $x$.xtrial_.max_norm( ));
$x$.htrial_ = daets::compInitialStepSize($x$.htrial_, $x$.tol_, $x$.e_, $x$.order_);
$x$.htrial_ = daets::restrictStepsize($x$.htrial_, params_→getHmin( ),
            params_→getHmax( ));
    }
    if (x.t_ ≡ t_end)
    {
       x.stats_→stopTimer( );
       return;
    }
    x.htrial_ = fabs(x.htrial_) ∗ direction;
    x.saveTCs(tcs_ad_);       /∗ for tcs_prev_ needed later for HO method ∗/
    while (step_count < num_steps)
    {
       double h_smallest = daets::compHmin(x, t_end, params_);
       params_→setHmin(h_smallest);
       do
       {
          do
          {
             if (fabs(x.htrial_) < params_→getHmin( ))
             {
                state = daets::htoosmall;
                x.state_ = daets::DAEsolution::EndOfPath;
```

```
        $x.stats\_\to stopTimer()$;
        **return**;
      }
      **daets**::$checkIfLastStep(x.ttrial\_, x.htrial\_, x.t\_, t\_end)$;
      $tcs\_ad\_\to compTSsolution(x.order\_, x.htrial\_, \&ts)$;
      $errorTS = estError(tcs\_ad\_, x.order\_, x.htrial\_)$;
      $tcs\_ad\_\to getX(\&ts\_saved)$;
      $projectTSsolution(x.ttrial\_, ts, params\_\to get\_ts\_proj\_tol(), jac\_, \&x.xtrial\_,$
          $x.sysJac\_, \&exitflag)$;
      $projected = \neg exitflag$;
      **if** $(\neg projected)$
      {
        $x.htrial\_ *= .5$;
        $x.stats\_\to countSteps(false)$;
        $tcs\_ad\_\to setX(ts\_saved)$;
      }
    } **while** $(\neg projected)$;
    $errorPRJ = (ts - x.xtrial\_).max\_norm()/$**std**::$max(x.xtrial\_.max\_norm(), 1.0)$;
    $x.e\_ = errorTS + errorPRJ$;
    $x.tol\_ = atol + rtol *$ **std**::$min(x.xtrial\_.max\_norm(), x.max\_norm())$;
    $accepted = (x.e\_ \leq x.tol\_)$;
```

```
if (accepted)
{
   t_prev_ = x.t_;
   tcs_prev_ = x.savedTCs_;
   h_prev_ = x.h_saved_tcs_;
   x.t_ = x.ttrial_;
   (daets::DAEpoint &) x = x.xtrial_;
   x.state_ = daets::DAEsolution::OnPath;
   computed_tcs = tcs_ad_→compTCsX(x);
   assert(computed_tcs);
   x.printData( );
   x.saveTCs(tcs_ad_);
   x.e_ = daets::estError(tcs_ad_, x.order_, x.htrial_);
   x.htrial_ = daets::compStepSize(x.htrial_, x.tol_, x.e_, x.order_);
   step_count++;
}
else
{
   x.htrial_ = daets::compStepSizeRej(x.htrial_, x.tol_, x.e_, x.order_);
   tcs_ad_→setX(ts_saved);
}
```

```
        $x$.htrial_ = **daets**::restrictStepsize($x$.htrial_, params_→getHmin( ),
              params_→getHmax( ));
        $x$.stats_→countSteps(accepted);
    } **while** (¬accepted);
    $x$.stats_→setHminMax(fabs($x$.htrial_));
    **if** ($x$.t_ ≡ t_end ∨ $x$.state_ ≡ **daets**::**DAEsolution**::EndOfPath)
    {
        state = **daets**::success;
        $x$.stats_→stopTimer( );
        **return**;
    }
    **if** ($x$.print_progress_ ≥ 0)
    {
        PrintProgress($x$.t_, $x$.getNumAccSteps( ), $x$.htrial_, $x$.e_, $x$.order_);
        sleep($x$.print_progress_);
    }
  }
}
```

Appendix B

The IrregularMatrix class

We store the definition of this class in the file `irregularmatrix.h`.

338 ⟨`irregularmatrix.h` 338⟩ ≡

```
#ifndef SRC_IRREGULARMATRIX_H_
#define SRC_IRREGULARMATRIX_H_
#include <assert.h>
#include <cmath>
#include <iostream>
#include <stdexcept>
#include <vector>
#include "daepoint.h"
  namespace sdaets
  {
```

```
template<class T>
class IrregularMatrix {
public:
  explicit IrregularMatrix(const std::vector<size_t> &d) { resize(d); }
  explicit IrregularMatrix(const std::vector<size_t> &d, const T &M);
  explicit IrregularMatrix() { }
  T &operator()(int i, int j) { return x_[i][j]; }
  T operator()(int i, int j) const { return x_[i][j]; }
  size_t num_rows() const { return x_.size(); }
  size_t num_cols(int i) const { return x_[i].size(); }
  size_t num_entries() const;
  void set(const std::vector<T> &v);
  void set(const double *v);
  void set(const daets::DAEpoint &x);
  void set(double a);
  void set_to_zero();
  void to_vector(double *v);
  IrregularMatrix<T> &operator+=(const IrregularMatrix<T> &x);
  IrregularMatrix<T> &operator-=(const IrregularMatrix<T> &x);
  IrregularMatrix<T> &operator*=(const std::vector<double> &v);
  IrregularMatrix<T> &operator/=(const std::vector<double> &v);
```

```
    IrregularMatrix<T> &operator*=(double v);

    double wrms_norm(const IrregularMatrix<double> &weight);

private:

    void resize(const std::vector<size_t> &d);

    std::vector<std::vector<T>> x_;

};

template<class T>

size_t IrregularMatrix<T>::num_entries() const

{

    size_t num = 0;

    for (size_t i = 0; i < num_rows(); i++)

        num += num_cols(i);

    return num;

}

template<class T>

void IrregularMatrix<T>::resize(const std::vector<size_t> &d)

{

    x_.resize(d.size());

    for (size_t i = 0; i < d.size(); i++)

        x_[i].resize(d[i]);

}
```

```
template⟨class T⟩
IrregularMatrix⟨T⟩::IrregularMatrix(const std::vector⟨size_t⟩ &d, const T
        &M)
{
  resize(d);
  for (size_t i = 0; i < num_rows(); i++)
    for (size_t j = 0; j < num_cols(i); j++)
      (*this)(i, j) = M;
}
template⟨class T⟩
void IrregularMatrix⟨T⟩::set(const std::vector⟨T⟩ &v)
{
  assert(num_entries() ≤ v.size());
  size_t k = 0;
  for (size_t i = 0; i < num_rows(); i++)
    for (size_t j = 0; j < num_cols(i); j++)
      (*this)(i, j) = v[k++];
}
template⟨class T⟩
void IrregularMatrix⟨T⟩::set(const double *v)
{
  size_t k = 0;
  for (size_t i = 0; i < num_rows( ); i++)
    for (size_t j = 0; j < num_cols(i); j++)
      (*this)(i, j) = v[k++];
}
template<class T>
void IrregularMatrix<T>::set(const daets::DAEpoint &x)
{
  assert(this->num_rows( ) ≤ x.getNumVariables( ));
  for (size_t i = 0; i < this->num_rows( ); i++)
  {
    assert(this->num_cols(i) ≤ x.getNumDerivatives(i));
    for (size_t j = 0; j < this->num_cols(i); j++)
      (*this)(i, j) = x.getX(i, j);
  }
}
template<class T>
void IrregularMatrix<T>::set(double a)
{
  for (size_t i = 0; i < num_rows( ); i++)
    for (size_t j = 0; j < num_cols(i); j++)
```

```
        (*this)(i, j) = a;
    }
    template⟨class T⟩
    IrregularMatrix⟨T⟩ &IrregularMatrix⟨T⟩::operator+=(const
            IrregularMatrix⟨T⟩ &x)
    {
      assert(x.num_rows() ≡ num_rows());
      for (size_t i = 0; i < x.num_rows(); i++)
      {
        assert(x.num_cols(i) ≡ num_cols(i));
        for (size_t j = 0; j < x.num_cols(i); j++)
          (*this)(i, j) += x(i, j);
      }
      return *this;
    }
    template⟨class T⟩
    IrregularMatrix⟨T⟩ &IrregularMatrix⟨T⟩::operator−=(const
            IrregularMatrix⟨T⟩ &x)
    {
      assert(x.num_rows() ≡ num_rows());
      for (size_t i = 0; i < x.num_rows(); i++)
```

```
  {
    assert(x.num_cols(i) ≡ num_cols(i));
    for (size_t j = 0; j < x.num_cols(i); j++)
      (*this)(i, j) -= x(i, j);
  }
  return *this;
}
template⟨class T⟩
IrregularMatrix⟨T⟩ &IrregularMatrix⟨T⟩::operator*=(const
        std::vector⟨double⟩ &v)
{
  for (size_t i = 0; i < num_rows( ); i++)
  {
    assert(v.size( ) ≥ this→num_cols(i));
    for (size_t j = 0; j < num_cols(i); j++)
      (*this)(i, j) *= v[j];
  }
  return *this;
}
template⟨class T⟩
IrregularMatrix⟨T⟩ &IrregularMatrix⟨T⟩::operator*=(double v)
```

```
{
  for (size_t i = 0; i < num_rows( ); i++)
  {
    for (size_t j = 0; j < num_cols(i); j++)
      (*this)(i, j) *= v;
  }
  return *this;
}
template<class T>
IrregularMatrix<T> &IrregularMatrix<T>::operator/=(const std::vector<double>
        &v)
{
  for (size_t i = 0; i < this->num_rows( ); i++)
    for (size_t j = 0; j < this->num_cols(i); j++)
      (*this)(i, j) /= v[j];
  return *this;
}
IrregularMatrix<double> operator*(double v, const IrregularMatrix<double>
    &M);
void multiply_add(const std::vector<double> &u, double z, std::vector<double>
    &v);
    void multiply_add(const IrregularMatrix⟨double⟩ &A, const std::vector⟨double⟩
          &z, IrregularMatrix⟨double⟩ &B);
    void multiply_add(const IrregularMatrix⟨std::vector⟨double⟩⟩ &A, const
          std::vector⟨double⟩ &z, IrregularMatrix⟨std::vector⟨double⟩⟩ &B);
    template⟨class T⟩
    IrregularMatrix⟨T⟩ operator−(const IrregularMatrix⟨T⟩ &A, const
                IrregularMatrix⟨T⟩ &B)
    {
      IrregularMatrix⟨T⟩ C = A;
      C −= B;
      return C;
    }
  }
#endif
```

We store the definition of all functions of the **IrregularMatrix** class in the file `irregularmatrix.cc`:

339 ⟨ `irregularmatrix.cc` 339 ⟩ ≡

```
#include "irregularmatrix.h"
#include "daepoint.h"
  namespace sdaets
  {
    template⟨ ⟩
```

```
void IrregularMatrix⟨double⟩::to_vector(double *v)
{
  size_t k = 0;
  for (size_t i = 0; i < num_rows( ); i++)
    for (size_t j = 0; j < num_cols(i); j++)
      v[k++] = (*this)(i, j);
}
template⟨ ⟩
void IrregularMatrix⟨std::vector⟨double⟩⟩::to_vector(double *v)
{
  size_t count = 0;
  for (size_t k = 0; k < num_entries( ); k++)
    for (size_t i = 0; i < num_rows( ); i++)
      for (size_t j = 0; j < num_cols(i); j++)
        v[count++] = (*this)(i, j)[k];
}
template⟨ ⟩
double IrregularMatrix⟨double⟩::wrms_norm(const IrregularMatrix⟨double⟩
        &w)
{
  size_t n = this→num_entries( );
  assert(w.num_entries() ≡ n);
  assert(w.num_rows() ≡ this→num_rows());
  double s = 0;
  for (size_t i = 0; i < this→num_rows(); i++)
    for (size_t j = 0; j < this→num_cols(i); j++)
    {
      double ew_ij = (*this)(i, j) * w(i, j);
      s += (ew_ij * ew_ij);
    }
  s /= n;
  return std::sqrt(s);
}
template⟨ ⟩
void IrregularMatrix⟨double⟩::set_to_zero()
{
  for (size_t i = 0; i < num_rows(); i++)
    for (size_t j = 0; j < num_cols(i); j++)
      (*this)(i, j) = 0;
}
template⟨ ⟩
void IrregularMatrix⟨std::vector⟨double⟩⟩::set_to_zero()
```

```
{
  for (size_t i = 0; i < num_rows(); i++)
    for (size_t j = 0; j < num_cols(i); j++)
      for (size_t k = 0; k < (*this)(i, j).size(); k++)
        (*this)(i, j)[k] = 0;
}
IrregularMatrix<double> operator*(double v, const IrregularMatrix<double>
        &M)
{
  IrregularMatrix<double> C = M;
  C *= v;
  return C;
}
void multiply_add(const std::vector<double> &u, double z, std::vector<double>
        &v)
{
  assert(v.size() == u.size());
  for (size_t i = 0; i < u.size(); i++)
    v[i] += u[i] * z;
}
void multiply_add(const IrregularMatrix<double> &A, const std::vector<double>
            &z, IrregularMatrix⟨double⟩ &B)
  {
    for (size_t i = 0; i < A.num_rows( ); i++)
      for (size_t j = 0; j < A.num_cols(i); j++)
        B(i, j) += A(i, j) * z[j];
  }
  void multiply_add(const IrregularMatrix⟨std :: vector⟨double⟩⟩ &A, const
            std :: vector⟨double⟩ &z, IrregularMatrix⟨std :: vector⟨double⟩⟩ &B)
  {
    for (size_t i = 0; i < A.num_rows( ); i++)
      for (size_t j = 0; j < A.num_cols(i); j++)
        multiply_add(A(i, j), z[j], B(i, j));
  }
}
```

Appendix C

KINSOL

340 ⟨ Nonlinear Solver Functions 59 ⟩ +≡

```
HoFlag NSolveKin(int n, double tol, int max_it, double *x0, KINSysFn
        fcn, KINLsJacFn jac, void *user_data)
{
  N_Vector x, s;
  x = N_VMake_Serial(n, x0);
  s = N_VNew_Serial(n);
  N_VConst_Serial(ONE, s);
     /* instantiate a KINSOL solver object */
  void *kmem = KINCreate();
  assert(kmem);
     /* specify the pointer to user-defined memory */
```

```
**int** $flag = KINSetUserData(kmem, user\_data)$;
$assert(flag \equiv$ KIN_SUCCESS$)$;
$flag = KINSetFuncNormTol(kmem, tol)$;
$flag = KINSetScaledStepTol(kmem, tol)$;
$flag = KINSetNumMaxIters(kmem, max\_it)$;
    /* disable all future error message output */
$flag = KINSetErrFile(kmem, \Lambda)$;
$assert(flag \equiv$ KIN_SUCCESS$)$;
    /* specify the problem defining function fcn, */
    /* allocate internal memory for kinsol, and initialize kinsol. */
$flag = KINInit(kmem, fcn, x)$;
$assert(flag \equiv$ KIN_SUCCESS$)$;
    /* create dense SUNMatrix */
**SUNMatrix** $J = SUNDenseMatrix(n, n)$;
    /* create dense SUNLinearSolver object */
**SUNLinearSolver** LS $= SUNLinSol\_Dense(x, J)$;
    /* attach the matrix and linear solver to KINSOL */
$flag = KINSetLinearSolver(kmem,$ LS$, J)$;
$assert(flag \equiv$ KINLS_SUCCESS$)$;
    /* Set the Jacobian function */
**if** $(jac)$
```

```
{
  flag = KINSetJacFn(kmem, jac);
  assert(flag ≡ KINLS_SUCCESS);
}
              /* maximum number of iterations between computing the Jacobian */
flag = KINSetMaxSetupCalls(kmem, 1);
assert(flag ≡ KIN_SUCCESS);
   /* solve the nonlinear system */
int strategy = KIN_NONE;      /* basic Newton iteration */
flag = KINSol(kmem, x, strategy, s, s);
HoFlag ho_flag;
if (flag < 0)
  ho_flag = HO_SUCCESS;
else
{
  ho_flag = HO_CONVERGENT;
  x0 = N_VGetArrayPointer_Serial(x);
}
N_VDestroy_Serial(x);
N_VDestroy_Serial(s);
KINFree(&kmem);
```

```
    SUNLinSolFree(LS);
    SUNMatDestroy($J$);
    return ho_flag;
}
```

Appendix D

Files

D.1 The HO class

We store the definition of this class in the file `ho.h`:

343 ⟨ho.h 343⟩ ≡

```
#ifndef SRC_HO_H_

#define SRC_HO_H_

#include <vector>

#include "constants.h"

#include "daepoint.h"

#include "fadbadts.h"

#include "fadiff.h"

#include "taylorseries.h"
```

```
#include "gradients.h"

#include "ho_enumtypes.h"

#include "irregularmatrix.h"

#include "sadata.h"

#include "sysjac.h"

#include "tadiff.h"

  namespace sdaets

  {

    ⟨ HO Declarations 23 ⟩

  }

#endif
```

D.1.1 Constructor

```
344 ⟨ HO Public Functions 344 ⟩ ≡

  HO(daets::TaylorSeries *ts, daets::Jacobian *jac, Gradients *grad)

  : ts_(ts), jac_(jac), grads_(grad) {

    sadata_ = ts_→get_sadata( );

    n_ = sadata_→get_size( );

    num_indep_tcs_ = 0;

    for (int j = 0; j < n_; j++)

            num_indep_tcs_ += sadata_→get_d(j);

    d_.resize(n_);
```

```
c_.resize(n_);
for (int i = 0; i < n_; i++)
{
  d_[i] = sadata_->get_d(i);
  c_[i] = sadata_->get_c(i);
}
sys_jac_ = new double[n_ * n_];
rhs_ = new double[n_];
ipiv_ = new int[n_];
tcs_stage0_ = new double[n_];
h_pow_.resize(sadata_->get_max_d() + daets::Constants::kMaxOrder_);
indep_tcs_ = IrregularMatrix<double>(d_);
psi_ = IrregularMatrix<double>(d_);
phi_ = IrregularMatrix<double>(d_);
f_ = IrregularMatrix<double>(d_);
tc_grad_.resize(num_indep_tcs_);
fr_prime_ = IrregularMatrix<std::vector<double>>(d_, tc_grad_);
f_prime_ = IrregularMatrix<std::vector<double>>(d_, tc_grad_);
coef_.resize(sadata_->get_max_d());
factorial_.resize(daets::Constants::kMaxOrder_);
daets::compFactorial(daets::Constants::kMaxOrder_,factorial_.data());
```

```
    residual_flat_ = new double[num_indep_tcs_];
    indep_tcs_flat_ = new double[num_indep_tcs_];
    ho_jacobian_ = new double[num_indep_tcs_ * num_indep_tcs_];
    ho_ipiv_ = new int[num_indep_tcs_];
  }
```

See also chunks 345 and 346

This code is used in chunk 23.

D.1.2 Destructor

345 ⟨HO Public Functions 344⟩ +≡

```
  ~HO()
  {
    delete[] sys_jac_;
    delete[] rhs_;
    delete[] ipiv_;
    delete[] tcs_stage0_;
    delete[] residual_flat_;
    delete[] indep_tcs_flat_;
    delete[] ho_jacobian_;
    delete[] ho_ipiv_;
  }
```

The class HO has the following public member functions

346 $\langle$ HO Public Functions 344 $\rangle$ $+\equiv$

HoFlag *CompF*(**const double** $*x$, **double** $*f$);

void *CompHoJac*(**double** $*$*ho_jac*);

void *EvalEqnsStageZero*(**const double** $*x$, **double** $*f$);

void *SetStageZeroTCsJac*(**const double** $*$*tc*);

void *CompA0*(**double** $*$*jac*);

The private member functions are listed bellow

347 $\langle$ HO Private Functions 35 $\rangle$ $+\equiv$

void *set_t*(**double** t);

void *set_h*(**double** h);

void *form_grad*(**int** i, **int** j, **int** k, **std** :: **vector**$\langle$**double**$\rangle$ &*tc_grad*);

void *FormFrPrime*(**int** r);

void *comp_a*(**int** r);

void *comp_b*(**int** r);

void *FormFr*(**int** r, **const std** :: **vector**$\langle$**vector**$\langle$**double**$\rangle\rangle$ &*tcs*);

void *FormFr*(**int** r);

void *CompPsi*(**const std** :: **vector**$\langle$**std** :: **vector**$\langle$**double**$\rangle\rangle$ &*tcs*);

void *set_pq*(**int** p, **int** q);

void *CompCpq*( );

void *CompCqp*( );

HoFlag *CompHoSolution*(**double** t, **double** h, **double** *tol*,

```
        const IrregularMatrix⟨double⟩ &weight, const std::vector⟨vector⟨double⟩⟩
        &tcs_prev, daets::DAEpoint &x);
    void SetIndepTCs( );
    void SetProjected(daets::DAEpoint &x);
    void SetIndepTCsJac( );
    HoFlag CompPhi(const double *x);
    void CompGradients(int q);
    HoFlag CompTCsNonlinear(double *tc);
    void SetStageZeroTCs(const double *tc);
    void CompTCsLinear(int k);
    void need_cond_jac(bool flag) { need_cond_jac_ = flag; }
    bool CompTCs(daets::DAEpoint &x);
    double CompCondJac( );
```

We store the definition of all functions of the HO class in the file ho.cc:

348 ⟨ho.cc 348⟩ ≡

```
#include <assert.h>
#include "ho.h"
#include "daepoint.h"
#include "ho_auxiliary.h"
#include "irregularmatrix.h"
#include "nsolve_functions.h"
```

```
#include "taylorseries.h"
  namespace sdaets {
    ⟨ Definitions of HO Private Functions 37 ⟩
    ⟨ Definitions of HO Public Functions 61 ⟩
  }
```

D.2 The Gradients class

We store the definition of this class in the file `gradients.h`.

350 ⟨ `gradients.h` 350 ⟩ ≡

```
#ifndef GRADIENTS_H_
#define GRADIENTS_H_
#include "constants.h"
#include "fadiff.h"
#include "tadiff.h"
#include "sadata.h"
  namespace sdaets
  {
    typedef fadbad::T⟨fadbad::F⟨double⟩⟩ TFdouble;
    typedef void(*TFadiff)(TFdouble t, const TFdouble *y, TFdouble *f, void *p);
    ⟨ Gradients Declarations 25 ⟩
  }
```

#endif

D.2.1 Constructor

In the constructor, we allocate the necessary memory, and generate the computational graph by calling *daefun*.

352 ⟨Gradients Data Members 80⟩ +≡

```
TFadiff daefun_;
```

353 ⟨Gradients Public Functions 82⟩ +≡

```
Gradients(TFadiff daefun, daets::SAdata *sadata, void *dae_params)
: sadata_(sadata), daefun_(daefun) {
  int n = sadata_→get_size();
  t_tfdouble_ = 0.0;
  grad_in_ = new TFdouble[2 * n];
  grad_out_ = grad_in_ + n;
  assert(grad_in_ ∧ grad_out_ ∧ daefun_);
  daefun_(t_tfdouble_, grad_in_, grad_out_, dae_params);
}
```

D.2.2 Destructor

354 ⟨Gradients Public Functions 82⟩ +≡

```
~Gradients()
{
```

delete[ ] *grad_in_*;

}

D.3 The StiffDAEsolver class

We store the definition of this class in the file `stiff_daesolver.h`:

356 ⟨`stiff_daesolver.h` 356⟩ ≡

#ifndef `INCLUDED_STIFF_DAESOLVER_H`

#define `INCLUDED_STIFF_DAESOLVER_H`

#include `"DAEsolver.h"`

#include `"ho.h"`

#include `"gradients.h"`

#include `"irregularmatrix.h"`

#include `"nsolve_functions.h"`

#include `"ho_enumtypes.h"`

#define `STIFF_DAE_FCN`(f)(**daets** :: **TdoubleOrgFun**)f, (**daets** :: **TFadiff**)

f, (**daets** :: **TFadiff**) f, f, f

namespace daets {

void *projectInitPoint*(**const DAEsolution** &*x_approx*, **double** *ipopt_proj_tol*,

Jacobian ∗*jac*, *IpoptFuncs* ∗ *ipopt_funcs*, **DAEpoint** ∗*x_projected*, **double**

∗*system_jac*, **int** ∗*exitflag*);

void *projectTSsolution*(**double** t, **const DAEpoint** &X, **double** *ts_proj_tol*, **Jacobian**

```
        *jac_, DAEpoint *Xproj, double *sysJac, int *exitflag);
    double estError(daets::TaylorSeries *auto_diff, int p, double h);
  }
  namespace sdaets {
    ⟨ StiffDAEsolver Declarations 27 ⟩
  }
#endif
```

D.3.1 Constructor

357 ⟨ StiffDAEsolver Public Functions 357 ⟩ ≡

```
  template⟨typename TSFun, typename JacFun, typename GradFun⟩
  StiffDAEsolver(int n, TSFun ts_fun, JacFun jac_fun,
          GradFun grads_fun, daets::SigmaMatrixFcn fcn3, daets::FCN_NONL
          fcn4, void *dae_params = Λ) throw(std::logic_error) :
          daets::DAEsolver(n, ts_fun, jac_fun, fcn3, fcn4, dae_params)
  {
    fadbad_grads_ = new Gradients((TFadiff) grads_fun, sadata_, dae_params);
    assert(fadbad_grads_);
    ho_ = new HO(tcs_ad_, jac_, fadbad_grads_);
    assert(ho_);
    user_min_order_ = 0;
    user_max_order_ = 0;
    ede_ = IrregularMatrix⟨double⟩(ho_→d_);
  }
```

See also chunks 358 and 359

This code is used in chunk 27.

D.3.2 Destructor

358 ⟨StiffDAEsolver Public Functions 357⟩ +≡

```
  ∼StiffDAEsolver( )
  {
    if (¬isIllPosed( ))
    {
      delete ho_;
      delete fadbad_grads_;
    }
  }
```

The StiffDAEsolver class has the following public member functions

359 ⟨StiffDAEsolver Public Functions 357⟩ +≡

```
  void integrate(daets::DAEsolution &x, double tend, daets::SolverExitFlag &state)
      throw(std::logic_error);
  void SetMinMaxOrder(int, int);
  void comp_cond( );
```

360 ⟨ StiffDAEsolver Private Functions 360 ⟩ ≡

```
void GetMinMaxOrder( );
void IntegrateByExplicitTS(daets :: DAEsolution &x, double t_end, unsigned int
    num_steps, daets :: SolverExitFlag &state);
void IntegrateByHO(daets :: DAEsolution &x, double t_end, daets :: SolverExitFlag
    &state);
void SetSavedTCs(const std :: vector⟨std :: vector⟨double⟩⟩ &tcs, int q);
double EstErrHO(int order, double epq, const IrregularMatrix⟨double⟩ &weight);
void tcs_to_ders(const std :: vector⟨std :: vector⟨double⟩⟩ &tcs,
    std :: vector⟨std :: vector⟨double⟩⟩ &ders);
void unscale_tcs(const std :: vector⟨double⟩ &pow_h,
    std :: vector⟨std :: vector⟨double⟩⟩ &tcs);
double CompErrorConstant(int p, int q);
void CompEpq(int p, int q, std :: vector⟨double⟩ &epq);
void set_pq_comp_coeffs(int p, int q, std :: vector⟨double⟩ &ho_epq);
OrderFlag SelectOrder(int m, double sigma_m, const IrregularMatrix⟨double⟩
    &weight, const std :: vector⟨double⟩ &epq);
```

This code is used in chunk 27.

361 ⟨ StiffDAEsolver Data Members 192 ⟩ +≡

```
Gradients *fadbad_grads_;
```

We store the definition of all functions of the **StiffDAEsolver** class in the file `stiff_daesolver.cc`:

362 ⟨stiff_daesolver.cc 362⟩ ≡

```
#include "stiff_daesolver.h"
  namespace sdaets
  {
    ⟨Definitions of StiffDAEsolver Private Functions 194⟩
    ⟨Definitions of StiffDAEsolver Public Functions 254⟩
  }
```

D.4 Nonlinear Solver

363 ⟨nsolve_functions.h 363⟩ ≡

```
#ifndef NSOLVE_FUNCTIONS_H
#define NSOLVE_FUNCTIONS_H
#include <kinsol/kinsol.h>
#include <nvector/nvector_serial.h>
#include <stdio.h>
#include <stdlib.h>
#include <sundials/sundials_math.h>
#include <sundials/sundials_types.h>
#include <sunlinsol/sunlinsol_dense.h>
#include <sunlinsol/sunlinsol_spgmr.h>
#include <sunmatrix/sunmatrix_dense.h>
```

```
#include "ho.h"

#include "ho_auxiliary.h"

#include "ho_enumtypes.h"

#include "norms.h"

#define ONERCONST (1.0)

  namespace sdaets {

    typedef HoFlag(*EvalF)(int, double *, double *, void *);

    typedef void(*EvalJ)(int, double *, double *, void *);

    HoFlag Fcn(int n, double *x, double *f, void *user_data);

    void Jac(int n, double *x, double *jac, void *user_data);

    HoFlag NSolve(int n, const IrregularMatrix<double> &weight, double *x0, EvalF
        fcn, EvalJ jac, double *residual, double *jacobian, int *ipiv, void *user_data);

    int FcnKinsol(N_Vector x, N_Vector f, void *user_data);

    int JacKinsol(N_Vector x, N_Vector f, SUNMatrix J, void *user_data, N_Vector
        tmp1, N_Vector tmp2);

    HoFlag NSolveKin(int n, double tol, int max_it, double *x0, KINSysFn
        fcn, KINLsJacFn jac, void *user_data);

  }

#endif
```

These functions are defined in the file

364 ⟨nsolve_functions.cc 364⟩ ≡

```
#include "nsolve_functions.h"
  namespace sdaets {
    ⟨Nonlinear Solver Functions 59⟩;
  }
```

D.5 Auxiliary functions

We create the header file `ho_auxiliary.h` to declare all auxiliary functions.

365 ⟨`ho_auxiliary.h` 365⟩ ≡

```
#ifndef INCLUDED_HO_AUXILIARY_H_
#define INCLUDED_HO_AUXILIARY_H_
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/select.h>
#include <unistd.h>
#include <vector>
#include <cstddef>
#include "daepoint.h"
#include "ho.h"
#include "ho_enumtypes.h"
#include "irregularmatrix.h"
```

```
#include "norms.h"

  namespace daets {

    extern double DAETS_H_SCALE;

    extern void LSolve(int n, double *Jac, int *ipiv, double *Fcn);

    extern void LU(int n, double *Jac, int *ipiv, int *info);

    extern void compFactorial(int n, double *factorial);

  }

  namespace sdaets {

  extern "C"

  {

    double dlange_(char *norm, int *m, int *n, double *a, int *lda, double *work);

    void dgecon_(char *norm, int *n, double *a, int *lda, double *anorm, double

        *rcond, double *work, int *iwork, int *info);

  }

  void comp_gen_divdif(const std::vector<double> &x, const std::vector<double>

      &f, std::vector<double> &c);

  void CompWeight(const daets::DAEpoint &y, double rtol, double

      atol, IrregularMatrix<double> &w);

  double CompWRMSnorm(const double *v, const IrregularMatrix<double> &w);

      void PrintProgress (double t, int no_steps, double h, double error , int order ) ;

  void update_pq(OrderFlag flag, int &p, int &q);
```

```
double comp_stepsize(double sigma, double max_sigma, double h_old);

double comp_sigma(int order, double ede_m, double safty);

void create_t_vec(double t_prev, double t_curr, int p, int q, std::vector<double>
    &t_vec);

void merge_ders(int k, int p, int q, const std::vector<double> &v_a, const
    std::vector<double> &v_b, std::vector<double> &y);

void CompNordsieck(size_t p, size_t q,
    const std::vector<double> &t_vec, const std::vector<std::vector<double>>
    &der_older, const std::vector<std::vector<double>> &der_old,
    IrregularMatrix<std::vector<double>> &c);

double eval_hermite(int p, int q, double a, double b, double t, std::vector<double> v);

void PredictSolution(int p, int q, double a, double b, double t, const
    IrregularMatrix<std::vector<double>> &nordsieck, daets::DAEpoint
    &prediction);

void subtract(int n, const double *x, double *y);

double cost_per_step(int n, int nn, int p, int q, double h);

OrderFlag min_cost(double cost1, double cost2, double cost3);

void CompPowersH(int size, double h, std::vector<double> &h_pow);

void scalar_times_vector(double a, int n, double *u, double *v);

double RCond(int n, double *mat, double mat_norm);

double MNorm(int n, double *mat);
```

```
  }
#endif
```

These functions are defined in the file `ho_auxiliary.cc`.

366 ⟨ `ho_auxiliary.cc` 366 ⟩ ≡

```
#include "ho_auxiliary.h"
  namespace sdaets {
    ⟨ Auxiliary functions 107 ⟩;
  }
```

D.5.1 Generalized divided differences

Given two vectors

$$\mathbf{x} = [x_0, x_1, \ldots, x_{n-1}] \quad \text{and}$$
$$\mathbf{y} = [y_0, y_1, \ldots, y_{n-1}],$$

we implement the function *comp_gen_divdif* to compute the generalized divided differences

$$c_j = y[x_0, \ldots, x_j], \quad \text{for } j = 0, \ldots, n-1.$$

367 ⟨ Auxiliary functions 107 ⟩ +≡

```
  void comp_gen_divdif(const std::vector⟨double⟩ &x, const std::vector⟨double⟩
          &y, std::vector⟨double⟩ &c)
  {
    size_t n = y.size();
  $assert(n \equiv x.size())$;
  $c = y$;
  $n--$;
  double $clast, temp$;
  for (size_t $j = 0$; $j < n$; $j{+}{+}$)
  {
    $clast = c[j]$;
    for (size_t $i = j + 1$; $i \leq n$; $i{+}{+}$)
    {
      if ($x[i] \equiv x[i - j - 1]$)
        $c[i]$ /= $(j + 1)$;
      else
      {
        $temp = c[i]$;
        $c[i] = (c[i] - clast)/(x[i] - x[i - j - 1])$;
        $clast = temp$;
      }
    }
  }
}
```

D.6 Enumerations

We create the header file `ho_enumtypes.h` to define all required enumerations.

369 ⟨ `ho_enumtypes.h` 369 ⟩ ≡

#ifndef `HO_ENUMTYPES_H_`

#define `HO_ENUMTYPES_H_`

  namespace sdaets {

    ⟨ enumeration type for order selection 202 ⟩;

    ⟨ enumeration type for HO method 24 ⟩;

  }

#endif

D.7 Examples

The interface to DAETS is in the file `stiff_daesolver.h`.

D.7.1 Van der Pol oscillator

The following program integrates (10.2).

372 ⟨ solve Van der Pol 372 ⟩ ≡

  int *main*(**int** *argc*, **char** $*$*argv*[ ])

  {

    ⟨ set size of Van der Pol and integration interval 373 ⟩;

    ⟨ create a solver 307 ⟩;

```
        ⟨create a DAEsolution object 309⟩;
        ⟨set order and tolerance 308⟩;
        ⟨set Van der Pol initial values 374⟩;
        ⟨integrate the problem 311⟩;
        ⟨output results 312⟩;
        return 0;
    }
```

This code is used in chunk 375.

373 ⟨set size of Van der Pol and integration interval 373⟩ ≡

```
    const int n = 1;
    double t0 = 0.0, tend = 2000;
```

This code is used in chunk 372.

374 ⟨set Van der Pol initial values 374⟩ ≡

```
    x.setT(t0)
        .setX(0, 0, 2)
        .setX(0, 1, 0.0);
```

This code is used in chunk 372.

375 ⟨vdpol.cc 375⟩ ≡

```
#include "stiff_daesolver.h"
#include <fstream>
    double CompSCD(daets::DAEsolution &x);
```

```
⟨ Van der Pol 316 ⟩;
⟨ solve Van der Pol 372 ⟩;
double CompSCD(daets::DAEsolution &x)
{
  double y[2];
  y[0] = 1.706167732170469;
  y[1] = −0.8928097010248125 · 10^{−3};
  double error_norm = 0;
  for (int i = 0; i < 2; i++)
  {
    double r = fabs((x.getX(0, i) − y[i])/y[i]);
    if (r > error_norm)
      error_norm = r;
  }
  return −log10(error_norm);
}
```

D.7.2 Oregonator

The following program integrates (10.3).

```
377 ⟨ solve Oregonator 377 ⟩ ≡
  int main(int argc, char *argv[ ])
```

{

⟨ size of Oregonator and the integration interval 378 ⟩;

⟨ create a solver 307 ⟩;

⟨ create a **DAEsolution** object 309 ⟩;

⟨ set order and tolerance 308 ⟩;

⟨ set Oregonator initial values 379 ⟩;

⟨ integrate the problem 311 ⟩;

⟨ output results 312 ⟩;

return 0;

}

This code is used in chunk 380.

378 ⟨ size of Oregonator and the integration interval 378 ⟩ ≡

const int $n = 3$;

double *t0* $= 0.0,$ *tend* $= 360$;

This code is used in chunk 377.

379 ⟨ set Oregonator initial values 379 ⟩ ≡

x.*setT* (*t0*)

.*setX* $(0, 0, 1)$

.*setX* $(1, 0, 2)$

.*setX* $(2, 0, 3)$;

This code is used in chunk 377.

380 ⟨ orego.cc 380 ⟩ ≡

```
#include "stiff_daesolver.h"
#include <fstream>
  double CompSCD(daets::DAEsolution &x);
  ⟨ Oregonator 319 ⟩;
  ⟨ solve Oregonator 377 ⟩;
  double CompSCD(daets::DAEsolution &x)
  {
    double y[3];
    y[0] = 0.1000814870318523 · 10^1;
    y[1] = 0.1228178521549917 · 10^4;
    y[2] = 0.1320554942846706 · 10^3;
    double error_norm = 0;
    for (int i = 0; i < 3; i++)
    {
      double r = fabs((x.getX(i, 0) − y[i])/y[i]);
      if (r > error_norm)
        error_norm = r;
    }
    return −log10(error_norm);
  }
```

D.7.3 Chemical Akzo Nobel

The main program is

382 ⟨ solve Chemical Akzo Nobel 382 ⟩ ≡

```
int main(int argc, char *argv[])
{
  ⟨ set size of Chemical Akzo Nobel and integration interval 383 ⟩;
  ⟨ create a solver 307 ⟩;
  ⟨ create a DAEsolution object 309 ⟩;
  ⟨ set order and tolerance 308 ⟩;
  ⟨ set Chemical Akzo Nobel initial values 384 ⟩;
  ⟨ integrate the problem 311 ⟩;
  ⟨ output results 312 ⟩;
  return 0;
}
```

This code is used in chunk 385.

383 ⟨ set size of Chemical Akzo Nobel and integration interval 383 ⟩ ≡

```
int n = 6;
double t0 = 0, tend = 180;
```

This code is used in chunk 382.

384 ⟨ set Chemical Akzo Nobel initial values 384 ⟩ ≡

```
x.setT(t0)
```

```
    .setX(0, 0, 0.444).setX(0, 1, 0)
    .setX(1, 0, 0.00123).setX(1, 1, 0)
    .setX(2, 0, 0.0).setX(2, 1, 0)
    .setX(3, 0, 0.007).setX(3, 1, 0)
    .setX(4, 0, 0.0).setX(4, 1, 0)
    .setX(5, 0, 0);
```

This code is used in chunk 382.

385 ⟨ chemakzo.cc 385 ⟩ ≡

```
#include "stiff_daesolver.h"
#include <fstream>
  double CompSCD(daets::DAEsolution &x);
  ⟨ Chemical Akzo Nobel 320 ⟩;
  ⟨ solve Chemical Akzo Nobel 382 ⟩;
  double CompSCD(daets::DAEsolution &x)
  {
    double y[6];
    y[0] = 0.1150794920661702;
    y[1] = 0.1203831471567715 · 10^-2;
    y[2] = 0.1611562887407974;
    y[3] = 0.3656156421249283 · 10^-3;
    y[4] = 0.1708010885264404 · 10^-1;
    $y[5] = 0.4873531310307455 \cdot 10^{-2}$;
    **double** *error_norm* $= 0$;
    **for** (**int** $i = 0$; $i < 6$; $i$++)
    {
      **double** $r =$ *fabs*(($x$.*getX*($i, 0$) − $y[i]$)/$y[i]$);
      **if** ($r >$ *error_norm*)
        *error_norm* $= r$;
    }
    **return** −*log10*(*error_norm*);
  }
```

D.7.4 A highly stiff index-2 DAE

The main program integrating (10.5) is

386 ⟨ solve Stiff index-2 386 ⟩ ≡

```
  **int** *main*(**int** *argc*, **char** ∗*argv*[ ])
  {
    ⟨ set size of Stiff index-2 and integration interval 387 ⟩;
    ⟨ create a solver 307 ⟩;
    ⟨ create a **DAEsolution** object 309 ⟩;
    ⟨ set order and tolerance 308 ⟩;
    ⟨ set Stiff index-2 initial values 388 ⟩;
    ⟨ integrate the problem 311 ⟩;
    ⟨output results 312⟩;
    return 0;
  }
```

This code is used in chunk 389.

387 ⟨set size of Stiff index-2 and integration interval 387⟩ ≡

```
  const int n = 3;
  double t0 = 0.0, tend = 2000;
```

This code is used in chunk 386.

388 ⟨set Stiff index-2 initial values 388⟩ ≡

```
  x.setT(t0)
      .setX(0, 0, 2).setX(0, 1, 0)
      .setX(1, 0, 3);
```

This code is used in chunk 386.

389 ⟨`vdpol_index2.cc` 389⟩ ≡

```
#include "stiff_daesolver.h"
#include <fstream>
  double CompSCD(daets::DAEsolution &x);
  ⟨Stiff index-2 322⟩;
  ⟨solve Stiff index-2 386⟩;
  double CompSCD(daets::DAEsolution &x)
  {
    double $y[3]$;
    $y[0] = 1.706167732170469$;
    $y[1] = -0.8928097010248125 \cdot 10^{-3}$;
    $y[2] = sqrt(y[0] * y[0] + 5)$;
    double $r$, $error\_norm = 0$;
    for (int $i = 0$; $i < 2$; $i{+}{+}$)
    {
      $r = fabs((x.getX(0, i) - y[i])/y[i])$;
      if ($r > error\_norm$)
        $error\_norm = r$;
    }
    $r = fabs((x.getX(1, 0) - y[2])/y[2])$;
    if ($r > error\_norm$)
      $error\_norm = r$;
    return $-log10(error\_norm)$;
  }
```

D.7.5 Car Axis

The main program for integrating this problem is

390 ⟨ solve Car Axis 390 ⟩ ≡

```
  int main(int argc, char *argv[ ])
  {
```

⟨ size of Car Axis and the integration interval 391 ⟩;

⟨ create a solver 307 ⟩;

⟨ create a **DAEsolution** object 309 ⟩;

⟨ set order and tolerance 308 ⟩;

⟨ set Car Axis initial values 392 ⟩;

⟨ integrate the problem 311 ⟩;

⟨ output results 312 ⟩;

return 0;

}

This code is used in chunk 393.

391 ⟨ size of Car Axis and the integration interval 391 ⟩ ≡

const int $n = 6$;

double *t0* $= 0.0,$ *tend* $= 3.0$;

This code is used in chunk 390.

The initial condition given in [37] is as follows.

392 ⟨ set Car Axis initial values 392 ⟩ ≡

x.*setT*(*t0*)

.*setX*$(0, 0, 0.0)$.*setX*$(0, 1, -0.5)$

.*setX*$(1, 0, 0.5)$.*setX*$(1, 1, 0.0)$

.*setX*$(2, 0, 1.0)$.*setX*$(2, 1, -0.5)$

.*setX*$(3, 0, 0.5)$.*setX*$(3, 1, 0.0)$;

This code is used in chunk 390.

```
393 ⟨caraxis.cc 393⟩ ≡
#include "stiff_daesolver.h"
#include <fstream>
  double CompSCD(daets::DAEsolution &x);
  ⟨Car Axis 323⟩;
  ⟨solve Car Axis 390⟩;
  double CompSCD(daets::DAEsolution &x)
  {
    double y[6], yp[4];
    y[0] = 0.493455784275402809122 · 10^-1;
    yp[0] = −0.770583684040972357970 · 10^-1;
    y[1] = 0.496989460230171153861;
    yp[1] = 0.744686658723778553466 · 10^-2;
    y[2] = 0.104174252488542151681 · 10^1;
    yp[2] = 0.175568157537232222276 · 10^-1;
    y[3] = 0.373911027265361256927;
    yp[3] = 0.770341043779251976443;
    y[4] = −0.473688659084893324729 · 10^-2;
    y[5] = −0.110468033125734368808 · 10^-2;
    double error_norm = 0;
    for (int i = 0; i < 6; i++)
    {
      double r = fabs((x.getX(i, 0) − y[i])/y[i]);
      if (r > error_norm)
        error_norm = r;
    }
    for (int i = 0; i < 4; i++)
    {
      double r = fabs((x.getX(i, 1) − yp[i])/yp[i]);
      if (r > error_norm)
        error_norm = r;
    }
    return −log10(error_norm);
  }
```

D.7.6 Multi Pendula

Our main program is

394 ⟨ solve Multi Pendula 394 ⟩ ≡

```
  int main(int argc, char *argv[ ])
  {
    ⟨ set size of Multi Pendula and integration interval 395 ⟩;
    sdaets::StiffDAEsolver solver(n, STIFF_DAE_FCN(fcn), parameters);
    ⟨ create a **DAEsolution** object 309 ⟩;
    ⟨ set order and tolerance 308 ⟩;
    ⟨ set Multi Pendula initial values 397 ⟩;
    ⟨ integrate the problem 311 ⟩;
    ⟨ output results 312 ⟩;
    return 0;
  }
```

This code is used in chunk 398.

395 ⟨ set size of Multi Pendula and integration interval 395 ⟩ ≡

```
  double t0 = 0.0, tend = 50;
  int P = 8;
  int n = 3 * P;
  double G = 9.8, L = 10, c = 0.1;
  double parameters[ ] = {G, L, c, (double) P};
```

This code is used in chunk 394.

We set initial conditions as follows.

396 ⟨ Multi Pendula 326 ⟩ +≡

```
  void SetInitialConditions(double t0, daets::DAEsolution &x, int P)
  {
    x.setT(t0);
    for (int i = 1; i ≤ P; i++)
    {
      x.setX(3 * i − 3, 0, 1).setX(3 * i − 3, 1, 0)
          .setX(3 * i − 2, 0, 0).setX(3 * i − 2, 1, 1);
      for (int k = 2; k < 2 * (P − i + 1); k++)
        x.setX(3 * i − 3, k, 0)
            .setX(3 * i − 2, k, 0)
            .setX(3 * i − 1, k − 2, 0);
    }
  }
```

397 ⟨ set Multi Pendula initial values 397 ⟩ ≡

```
  SetInitialConditions(t0, x, P);
```

This code is used in chunk 394.

398 ⟨ multipend.cc 398 ⟩ ≡

```
#include "stiff_daesolver.h"
#include <fstream>
  double CompSCD(daets::DAEsolution &x);
  void SetInitialConditions(double t0, daets::DAEsolution &x, int P);
  ⟨ Multi Pendula 326 ⟩;
  ⟨ solve Multi Pendula 394 ⟩;
  double CompSCD(daets::DAEsolution &x)
  {
```

double $y[24]$;

$y[0] = -6.4831571036071827$;

$y[1] = 7.6137161734561474$;

$y[2] = 2.2484325549960991$;

$y[3] = -4.0364902310106219$;

$y[4] = 9.3943688566338341$;

$y[5] = 3.0461516420251065$;

$y[6] = -3.6572225101397066 \cdot 10^{-1}$;

$y[7] = 1.0298123174512218 \cdot 10^{1}$;

$y[8] = 2.7030793890008225$;

$y[9] = -9.9545920353178730$;

$y[10] = -2.5269195812728182$;

$y[11] = -1.8563855044649777 \cdot 10^{-1}$;

$y[12] = -9.3676724491232601$;

$y[13] = 3.4461254187504720$;

$y[14] = 7.5958908054680752 \cdot 10^{-1}$;

$y[15] = -9.9849301250615969$;

$y[16] = 1.3513394519675139$;

$y[17] = 4.0315262620453329 \cdot 10^{-1}$;

$y[18] = -9.9383198467905878$;

$y[19] = 1.4274905238555882$;

```
  $y[20] = 4.0229079570012122 \cdot 10^{-1}$;
  $y[21] = -9.9318259110677296$;
  $y[22] = 1.4714054650187212$;
  $y[23] = 4.1054226578576264 \cdot 10^{-1}$;
  double error_norm = 0;
  for (int i = 0; i < 24; i++)
  {
    double r = fabs((x.getX(i, 0) - y[i])/y[i]);
    if (r > error_norm)
      error_norm = r;
  }
  return -log10(error_norm);
}
```

Bibliography

[1] R. C. Aiken, *Stiff computation*, vol. 169, Oxford University Press New York, 1985.

[2] U. M. Ascher, H. Chin, and S. Reich, *Stabilization of DAEs and invariant manifolds*, Numerische Mathematik, 67 (1994), pp. 131–149.

[3] U. M. Ascher and L. R. Petzold, *Projected implicit Runge-Kutta methods for differential-algebraic equations*, SIAM Journal on Numerical Analysis, 28 (1991), pp. 1097–1120.

[4] U. M. Ascher and L. R. Petzold, *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*, SIAM, Philadelphia, 1998.

[5] K. Atkinson, W. Han, and D. E. Stewart, *Numerical solution of ordinary differential equations*, vol. 108, John Wiley & Sons, 2011.

[6] R. Barrio, *Performance of the Taylor series method for ODEs/DAEs*, Appl. Math. Comp., 163 (2005), pp. 525–545.

[7] K. E. BRENAN, S. L. CAMPBELL, AND L. R. PETZOLD, *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*, SIAM, Philadelphia, second ed., 1996.

[8] L. BRUGNANO AND C. MAGHERINI, *The BiM code for the numerical solution of ODEs*, Journal of Computational and Applied Mathematics, 164 (2004), pp. 145–158.

[9] O. BRUNO AND D. HOCH, *Numerical differentiation of approximated functions with limited order-of-accuracy deterioration*, SIAM Journal on Numerical Analysis, 50 (2012), pp. 1581–1603.

[10] M. CALVO, F. LISBONA, AND J. MONTIJANO, *On the stability of variable-stepsize Nordsieck BDF methods*, SIAM journal on numerical analysis, 24 (1987), pp. 844–854.

[11] S. CAMPBELL AND C. GEAR, *The index of general nonlinear DAEs*, Numerische Mathematik, 72 (1995), pp. 173–196.

[12] S. L. CAMPBELL AND E. GRIEPENTROG, *Solvability of general differential-algebraic equations*, SIAM Journal on Scientific Computing, 16 (1995), pp. 257–270.

[13] J. CASH, *Efficient numerical methods for the solution of stiff initial-value problems and differential algebraic equations*, in Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences, vol. 459, The Royal Society, 2003, pp. 797–815.

[14] G. F. Corliss, A. Griewank, P. Henneberger, G. Kirlinger, F. A. Potra, and H. J. Stetter, *High-order stiff ODE solvers via automatic differentiation and rational prediction*, in International Workshop on Numerical Analysis and Its Applications, Springer, 1996, pp. 114–125.

[15] G. Dahlquist et al., *Problems related to the numerical treatment of stiff differential equations*, in International Computing Symposium, 1973, pp. 307–314.

[16] G. G. Dahlquist, *A special stability problem for linear multistep methods*, BIT Numerical Mathematics, 3 (1963), pp. 27–43.

[17] I. Duff and C. Gear, *Computing the structural index*, SIAM Journal on Algebraic and Discrete Methods, 7 (1986), pp. 594–603.

[18] B. L. Ehle, *On Padé approximations to the exponential function and A-stable methods for the numerical solution of initial value problems*, PhD thesis, University of Waterloo, Waterloo, Ontario, 1969.

[19] E. Eich-Soellner and C. Führer, *Numerical methods in multibody dynamics*, vol. 45, Springer, 1998.

[20] E. Gad, M. Nakhla, R. Achar, and Y. Zhou, *A-stable and L-stable high-order integration methods for solving stiff differential equations*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 28 (2009), pp. 1359–1372.

[21] C. GEAR, *Simultaneous numerical solution of differential-algebraic equations*, IEEE transactions on circuit theory, 18 (1971), pp. 89–95.

[22] C. W. GEAR, *Differential-algebraic equation index transformations*, SIAM Journal on Scientific and Statistical Computing, 9 (1988), pp. 39–47.

[23] ——, *Differential algebraic equations, indices, and integral algebraic equations*, SIAM Journal on Numerical Analysis, 27 (1990), pp. 1527–1534.

[24] E. GRIEPENTROG AND R. MARZ, *Differential-algebraic equations and their numerical treatment*, in Teubner-Texte zur Mathematik [Teubner Texts in Mathematics], 88., BSB B. G. Teubner Verlagsgesellschaft, Leipzig, 1986. With German, French and Russian summaries.

[25] A. GRIEWANK, *ODE solving via automatic differentiation and rational prediction*, Pitman Research Notes in Mathematics Series, (1996), pp. 36–56.

[26] ——, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, Frontiers in applied mathematics, SIAM, Philadelphia, PA, 2000.

[27] A. GRIEWANK, D. JUEDES, AND J. UTKE, *ADOL-C, a package for the automatic differentiation of algorithms written in C/C++*, ACM Trans. Math. Software, 22 (1996), pp. 131–167.

[28] E. HAIRER AND G. WANNER, *Solving Ordinary Differential Equations II. Stiff and Differential–Algebraic Problems*, Springer Verlag, Berlin, second ed., 1991.

[29] A. HARO, *Automatic differentiation tools in computational dynamical systems*, Univ. of Barcelona Preprint, (2008).

[30] C. HERMITE, *Oeuvres de Charles Hermite (Gautheir–Villar, Paris), vol*, 1912.

[31] A. HINDMARSH, P. BROWN, K. GRANT, S. LEE, R. SERBAN, D. SHUMAKER, AND C. WOODWARD, *SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers*, ACM TOMS, 31 (2005), pp. 363–396.

[32] D. E. KNUTH, *Literate Programming*, Center for the Study of Language and Information, Stanford, CA, USA, 1992.

[33] D. E. KNUTH AND S. LEVY, *The CWEB System of Structured Documentation*, Addison-Wesley, Reading, Massachusetts, 1993.

[34] P. KUNKEL AND V. MEHRMANN, *Index reduction for differential-algebraic equations by minimal extension*, ZAMM-Journal of Applied Mathematics and Mechanics/Zeitschrift für Angewandte Mathematik und Mechanik, 84 (2004), pp. 579–597.

[35] P. KUNKEL AND V. L. MEHRMANN, *Differential-algebraic equations: analysis and numerical solution*, European Mathematical Society, Zürich, Switzerland, 2006.

[36] J. LAMBERT, *Computational Methods in Ordinary Differential Equations*, Wiley, 1977.

[37] F. Mazzia and F. Iavernaro, *Test set for initial value problem solvers*, Tech. Rep. 40, Department of Mathematics, University of Bari, Italy, 2003. http://pitagora.dm.uniba.it/~testset/.

[38] R. McKenzie and J. Pryce, *Structural analysis based dummy derivative selection for differential algebraic equations*, BIT Numerical Mathematics, 57 (2017), pp. 433–462.

[39] V. Mehrmann, *Index concepts for differential-algebraic equations*, Encyclopedia of Applied and Computational Mathematics, (2015), pp. 676–681.

[40] P. Milenkovic, *Numerical solution of stiff multibody dynamic systems based on kinematic derivatives*, Journal of Dynamic Systems, Measurement, and Control, 136 (2014), p. 061001.

[41] W. E. Milne, *A note on the numerical integration of differential equations*, Journal of Research of the National Bureau of Standards, 43 (1949), pp. 537–542.

[42] R. Moore, *Interval Analysis*, Prentice-Hall, Englewood, N.J., 1966.

[43] N. Nedialkov and J. Pryce, *DAETS user guide*, Tech. Rep. CAS 08-08-NN, Department of Computing and Software, McMaster University, Hamilton, ON, Canada, June 2013. 68 pages, DAETS is available at http://www.cas.mcmaster.ca/~nedialk/daets.

[44] N. S. NEDIALKOV, *Implementing a rigorous ODE solver through literate programming*, in Modeling, Design, and Simulation of Systems with Uncertainties, A. Rauh and E. Auer, eds., Springer, 2011, pp. 3–19.

[45] N. S. NEDIALKOV AND J. D. PRYCE, *Solving differential-algebraic equations by Taylor series (I): Computing Taylor coefficients*, BIT Numerical Mathematics, 45 (2005), pp. 561–591.

[46] ——, *Solving differential-algebraic equations by Taylor series (II): Computing the system Jacobian*, BIT Numerical Mathematics, 47 (2007), pp. 121–135.

[47] ——, *Solving differential-algebraic equations by Taylor series (III): the DAETS code*, JNAIAM J. Numer. Anal. Indust. Appl. Math, 3 (2008), pp. 61–80.

[48] T. NGUYEN-BA, H. YAGOUB, H. HAO, AND R. VAILLANCOURT, *Pryce pre-analysis adapted to some DAE solvers*, Applied Mathematics and Computation, 217 (2011), pp. 8403–8418.

[49] N. OBRECHKOFF, *Sur les quadrature mecaniques*, Spisanic Bulgar Akad Nauk, 65 (1942), pp. 191–289.

[50] N. OBRESHKOV, *Neue quadraturformeln*, Verlag der Akademie der Wissenschaften, in Kommission bei W. de Gruyter, 1940.

[51] C. C. PANTELIDES, *The consistent initialization of differential-algebraic systems*, SIAM J. Sci. Stat. Comput., 9 (1988), pp. 213–231.

[52] M. PHARR AND G. HUMPHREYS, *Physically Based Rendering: From Theory to Implementation*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[53] F. A. POTRA AND W. C. RHEINBOLD, *On the numerical solution of Euler-Lagrange equations*, Journal of Structural Mechanics, 19 (1991), pp. 1–18.

[54] J. D. PRYCE, *A simple structural analysis method for DAEs*, BIT Numerical Mathematics, 41 (2001), pp. 364–394.

[55] G. REISSIG, W. S. MARTINSON, AND P. I. BARTON, *Differential–algebraic equations of index 1 may have an arbitrarily high structural index*, SIAM J. Sci. Comput., 21 (1999), pp. 1987–1990.

[56] W. C. RHEINBOLDT, *Differential-algebraic systems as differential equations on manifolds*, Mathematics of computation, 43 (1984), pp. 473–482.

[57] L. SCHOLZ AND A. STEINBRECHER, *Regularization of DAEs based on the Signature method*, BIT Numerical Mathematics, 56 (2016), pp. 319–340.

[58] ——, *Structural-algebraic regularization for coupled systems of DAEs*, BIT Numerical Mathematics, 56 (2016), pp. 777–804.

[59] L. F. SHAMPINE, *Implementation of implicit formulas for the solution of ODEs*, SIAM Journal on Scientific and Statistical Computing, 1 (1980), pp. 103–118.

[60] ——, *Efficient use of implicit formulas with predictor-corrector error estimate*, Journal of computational and applied mathematics, 7 (1981), pp. 33–35.

[61] B. SIMEON, *MBSPACK-Numerical integration software for constrained mechanical motion*, Surveys on Mathematics for Industry, 5 (1995), pp. 169–201.

[62] M. SMITH, *Towards modern literature programming*, University of Canterbury. Department of Computer Science, (2001).

[63] O. STAUNING AND C. BENDTSEN, *FADBAD++ web page*, May 2003. http://www.imm.dtu.dk/fadbad.html.

[64] J. STOER AND R. BULIRSCH, *Introduction to numerical analysis*, vol. 12, Springer Science & Business Media, 2013.

[65] P. VAN DER HOUWEN AND J. DE SWART, *Parallel linear system solvers for Runge-Kutta methods*, Advances in Computational Mathematics, 7 (1997), pp. 157–181.

[66] O. B. WIDLUND, *A note on unconditionally stable linear multistep methods*, BIT Numerical Mathematics, 7 (1967), pp. 65–70.

[67] Y. ZHOU, *Stable high order methods for circuit simulation*, PhD thesis, Carleton University, 2011.

[68] R. ZOLFAGHARI AND N. S. NEDIALKOV, *Structural analysis of linear integral-algebraic equations*, Journal of Computational and Applied Mathematics, 353 (2019), pp. 243–252.

List of Refinements

$\langle$compute the smallest allowed stepsize 245$\rangle$ Used in chunks 246 and 275.

$\langle$compute *iteration_error* $= \dfrac{\rho}{1-\rho}\|\boldsymbol{\delta}^m\|$ 115$\rangle$ Used in chunk 101.

$\langle$create a solver 307$\rangle$ Used in chunks 305, 372, 377, 382, 386, and 390.

$\langle$create a **DAEsolution** object 309$\rangle$ Used in chunks 305, 372, 377, 382, 386, 390, and 394.

$\langle$declare variables for integration 218 224 227 231 236 239 241 243 249 251 259 267 269 277 283 287 288$\rangle$ Used in chunk 215.

$\langle$declare variables for nonlinear solver 113 114$\rangle$ Used in chunk 101.

$\langle$determine the order for next step 290$\rangle$ Used in chunk 275.

$\langle$enumeration type for HO method 24$\rangle$ Used in chunk 369.

$\langle$enumeration type for order selection 202$\rangle$ Used in chunk 369.

$\langle$estimate the error 270 271 272 273 274$\rangle$ Used in chunk 215.

$\langle$evaluate $\|\boldsymbol{\delta}^m\|$ 112$\rangle$ Used in chunk 101.

$\langle$evaluate $\mathbf{f}_{I_0}$ 64$\rangle$ Used in chunk 59.

$\langle$find $\min\{\mathcal{C}_{\kappa-1}, \mathcal{C}_{\kappa}, \mathcal{C}_{\kappa+1}\}$ to determine the possible order change 211$\rangle$ Used in chunk 203.

$\langle$find LU decomposition of $\mathbf{A}_0$ 265$\rangle$ Used in chunk 262.

$\langle$find LU factorization of $\mathbf{J}_{\mathrm{HO}}$ 104 105$\rangle$ Used in chunk 101.

$\langle$find LU factorization of $\mathbf{A}_0$ 142$\rangle$ Used in chunk 141.

$\langle$find an initial guess for $\mathbf{x}_{J_{\leq\alpha}}$ 250$\rangle$ Used in chunk 215.

$\langle$get $\mathbf{x}^0_{J_{\leq\alpha}}$ from x 175 176$\rangle$ Used in chunk 179.

$\langle$`gradients.h` 350$\rangle$

$\langle$`ho.cc` 348$\rangle$

⟨`ho.h` 343⟩

⟨`ho_auxiliary.cc` 366⟩

⟨`ho_auxiliary.h` 365⟩

⟨`ho_enumtypes.h` 369⟩

⟨initialize $\nabla \mathbf{x}_{J_{<q}}$ 86⟩ Used in chunk 79.

⟨integrate by HO method up to the final time 301⟩ Used in chunk 294.

⟨integrate by explicit TS method on the first step 300⟩ Used in chunk 294.

⟨integrate the problem 311⟩ Used in chunks 305, 372, 377, 382, 386, 390, and 394.

⟨`irregularmatrix.cc` 339⟩

⟨`irregularmatrix.h` 338⟩

⟨`multipend.cc` 398⟩

⟨`nsolve_functions.cc` 364⟩

⟨`nsolve_functions.h` 363⟩

⟨optional output 292 293⟩ Used in chunk 215.

⟨`orego.cc` 380⟩

⟨output results 312⟩ Used in chunks 305, 372, 377, 382, 386, 390, and 394.

⟨predict the stepsize for next step 284 286⟩ Used in chunk 275.

⟨prepare for integration 216 219 225 228 232 237 240 242 244 246 255 268⟩ Used in chunk 215.

⟨prepare for next step 275⟩ Used in chunk 215.

⟨project $\mathbf{x}^{\mathrm{HO}}_{J_{\leq 0}}$ onto constraints 260 261⟩ Used in chunk 215.