A SOLVER FOR HIGH-INDEX DAES

Design and Implementation of a Solver for High-Index Differential-Algebraic Equations

By
Wanhe Zhang, B.Eng.

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements
for the Degree of

Master of Science

McMaster University

MASTER OF SCIENCE (2005) McMaster University
(Computing and Software) Hamilton, Ontario

TITLE: Design and Implementation of a Solver for High-Index Differential-Algebraic Equations

AUTHOR: Wanhe Zhang
B.Eng. (Nankai University)

SUPERVISOR: Dr. Ned Nedialkov

NUMBER OF PAGES: xii, 134

Abstract

Systems of differential-algebraic equations (DAEs) arise in numerious applications, and there has been considerable research on solving DAE initial value problems (IVPs). Existing methods and software for solving DAEs usually handle at most index-three problems. However, DAE problems of index three and higher do arise, for example, in actuator dynamics, multi-stage processes, and optimization.

We present the method of J. Pryce and N. Nedialkov for solving DAEs, which can be of high index, fully implicit, and contain derivatives of order higher than one. We solve such DAEs by expanding their solution in Taylor series (TS). To compute Taylor coefficients, we employ J. Pryce's structural analysis and automatic differentiation. Then we compute an approximate TS solution with appropriate stepsize and project this solution to satisfy the constraints (explicit and hidden) of the problem.

This thesis discusses the algorithms involved in this method, including the algorithms for Taylor coefficients computation, consistent point projection, error estimation, stepsize control, and the overall integration process. The author has implemented a software package named HIDAETS (High-Index DAE by Taylor Series). In this thesis, we present the specification, design, implementation, and usage of HIDAETS. Numerical results on several high-index DAEs are reported. These results demonstrate that HIDAETS is efficient and accurate for solving IVP in DAEs.

Acknowledgements

I would like to express my sincere thanks and deep appreciation to my supervisor, Ned Nedialkov, for his constant support, encouragement, thoughtful guidance throughout my research and the thesis write-up. His detailed comments, helpful suggestions, and careful corrections have improved this thesis significantly. I have learned much from him in both academic research and non-academic fields.

I am grateful to Andrea Walther and Andreas Griewank for their help with ADOL-C package, to Ole Stauning for helping with his FADBAD++ package, and to Andreas Wächther for prompt assistance with his IPOPT software. Without their help, developing HIDAETS would have been far more difficult.

I also thank Dr. Spencer Smith and Dr. Jacques Carette, for reviewing this thesis and for their valuable suggestions and comments.

Finally, I appreciate the help and suggestions from John Pryce (Royal Military College of Science, Cranfield University), and faculty in the Department of Computing and Software, McMaster University.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

We consider a differential-algebraic equation (DAE) initial value problem (IVP) in n *dependent variables* $x_j = x_j(t)$, with t a scalar independent variable, of the form

$$f_i(t, \text{ the } x_j \text{ and the derivatives of them}) = 0, \quad 1 \leq i \leq n. \tag{1.1}$$

We assume that f_i are suitably smooth. We allow derivatives of order higher than one and derivatives of x_j to appear nonlinearly in (1.1).

The goal of this thesis is studying, designing, implementing, and documenting a numerical method [NP03] for solving (1.1) directly by expanding its solution in Taylor series (TS).

1.1 Motivation

The importance of solving DAE initial value problems numerically has been recognized for over 20 years [BCP96]. Interest in DAEs arose because many mathematical models in mechanical, chemical, or electrical engineering, occur naturally as systems of differential equations with algebraic constraints.

The development of efficient numerical methods for solving DAEs has been an active research area and a variety of efficient methods already exist [BCP96]. Several numerical methods have been proposed, including backward differentiation formula (BDF) or implicit Runge-Kutta (IRK) methods. There are also several methods designed specifically for particular applications such as constrained mechanics or electrical circuits. These approaches have proven very useful, and the availability of codes

has encouraged a wider consideration of DAE models. However, these methods are limited to problems of low index or special structure. Informally, the *index* of a DAE is the minimum number of differentiations needed to convert it to an ODE [AP98]. An ODE is of index zero. Generally, the higher the index of a DAE, the more difficult it is to solve.

Initially most of the numerical work on DAEs assumed that a DAE was of index one [Cam95]. High-index DAEs (index ≥ 2) were thought perhaps not important in applications. This has changed within the past ten years, with the realization that many of the problems in mechanics are initially formulated as index-two or higher-index DAEs. However, there were no general code available for even index-two problems before 1995 [CH96]. Even till now, existing methods and software for solving DAEs are restricted to at most index-three problems. Most of these solvers first reformulate the problem to first-order, lower-index forms, and then use existing numerical methods and codes for the reformulated problem to solve the original high-index, high-order problem.

Nedialkov and Pryce [NP03] present a new approach for solving numerically DAEs in the general form (1.1). The DAEs can be of high index, fully implicit, and contain derivatives of order higher than one. Their method does not reduce a DAE to a first-order, lower-index form — they solve it directly by expanding its solution in Taylor series. To compute Taylor coefficients (TCs), they employ the structural analysis (SA) of Pryce [Pry01] and automatic differentiation (AD). Nedialkov [NP03] implements the method into a C++ prototype DAE solver named DAETS.

The algorithms behind a high-index DAE solver employing SA and TS methods need to be studied and presented in detail. These algorithms include Taylor coefficients computation. Jacobian computation, error estimation, stepsize control, TS solution projection, and the whole integration process.

Besides the algorithms, a well-designed, flexible, easy-to-use, and well-documented software package is needed. First, the software package must be open to the user. For example, the user can use his/her own numerical routines for computing TCs, or employ his/her own algorithm in stepsize control. Second, this new software package may be used by people without knowledge of the particular DAE algorithms; therefore, it must be easy-to-use. Last, a well-documented software package is important for later maintenance and improvement.

1.2 Background

DAE solvers. There are several excellent and widely used software packages for solving DAEs. There are also many codes designed specifically for simulating constrained mechanical systems. Table 1.1 lists some standard DAE solvers. In [MI03], the authors summarize more DAE solvers and report various numerical results.

Name	Author(s)	Methods	DAE index
DASSL	Petzold [BS96a]	fixed-leading-coefficient BDF	≤ 1
GAMD	Iavernaro and Mazzia [MI03]	Generalized Adams Methods (GAMs)	≤ 3
MEBDFI	Abdulla and Cash [HW96]	Modified Extended BDF of Cash	≤ 3
RADUA5	Hairer and Wanner [HW96]	Implicit Runge-Kutta method (Radau IIa) of order 5	≤ 3

Table 1.1: Some existing DAE solvers.

All of these solvers are only applicable to at most index-three problems. Most of them are restricted to DAEs of special form or derivatives at most the first. This may be inconvenient, since a high-index, high-order DAE needs to be converted to a lower-index, first-order DAE.

DAE structural analysis. Pantelides [Pan88] proposes a graph-theoretical algorithm to locate subsets of the system equations which need to be differentiated. Pryce [Pry98, Pry01] compares Pantelides' method with his structural analysis and proves these two methods are equivalent to each other. However, the algorithms of Pantenlides can only apply to first-order DAEs, which is in the form of $f(t, x, x') = 0$.

Mattsson and Söderlind [MS93] present a technique for solving high-index DAE problems by index reduction approach. (Their numerical method is related to Pryce's structural analysis.)

Campbell-Gear's derivative array equations [CG95] is another approach for high-index DAEs, but more complex than the structural analysis of Pryce. Their approach requires symbolic software to preprocess the equations, which may be hard to apply automatically to program code.

Pryce in [Pry98, Pry01] presents the theory of his structural analysis. From the results in [Pry98, Pry01], a DAE given in the form (1.1) can be either solved directly using Taylor series or converted to an ODE which can then be solved.

Taylor series solution of DAEs. Taylor series method for the solution of IVP for an ODE has been well studied. Successful software packages for solving ODE by Taylor series are AUTOMFT [CC94], VNODE [NJ02], and COSY [Ber97].

Chang and Corliss [CC94] present how to generate Taylor series for the simple pendulum (2.2) in an ad hoc way. Then Corliss and Lodwick [CL96] extend it to validated solution for (2.2) using Lohner's Anfangswertaufgabe (AWA) program.

Pryce [Pry98] presents a Taylor method for solving high-index DAE systems by expanding the solution as a high-order Taylor series. He has implemented a prototype in MATLAB code that solves the DAE using Taylor coefficients, without converting to an ODE.

Nedialkov [NP03] has implemented the method into a prototype DAE solver named DAETS in C++ program. He computes TCs by operator overloading without parsing and code generation. As in Pryce's structural analysis, the DAE is solved without reducing to low-order, low-index forms.

1.3 Contributions

The contributions of this thesis include two parts. The first part is the presentation of algorithms for a general high-index DAE solver based on structural analysis and Taylor series. Numerical algorithms and methods such as TCs computation, error estimation, and stepsize control are presented in both pseudo code and C++ code.

The second contribution is the design, documentation, and numerical studies for HIDAETS. We employ the object-oriented design method and present in detail documentations for specification, design, implementation, and usage of HIDAETS. In addition, more than ten DAE and ODE and the testing results for five high-index DAE problems are presented in this thesis.

Numerical results demonstrate that HIDAETS can be accurate, efficient, and suitable for solving DAEs of an index too high for the existing methods and solvers to handle.

1.4 Thesis structure

This thesis is organized as follows.

Chapter 2 presents the main steps of Pryce's structural analysis and describes how the Taylor series method works using a simple example.

Chapter 3 presents numerical algorithms in HIDAETS. We describe algorithms for signature matrix and offsets computation, computing Jacobians, projecting initial point, Taylor coefficients computation, error estimation, and stepsize control. We also present the overall integration process of HIDAETS. All of these algorithms are demonstrated in pseudo code after each description. Corresponding C++ code is given in Appendix D.

Chapter 4 discusses numerical software issues. First, we present specification and design documentation. Since we employ object-oriented design, we give both high-level design and low-level design with detailed documentation. We also illustrate how to install and use HIDAETS at the end of this chapter.

Chapter 5 reports numerical results of HIDAETS. We have tested more than ten DAE and ODE problems with HIDAETS. In this thesis, we study five high-index DAEs in detail. We present results such as work precision diagrams, stepsize behavior, and dependence of the work on the order of the method.

In the last chapter, Chapter 6, we draw conclusions and suggest directions for future work.

In appendix A, we give the acronyms. Appendix B is the knowledge about automatic differentiation. Appendix C is some UML legends used in this thesis. In appendix D, we present some source code for the algorithms discussed in Chapter 3.

Chapter 2

Theoretical Background

2.1 Structural analysis

Here we present the main steps of Pryce's structural analysis. More details can be found in [Pry01, NP03]. We first give some definitions and then illustrate the process with the single pendulum problem (2.2) step by step.

A *transversal* T of an $n \times n$ matrix (σ_{ij}) is a set of n positions in this matrix with one element in each row and column. The *value* of T is $\|T\| = \sum_{(i,j)\in T} \sigma_{ij}$. A *highest value transversal* (HVT) is a transversal T that makes $\|T\|$ as large as possible.

A *consistent point* for (1.1) is a set of the x_j and derivative of them, at a time t, that specify a unique solution.

The *degrees of freedom* (DOF) of a DAE system is the number of independent initial conditions required.

Given a DAE in the form of (1.1), we perform the following steps.

1. Form the $n \times n$ *signature matrix* $\Sigma = (\sigma_{ij})$, according to

$$\sigma_{ij} = \begin{cases} \text{highest order of derivative to } x_j \text{ occurs in } f_i; & \text{or} \\ -\infty \text{ if } x_j \text{ does not occur in } f_i. \end{cases} \tag{2.1}$$

Example 2.1. Throughout this chapter, we give examples based on the single

pendulum [AP98]:

$$\begin{aligned} 0 = f &= x'' + x\lambda, \\ 0 = g &= y'' + y\lambda - G, \\ 0 = h &= x^2 + y^2 - L^2, \end{aligned} \tag{2.2}$$

where $G > 0$, $L > 0$ are constants, and the dependent variables are $x(t)$, $y(t)$, and $\lambda(t)$.

Its signature matrix Σ, labeled by equations and variables, is

$$\begin{array}{c} \\ f \\ g \\ h \end{array} \begin{array}{c} \begin{array}{ccc} x & y & \lambda \end{array} \\ \begin{pmatrix} 2 & -\infty & 0 \\ -\infty & 2 & 0 \\ 0 & 0 & -\infty \end{pmatrix} \end{array}.$$

2. Find an HVT for Σ.

Example 2.2. There are two HVTs for the single pendulum problem: one in positions $(f,\lambda),(g,y),(h,x)$, the other in positions $(f,x),(g,\lambda),(h,y)$. In both HVTs, $\|T\| = 2$.

3. Find n-dimensional integer vectors $\mathbf{c}$ and $\mathbf{d}$, with all $c_i \geq 0$, satisfying

$$\begin{cases} d_j - c_i \geq \sigma_{ij} & \text{for all } i,j = 1,\ldots,n, \\ d_j - c_i = \sigma_{ij} & \text{for all } (i,j) \in T, \end{cases} \tag{2.3}$$

where T is an HVT. We call the smallest $\mathbf{c}$ and $\mathbf{d}$ (in the sense of $\mathbf{a} \leq \mathbf{b}$ if $a_i \leq b_i$ for each i) the *offsets* of the problem. We name $\mathbf{c}$ *equation offsets*, and $\mathbf{d}$ *variable offsets.*

Example 2.3. For the single pendulum problem, the offsets are $\mathbf{c} = (0,0,2)$ and $\mathbf{d} = (2,2,0)$ for both HVTs[1]:

[1]The * inside of the signature matrix annotates the positions of HVT.

$$\begin{array}{c} \\ f \\ g \\ h \\ d_j \end{array}\begin{array}{c} \begin{array}{cccc} x & y & \lambda & c_i \end{array} \\ \begin{pmatrix} 2 & -\infty & 0* \\ -\infty & 2* & 0 \\ 0* & 0 & -\infty \end{pmatrix}\begin{array}{c} 0 \\ 0 \\ 2 \end{array} \\ \begin{array}{ccc} 2 & 2 & 0 \end{array} \end{array} \quad \text{and} \quad \begin{array}{c} \\ f \\ g \\ h \\ d_j \end{array}\begin{array}{c} \begin{array}{cccc} x & y & \lambda & c_i \end{array} \\ \begin{pmatrix} 2* & -\infty & 0 \\ -\infty & 2 & 0* \\ 0 & 0* & -\infty \end{pmatrix}\begin{array}{c} 0 \\ 0 \\ 2 \end{array} \\ \begin{array}{ccc} 2 & 2 & 0 \end{array} \end{array}$$

4. Form the $n \times n$ System Jacobian matrix

$$\mathbf{J} = \frac{\partial\left(f_1^{(c_1)}, \ldots, f_n^{(c_n)}\right)}{\partial\left(x_1^{(d_1)}, \ldots, x_n^{(d_n)}\right)}. \tag{2.4}$$

From Griewank's lemma in [NP03], (2.4) is equivalent to

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

Example 2.4. For the single pendulum problem, we have

$$\mathbf{J} = \begin{bmatrix} \frac{\partial f}{\partial x''} & 0 & \frac{\partial f}{\partial \lambda} \\ 0 & \frac{\partial g}{\partial y''} & \frac{\partial g}{\partial \lambda} \\ \frac{\partial h}{\partial x} & \frac{\partial h}{\partial y} & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 2x & 2y & 0 \end{bmatrix}.$$

5. Find values for $x_j^{(k_j)}$ satisfying $f_i^{(l_i)} = 0$, where $i, j = 1, \ldots, n$; $k_j = 0, \ldots, d_j$; $l_i = 0, \ldots, c_i$. If such values are found, and $\mathbf{J}$ is nonsingular, a consistent point as defined above is found. Then the structural analysis method succeeds.

Example 2.5. For the single pendulum, we need to solve

$$\begin{pmatrix} f \\ g \\ h \\ h' \\ h'' \end{pmatrix} = \begin{pmatrix} x'' + x\lambda \\ y'' + y\lambda - G \\ x^2 + y^2 - L^2 \\ 2xx' + 2yy' \\ 2xx'' + 2yy'' + 2x'^2 + 2y'^2 \end{pmatrix} = 0.$$

To find values for $(x, x', x''; y, y', y''; \lambda)$, we need to solve $h = 0$ for x, y, and then $h' = 0$ for x', y'. Then f, g, h'' form a linear system of x'', y'', and λ as

$$\begin{bmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 2x & 2y & 0 \end{bmatrix} \begin{bmatrix} x'' \\ y'' \\ \lambda \end{bmatrix} + \begin{bmatrix} 0 \\ -G \\ 2x'^2 + 2y'^2 \end{bmatrix} = 0,$$

where the first matrix is just $\mathbf{J}$. Since $\det \mathbf{J} = -2(x^2 + y^2) = -2L^2 \neq 0$, we can solve this linear system.

Thus the method succeeds for the single pendulum problem.

When the method succeeds, there are some properties:

- The DAE has $\sum_j d_j - \sum_i c_i$ DOF, which also equals the value of the HVT.
- The differentiation index ν_d, see [CG95], of the DAE is less than or equal to the *Taylor index*

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

In many cases, $\nu_T = \nu_d$.

Example 2.6. The single pendulum problem has 2 DOF, and $\nu_d = \nu_T = 3$.

2.2 Taylor series

A *Taylor series* expansion of a C^∞ function $f(t)$ about a point $t = t^*$ is given by

$$f(t) = f(t^*) + f'(t^*)(t - t^*) + \frac{f''(t^*)}{2!}(t - t^*)^2 + \ldots + \frac{f^{(n)}(t^*)}{n!}(t - t^*)^n + \ldots.$$

We assume that for some $N \geq 1$, each function f_i in (1.1) has $(N + c_i)$ continuous derivatives in a neighborhood of a point $(t^*, \mathbf{x}^*)$ at which $\mathbf{J}$ is nonsingular. Thus, from Theorem 4.2 in [NP03], we can compute TCs for $x_j(t)$ up to order $(N + d_j)$.

For ODE initial value problems, Taylor series methods are well known [CC82, Pry98, Ned99]. Their implementation is based on AD to evaluate Taylor coefficients to arbitrary order.

For DAE systems, this method meets some difficulties.

1. The required initial values are not obvious in general.
2. Simultaneous equations must be solved to find Taylor coefficients.
3. It is no longer possible to find first all the first coefficients, then all the second coefficients, and so on, as done in standard ODE TS methods.

Pryce [Pry98, Pry01] and Nedialkov and Pryce [NP03] have presented a "staggered" way to find TCs for a general DAE. Here, we outline how their method works.

We denote the lth TC of a function f of a real variable t at a point t^* by

$$(f)_l = \frac{f^{(l)}(t^*)}{l!}. \tag{2.5}$$

Denote

$$k_c = -\max_i c_i \quad \text{and} \quad k_d = -\max_j d_j, \tag{2.6}$$

where c_i and d_j are the offsets defined in (2.3).

The solution scheme for computing TCs is to solve a system of equations on each stage $k = k_d, k_d + 1, \ldots$. Each system contains

$$(f_i)_{k+c_i} = 0 \quad \text{for all } i \text{ such that} \quad k + c_i \geq 0, \tag{2.7}$$

and we solve for

$$(x_j)_{k+d_j} \quad \text{for all } j \text{ such that} \quad k + d_j \geq 0,$$

where all previously computed $(x_j)_l$ occurring in (2.7) are to be treated as constants.

Example 2.7. For the pendulum, we have the following recipe[2]:

Stage	uses equations	to obtain
$k = -2$	$0 = h_0$	x_0, y_0
$k = -1$	$0 = h_1$	x_1, y_1
$k = 0$	$0 = f_0, g_0, h_2$	x_2, y_2, λ_0
$k = 1$	$0 = f_1, g_1, h_3$	x_3, y_3, λ_1
$\ldots$	$\ldots$	$\ldots$

[2]This example is from [NP03]. For simplicity, we use x_i instead of $(x)_i$ for the single pendulum problem. A similar rule applies to other variables in that problem.

At stage $k = k_d = -2$, we find x_0, y_0 that satisfy

$$0 = h_0 = x_0^2 + y_0^2 - L^2.$$

At stage $k = -1$, we find x_1, y_1 that satisfy

$$0 = h_1 = 2x_0x_1 + 2y_0y_1,$$

taking the previously computed x_0, y_0 as constants.

At stage $k = 0$, we find x_2, y_2, λ_0 that satisfy

$$\begin{aligned} 0 &= f_0 = 1 \cdot 2x_2 + x_0\lambda_0, \\ 0 &= g_0 = 1 \cdot 2y_2 + y_0\lambda_0 - G, \\ 0 &= h_2 = 2x_0x_2 + x_1^2 + 2y_0y_2 + y_1^2, \end{aligned} \tag{2.8}$$

taking the previous computed x_0, y_0, x_1, y_1 as knowns.

In general, at stage $k \geq 0$, we find x_{k+2}, y_{k+2}, λ_k satisfying $f_k, g_k, h_{k+2} = 0$, subject to all the already found values.

By summing relevant TCs at current time t^* with appropriate stepsize h, we obtain an approximate TS solution at $t^* + h$.

Thus, the whole algorithm employing TS processes is summarized as follows.

1. At current $t = t^*$, an approximate solution is given, which comprises $(x_j)_l$, where $0 \leq l \leq d_j$.

2. Stages $k = k_d, \ldots, 0$ convert $(x_j)_l$ to a consistent initial point.

3. Stages $k = 1, 2, \ldots$ compute further TCs up to some specified order.

4. Summing relevant TCs with appropriate stepsize h obtains the approximate $(x_j)_l$ at point $t = t^* + h$. Go back to 1. and update $t = t^* + h$. The process repeats.

Chapter 3

Numerical Algorithms

This chapter describes the algorithms in HIDAETS. First we introduce some of the notation we need later. Then we describe the overall integration process and each step in detail.

3.1 Notation

The notation here follows closely [NP03].

Let

$$\mathcal{I} = \{(i,l) \mid i = 1,\ldots,n;\ l = 0,1,\ldots\}.$$

Given the n-dimensional equation offsets $\mathbf{c}$ and variable offsets $\mathbf{d}$, we define for all $k \in \mathbb{Z}$,

$$I_k = \{(i,l) \in \mathcal{I} \mid l = k + c_i\},$$
$$J_k = \{(j,l) \in \mathcal{I} \mid l = k + d_j\},$$

or equivalently,

$$I_k = \{(i,l) \mid l = k + c_i \geq 0\},$$
$$J_k = \{(j,l) \mid l = k + d_j \geq 0\}.$$

Here I_k is empty for $k < k_c$, and J_k is empty for $k < k_d$, where k_c, k_d are defined by (2.6).

3.1.1 Notation for Taylor coefficients

We interpret the $(x_j)_l$ defined by (2.5) as variables, and $(f_i)_l$ as the function for evaluating the lth TC of f_i. The lth TC for $x_j(t)$ at t^* is denoted by $(x_j)_l^*$. An approximation for the lth TC of $x_j(t)$ at t^* is denoted by $(x_j)_l^a$.

Let

$$m_k = |I_k| \quad \text{and} \quad n_k = |J_k|,$$

where $|S|$ denotes the number of elements in set S.

We denote by $\mathbf{x}_{J_k}$ the n_k-dimensional vector with components $\{(x_j)_l \mid (j,l) \in J_k\}$ ordered in increasing j. That is,

$$\mathbf{x}_{J_k} = \left((x_{j_1})_{l_1}, \ldots, (x_{j_{n_k}})_{l_{n_k}}\right)^T, \tag{3.1}$$

where $j_r, r = 1, \ldots, n_k$, are in increasing order.

Similarly, we denote by $\mathbf{f}_{I_k}$ the m_k-dimensional vector with components $\{(f_i)_l \mid (i,l) \in I_k\}$ ordered in increasing i:

$$\mathbf{f}_{I_k} = \left((f_{i_1})_{l_1}, \ldots, (f_{i_{m_k}})_{l_{m_k}}\right)^T,$$

where $i_r, r = 1, \ldots, m_k$, are in increasing order.

We define $J_{\leq k}$ to be the union of the J_r for $r \leq k$. This is the union for $r = k_d, \ldots, k$, since J_r is empty for $r < k_d$. Then we define $\mathbf{x}_{J_{\leq k}}$ to be the vector of $\{(x_j)_l \mid (j,l) \in J_{\leq k}\}$. In block-vector notation, the components of $\mathbf{x}_{J_{\leq k}}$ are arranged in the order

$$\mathbf{x}_{J_{\leq k}} = \left(\mathbf{x}_{J_{k_d}}, \mathbf{x}_{J_{k_d+1}}, \ldots, \mathbf{x}_{J_k}\right)^T.$$

We define $I_{\leq k}$ to be the union of the I_r for $r \leq k$. This is the union for $r = k_c, \ldots, k$, since I_r is empty for $r < k_c$. Then we define $\mathbf{f}_{I_{\leq k}}$ to be the vector of $\{(f_i)_l \mid (i,l) \in I_{\leq k}\}$. In block-vector notation, the components of $\mathbf{f}_{I_{\leq k}}$ are arranged in the order

$$\mathbf{f}_{I_{\leq k}} = \left(\mathbf{f}_{I_{k_c}}, \mathbf{f}_{I_{k_c+1}}, \ldots, \mathbf{f}_{I_k}\right)^T.$$

In a similar manner, we define $\mathbf{x}_{J_{<k}}$ and $\mathbf{f}_{I_{<k}}$, respectively. In block-vector notation, we may write

$$\mathbf{x}_{J_{\leq k}} = \left(\mathbf{x}_{J_{<k}}, \mathbf{x}_{J_k}\right)^T \quad \text{and} \quad \mathbf{f}_{I_{\leq k}} = \left(\mathbf{f}_{I_{<k}}, \mathbf{f}_{I_k}\right)^T.$$

3.1.2 Notation for derivatives

We denote the lth derivative of x_j by $x_j^{(l)}$ and the lth derivative of f_i by $f_i^{(l)}$.

Similar to the notation for Taylor coefficients, we denote

$$x_{J_k} = \left(x_{j_1}^{(l_1)}, \ldots, x_{j_{n_k}}^{(l_{n_k})}\right)^T, \tag{3.2}$$

where j_r, $r = 1, \ldots, n_k$, are in increasing order[1].

Let

$$f_{I_k} = \left(f_{i_1}^{(l_1)}, \ldots, f_{i_{m_k}}^{(l_{m_k})}\right)^T,$$

where i_r, $r = 1, \ldots, m_k$, are in increasing order.

Denote also

$$\begin{aligned} x_{J_{\leq k}} &= \left(x_{J_{k_d}}, x_{J_{k_d+1}}, \ldots, x_{J_k}\right)^T, \\ f_{I_{\leq k}} &= \left(f_{I_{k_c}}, f_{I_{k_c+1}}, \ldots, f_{I_k}\right)^T. \end{aligned}$$

We also define $x_{J_{<k}}$ and $f_{I_{<k}}$, respectively. In block-vector notation, we have

$$x_{J_{\leq k}} = \left(x_{J_{<k}}, x_{J_k}\right)^T \quad \text{and} \quad f_{I_{\leq k}} = \left(f_{I_{<k}}, f_{I_k}\right)^T.$$

In HIDAETS, all variables are represented in Taylor coefficient format. Most of the following algorithms handle Taylor coefficients directly except for the integration process and tolerance computation, which use variables in derivative format.

3.2 The integration process

To help describe the integration process, we denote

- n: number of variables;
- h: stepsize;
- p: order of Taylor series;
- t_0: starting point of the integration interval;

[1]The only difference between (3.1) and (3.2) is the font of x: bold for Taylor coefficients, and normal for derivatives.

- t_{end}: end point of the integration interval;
- x_{ICs}: set of initial values (specified by the user);
- atol: absolute tolerance;
- rtol: relative tolerance.

In subsection 4.1, we give detailed explanations for these variables.

The user provides

- function for evaluating (1.1);
- t_0, t_{end};
- x_{ICs}.

Optionally, the user provides atol, rtol, and p.

By evaluating the function, we compute the signature matrix and generate computational graphs for computing TCs. After finding the offsets, we try to compute a consistent initial point. If HIDAETS fails to compute one, it cannot solve the problem with the given initial point x_{ICs}. If it succeeds, we continue with the integration.

After obtaining a consistent initial point, we compute Taylor coefficients. We also form tolerance tol using the solution at t_0, atol, and rtol, and then we compute stepsize h. With this h, we compute an approximate TS solution at $t = t_0 + h$ by summing the series.

Now, we have to ensure that this numerical solution satisfies the equation constraints (obvious and hidden). For this purpose, we find the closest point by projecting it as illustrated in Figure 3.1. If HIDAETS fails to obtain such a consistent point, we reduce the stepsize h, recompute a TS solution with the new h, and project it again. We repeat this process, till we obtain a consistent solution.

We iterate the above process by updating t, till $t = t_{\text{end}}$. If t_{end} is reached, HIDAETS succeeds in solving the given DAE problem.

The integration process is given in the pseudo code below and in the flow chart in Figure 3.2. Here, compDAE denotes the function for evaluating the DAE.

INTEGRATE-DAE($n, t_0, t_{\text{end}}, x_{\text{ICs}}$, compDAE)

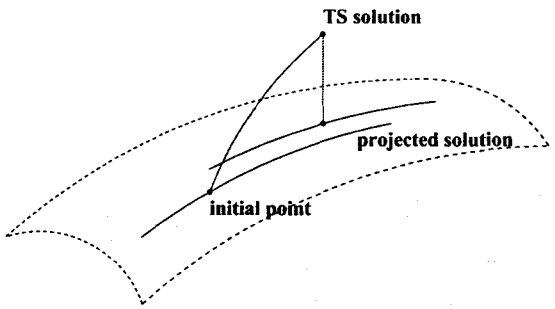


Figure 3.1: One step of the integration process.

```
1  Evaluate compDAE
2  Compute signature matrix and offsets
3  if computing a consistent initial point fails
4    then error "solver fails to compute a consistent initial point"
5    else t ← t_0
6         while t < t_end
7           do Compute Taylor coefficients at t
8              Select stepsize h
9              if |h| < hmin                ▷ h too small
10               then error "stepsize is too small"
11                    return
12             Compute TS solution with h
13             while projecting TS solution fails
14               do reduce stepsize h
15                  if |h| < hmin
16                    then error "stepsize is too small"
17                         return
18                  Compute TS solution with h
19             t ← t + h
```

The corresponding C++ code is on page 121.

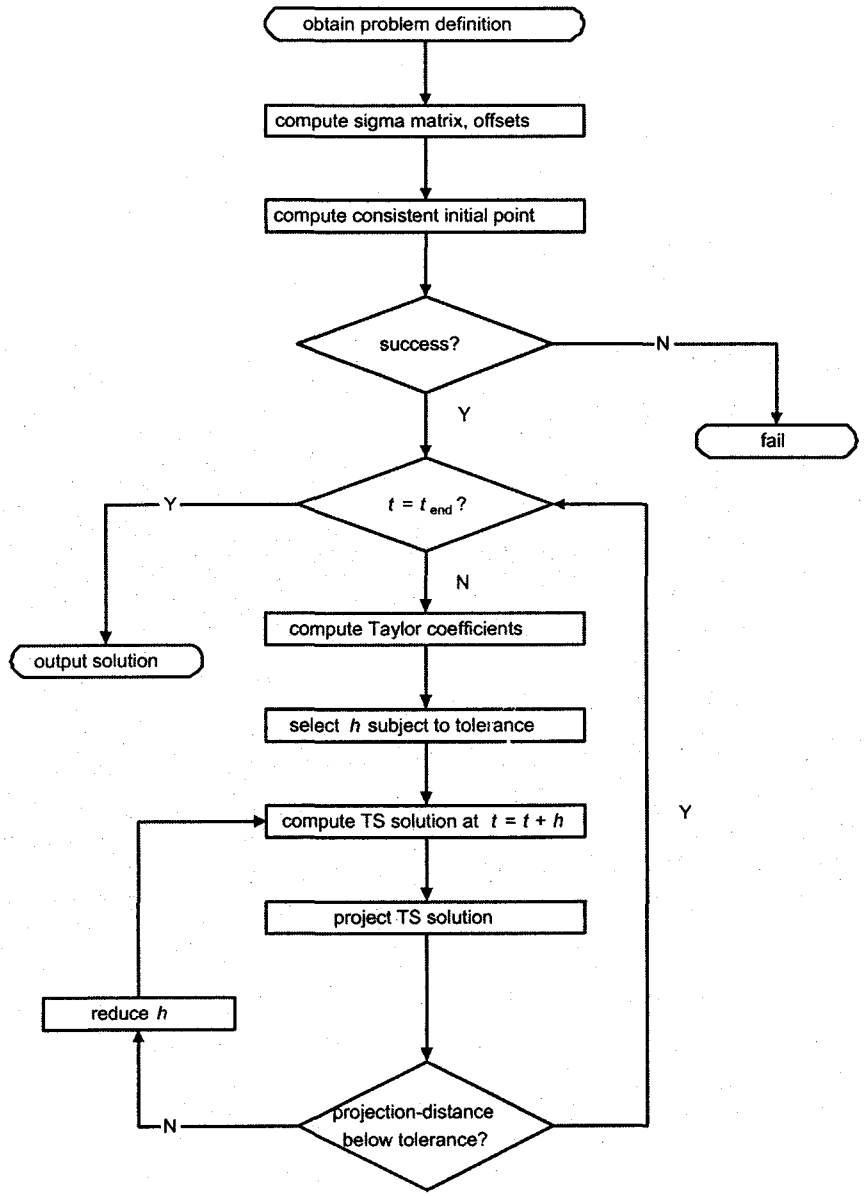


Figure 3.2: The integration process of HIDAETS.

3.3 Computing signature matrix and offsets

Given a DAE in the form of (1.1), we first form the $n \times n$ signature matrix $\Sigma = (\sigma_{ij})$ following (2.1).

To compute the signature matrix, we need to do some work like parsing. In HIDAETS, we use a C++ class *sigma* implemented by Nedialkov. It generates a Σ matrix through operator overloading by executing the function for evaluating the DAE. The algorithm for computing Σ is presented in [NP03].

After obtaining the signature matrix, we compute the equation offsets $\mathbf{c}$ and the variable offsets $\mathbf{d}$. From Pryce's structural analysis, we need first to find an HVT. Then we compute the offsets by finding n-dimensional integer vectors $\mathbf{c}$ and $\mathbf{d}$ that satisfy (2.3).

Finding HVT can be written as a Linear Programming (LP) problem: compute

$$\begin{array}{lll} \text{Max} & \sum_{i,j} \sigma_{ij} x_{ij} & \\ \text{s.t.} & \sum_{i,j} x_{ij} = 1 & \text{for each } i, \\ & \sum_{i,j} x_{ij} = 1 & \text{for each } j, \\ & x_{ij} \in \{0,1\} & i,j = 1,2,\ldots,n. \end{array} \tag{3.3}$$

The HVT is the set of (i,j), where $x_{ij} = 1$ in the solution of (3.3).

In HIDAETS, we use the LAP [JV87] program to solve (3.3). It returns with an n-vector column solution α, such that $\alpha_i = j$ is the position (j,i) in HVT.

Example 3.1. In the single pendulum problem, we have two HVTs as

$$\begin{array}{c} \\ f \\ g \\ h \end{array} \begin{array}{c} \begin{array}{ccc} x & y & \lambda \end{array} \\ \begin{pmatrix} 2 & -\infty & 0* \\ -\infty & 2* & 0 \\ 0* & 0 & -\infty \end{pmatrix} \end{array} \quad \text{and} \quad \begin{array}{c} \\ f \\ g \\ h \end{array} \begin{array}{c} \begin{array}{ccc} x & y & \lambda \end{array} \\ \begin{pmatrix} 2* & -\infty & 0 \\ -\infty & 2 & 0* \\ 0 & 0* & -\infty \end{pmatrix} \end{array}$$

Then, for the first HVT, $\alpha = (3,2,1)^T$; for the second HVT, $\alpha = (1,3,2)^T$.

From [Pry01], finding $\mathbf{c}$ and $\mathbf{d}$ can be treated as a dual LP problem which has the same optimal value with (3.3). After obtaining HVT, we first initialize $\mathbf{c}$ and $\mathbf{d}$ to 0. Then we set d_j to the maximum of $\sigma_{ij} + c_i$, for $i = 1,\ldots,n$. We update $\mathbf{c}$

following the relation of $\mathbf{c}$ and $\mathbf{d}$, that is $d_j - c_i = \sigma_{ij}$, for $(i,j) \in \mathrm{HVT}$. To ensure that all $c_i \geq 0$, we iterate the above process till $\mathbf{d}$ equals $\mathbf{d}_{\mathrm{old}}$, where $\mathbf{d}_{\mathrm{old}}$ stores $\mathbf{d}$'s previous value. This method for computing the offsets vectors is due to Pryce [Pry01].

COMPUTE-OFFSETS(n, c, d, Σ)

```
1  compute HVT of Σ, obtain column solution α
2  c ← 0
3  d ← 0
4  while true
5     do d_old ← d
6        for j ← 1 to n
7           do d_j ← max_i (σ_ij + c_i)
8        if d = d_old
9          then break
10         else  for j ← 1 to n
11                  do i ← α_j
12                     c_i ← d_j − σ_ij
```

The corresponding C++ code is on page 125.

3.4 Computing Jacobians

Instead of computing the system Jacobian $\mathbf{J}$ (2.4) directly, we compute a Jacobian of Taylor coefficients $\partial \mathbf{f}_{I_k} / \partial \mathbf{x}_{J_k}$.

3.4.1 Computing $\partial \mathbf{f}_{I_k} / \partial \mathbf{x}_{J_k}$

We apply the forward mode of AD to differentiate each component of $\mathbf{f}_{I_k}$ with respect to $\mathbf{x}_{J_k}$. We first set all gradients of each component of $\mathbf{x}_{J_{\leq k}}$ corresponding to $\mathbf{x}_{J_k}$ to 0, and then set the gradients of each component of $\mathbf{x}_{J_k}$ corresponding to itself to 1. Then we propagate gradients through the code list[2] of $\mathbf{f}_{I_k}$. Finally, we can evaluate

[2]The code list means a sequence of expressions containing elementary arithmetic operations and standard functions. See also Appendix B.

$\partial \mathbf{f}_{I_k}/\partial \mathbf{x}_{J_k}$ with the computation of $\mathbf{f}_{I_k}$.

The above explanation is based on how FADBAD++ computes Jacobians, while ADOL-C has a different scheme presented in [GW04]. The method here is not efficient but convenient to implement; a source code translation approach is given in [NP03], which is more efficient, but also more difficult to implement.

Example 3.2. For the single pendulum, we want to compute $\partial \mathbf{f}_{I_0}/\partial \mathbf{x}_{J_0}$. At stage $k = 0$, we have

$$\begin{aligned} \mathbf{f}_{I_0} &= (f_0, g_0, h_2)^T, \\ \mathbf{x}_{J_0} &= (x_2, y_2, \lambda_0)^T, \\ \mathbf{x}_{J_{\leq 0}} &= (x_0, x_1, x_2, y_0, y_1, y_2, \lambda_0)^T, \end{aligned}$$

and the system equations satisfy (2.8).

First, we initialize $\partial \mathbf{x}_{J_{\leq 0}}/\partial \mathbf{x}_{J_0}$ to

∇x_0	$(0,0,0)$
∇x_1	$(0,0,0)$
∇x_2	$(1,0,0)$
∇y_0	$(0,0,0)$
∇y_1	$(0,0,0)$
∇y_2	$(0,1,0)$
$\nabla \lambda_0$	$(0,0,1)$

Then, we evaluate $\partial \mathbf{f}_{I_0}/\partial \mathbf{x}_{J_0} = (\nabla f_0, \nabla g_0, \nabla h_2)^T$ as shown in Tables 3.1, 3.2, and 3.3.

$t_1 = x_2$	$\nabla t_1 = \nabla x_2$	$(1,0,0)$
$t_2 = x_0$	$\nabla t_2 = \nabla x_0$	$(0,0,0)$
$t_3 = \lambda_0$	$\nabla t_3 = \nabla \lambda_0$	$(0,0,1)$
$t_4 = 2t_1$	$\nabla t_2 = 2\nabla x_2$	$(2,0,0)$
$t_5 = t_2 t_3$	$\nabla t_5 = t_2 \nabla t_3 + t_3 \nabla t_2$	$(0,0,x_0)$
$t_6 = t_4 + t_5$	$\nabla t_6 = \nabla t_4 + \nabla t_5$	$(2,0,x_0)$
$f_0 = t_6$	$\nabla f_0 = \nabla t_6$	$(2,0,x_0)$

Table 3.1: Evaluation of the gradient code list for $f_0 = 2x_2 + \lambda_0 x_0$.

$t_7 = y_2$	$\nabla t_7 = \nabla y_2$	$(0,1,0)$
$t_8 = y_0$	$\nabla t_8 = \nabla y_0$	$(0,0,0)$
$t_9 = \lambda_0$	$\nabla t_9 = \nabla \lambda_0$	$(0,0,1)$
$t_{10} = 2t_7$	$\nabla t_{10} = 2\nabla t_7$	$(0,2,0)$
$t_{11} = t_8 t_9$	$\nabla t_{11} = t_8 \nabla t_9 + t_9 \nabla t_8$	$(0,0,y_0)$
$t_{12} = t_{10} + t_{11}$	$\nabla t_{12} = \nabla t_{10} + \nabla t_{11}$	$(0,2,y_0)$
$g_0 = t_{12}$	$\nabla g_0 = \nabla t_{12}$	$(0,2,y_0)$

Table 3.2: Evaluation of the gradient code list for $g_0 = 2y_2 + \lambda_0 y_0 - G$.

$t_{13} = x_0$	$\nabla t_{13} = \nabla x_0$	$(0,0,0)$
$t_{14} = x_1$	$\nabla t_{14} = \nabla x_1$	$(0,0,0)$
$t_{15} = x_2$	$\nabla t_{15} = \nabla x_2$	$(1,0,0)$
$t_{16} = y_0$	$\nabla t_{16} = \nabla y_0$	$(0,0,0)$
$t_{17} = y_1$	$\nabla t_{17} = \nabla y_1$	$(0,0,0)$
$t_{18} = y_2$	$\nabla t_{18} = \nabla y_2$	$(0,1,0)$
$t_{19} = 2t_{13}$	$\nabla t_{19} = 2\nabla t_{13}$	$(0,0,0)$
$t_{20} = t_{19} t_{15}$	$\nabla t_{20} = t_{19} \nabla t_{15} + t_{15} \nabla t_{19}$	$(2x_0,0,0)$
$t_{21} = t_{14} t_{14}$	$\nabla t_{21} = 2t_{14} \nabla t_{14}$	$(0,0,0)$
$t_{22} = 2t_{16}$	$\nabla t_{22} = 2\nabla t_{16}$	$(0,0,0)$
$t_{23} = t_{22} t_{18}$	$\nabla t_{23} = t_{22} \nabla t_{18} + t_{18} \nabla t_{22}$	$(0,2y_0,0)$
$t_{24} = t_{17} t_{17}$	$\nabla t_{24} = 2t_{17} \nabla t_{17}$	$(0,0,0)$
$t_{25} = t_{20} + t_{21} + t_{23} + t_{24}$	$\nabla t_{25} = \nabla t_{20} + \nabla t_{21} + \nabla t_{23} + \nabla t_{24}$	$(2x_0,2y_0,0)$
$h_2 = t_{25}$	$\nabla h_2 = \nabla t_{25}$	$(2x_0,2y_0,0)$

Table 3.3: Evaluation of the gradient code list for $h_2 = 2x_0x_2 + x_1^2 + 2y_0y_2 + y_1^2$.

Thus, we obtain

$$\frac{\partial \mathbf{f}_{I_0}}{\partial \mathbf{x}_{J_0}} = \begin{bmatrix} \nabla f_0 \\ \nabla g_0 \\ \nabla h_2 \end{bmatrix} = \begin{bmatrix} 2 & 0 & x_0 \\ 0 & 2 & y_0 \\ 2x_0 & 2y_0 & 0 \end{bmatrix}. \tag{3.4}$$

In HIDAETS, we use FADBAD++ as our default AD package. It implements automatic differentiation through operator overloading. It provides data types F<double> for the forward mode of AD and T<double> for the Taylor series computation. The T<double> type can be built on top of F<double>. Thus, when a Taylor series computation is performed on F<double> objects, both TCs and their gradients are produced.

Remark. In this computation, FADBAD++ performs operations with gradients containing only zeros. If such operations are recognized, for example, through "if" statements, and avoided, the computation may be more efficient.

COMPUTE-JACOBIAN(k, d)
1 $\frac{\partial \mathbf{x}_{J_{\leq k}}}{\partial \mathbf{x}_{J_k}} \leftarrow \mathbf{0}$
2 **for** $j \leftarrow 1$ **to** n
3 **do** $L_j \leftarrow k + d_j$
4 **if** $L_j \geq 0$
5 **then** $\frac{\partial (x_j)_{L_j}}{\partial (x_j)_{L_j}} = 1$
6 Evaluate $\partial \mathbf{f}_{I_k} / \partial \mathbf{x}_{J_k}$ by computing $\mathbf{f}_{I_k}$ with T< F<double> > objects

The corresponding C++ code is on page 127.

3.4.2 The relation between $\partial \mathbf{f}_{I_k} / \partial \mathbf{x}_{J_k}$ and J

From COMPUTE-JACOBIAN, we can compute $\partial \mathbf{f}_{I_k} / \partial \mathbf{x}_{J_k}$ for any k. How to form the system Jacobian from it? Do we really need to compute Jacobian for all k? Nedialkov and Pryce [NP03] present some useful propositions to answer these questions. Here, we first quote these propositions, and then draw our algorithms for printing Jacobian $\mathbf{J}_k$ as defined in (3.5).

From Proposition 4.1 in [Pry01], we have

$$\mathbf{J} = \frac{\partial f_{I_0}}{\partial x_{J_0}} \quad \text{and} \quad \mathbf{J}_k = \frac{\partial f_{I_k}}{\partial x_{J_k}}, \tag{3.5}$$

and $\mathbf{J}_k$ is the submatrix of $\mathbf{J}$ by deleting those rows i where $k + c_i < 0$ and columns j where $k + d_j < 0$. If $k \geq 0$, then $\mathbf{J}_k = \mathbf{J}$, since all $k + c_i \geq 0$ and $k + d_j \geq 0$.

From [NP03], the relation between $\partial \mathbf{f}_{I_k} / \partial \mathbf{x}_{J_k}$ and $\mathbf{J}$ is

$$\frac{\partial \mathbf{f}_{I_k}}{\partial \mathbf{x}_{J_k}} = C_k^{-1} \mathbf{J} D_k \quad \text{or} \quad \mathbf{J} = C_k \frac{\partial \mathbf{f}_{I_k}}{\partial \mathbf{x}_{J_k}} D_k^{-1}, \tag{3.6}$$

where C_k, D_k are diagonal matrices defined by

$$C_k = \operatorname{diag}\big[(k + c_i)! \mid \text{for those } i \text{ with } k + c_i \geq 0\big],$$
$$D_k = \operatorname{diag}\big[(k + d_j)! \mid \text{for those } j \text{ with } k + d_j \geq 0\big].$$

Example 3.3. For the single pendulum, when $k = 0$,

$$C_0 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 2 \end{bmatrix}, \quad D_0 = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Thus, we just need to compute $\partial \mathbf{f}_{I_k} / \partial \mathbf{x}_{J_k}$ for $k = k_d, \ldots, 0$. Then we can compute the system Jacobian $\mathbf{J}$ from $\partial \mathbf{f}_{I_0} / \partial \mathbf{x}_{J_0}$, and obtain $\partial \mathbf{f}_{I_k} / \partial \mathbf{x}_{J_k}$ for $k > 0$ by scaling $\mathbf{J}$.

Example 3.4. For the single pendulum, from (3.4) and (3.6), we have

$$\begin{aligned} \mathbf{J} &= C_0 \frac{\partial \mathbf{f}_{I_0}}{\partial \mathbf{x}_{J_0}} D_0^{-1} \\ &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 2 \end{bmatrix} \begin{bmatrix} 2 & 0 & x_0 \\ 0 & 2 & y_0 \\ 2x_0 & 2y_0 & 0 \end{bmatrix} \begin{bmatrix} \frac{1}{2} & 0 & 0 \\ 0 & \frac{1}{2} & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & x_0 \\ 0 & 1 & y_0 \\ 2x_0 & 2y_0 & 0 \end{bmatrix}. \end{aligned}$$

In fact, if just obtaining system Jacobian and $\partial \mathbf{f}_{I_k} / \partial \mathbf{x}_{J_k}$, we only need to compute $\partial \mathbf{f}_{I_0} / \partial \mathbf{x}_{J_0}$. In general, from Pryce's structural analysis, the values of $\mathbf{x}_{J_{\leq 0}}$ cannot be determined without processing stages $k \leq 0$, so we need to compute $\partial \mathbf{f}_{I_k} / \partial \mathbf{x}_{J_k}$ for $k = k_d, \ldots, 0$ to obtain a consistent initial point.

PRINT-JACOBIAN(k, c, d)
1 **for** $i \leftarrow 0$ **to** n
2 **do** $L_i = k + c_i$
3 **if** $L_i \geq 0$
4 **then for** $j \leftarrow 0$ **to** n
5 **do** $L_j = k + d_j$
6 **if** $L_j \geq 0$
7 **then** Print $\dfrac{L_i!}{L_j!} \dfrac{\partial(f_i)_{L_i}}{\partial(x_j)_{L_j}}$

The corresponding C++ code is on page 127.

Note that the Jacobian can be obtained in sparse or dense format. In HIDAETS, we obtain it in both formats and implement it by function overloading.

3.5 Computing TCs

From Pryce's structural analysis [Pry98, Pry01], for stages $k \geq k_d$, the $\mathbf{f}_{I_{\leq k}}$ functions of $\mathbf{x}_{J_{\leq k}}$, must satisfy

$$\mathbf{f}_{I_k}(t^*, \mathbf{x}^*_{J_{<k}}, \mathbf{x}^*_{J_k}) = 0.$$

At each stage $k = k_d, k_d + 1, \ldots$, the $\mathbf{x}^*_{J_{<k}}$ have already been solved, and the $\mathbf{x}^*_{J_k}$ are the unknowns.

From this structural analysis, stages $k \leq 0$ form nonlinear systems in general to comprise the projection on the constraints manifolds, while stages $k > 0$ form square linear systems only.

3.5.1 Computing a consistent point

For stages $k \leq 0$, we solve the system[3]

$$\min \|\mathbf{x}^a_{J_k} - \mathbf{x}_{J_k}\|_2 \quad \text{subject to} \quad \mathbf{f}_{I_k}(t^*, \mathbf{x}^*_{J_{<k}}, \mathbf{x}_{J_k}) = \mathbf{f}_{I_k}(\mathbf{x}_{J_k}) = 0. \tag{3.7}$$

Example 3.5. For the simple pendulum, at stage $k = -2$,

$$\mathbf{f}_{I_{-2}} = h_0 \quad \text{and} \quad \mathbf{x}_{J_{-2}} = (x_0, y_0)^T.$$

[3] $\mathbf{f}_{I_k}$ may be linear in $\mathbf{x}_{J_k}$. However, we still solve (3.7) to obtain a consistent point.

Given an initial guess $\mathbf{x}^a_{J_{-2}}$ for $\mathbf{x}_{J_{-2}}$, we solve

$$\min \|\mathbf{x}^a_{J_{-2}} - \mathbf{x}_{J_{-2}}\|_2 \quad \text{subject to} \quad \mathbf{f}_{I_{-2}} = x_0^2 + y_0^2 = 0.$$

In HIDAETS, we employ IPOPT [WB04] to solve (3.7). To use IPOPT, we need to provide the objective function, the gradients of the objective function, the evaluation of the constraints, and the Jacobian of the constrains. For simplicity, we replace $\|\mathbf{x}^a_{J_{-k}} - \mathbf{x}_{J_{-k}}\|_2$ by $0.5\|\mathbf{x}^a_{J_{-k}} - \mathbf{x}_{J_{-k}}\|_2$, then the gradient becomes $\mathbf{x}^a_{J_{-k}} - \mathbf{x}_{J_{-k}}$. We use AD to compute the constraints and their Jacobians, as illustrated in subsection 3.4.2.

Solving (3.7) from $k = k_d$ to $k = 0$ in order, we obtain a consistent initial point. In fact, at each integration step, data is available in the form of "guesses" $\mathbf{x}^a_{J_{\leq 0}}$ for the desired values of $\mathbf{x}^*_{J_{\leq 0}}$. Thus, we use (3.7) to compute a consistent point for both the first step and each integration step afterwards.

COMPUTE-CONSISTENT-POINT$(k, \mathbf{x}^a_{J_k})$
1 Define objective function $0.5\,||\mathbf{x}^a_{J_k} - \mathbf{x}_{J_k}||_2$
2 Define constraints function $\mathbf{f}_{I_k}(t^*, \mathbf{x}^*_{J_{<k}}, \mathbf{x}_{J_k})$
3 Define functions needed by optimization package
4 Compute optimal solution by an optimization package

The corresponding C++ code is on page 128.

3.5.2 The linear case

Let

$$\mathbf{z}_k = \mathbf{x}^a_{J_k} - \mathbf{x}^*_{J_k}.$$

When $\mathbf{f}_{I_k}$ is linear in $\mathbf{x}_{J_k}$, from

$$0 = \mathbf{f}_{I_k}(t^*, \mathbf{x}^*_{J_{<k}}, \mathbf{x}^*_{J_k}) = \mathbf{f}_{I_k}(t^*, \mathbf{x}^*_{J_{<k}}, \mathbf{x}^a_{J_k}) - \frac{\partial \mathbf{f}_{I_k}}{\partial \mathbf{x}_{J_k}}(t^*, \mathbf{x}^*_{J_{<k}}, \mathbf{x}^a_{J_k})\mathbf{z}_k,$$

we have

$$\frac{\partial \mathbf{f}_{I_k}}{\partial \mathbf{x}_{J_k}}(t^*, \mathbf{x}^*_{J_{<k}}, \mathbf{x}^a_{J_k})\mathbf{z}_k = \mathbf{f}_{I_k}(t^*, \mathbf{x}^*_{J_{<k}}, \mathbf{x}^a_{J_k}). \tag{3.8}$$

From (3.6), we transform (3.8) into

$$C_k^{-1}\mathbf{J}D_k\mathbf{z}_k = \mathbf{f}_{I_k}(t^*, \mathbf{x}^*_{J_{<k}}, \mathbf{x}^a_{J_k}). \tag{3.9}$$

If $\partial \mathbf{f}_{I_k}/\partial \mathbf{x}_{J_k}$ is nonsingular, we can solve (3.8) for $\mathbf{z}_k$ in the usual sense. Then,

$$\mathbf{x}^*_{J_k} = \mathbf{x}^a_{J_k} - \mathbf{z}_k.$$

Example 3.6. At stage $k = 1$,

$$\mathbf{f}_{I_1} = (f_1, g_1, h_3)^T \quad \text{and} \quad \mathbf{x}_{J_1} = (x_3, y_3, \lambda_1)^T.$$

We have

$$\begin{aligned}
\frac{\partial \mathbf{f}_{I_1}}{\partial \mathbf{x}_{J_1}} &= C_1^{-1} \mathbf{J} D_1 \\
&= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & \frac{1}{3} \end{bmatrix} \begin{bmatrix} 1 & 0 & x_0^* \\ 0 & 1 & y_0^* \\ 2x_0^* & 2y_0^* & 0 \end{bmatrix} \begin{bmatrix} 3 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} 3 & 0 & x_0^* \\ 0 & 3 & y_0^* \\ 3x_0^* & 3y_0^* & 0 \end{bmatrix}.
\end{aligned}$$

Since $\partial \mathbf{f}_{I_1}/\partial \mathbf{x}_{J_1}$ is nonsingular, we compute $\mathbf{z}_1$ by solving the 3×3 system

$$\frac{\partial \mathbf{f}_{I_1}}{\partial \mathbf{x}_{J_1}} \mathbf{z}_1 = \mathbf{f}_{I_1}(\mathbf{x}^*_{J_{<1}}, \mathbf{x}^a_{J_1}).$$

Hence, we need first to solve $\mathbf{z}_k$ for the linear case. Generally, we first obtain the system Jacobian without scaling. Then we scale it following (3.6) to obtain $\partial \mathbf{f}_{I_k}/\partial \mathbf{x}_{J_k}$. We compute $\mathbf{f}_{I_k}(\mathbf{x}^*_{J_{<k}}, \mathbf{x}^a_{J_k})$ with an initial guess $\mathbf{x}^a_{J_k}$ for $\mathbf{x}_{J_k}$ by AD. After that, we solve (3.8) to obtain $\mathbf{z}_k$. Finally, we compute $\mathbf{x}^*_{J_k}$ by

$$\mathbf{x}^*_{J_k} = \mathbf{x}^a_{J_k} - \mathbf{z}_k.$$

However, to solve (3.8), we need to do LU factorization for $\partial \mathbf{f}_{I_k}/\partial \mathbf{x}_{J_k}$ first, whose running time is $O(n^3)$. If we do it for each stage k, it is obviously not efficient. We want to do LU factorization once for all stages, and then do some scaling to solve $\mathbf{x}_{J_k}$ for each stage k.

We denote $m_c = \max_i c_i$ and by $\|\cdot\|$ the max norm of a vector or a matrix. Then $\|C_k\| = (k + m_c)!$ and

$$\begin{aligned}
\frac{1}{\|C_k\|} C_k &= \operatorname{diag}\left[\frac{(k+c_1)!}{(k+m_c)!}, \ldots, \frac{(k+c_n)!}{(k+m_c)!}\right], \\
\|C_k\| \, C_k^{-1} &= \operatorname{diag}\left[\frac{(k+m_c)!}{(k+c_1)!}, \ldots, \frac{(k+m_c)!}{(k+c_n)!}\right].
\end{aligned}$$

For the pendulum problem,

$$\begin{aligned}\frac{1}{\|C_k\|}C_k &= \operatorname{diag}\left[\frac{k!}{(k+2)!}, \frac{k!}{(k+2)!}, \frac{(k+2)!}{(k+2)!}\right] \\ &= \operatorname{diag}\left[\frac{1}{(k+1)(k+2)}, \frac{1}{(k+1)(k+2)}, 1\right],\end{aligned}$$

$$\begin{aligned}\|C_k\|\, C_k^{-1} &= \operatorname{diag}\left[\frac{(k+2)!}{k!}, \frac{(k+2)!}{k!}, \frac{(k+2)!}{(k+2)!}\right] \\ &= \operatorname{diag}\big[(k+1)(k+2), (k+1)(k+2), 1\big].\end{aligned}$$

Similar for $D_k/\|C_k\|$ and $\|C_k\|\, D_k^{-1}$.

Multiplying both sides of (3.9) by $C_k/\|C_k\|$, we obtain

$$\mathbf{J}\frac{1}{\|C_k\|}D_k\mathbf{z}_k = \frac{1}{\|C_k\|}C_k\mathbf{f}_{I_k}(\mathbf{x}^*_{J_{<k}}, \mathbf{x}^a_{J_k}).$$

Let $\mathbf{y}_k = D_k/\|C_k\|\, \mathbf{z}_k$, then

$$\mathbf{J}\mathbf{y}_k = \frac{1}{\|C_k\|}C_k\mathbf{f}_{I_k}(\mathbf{x}^*_{J_{<k}}, \mathbf{x}^a_{J_k}). \tag{3.10}$$

To solve (3.10), we do not need to do LU factorization for each k. We just need to do LU factorization of $\mathbf{J}$ once, and then use it directly for each stage k.

Finally, we compute $\mathbf{x}^*_{J_k}$ by

$$\mathbf{x}^*_{J_k} = \mathbf{x}^a_{J_k} - \|C_k\|\, D_k^{-1}\mathbf{y}_k.$$

For the Taylor series, we repeat the above process from 1 to the required order p.

COMPUTE-TS(n, c, d, x, p)

```
1  J ← System Jacobian at stage 0
2  compute LU factorization of J
3  for i ← 1 to n
4      do u_i ← 1/c_i!
5         v_i ← 1/d_i!
6  for k ← 1 to p
```

```
7     do COMPUTE-TERM(n, x, k, u, v)
```

```
COMPUTE-TERM(n, x, k, u, v)
1  for i ← 1 to n
2      do u_i ← u_i/(k + c_i)
3         v_i ← v_i/(k + d_i)
4  for i ← 1 to n
5      do g_i ← compute (f_i)_{k+c_i}
6         g_i ← g_i/(u_i · (k + m_c)!)
7  s ← solution of (3.10) using the LU factors
8  for j ← 1 to n
9      do (x_j)_{k+d_j} ← −s_j v_i (k + m_c)!
```

The corresponding C++ code is on page 130.

3.6 Error estimation

If the x_j have infinite Taylor series expansions, for a given $h = t - t^*$,

$$x_j^{(l)}(t) = \sum_{k=0}^{\infty} \frac{x_j^{(k+l)}(t^*)}{k!}(t-t^*)^k = \sum_{k=0}^{\infty} \frac{x_j^{(k+l)}(t^*)}{k!}h^k.$$

If we approximate $x_j^{(l)}$ by a Taylor series with highest-order term of order p, we can write

$$x_j^{(l)}(t) = \sum_{k=0}^{p} \frac{x_j^{(k+l)}(t^*)}{k!}h^k + \sum_{k=p+1}^{\infty} \frac{x_j^{(k+l)}(t^*)}{k!}h^k, \tag{3.11}$$

$$\widehat{x}_j^{(l)}(t) = \sum_{k=0}^{p} \frac{x_j^{(k+l)}(t^*)}{k!}h^k. \tag{3.12}$$

For simplicity, we shall omit t in $x_j^{(l)}(t)$ and $\widehat{x}_j^{(l)}(t)$, and use $x_j^{(l)}$ and $\widehat{x}_j^{(l)}$, respectively.

In general, from (3.11) and (3.12), we can use $|x_j^{(p+1+i)}|/(p+1)!\,|h|^{p+1}$ as the error estimation for $\widehat{x}_j^{(i)}$.

In HIDAETS, we compute $\widehat{x}_j^{(i)}$ by

$$\widehat{x}_j^{(i)} = \sum_{k=i}^{p+d_j} \frac{x_j^{(k)}}{(k-i)!} h^{k-i}$$

and we use $|x_j^{(p+d_j)}|/(p+d_j-i)!\,|h|^{p+d_j-i}$ as the error estimation for $\widehat{x}_j^{(i)}$ at t.

Since $x_j^{(l)}$, for all $j = 1, \ldots, n$ and all $l = 0, \ldots, d_j$, are used to obtain an approximate solution, we must consider all these $x_j^{(l)}$ when estimating the error.

Hence, an error estimation for $\widehat{x}_j^{(l)}$ is $|x_j^{(p+d_j)}|/(p+d_j-l)!\,|h|^{p+d_j-l}$, where $l = 0, \ldots, d_j$. Usually, $|h| \ll p$. We have

$$|h|^p/p! > |h|^{p+1}/(p+1)! > \ldots > |h|^{p+d_j}/(p+d_j)!$$

This implies

$$\frac{|x_j^{(p+d_j)}|}{(p+d_j-l)!}|h|^{p+d_j-l} < \frac{|x_j^{(p+d_j)}|}{p!}|h|^p, \quad \text{for all} \quad l = 0, \ldots, d_j.$$

Then we can use

$$e_j = \frac{|x_j^{(p+d_j)}|}{p!}|h|^p$$

as an error estimation for $x_j^{(l)}$, for all $l = 0, \ldots, d_j$. The error for the whole system is $\|e\|$.

Therefore, we can form error estimate for the current step as

$$\text{est} = \|e\|, \quad \text{where} \quad e = \frac{|h|^p}{p!}\left(|x_1^{(p+d_1)}|, \ldots, |x_n^{(p+d_n)}|\right)^T. \tag{3.13}$$

Since the variables are represented as Taylor coefficients in HIDAETS, by replacing derivatives by TCs in (3.13), we have

$$e = \frac{|h|^p}{p!}\left(|(x_1)_{p+d_1}(p+d_1)!|, \ldots, |(x_n)_{p+d_n}(p+d_n)!|\right)^T. \tag{3.14}$$

ESTIMATE-ERROR(n, x, d, h, p)

```
1  for j ← 1 to n
2      do e_j ← |(x_j)_{p+d_j}||h|^p (p+1) ··· (p+d_j)
3  return NORM(e)
```

The corresponding C++ code is on page 132.

3.7 Stepsize control

3.7.1 Stepsize selection

Tolerance computation

We use a mixture of relative and absolute error control. The user gives absolute tolerance atol and relative tolerance rtol. Then

$$\text{tol} = \text{rtol}\,\|x_{J_{\leq 0}}\| + \text{atol}.$$

COMPUTE-TOLERANCE$(n, x, d, \text{atol}, \text{rtol})$

1 **for** $j \leftarrow 1$ **to** n
2 **do** $\beta_j \leftarrow \text{rtol} \left\|\left(x_j^{(0)}, \ldots, x_j^{(d_j)}\right)\right\|$
3 $\text{tol} \leftarrow \text{atol} + \|\beta\|$
4 **return** tol

The corresponding C++ code is on page 132.

Selecting stepsize

After obtaining a consistent initial point, we compute the Taylor coefficients at current integration time. Then we need the error estimate to be $\leq$ tol for this step. From (3.13) and (3.14), we have

$$\gamma\,|h|^p \leq \text{tol},$$

where,

$$\gamma = \frac{1}{p!}\left\|\left((x_1)_{p+d_1}(p+d_1)!, \ldots, (x_n)_{p+d_n}(p+d_n)!\right)^T\right\|.$$

Since we have already obtained a consistent initial point, we can compute tol. We can also compute γ by the Taylor coefficients which are knowns. Then, we use formula

$$h = (\text{tol}/\gamma)^{1/p}$$

to compute the stepsize.

COMPUTE-STEPSIZE(n, x, d, p, tol)

1 $\gamma \leftarrow$ ESTIMATE-ERROR$(n, x, d, 1.0, p)$
2 $h \leftarrow (\text{tol}/\gamma)^{1/p}$
3 **return** h

The corresponding C++ code is on page 133.

3.7.2 Final stepsize selection

About the final step, one may think of using $t_{\text{end}} - t$ as the final stepsize. However, it may be too small. In HIDAETS, after computing the stepsize h, we compare $|t_{\text{end}} - t|$ with $|h|$. Table 3.4 shows how we deal with different conditions. Here,

Condition	Stepsize
$2\|h\| < \|t_{\text{end}} - t\|$	$\text{sign}(t_{\text{end}} - t)\|h\|$
$\|h\| < \|t_{\text{end}} - t\| \leq 2\|h\|$	$(t_{\text{end}} - t)/2$
$\|h\| \geq \|t_{\text{end}} - t\|$	$t_{\text{end}} - t$

Table 3.4: Selection of the final stepsize.

$$\text{sign}(x) = \begin{cases} 1 & \text{when} \quad x \geq 0, \\ -1 & \text{when} \quad x < 0. \end{cases}$$

We use $|h|$ to ensure that the integration handles special cases such as $h < 0$, or $t_{\text{end}} < t_0$. For instance, we could integrate the single pendulum problem from $t_0 = 100$ to $t_{\text{end}} = 0$. Note that this is the first time we use $|h|$, and we use h in COMPUTE-STEPSIZE.

Besides these, we must ensure that the stepsize does not become too small. We set a threshold hmin in HIDAETS. After computing the final stepsize or reducing stepsize in the integration process as illustrated in subsection 3.2, we check whether $h \geq$ hmin. If $h <$ hmin, HIDAETS sets Ind, an indicator defined in subsection 4.1, appropriately and exits.

CHECK-FINAL-STEPSIZE(t, t_{end}, h)
1 **if** $2|h| < |t_{\text{end}} - t|$

```
2     then if t_end > t
3             then return |h|
4             else  return  − |h|
5     else  if |h| < |t_end − t|
6             then return (t_end − t)/2
7             else  return t_end − t
```

The corresponding C++ code is on page 133.

Chapter 4

Numerical Software

In this chapter, we follow general software development process to describe HIDAETS as we did in developing it. First we give an informal specification for HIDAETS, then discuss design issues, and finally illustrate how to install and use it.

4.1 Informal specification

The requirement specification is organized as follows. Section 4.1.1 provides the overall description of the system to make the requirements in section 4.1.2 easier to understand. Section 4.1.2 contains all detailed requirements. Section 4.1.3 includes supporting information important to a system's development. Section 4.1.4 lists all possible changes in the software development life cycle.

4.1.1 General system description

This section gives a general description of the system.

System context

Figure 4.1 shows the sketch of the system context. A circle represents an external entity outside the system and a rectangle is the system itself. Arrows represent the data flows between the system and its environment.

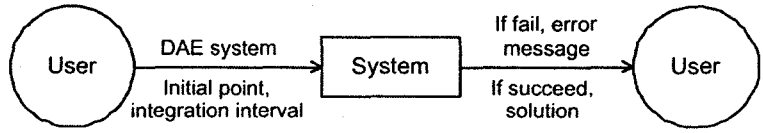


Figure 4.1: Sketch of system context diagram.

This system is independent and self-contained. The main external interaction of the system is the user interface. The responsibilities of the user and the system are listed as follows.

- User responsibilities:
  1. Prepare a set of input information including system equations, an initial point, and the integration interval.
  2. Assure there is not input data error caused by human oversight like typing mistakes.
- System responsibilities:
  1. Decide whether the problem is solvable. If not, output relevant error message.
  2. If the problem solvable, compute a numerical solution at the end point of the integration interval.

User characteristics

The typical user would be from an Engineering area who may have no knowledge about the algorithms and the programming languages of the software system. The user must have the necessary prerequisite knowledge and skills to use the system. The knowledge and skills include defining the DAE problem, providing the system input following the instructions, and running script files.

System constraints

The system will employ Pryce's structural analysis and Taylor series methods. To compute Taylor coefficients, we use automatic differentiation.

In numerical methods for ODEs/DAEs, we normally want a solution at an end point. In this work, we do not provide a continuous numerical solution.

The software system needs several numerical packages, such as AD, optimization, and linear assignment problem. Table 4.1 lists these packages.

Name	Description
ADOL-C	AD package in C/C++
FADBAD++	AD package in C/C++
IPOPT	optimization package in Fortran with C interface
LAP	linear assignment problem solver in C/C++
LAPACK	subroutines for solving linear systems in Fortran

Table 4.1: Packages in HIDAETS

For these external packages, the user needs to download and install them following their instructions.

As a numerical software package, considering efficient and usability, we usually choose C/C++ or Fortran to implement it. Since ADOL-C, FADBAD++, and LAP only have C/C++ interfaces, it is very difficult to call C/C++ function from Fortran. Therefore, we implement the system in C/C++ language.

We want to use a free compiler which is easy to manage and close to ANSI C++, thus we choose g++ 3.2.

The system is implemented and tested in a Unix environment and is expected to be portable to other Unixes, Linux, and MacOS X.

4.1.2 System description

This section describes the system requirements in detail.

Functional requirements

- Problem description

  A DAE initial value problem includes the following items.

  1. System of equations

     The problem is a DAE initial value problem in the form of (1.1).

The number of equations must match the number of dependent variables in the DAE system. The functions are assume to be real-valued.

This system is represented in a computer program using a finite number of constants, variables, elementary operations (+, −, ∗, and /), differentiation operators, and smooth functions (sin, cos, exp, log, etc.). The representation excludes nonsmooth functions, such as, branches, abs, and min.

2. Set of initial values

   The initial values for the DAE system. Generally, this is a subset of $x_{J_{\leq 0}}$. The values are in the floating point format.

3. Integration interval

   $[t_0, t_{\text{end}}]$. The values are in the floating point format. t_0 is the starting point of the integration interval, and t_{end} is the end point of the integration interval. Here, t_{end} can be less than t_0, such as integrating the problem from $t_0 = 0$ to $t_{\text{end}} = -10$.

- Goals statement

Basic. We want to solve numerically the DAE initial value problem described above. If the system fails to solve the problem, it outputs the relative error message. If it succeeds, it computes the solution at t_{end}, and outputs relative information as described in the output requirement.

Besides these, the system could take the user's optional input as specified below.

Advanced. Since the system needs some external packages, we also allow the "expert" user, familiar with the particular algorithms and the C++ programming language to substitute a package with another one providing the same functionality. The exchangeable packages are specified in section 4.1.4.

- Optional input

1. Tolerances

Absolute and relative error tolerances, which indicate how accurately the solution is to be computed. The values are in the floating point format. Default values are 10^{-13}.

2. Order

   This is Taylor series order. The values are in the integer format. Default value is 20.

3. Indicator

   On initial entry, it must be set to tillend, onestep, or jacobian. Its default value is tillend. The user can re-enter the integration with Ind = tillend, onestep, or Jacobian. Ind = Jsingular, conspoint, and smallstepsize indicate errors as shown below.

Ind	DAE solver
tillend	integrates from current t to t_{end}
onestep	takes a step and exits
jacobian	computes Σ, $\mathbf{c}, \mathbf{d}$, $\mathbf{J}$ and exits
normalexit	normal exit
Jsingular	$\mathbf{J}$ is singular
conspoint	$\mathbf{J}$ is nonsingular, fails to obtain a consistent point
smallstepsize	stepsize is too small

4. Output parameters

   The integration system output includes several levels, as listed below:

outpuInd	DAE solver
solution	outputs the solution at t_{end}
stat	solution + statistics information
process	stat + integration process
offsets	process + signature matrix and offsets
initpoint	offsets + consistent initial point
scheme	initpoint + solution scheme

- Printed output

Based on the system input, the output system has several levels.

1. Returns

   The system returns success or fail. If success, output information based on the output parameters. If fail, output the failure reason.

2. Solution at t_{end}

   Solution values and derivative values of the variables in the system. In practice, we output $x_{J_{\leq 0}}$ at t_{end}. These values are in the floating point format.

3. Statistics information

   It includes

   (a) number of steps

   (b) number of rejected steps

   (c) user CPU time

4. Signature matrix, degrees of freedom, and structural index

   This prints the system signature matrix, offsets, degree of freedom, and structural index.

5. Integration process

   This prints the system integration process: current integration time and the estimated local error.

6. Consistent initial point

   This prints the consistent initial point generated by the system. In practice, we output $x_{J_{\leq 0}}$ at t_0. These values are in the floating point format.

7. Solution scheme

   This part illustrates how to obtain a consistent initial point. It shows at each stage, which functions are used and for which variables are solved.

Non-functional requirement

- Accuracy of the input data

  The input data is given directly by the user of the system, assuming no human error, there are no input data error or measurement error. In other word, the input data of the system are assumed to be accurate.

- Tolerance of the solution

  It is always hard to find a true solution for the DAE problem we solve. We compare the solution with the reference solution generated by the system with small tolerances (for example, 10^{-16}), which is supposed correct. We can also compare our reference solution with other reference solutions presented in the literature or generated by other existing solver to assess the correctness of our reference solution.

- Solution validation strategies

  We compute the number of signature digits at the end point of the integration interval for different tolerances. If the number of significant digits increases with the tolerance's decreasing, the solution is assumed reliable.

- Look and feel requirements

  The user's input interface is to define the problem by implementing relative C++ functions almost in mathematical form. The output interface should be straightforward enough to provide the information.

- Usability requirements

  The system should be easy to learn and use, and it should take a considerable amount of time for a user with the necessary background specified in subsection 4.1.1 to use the system to solve a problem.

- Performance requirements

  The system is facing to solve high-index DAE problems. It can solve the problems whose index is too high for existing solvers to handle. The system is also expected to be competitive to existing solvers when the problems can be solved by both and high accuracy (global error $\leq 10^{-10}$) is demanded.

- Maintainability requirements

  This system should be developed in a way that the efforts spent on maintaining the system (including document and code) or upgrading the system would be minimal. The release frequency of the system should be less than once within

one year. The time and effort for each upgrade should be less than 1/5 of the time and effort for developing the original system.

- Portability requirements

  The system should be easily ported to Mac OS, Linux, and Unix environment with g++ 3.2 or later. It depends on the g++ compiler and some Unix system functions (like time()), which are also available on Mac OS and Linux. Thus the system takes same time and effort to be installed on these operating systems.

4.1.3 Other system issues

Open issues

Due to roundoff and truncation error, our computed solutions are not the mathematically correct ones. If an interval computation can be applied, we will be able to assess the accuracy of the computed solution.

Waiting room

Due to the explicit nature of the Taylor series methods, the system is not efficient for stiff DAE problems. New algorithms and implementation needs to be developed for stiff problems.

The user can change algorithms in one part without influencing other parts. These parts include

1. Computing signature matrix and offsets. The algorithms for computing signature matrix from the evaluation of the DAE system and generating offsets from the signature matrix.

2. Projecting a consistent solution. The algorithms for computing a consistent point from an approximate one.

3. Computing TCs. The algorithms for computing Taylor coefficients to some specified order by automatic differentiation.

4. Computing Jacobians. The algorithms for computing the related Jacobians from the evaluation of the DAE system by automatic differentiation.

5. Error estimation. The algorithms for estimating the local error per step.

6. Stepsize control. The algorithms for predicting an appropriate stepsize.

4.1.4 Likely changes

Numerical packages

The user can substitute any one of the packages in the system by his/her own same functional packages without influencing others. These packages include

1. AD packages for computing TCs and Jacobians. If the package is a non-C++ based AD package, the interfaces for computing TCs and Jacobians must be possible to be called by C++. If the package is a non-overloading based AD package, it must take the function (in the computer program) used to generate the computational graph in the form of (1.1).

2. Optimization packages for projecting solutions. The projection problem is for $k \leq 0$, we solve the system

$$\min \|\mathbf{x}^a_{J_k} - \mathbf{x}_{J_k}\|_2 \quad \text{subject to} \quad \mathbf{f}_{I_k}(t^*, \mathbf{x}^*_{J_{<k}}, \mathbf{x}_{J_k}) = \mathbf{f}_{I_k}(\mathbf{x}_{J_k}) = 0$$

   to compute $\mathbf{x}^*_{J_k}$ by standard least-square methods.

3. Linear assignment problem solver for computing an HVT.

4. Linear algebra packages for solving linear system.

Printed output

The system output may have the following changes:

1. The order of output items. We define the levels for the output system for current version. It may need to change the order of the levels and contents in each level.

2. More output information. In the future, the system may need to output more information.

(a) Feedback to user's input. There may be human errors from the user, the output system should response with relevant indicating information.

(b) Plots of dependent variables over the integration interval.

(c) Plots of the stepsize behavior. The user can investigate the stepsize behavior, therefore, the stepsize control algorithms.

(d) Dynamic simulation of the problem. This will be useful in the real application. We can show the user dynamic simulation of the problem by animation or other graph representation method.

4.2 Design

In this section, we follow an object-oriented approach. First we decompose the system into components. Then, for each component, we display class diagrams and give detailed interfaces.

4.2.1 High-level design

This subsection describes the structure of HIDAETS. It consists of the following components: DAEProblem, IntegrationParameters, SignatureMatrix, Offsets, ConsistentInitialPoint, AD, ErrorEstimation, StepsizeSelection, and Projection. Figure 4.2 depicts the solver structure. We describe each part in turn.

In each interface description, we use the following form

Component	*Description*

where the column with *Component* is the name of the component using this interface, and the column with *Description* is the description how the component uses this interface.

1. DAEProblem

   This component obtains the definition of a DAE problem, which includes system equations, initial point x_{ICs}, t_0, and t_{end}.

   Interface: *computeDAE, getInitialPoint*

   (a) computeDAE

   This interface provides the system equations of the DAE problem. It is used by:

Component	*Description*
SignatureMatrix	It computes the signature matrix.
AD	It uses system equations to generate computational graphs for computing Taylor coefficients and Jacobians.

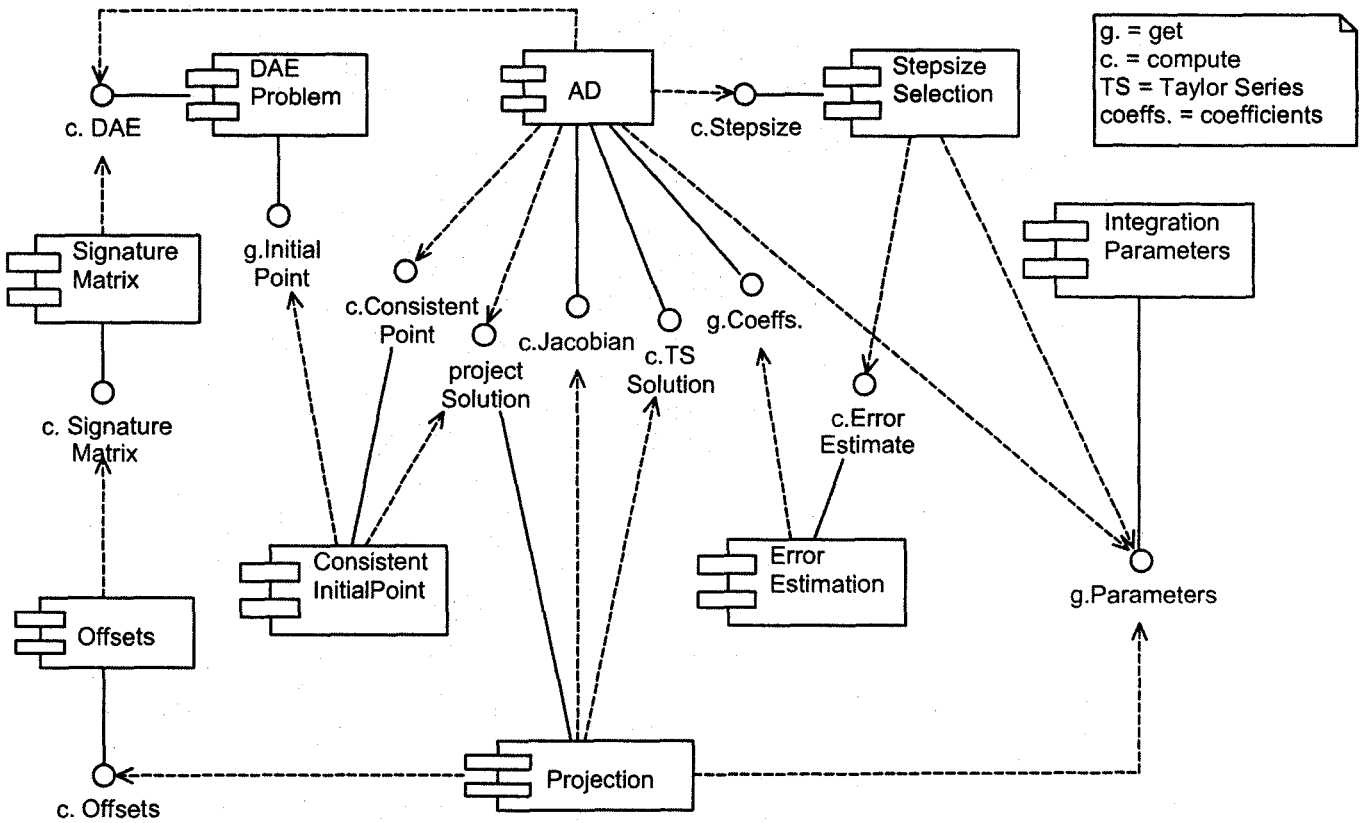


Figure 4.2: Structure of HIDAETS.

(b) getInitialPoint

This interface obtains the initial point from the user. It is used by:

Component	*Description*
ConsistentInitialPoint	It first checks whether the initial point is consistent; If not, it tries to find a consistent one.

2. IntegrationParameters

This component obtains integration parameters from the user. They include absolute tolerance (atol), relative tolerance (rtol), tolerance for optimization package (dtol[1]), order (Ord), indicator (Ind), and output control information (outputInd). More details are given in subsection 4.1.

Interface: *getParameters*

This interface provides the integration parameters. It is used by:

Component	*Description*
AD	It sets the order of Taylor series to Ord.
StepsizeSelection	It uses atol, rtol, and Ord when selecting a stepsize for next step.
Projection	It uses dtol when projecting an initial point or approximate TS solution.

3. SignatureMatrix

This component obtains the signature matrix from the definition of the DAE system.

Interface: *computeSignatureMatrix*

This interface computes the system signature matrix. It is used by:

Component	*Description*
Offsets	It computes offsets based on the generated signature matrix.

[1]In HIDAETS, we integrate IPOPT as our nonlinear system solver. When an error estimate [WB04] becomes less than dtol, IPOPT succeeds and returns with solution.

4. Offsets

   This component obtains offsets **c** and **d** from the signature matrix of the DAE system.

   Interface: *computeOffsets*

   This interface computes equation offsets and variable offsets. It is used by:

Component	*Description*
Projection	It uses offsets when projecting an initial point or TS solution.

5. ConsistentInitialPoint

   This component computes a consistent point based on the structural analysis. It checks whether the initial point given by the user is consistent; if not, it tries to find a consistent one.

   Interface: *computeConsistentPoint*

   This interface tries to find a consistent point. It is used by:

Component	*Description*
AD	It uses the consistent initial point to compute Taylor coefficients and Jacobians.

6. AD

   This component computes and provides Taylor coefficients, Taylor series solution, and Jacobians. It uses interfaces:

   (a) computeDAE from DAEProblem;

   (b) computeConsistentPoint from ConsistentInitialPoint;

   (c) computeStepsize from StepsizeSelection;

   (d) projectSolution from Projection;

   (e) getParameters from IntegrationParameters.

Interface: *getCoefficients, computeTSSolution, computeJacobian*

(a) getCoefficients

This interface provides the Taylor coefficients. It is used by:

Component	*Description*
ErrorEstimation	It estimates error for the approximate TS solution.

(b) computeJacobian

This interface computes Jacobians. It is used by:

Component	*Description*
Projection	It uses Jacobian when projecting an initial point or approximate solution.

(c) computeTSSolution

This interface computes Taylor series solution based on Taylor coefficients with a given stepsize and order. It is used by:

Component	*Description*
Projection	It uses TS solution when projecting a Taylor series solution.

7. ErrorEstimate

This component estimates the error of an approximate TS solution.

Interface: *computeErrorEstimate*

This interface estimates error from the Taylor coefficients. It is used by:

Component	*Description*
StepsizeSelection	It decides next stepsize based on the estimated error.

8. StepsizeSelection

   This component selects stepsize for the next integration step. It computes next stepsize based on the estimated error and the tolerances.

   Interface: *computeStepsize*

   This interface computes next stepsize. It used by:

Component	*Description*
AD	It uses stepsize when computing Taylor series solution.

9. Projection

   This component projects the initial point provided by the user or the approximate solution computed by the AD component. It uses the interfaces:

   (a) computeTSSolution from AD component;

   (b) computeJacobian from AD component;

   (c) computeOffsets from Offsets component;

   (d) getParameters from IntegrationParameters component.

   After it obtains corresponding data, it uses an optimization package (IPOPT) to project an initial point or solution.

   Interface: *projectSolution*

   This interface projects an initial point or the computed TS solution.

Component	*Description*
AD	It computes Taylor coefficients and Jacobians using the projected consistent solution.
ConsistentInitialPoint	It projects the initial point provided by the user to a consistent one.

4.2.2 Low-level design

This subsection describes the class diagrams in HIDAETS. We have the following classes: DAEProblem, Parameters[2], SignatureMatrix, Offsets, InitialPoint, AD, ErrorEst, Stepsize, Projection, and DAESolver. For each class, we present the corresponding class diagram and a description.

1. DAEProblem

   An AD package based on operator overloading usually requests the parameters of compDAE in its own defined type, and the Sigma class also demands the variables of the evaluation function in Sigma type. We could employ templates to implement it as follows:

```
template < typename T >
class DAEProblem
{
 public:
  virtual void compDAE( T *f, T *x, T &t ) = 0;
  // ...

 private:
  // ...
}

template < typename T >
class DAEProblem_1 : public DAEProblem < T >
{
 public:
  void compDAE( T *f, T *x, T &t );
  // ...

 private:
  // ...
}
```

   The advantage is that the user can declare his/her own variables to set problem parameters. For example, the user can define the single pendulum problem as

[2]Usually, text that is related to programs is typed in typewriter font. We use normal font, as it is clear from the context.

```
template < typename T >
class Pendulum : public DAEProblem < T >
{
 private:
  double g, L;
  // ...

 public:
  void setParam( double *param )
  {
    g = param[0];
    L = param[1];
  }

  void compDAE( T *f, T *x, T &t )
  {
    f[0] = diff(Y[0],2) + Y[0]*Y[2];
    f[1] = diff(Y[1],2) + Y[1]*Y[2] - g;
    f[2] = Y[0]*Y[0]    + Y[1]*Y[1] - L*L;
  }
  // ...

}
```

Then, in the main function, the user could change g, L and integrate the new problem without changing the definition of it.

However, the interfaces become quite complicated. First, to implement the class Pendulum, the user needs to know about class inheritance, templates, and overloading. Second, in the main function, we have to create more than one Pendulum objects with different types. That means when we change some values in the DAE problem, we have to do the same thing on all the Pendulum objects, which also complicates usage. For example, to use FADBAD++ and set problem parameters, we would have the following code in the main function.

```
int main()
{
  // ...
```

```
  DAEProblem <Sigma> *ptrPend1 = new Pendulum <Sigma>();
  DAEProblem < F<double> > *ptrPend2 = new Pendulum < F<double> >();
  DAEProblem < T< F<double> > > *ptrPend3 = new Pendulum < T< F<double> > >();

  double param[2];
  param[0] = 9.8; param[1] = 10;

  ptrPend1->setParam( param );
  ptrPend2->setParam( param );
  ptrPend3->setParam( param );

  // ...

}
```

The above main function seems complicated. To simplify it, in HIDAETS we define a DAE problem by a template function. The user needs only to provide such function as shown in subsection 4.3.2. To enable the user to change problem parameters, we add one parameter, a void pointer to the problem parameters, in the template function. For the single pendulum problem, we define it as below.

```
template < typename T >
void Pendulum( T *f, T *x, T &t, void *DAEParam)
{
  double *d = (double *) DAEParam;
  g = d[0];
  L = d[1];

  f[0] = diff(Y[0],2) + Y[0]*Y[2];
  f[1] = diff(Y[1],2) + Y[1]*Y[2] - g;
  f[2] = Y[0]*Y[0]    + Y[1]*Y[1] - L*L;
}
```

Because there are many relations between DAEProblem and other classes, we need a class diagram consisting of necessary methods to represent this component and illustrate associated relations. Then, we could present the class diagrams by UML easily and demonstrate the relations among components clearly.

2. Parameters

   This class provides functions for setting and obtaining parameters such as atol, rtol, Ord, Ind, and outputInd. For more details, refer to subsection 4.1; see also Figure 4.3 and Table 4.2.

   Remark. All methods in Parameters are inline functions.

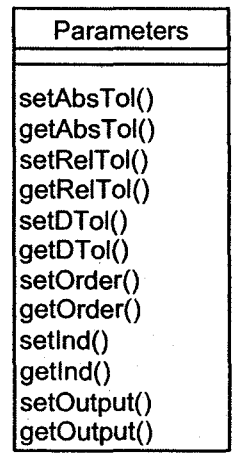


Figure 4.3: Class diagram: Parameters.

3. SignatureMatrix

   The SignatureMatrix class obtains the signature matrix from system equations. It uses *compDAE*[3] method to obtain system equations as illustrated in Figure 4.4. Table 4.3 demonstrates its document in detail.

4. Offsets

   Figure 4.4 shows the Offsets class diagram. The Offsets class computes problem offsets based on the computed signature matrix. There are four classes in this part: Offsets, LAPSolver, LAP. Offsets class inherits from the SignatureMatrix class, and has an LAPSolver class. LAP denotes class that implements interfaces defined in the LAPSolver class; see Tables 4.4, 4.5, and 4.6.

[3]This compDAE refers to the function describing the DAE, not a method of non-existing class.

Parameters		
Methods	*Type*	*Description*
setAbsTol	void	Sets absolute tolerance.
getAbsTol	double	Obtains absolute tolerance.
setRelTol	void	Sets relative tolerance.
getRelTol	double	Obtains relative tolerance.
setDTol	void	Sets tolerance for the optimization packages.
getDTol	double	Obtains tolerance for the optimization packages.
setOrder	void	Sets order value.
getOrder	unsigned integer	Obtains order value.
setInd	void	Sets Ind value.
getInd	unsigned integer	Obtains Ind value.
setOutput	void	Sets output information value.
getOutput	unsigned integer	Obtains output information value.

Table 4.2: Methods of the Parameters class.

SignatureMatrix		
Methods	*Type*	*Description*
compSignatureMatrix	void	Computes the signature matrix from system equations.

Table 4.3: Method of the SignatureMatrix class.

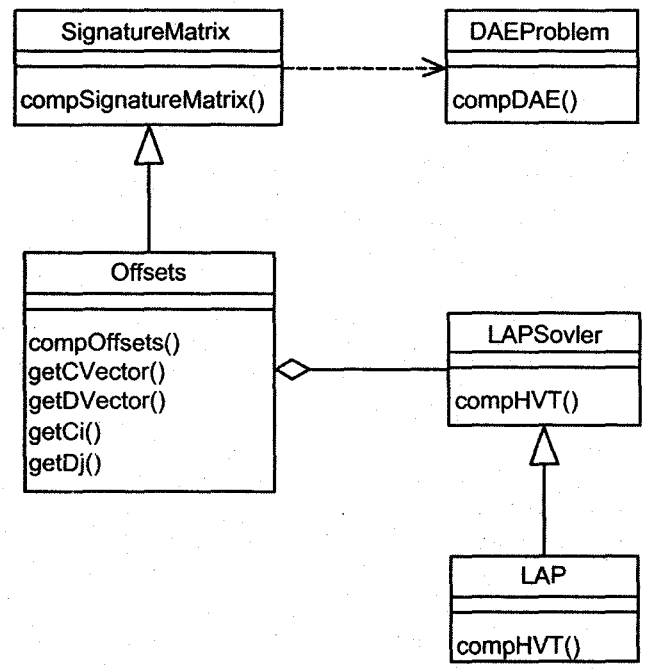


Figure 4.4: Class diagram: SignatureMatrix and Offsets.

From Figure 4.4, the user can use his/her own linear assignment problem solver by inheriting the LAPSolver class and implementing it.

Offsets		
Methods	*Type*	*Description*
compOffsets	void	Computes the equation offsets and variable offsets from the HVT of signature matrix.
getCVector	int *	Obtains equation offsets.
getDVector	int *	Obtains variable offsets.
getCi	int	Returns equation offset for a given index.
getDj	int	Returns variable offset for a given index.

Table 4.4: Methods of the Offsets class.

LAPSolver		
Methods	*Type*	*Description*
compHVT	void	Virtual function to compute the HVT from the signature matrix of DAE system.

Table 4.5: Method of the LAPSolver class.

LAP		
Methods	*Type*	*Description*
compHVT	void	Inherits from LAPSolver class and provides an implementation.

Table 4.6: Method of the LAP class.

5. InitialPoint

   Figure 4.5 illustrates the class diagram of InitialPoint. It uses interfaces *getInitialPoint* from DAEProblem and *projectSolution* from Projection. It first decides whether the initial point provided by the user is consistent; if not, it tries to compute a consistent one. Table 4.7 gives some details.

6. AD

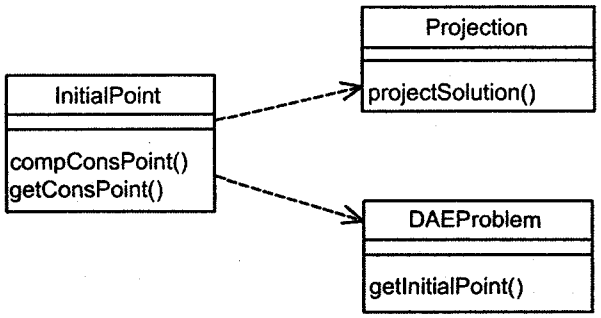


Figure 4.5: Class diagram: InitialPoint.

InitialPoint		
Methods	*Type*	*Description*
compConsPoint	void	Decides whether the initial point is consistent; if not, it computes a consistent point.
getConsPoint	InitialPoint *	Obtains a consistent initial point, computed by compConsPoint.

Table 4.7: Methods of the InitialPoint class.

The AD class is a key class in HIDAETS. It computes Taylor coefficients, Jacobians, and Taylor series solution. This part includes three classes: AD, ADOL-C, and FADBAD++. AD class defines interfaces for automatic differentiation packages. It uses interfaces provided by other classes as shown on the right side of Figure 4.6. Currently HIDAETS integrates FADBAD++ and ADOL-C. Their class diagrams are shown in Figure 4.6. Tables 4.8 and 4.9 list the methods in these classes with brief descriptions. Tables 4.10, 4.11, 4.12, 4.13, 4.14, 4.15, 4.16 show the methods in class AD.

From Figure 4.6, the user can use his/her own AD packages by inheriting the AD class and implementing it.

AD		
Methods	*Type*	*Description*
compJacobian	void	Virtual function to compute the Jacobian matrix for DAE system.
getJacobian	void	Virtual function to obtain the computed Jacobian matrix.
compConstraints	void	Virtual function to compute constraints based on stage.
getConstraints	double	Virtual function to obtain the specified constraint value.
compCoefficients	void	Virtual function to compute the Taylor coefficients for DAE system.
getCoefficients	double	Virtual function to obtain the specified Taylor coefficient.
compTSSolution	void	Virtual function to compute Taylor series solution at current time.
getTSSolution	void	Virtual function to obtain Taylor series solution at current time.
getNormX	double	Virtual function to obtain the norm of $x_{J_{<0}}$.

Table 4.8: Methods of the AD class.

7. ErrorEst

The ErrorEst class estimates the error for the current step. It uses *getCoefficients* from AD class to obtain needed Taylor coefficients. Figure 4.7 and Table

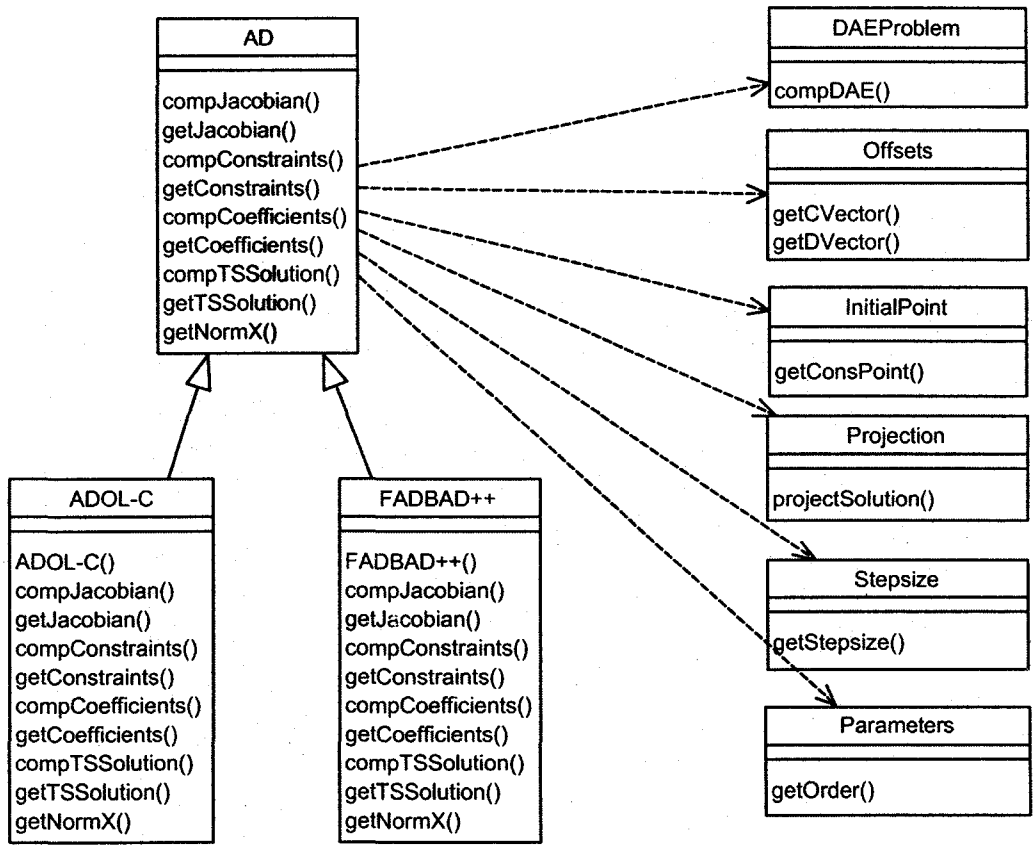


Figure 4.6: Class diagram: AD.

ADOL-C and FADBAD++		
Methods	*Type*	*Description*
ADOL-C/FADBAD++		Constructor to generate the computational graph for Taylor coefficients and Jacobians.
compJacobian	void	Inherits from AD class and provide an implementation.
getJacobian	void	
compConstraints	void	
getConstraints	double	
compCoefficients	void	
getCoefficients	double	
compTSSolution	void	
getTSSolution	void	
getNormX	double	

Table 4.9: Methods of classes ADOL-C and FADBAD++.

compJacobian(stage)		
Parameters	*Type*	*Description*
stage	int	stage number.

Table 4.10: Description of the compJacobian method.

getJacobian(stage, m, n, Jac)		
Parameters	*Type*	*Description*
stage	int	Stage number.
m	int &	Number of rows in Jacobian.
n	int &	Number of columns in Jacobian.
Jac	double **	Pointer to a Jacobian.

Table 4.11: Description of the getJacobian method.

compConstraints(stage)		
Parameters	*Type*	*Description*
stage	int	Stage number.

Table 4.12: Description of the compConstraints method.

getConstraints(stage, c)		
Parameters	*Type*	*Description*
stage	int	Stage number.
c	double *	Pointer to values of the constraints.

Table 4.13: Description of the getConstraints method.

compCoefficients(p)		
Parameters	*Type*	*Description*
p	int	Taylor series order.

Table 4.14: Description of the compCoefficients method.

getCoefficients(j, k)		
Parameters	*Type*	*Description*
j	int	index of variable.
k	int	order of Taylor coefficients.

Table 4.15: Description of the getCoefficients method.

compTSSolution(p, h)		
Parameters	*Type*	*Description*
p	int	Taylor series order.
h	double	stepsize.

Table 4.16: Description of the compTSSolution method.

4.17 show its class diagram and documentation.

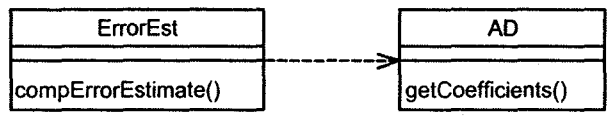


Figure 4.7: Class diagram: ErrorEst.

ErrorEst		
Methods	*Type*	*Description*
compErrorEstimate	double	Estimates error for current TS solution.

Table 4.17: Method of the ErrorEst class.

8. Stepsize

   The Stepsize class forms the tolerance and select stepsize for next integration step. It uses *compErrorEstimate* from class ErrorEst, *getNormX* from class AD, and obtains needed parameters from Parameters class. See Figure 4.8 and Table 4.18.

Stepsize		
Methods	*Type*	*Description*
compStepsize	double	Computes stepsize for next step.
compFinalStep	double	Computes final stepsize.
compTolerance	double	Computes tolerance for the DAE system.

Table 4.18: Methods of the Stepsize class.

9. Projection

   The Projection class tries to compute a consistent initial point or Taylor series solution. This part includes four classes: Projection, OptimizationPackage, and IPOPT. Projection uses interfaces from classes AD, Offsets, Parameters as shown on the right side of Figure 4.9. It also aggregates an OptimizationPackage class, which defines virtual interfaces for optimization package. IPOPT is class

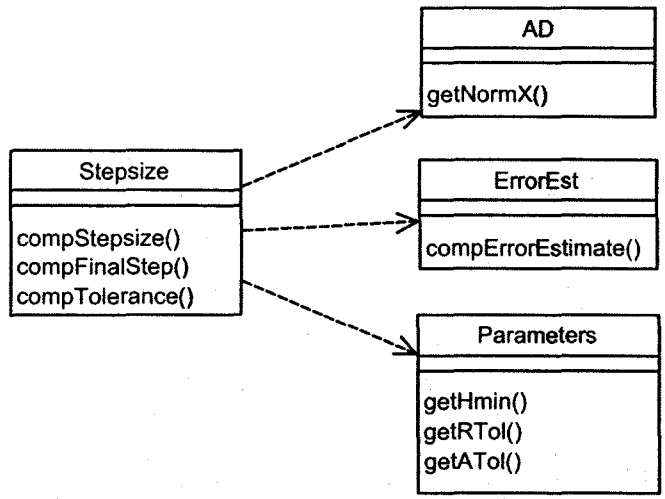


Figure 4.8: Class diagram: Stepsize.

to implement these functions. Their class diagrams are shown in Figure 4.9. Tables 4.19, 4.20, and 4.21 describe these classes.

From Figure 4.9, the user can use his/her own optimization package by inheriting the OptimizationPackage class and implementing it.

Projection		
Methods	*Type*	*Description*
projectSolution	bool	Projects an initial point or TS solution. If it obtains a consistent one, returns true. If it fails, returns false.

Table 4.19: Method of the Projection class.

10. DAESolver

The DAESolver class aggregates all the classes above to provide an integration interface for the user. Figure 4.10 illustrates its class diagram. See also Table 4.22.

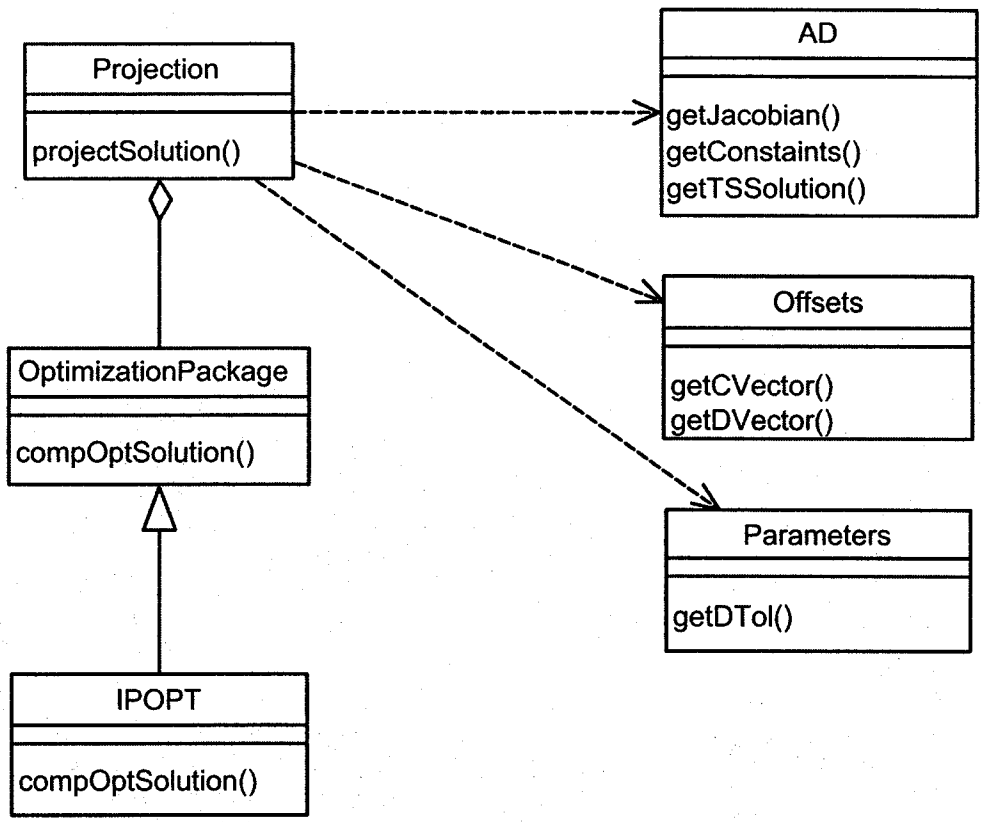


Figure 4.9: Class diagram: Projection.

OptimizationPackage		
Methods	*Type*	*Description*
compOptSolution	void	Virtual method to compute a solution for an optimization problem.

Table 4.20: Method of the OptimizationPackage class.

IPOPT		
Methods	*Type*	*Description*
compOptSolution	void	Inherits from OptimizationPackage class and provides an implementation.

Table 4.21: Method of the IPOPT class.

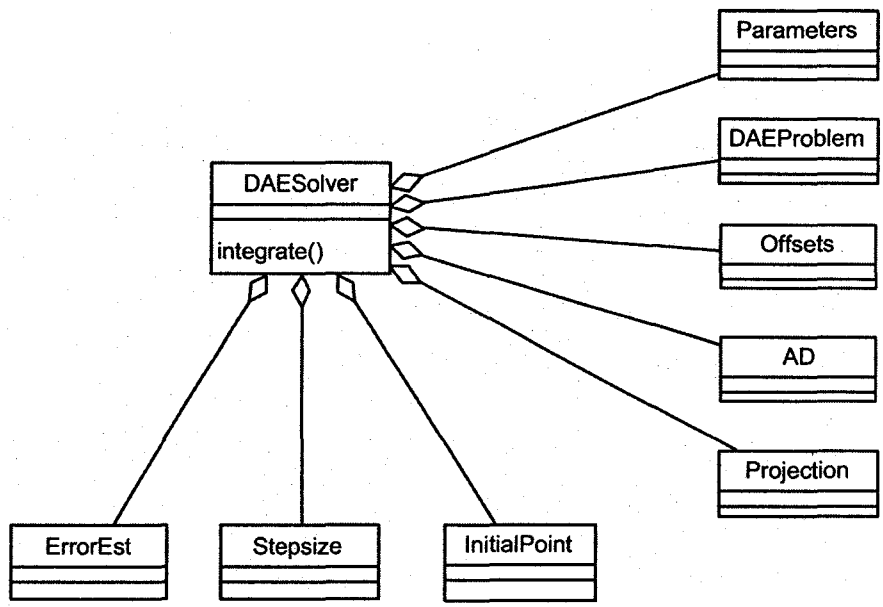


Figure 4.10: Class diagram: DAESolver.

DAESolver		
Methods	*Type*	*Description*
integrate	void	Integrates DAE problem depending on Ind.

Table 4.22: Method of the DAESolver class.

4.3 Installation and usage

This section describes how to install and use HIDAETS on operating systems (Linux, Mac OS X, and Unix) with GNU g++. We have used version after 3.2.

HIDAETS is developed on Sun Solaris 9 environment with C/C++ mixed with packages in FORTRAN, which are BLAS, LAPACK, and IPOPT. The source code of HIDAETS includes 4845 lines. We provide ten DAE and ODE examples. HIDAETS has been successfully tested on Mac OS X, Linux, and Solaris.

4.3.1 Installation

Essentially, we have to do the following steps to install HIDAETS:

1. Download HIDAETS;
2. Download third party components;
3. Compile third party components and HIDAETS.

Content of the package

HIDAETS is distributed as a gzipped tar file HIDAETS.tar.gz, which can be downloaded from `www.cas.mcmaster.ca/~hidaets`. Online documentation is also available at this Web site. To extract the files, type

```
gunzip < HIDAETS.tar.gz | tar xvf -
```

The directory structure of HIDAETS is shown in Figure 4.11. A top-level makefile in the HIDAETS directory is provided to perform the entire installation procedure. There are also separate makefiles inside NUMLIB, SRC, and EXAMPLES directories.

The following third party software components are required for building the HIDAETS package. The user needs to download the source code for those components as described below.

1. **ADOL-C.** The user may download adolc_1.8.7.tar.gz freely from anonymous ftp at `ftp://ftp.math.tu-dresden.de/pub/ADOLC/ADOLC_1.8` and unpack it into

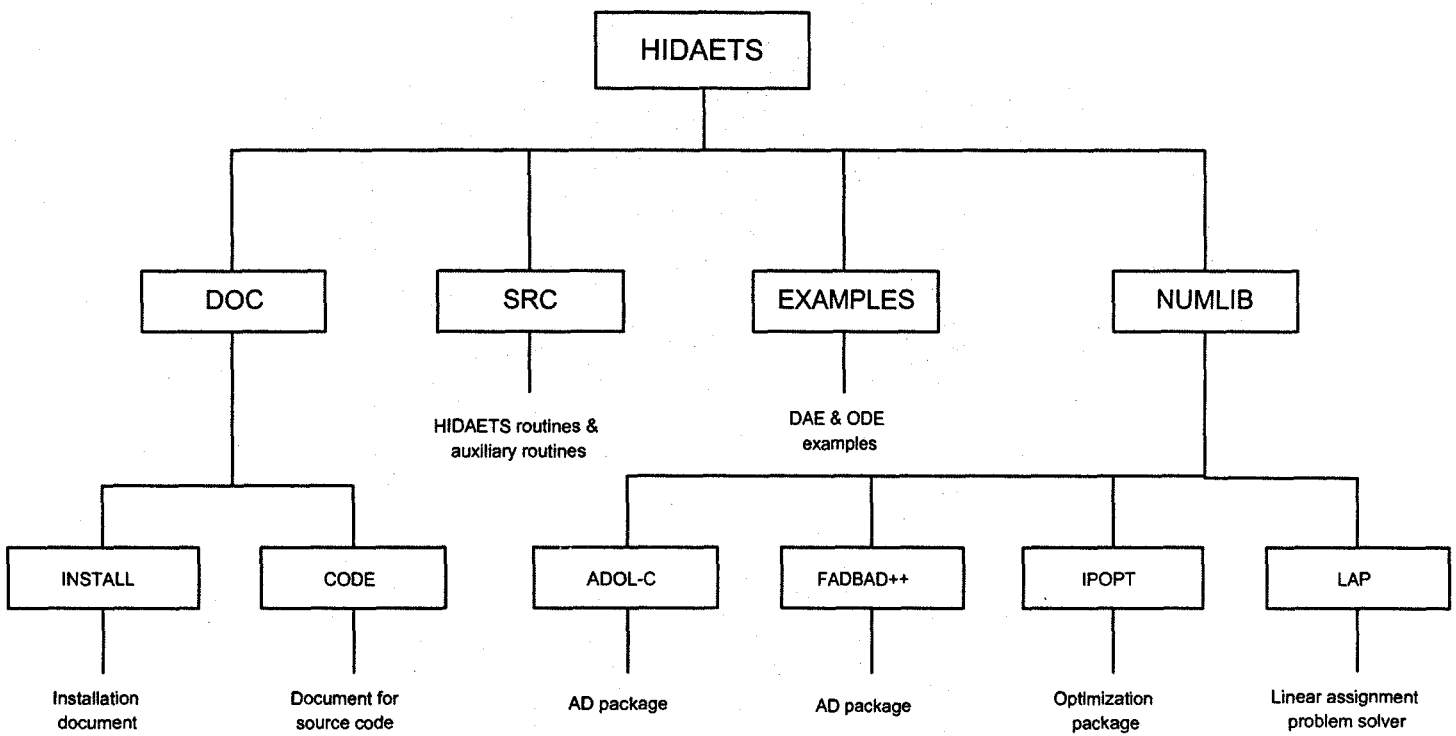


Figure 4.11: Organization of HIDAETS.

the NUMLIB directory. Then, a directory named ADOLC18 is created in the NUMLIB directory. For more details, refer to `http://www.math.tu-dresden.de/wir/project/adolc/`.

2. **FADBAD++**. The user may download it from `http://www.imm.dtu.dk/nag/proj_km/fadbad/` and unpack it into the NUMLIB directory. Then, a directory named FADBAD++ is created in the NUMLIB directory. For more details, refer to `http://www.imm.dtu.dk/nag/proj_km/fadbad/`.

3. **IPOPT**. The user may download it from the COIN-OR web page at `http://www.coin-or.org`. After that, the user needs to unpack it into the NUMLIB directory. Then, a directory named COIN is created in the NUMLIB directory. IPOPT needs more third party components as described in its documentation. For more details, refer to `http://www.coin-or.org/Ipopt/index.html`.

4. **LAP**. The user may download it from `http://www.magiclogic.com/assignment.html` and unpack all files into the NUMLIB/LAP directory.

Alternatively, the user may have these third party components in other directories. Then he/she needs to specify the path by changing relative variables in Makefile.inc in the HIDAETS directory. Below is the Makefile.inc with IPOPT at /usr/local/COIN/Ipopt.

```
#
#  HIDAETS make include file
#  April 25, 2005
#
PREFIX = $(PWD)

ADOLC  = $(PREFIX)/NUMLIB/ADOLC18
FADBAD = $(PREFIX)/NUMLIB/FADBAD++
IPOPT  = /usr/local/COIN/Ipopt
LAP    = $(PREFIX)/NUMLIB/LAP

AD     = $(ADOLC)/SRC
IPOPTH = $(IPOPT)/include
IPOPTL = $(IPOPT)/lib
```

```
CC      = gcc
CXX     = g++
```

Installing HIDAETS in systems with g++

Once the user has all the necessary third party components in place, the user should run the top-level makefile by typing

```
make all
```

We provide other options for installation and cleaning. Table 4.23 lists all of them.

Argument	Description
all	Installs source code, third party components, and examples
install	Installs source code and third party components
installsrc	Installs source code
clean	Clean all object files
cleansrc	Clean all object files in SRC
cleannumlib	Clean all object files in NUMLIB

Table 4.23: Options for installation.

After installation, the user may either test HIDAETS by the examples in the EXAMPLES directory, or solve his/her own problems.

4.3.2 Usage

Basic usage

The user must provide system equations, dimension of the DAE system, t_0, t_{end}, and x_{ICs}. The user may set (optionally) order, Ind, tolerances, and output parameters. Below we illustrate how to use HIDAETS on the single pendulum example.

1. **Set system equations.** The user has to provide the system equations in a template function. Below is the definition for the single pendulum problem in pendulum.h.

```
template <typename T>
```

```
void Pendulum( int n, T *f, T *Y, T & t, void *param )
{
  double g = 1.0;
  double L = 1.0;

  f[0] = diff(Y[0],2) + Y[0]*Y[2];
  f[1] = diff(Y[1],2) + Y[1]*Y[2] - g;
  f[2] = Y[0]*Y[0]    + Y[1]*Y[1] - L*L;
}
```

2. **Set dimension of DAE system, t_0, and t_{end}.** The user needs to give dimension of DAE problem, t_0, and t_{end} at the beginning of the implementation file. Below is the main function for the single pendulum problem in pendulum.cc.

```
1  int main()
2  {
3    int n = 3;
4    double t0 = 0, tend = 100.0;
5
6    DAESolver *ptrDAESolver = new DAESolver( n, Pendulum, Pendulum, Pendulum );
7
8    InitPoint x( n, ptrDAESolver->getDVector() );
9    setInitialValues( x );
10
11   ptrDAESolver->integrate( t0, tend, x );
12
13   delete ptrDAESolver;
14
15   return 0;
16 }
```

Line 3 sets the dimension, and line 4 sets t_0 and t_{end}. Line 6 declares an object of DAESolver[4]. Lines 8 to 9 set initial point. Line 11 integrates the DAE problem. For basic usage, the user usually does not need to change anything between line 6 and line 16.

[4]In line 6, the user needs to give the Pendulum function three times to the constructor of DAESolver, since FADBAD++ and the Sigma class require different types.

3. **Set an initial point.** The user can set an initial point in a function in the implementation file. Below is the function for the single pendulum problem in the pendulum.cc.

```
void setInitialValues( int n, InitPoint &x )
{
  //initial values
  x(0, 0) = 1; x(0, 1) = 0;
  x(1, 0) = 0; x(1, 1) = 1;
}
```

Alteratively, the user can set it directly in the main function by

```
x(0, 0) = 1;
x(0, 1) = 0;
x(1, 0) = 0;
x(1, 1) = 1;
```

instead of line 9 in the above main function and the setInitialValues function.

4. **Set order, Ind, tolerances, output parameters (optional).** The default values of these variables are

order	=	20	Ind	=	tillend
atol	=	10^{-13}	outputInd	=	initpoint
rtol	=	10^{-13}	filename	=	"result"
dtol	=	$0.5 \cdot$ atol	digits	=	16

Here, outputInd is the output control indicator, filename is the file name to output the integration information, and digits is the number of output digits.

The integration information at t is stored in the form

column	1	2	3	...	$n+1$	$n+2$
data	t	x_1	x_2	...	x_n	h

That is in column 1, we store the integration time t; from column 2 to $n+1$, we store the solution at t; in column $n+2$, we store the stepsize taken at t.

We could call the corresponding member functions of the Parameters class to set these parameters. Below is an example of how we set order, absolute tolerance, and relative tolerance in pendulum.cc.

```
1  int main()
2  {
3
4    int n = 3;
5    double t0 = 0, tend = 100.0;
6
7    Parameters *parameters = new Parameters();
8    parameters->setOrder( 30 );
9    parameters->setATol( 1e-10 );
10   parameters->setRTol( 1e-11 );
11
12   DAESolver *ptrDAESolver = new DAESolver( n, Pendulum, Pendulum,
13                                            Pendulum, parameters);
14
15   InitPoint x( n, ptrDAESolver->getDVector() );
16   setInitialValues( x );
17
18   ptrDAESolver->integrate( t0, tend, x );
19
20   delete ptrDAESolver;
21   delete parameters;
22
23   return 0;
24 }
```

Lines 7 to 10 declare a new object of Parameters and set corresponding values. Lines 12 to 13 create an object of DAESolver and pass the object of Parameters to it. Line 21 deallocates the Parameters object.

Advanced usage

1. **Change problem parameters.** The user can change the problem parameters in the system equations, and he/she may integrate the changed problem. To do so, the user needs to change the definition of system equations and some lines in the main function. Below is an example of using the single pendulum problem.

```
1  template <typename T>
2  void Pendulum( int n, T *f, T *Y, T & t, void *param )
3  {
4    double *d = (double*) param;
5    double g = d[0];
6    double L = d[1];
7
8    f[0] = diff(Y[0],2) + Y[0]*Y[2];
9    f[1] = diff(Y[1],2) + Y[1]*Y[2] - g;
10   f[2] = Y[0]*Y[0]    + Y[1]*Y[1] - L*L;
11 }
```

Lines 4 to 6 obtain the user defined parameters. The user must ensure that the problem parameters are set and retrieved correctly.

```
1  int main()
2  {
3
4    int n = 3;
5    double t0 = 0, tend = 100.0;
6
7    Parameters *ptrParameters = new Parameters();
8    DAESolver *ptrDAESolver = new DAESolver( n, Pendulum, Pendulum,
9                                             Pendulum, ptrParameters );
10
11   InitPoint x( n, ptrDAESolver->getDVector() );
12   setInitialValues( x );
13
14   double param[2];
15   param[0] = 1; param[1] = 1;
16   ptrDAESolver->integrate( t0, tend, x, param );
17
18   param[0] = 9.8; param[1] = 10;
19   ptrParameters->setInd( tillend );
20   ptrDAESolver->integrate( t0, tend, x, param );
21
22   delete ptrDAESolver;
23   delete ptrParameters;
24
```

```
25   return 0;
26 }
```

Lines 14 to 15 define the problem parameters and set values. Line 16 integrates the problem with specified parameters. Line 18 changes the problem parameters, and Line 19 sets Ind to tell the solver that the problem parameters have been changed. Line 20 integrates the problem with new parameters.

2. **Set method classes.** For a more advanced usage, the user can implement his/her own classes, following the description in section 4.2. After that, the user needs to recompile the relevant source code, create objects of classes in the main function, and set as parameters in the constructor function of DAESolver when creating a new DAESolver class. Below is the main function for a new AD class ADOL-C.

```
1  int main()
2  {
3
4    int n = 3;
5    double t0 = 0, tend = 100.0;
6
7    LAPSolver_1 lapSolver;
8    Offsets     offsets( n );
9    offsets.compSignatureMatrix( Pendulum );
10   offsets.compOffsets( lapSolver );
11
12   Parameters *parameters = new Parameters();
13   AD *adPackage = new ADOL_C( n, parameters->getOrder(), Pendulum, offsets );
14   OptPackage *opt_package = new OptPackage_Ipopt();
15   Projection *projection = new Projection( offsets, adPackage,
16                                            parameters, opt_package );
17   ErrorEst *ptrErrorEst = new ErrorEst( n, &offsets );
18   StepSize *ptrStepSize = new StepSize( );
19
20   DAESolver *ptrDAESolver = new DAESolver( &offsets, parameters,
21                                            adPackage, projection,
22                                            ptrErrorEst, ptrStepSize );
23
```

```
24   InitPoint x( n, ptrDAESolver->getDVector() );
25   setInitialValues( n, x );
26
27   ptrDAESolver->integrate( t0, tend, x );
28
29   delete ptrDAESolver;
30
31   delete ptrStepSize;
32   delete ptrErrorEst;
33   delete opt_package;
34   delete projection;
35   delete adPackage;
36   delete parameters;
37
38   return 0;
39 }
```

Lines 7 to 10 create new objects of class LAPSolver and Offsets to compute the signature matrix and offsets. Lines 12 to 18 declare new objects of class Parameters, AD, OptPackage, Projection, ErrorEst, and Stepsize, where line 13 creates an object of class ADOL_C. Lines 20 to 22 pass these objects to object of DAESolver. Lines 29 to 36 deallocate these objects.

Screen output

Below is the screen output of HIDAETS for the single pendulum problem with outputInd = initpoint. The user can set the output with reference to subsection 4.1.

```
SIGNATURE MATRIX & OFFSETS:

       0   1   2  |c_i
     |-----------------
   0|  2   -   0* | 0
   1|  -   2*  0  | 0
   2|  0*  0   -  | 2
     |-----------------
 d_j|  2   2   0
```

```
INITIAL POINT

   x(0,0) =  1.0000000000000000e+00
   x(0,1) =  0.0000000000000000e+00
   x(0,2) = -1.0000000000000000e+00

   x(1,0) =  0.0000000000000000e+00
   x(1,1) =  1.0000000000000000e+00
   x(1,2) =  1.0000000000000000e+00

   x(2,0) =  1.0000000000000000e+00


INTEGRATION PROCESS

   Integrated at     Local Error
    1.0000e+02        1.5004e-11


SOLUTION AT t = 100.000000

   x(0,0) = -4.5766268833405888e-01
   x(0,1) =  1.4820029313357552e+00
   x(0,2) =  1.6784219355066259e+00

   x(1,0) =  8.8912589868201786e-01
   x(1,1) =  7.6283622676985696e-01
   x(1,2) = -2.2607604897991043e+00

   x(2,0) =  3.6673776960412945e+00


STATS INFO

   CPU time (sec).....1.750
   CPU time/step.....0.007
   Steps.....246
     accepted.....246  (100.0%)
     rejected.....0  (0.0%)
```

Chapter 5

Numerical Results

We have tested HIDAETS on more than ten DAE and ODE problems including single pendulum [AP98], double pendula [Pry98], car axis problem [MI03], two-link robotic arm [Pry98], transistor amplifier [MI03], modified[1] car axis problem [MI03], chemical Akzo Nobel problem [MI03], modified[2] chemical Akzo Nobel problem [MI03], and Van der Pol problem [MI03]. In this chapter, we give numerical results for the first five high-index DAE problems in detail.

5.1 Format of the problem descriptions

For each problem, we provide

1. general information,
2. mathematical description, and
3. numerical results.

In 3., we present the following information.

1. **Reference solution at the end of the integration interval.** The values of the components of a reference solution at the end of the integration interval

[1]We modify the car axis problem in [MI03] to a DAE consisting 4 differential and 2 algebraic equations.

[2]We modify the chemical Akzo Nobel problem in [MI03] to a DAE consisting 5 differential and 1 algebraic equation.

are listed. If possible, we compare a solution generated by HIDAETS with that presented in the literature, to assess the accuracy of our solution.

2. **Behavior of the numerical solution.** Plots of (some of) the solution components over the integration interval or part of it are presented.

3. **Run characteristics.** The experiments are done on Sun Ultra 5/10, Ultra SPARC-IIi 360MHZ, 512MB memory, Solaris 9. We use gcc 3.2 with optimization flag -O2. The following characteristics are reported:

   - tol

     We set absolute error tolerance equal to relative error tolerance.

   - steps

     Total number of steps taken by the solver.

   - CPU time (sec)

     The sum of user and system time in seconds taken by HIDAETS.

   - CPU time per step

     Average CPU time per step.

4. **Work-precision diagram.** We define significant correct digits (SCD) by

   $$\text{SCD} := -\log_{10}(\| \text{ relative error at the end of integration interval } \|_\infty).$$

   For HIDAETS, we plot CPU time against SCD.

5. **Error versus tolerance.** For each problem, we estimate the relative error of the solution components, and therefore SCD, and plot SCD against its input tolerance.

6. **Stepsize.** We plot the stepsize from t_0 to t_{end} to illustrate the stepsize behavior.

7. **CPU time versus order.** We compute the CPU time with different orders, different tolerances, and for each tolerance, we plot CPU time against order. With the CPU time versus order diagram, we could decide which order may be optimal for the tested DAE problem.

5.2 Single pendulum

5.2.1 General information

The problem is a nonstiff DAE of index 3, consisting of 2 differential and 1 algebraic equations.

5.2.2 Mathematical description of the problem

The problem is of the form

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

Here L is the length of the pendulum, g is gravity, both of which are constants; we take $L = 1$, $g = 1$. x, y, and λ are the dependent variables.

We integrate this problem from $t_0 = 0$ to $t_{\text{end}} = 100$ with initial conditions

$$\begin{aligned} x_0 = 1, \quad x_0' = 0; \\ y_0 = 0, \quad y_0' = 1. \end{aligned}$$

5.2.3 Numerical results

For this problem, we use order = 20 except for the CPU time versus order diagram. Since IPOPT can reach the desired tolerance dtol = 10^{-14} as its best, we perform all tests with tol $\geq 10^{-14}$.

Table 5.1 presents the reference solution at the end of the integration interval. The reference solution is computed with atol = rtol = 10^{-16} and dtol = 10^{-14} for IPOPT on the described setting before.

x	$-4.5766268835131380 \cdot 10^{-1}$
y	$8.8912589867298431 \cdot 10^{-1}$
λ	3.6673776960190421

Table 5.1: Reference solution for the pendulum problem.

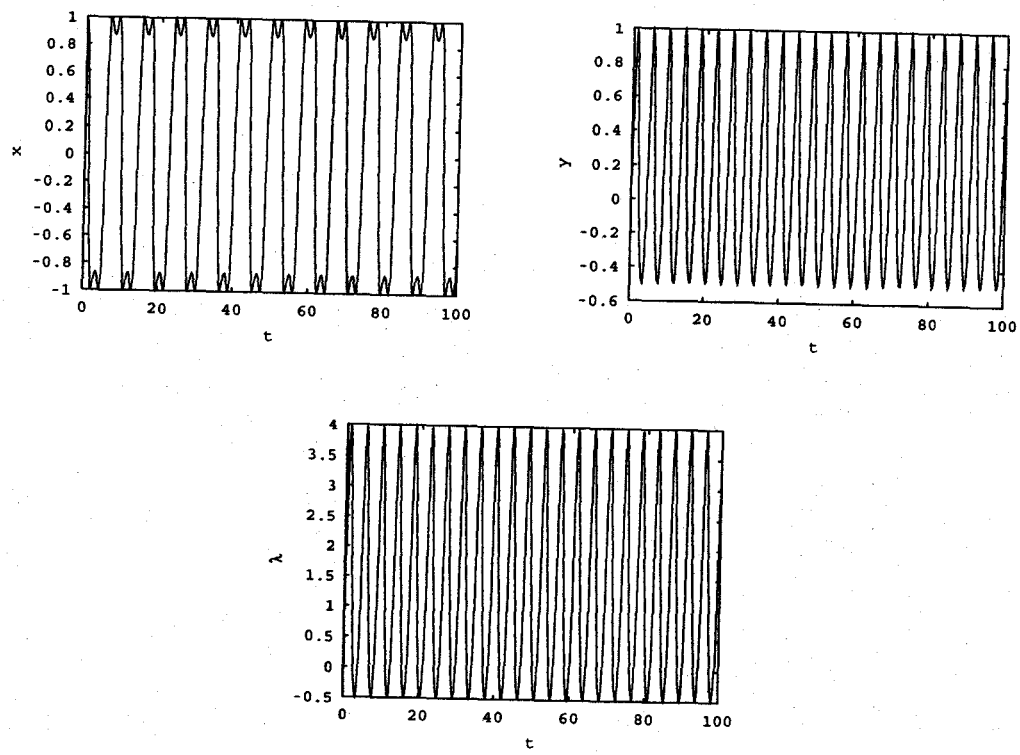


Figure 5.1: Plots of x, y, and λ versus time for the pendulum problem.

Figure 5.1 shows the behavior of x, y, λ over the integration interval.

Table 5.2 presents the run characteristics. We use atol = rtol = $10^{-(2m+1)}$, $m = 2, \ldots, 6$ and dtol = $0.5 \cdot$ atol for IPOPT. The columns with ADOL-C and FADBAD contain timing results produced with ADOL-C and FADBAD++, respectively. In the run characteristics table, HIDAETS takes the same number of steps with ADOL-C and FADBAD++.

tol	steps	CPU time (sec)		CPU time per step	
		ADOL-C	FADBAD	ADOL-C	FADBAD
10^{-5}	123	0.73	0.69	0.006	0.006
10^{-7}	155	1.10	1.13	0.007	0.007
10^{-9}	196	1.32	1.33	0.007	0.007
10^{-11}	246	1.51	1.50	0.006	0.006
10^{-13}	310	1.76	1.89	0.006	0.006

Table 5.2: Run characteristics for the pendulum problem.

Figure 5.2 shows the work-precision diagrams. We use atol = 10^{-m}, $m = 5, \ldots, 14$, rtol = atol, and

$$\text{dtol} = \begin{cases} 0.5 \cdot \text{atol}, & m = 5, \ldots, 13, \\ 10^{-14}, & m = 14. \end{cases} \tag{5.1}$$

Figure 5.3 illustrates error against different tolerances which are the same as the tolerances in the work precision diagram.

Figure 5.4 exhibits stepsize behavior over the integration interval. One of the plots is with tol = 10^{-7}, the other is with tol = 10^{-13}.

Figure 5.5 displays the CPU time against order with different tolerances. We use FADBAD++ as the AD package and the tolerances are the same as the ones used in the run characteristics table. We integrate the problem to $t_{\text{end}} = 300$. From the figure, we can conclude that for the single pendulum problem, order ≈ 30 is optimal for different tolerances.

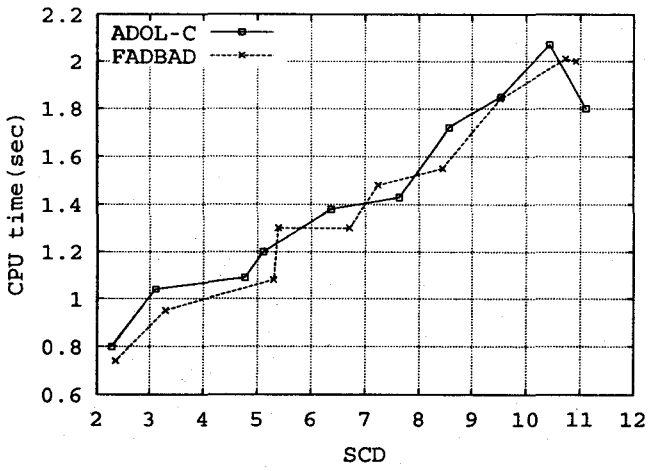


Figure 5.2: Work-precision diagram for the pendulum problem.

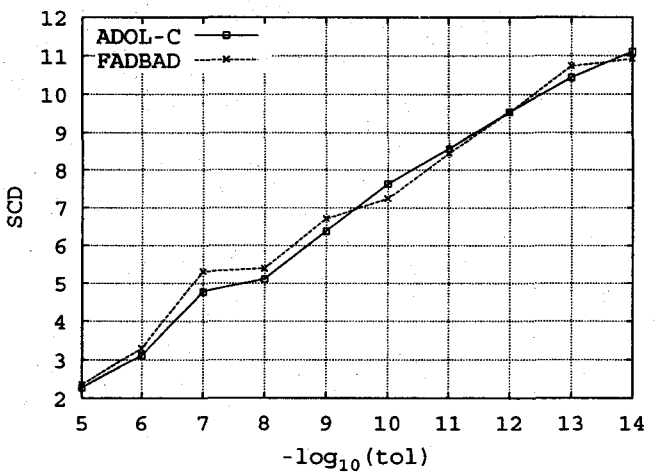


Figure 5.3: Error versus tolerance for the pendulum problem.

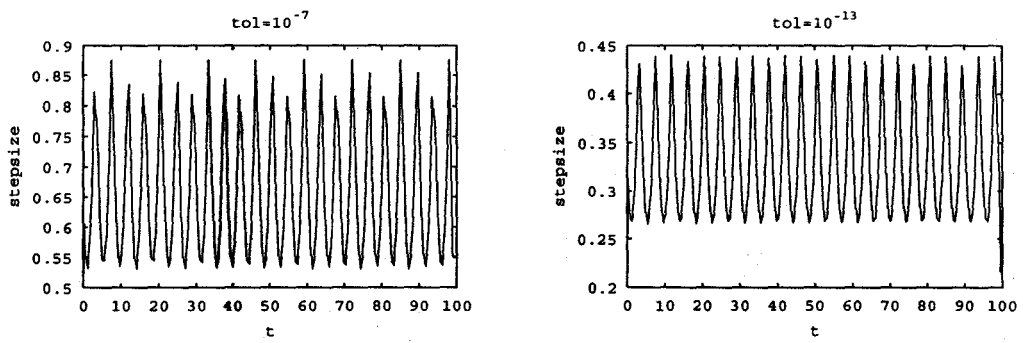


Figure 5.4: Stepsize versus time with tolerances 10^{-7} and 10^{-13}.

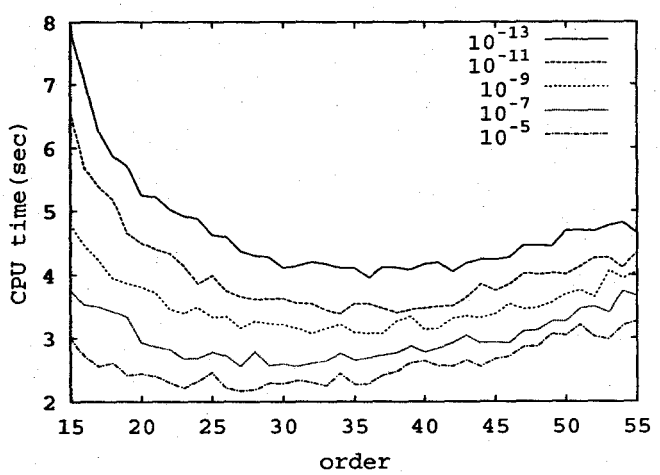


Figure 5.5: CPU time versus order with different tolerances.

5.3 Double pendula

5.3.1 General information

The problem is a nonstiff DAE of index 5, consisting of 4 differential and 2 algebraic equations.

5.3.2 Mathematical description of the problem

The problem is of the form

$$\begin{aligned} 0 &= x'' + x\lambda, & 0 &= u'' + u\kappa, \\ 0 &= y'' + y\lambda - g, & 0 &= v'' + v\kappa - g, \\ 0 &= x^2 + y^2 - L^2, & 0 &= u^2 + v^2 - (L + c\lambda)^2. \end{aligned}$$

Here we take $L = 1$, $g = 1$, and $c = 0.1$; x, y, λ, u, v, and κ are the dependent variables.

We integrate this problem from $t_0 = 0$ to $t_{\text{end}} = 100$ with initial conditions

$$\begin{aligned} x_0 &= 1, \quad x_0' = 0; & u_0 &= 1, \quad u_0' = 0; \\ y_0 &= 0, \quad y_0' = 1; & v_0 &= 0, \quad v_0' = 1. \end{aligned} \tag{5.2}$$

5.3.3 Numerical results

For this problem, we use order = 20 except for the CPU time versus order diagram.

The solution of double pendula problem shows chaotic behavior [Pry98]. Figure 5.6 demonstrates this behavior. We integrate the problem with two slightly perturbed initial points which are shown in Table 5.3 and using atol = rtol = $2 \cdot$ dtol = 10^{-10}. The column with InitPoint-1 in Table 5.3 is the projected initial point corresponding to (5.2), while the column with InitPoint-2 is the projected initial point with changed v_0 from 0 to 0.001 in (5.2). Since the consistent initial values for x, y, λ are same in InitPoint-1 and InitPoint-2, we just give the initial values of u, v, and κ, where we also include these values corresponding to stage 0.

From Figure 5.6, we can find that the two solutions of u, v, and κ are very close till about $t = 30$, and clearly diverging from there. This is strong evidence of chaotic

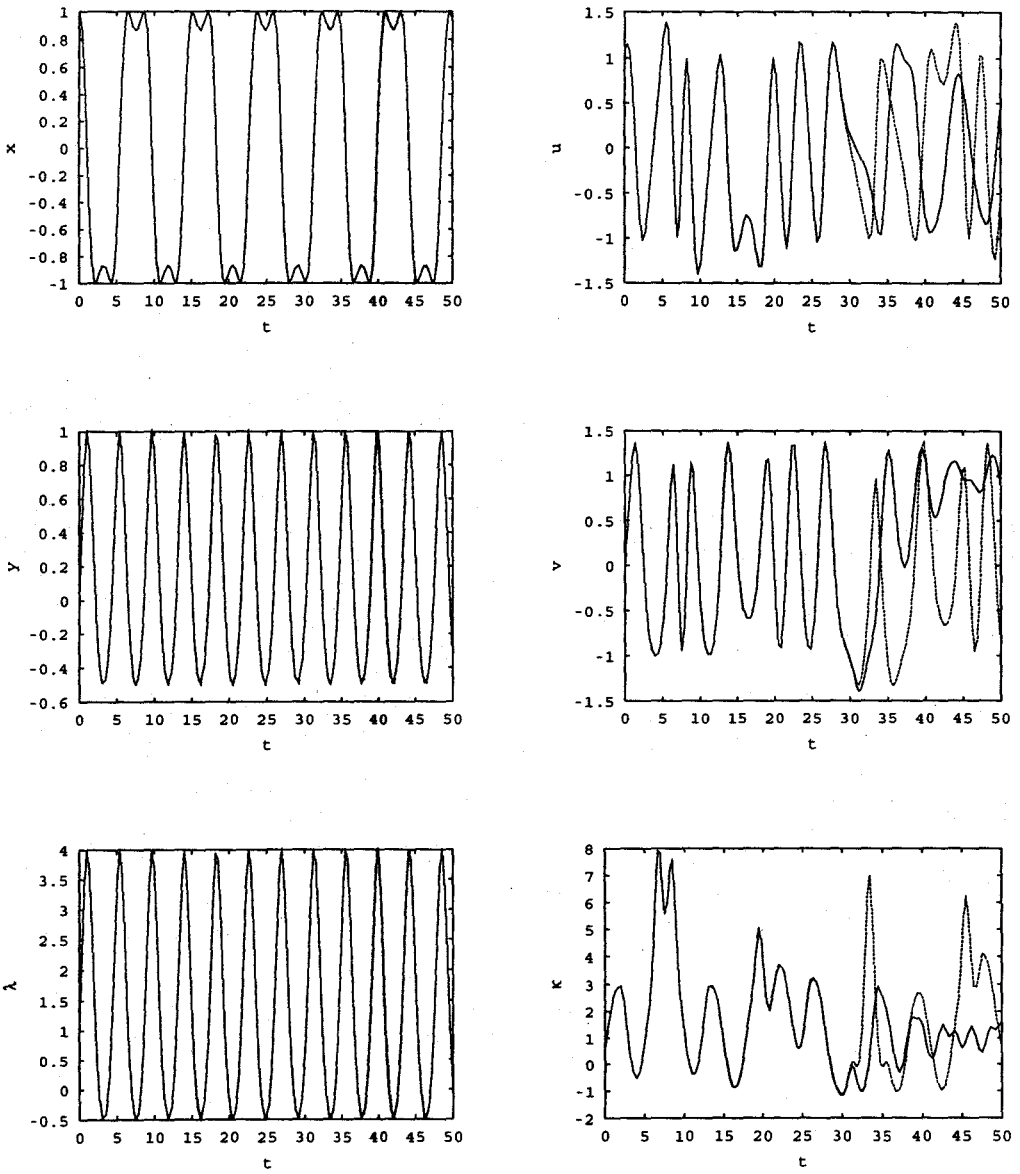


Figure 5.6: Plots of x, y, λ, u, v, and κ versus time with the initial points in Table 5.3.

	InitPoint-1	InitPoint-2
u_0	1.1000000000	1.0999994500
u_0'	$3.0000000000 \cdot 10^{-1}$	$2.9899985100 \cdot 10^{-1}$
u_0''	$-6.0909090909 \cdot 10^{-1}$	$-6.1008969446 \cdot 10^{-1}$
v_0	0.0000000000	$1.0999994500 \cdot 10^{-3}$
v_0'	1.0000000000	1.0002989999
v_0''	$1.0000000000 \cdot 10^{-1}$	$9.9938991030 \cdot 10^{-1}$
κ_0	$5.5371900826 \cdot 10^{-1}$	$5.5462727227 \cdot 10^{-1}$

Table 5.3: Initial values of u, v, and κ for the double pendula problem.

behavior. Hence, we do not discuss reference solution, work-precision diagram, and error versus tolerance diagram for the double pendula problem.

Figure 5.7 shows the behavior of x, y, λ, u, v, and κ over the integration interval with atol = rtol = 10^{-13}, dtol = $0.5 \cdot$ atol.

Table 5.4 presents the run characteristics. We use atol = rtol = $10^{-(2m+1)}$, $m = 2, \ldots, 6$ and dtol = $0.5 \cdot$ atol. In this table, HIDAETS takes the same number of steps with ADOL-C and FADBAD++.

tol	steps	CPU time (sec)		CPU time per step	
		ADOL-C	FADBAD	ADOL-C	FADBAD
10^{-5}	166	2.10	2.71	0.013	0.016
10^{-7}	197	2.68	3.26	0.014	0.017
10^{-9}	250	3.04	4.06	0.012	0.016
10^{-11}	315	3.97	4.91	0.013	0.016
10^{-13}	396	5.00	6.55	0.013	0.017

Table 5.4: Run characteristics for the double pendula problem.

Figure 5.8 exhibits stepsize behavior over the integration interval. One of the plots is with tol = 10^{-7}, the other is with tol = 10^{-13}.

Figure 5.9 displays the CPU time against order with different tolerances. From the figure, we can conclude that for the double pendulum problem, order ≈ 30 is optimal for different tolerances.

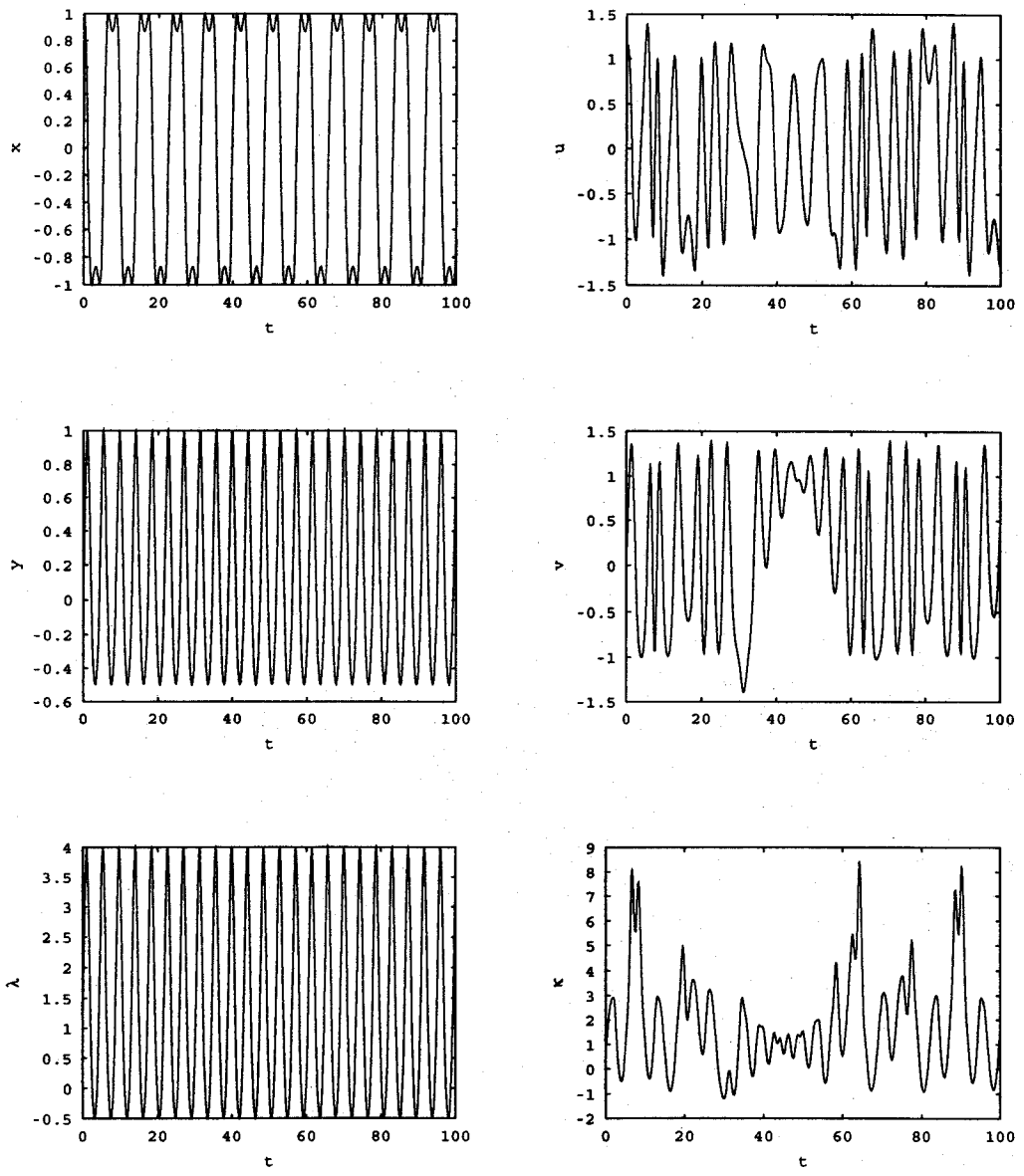


Figure 5.7: Plots of x, y, λ, u, v, and κ versus time for the double pendula problem.

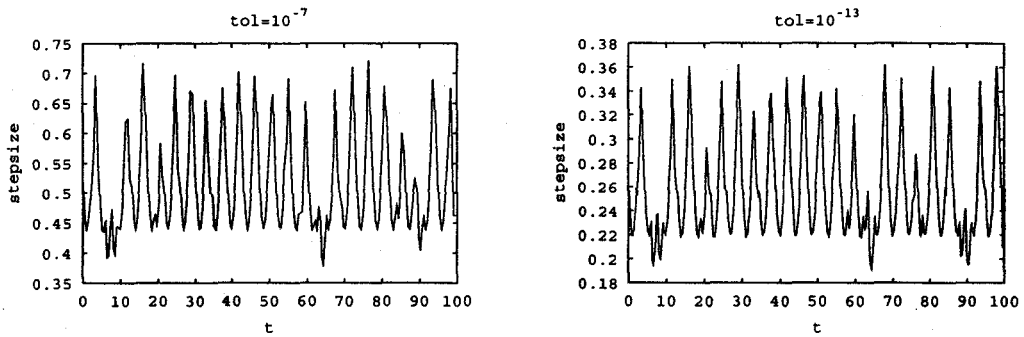


Figure 5.8: Stepsize versus time with tolerances 10^{-7} and 10^{-13}.

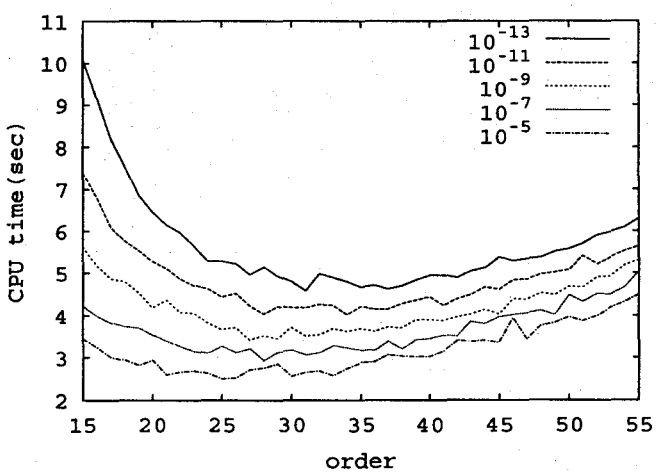


Figure 5.9: CPU time versus order with different tolerances.

5.4 Car axis

5.4.1 General information

This problem is a mildly stiff DAE of index 3, consisting of 8 differential and 2 algebraic equations.

5.4.2 Mathematical description of the problem

The problem is of the form

$$\begin{aligned} p' &= q, \\ Kq' &= f(t,p,\lambda), \qquad p,q \in \mathbb{R}^4,\ \lambda \in \mathbb{R}^2,\ 0 \le t \le 3, \\ 0 &= \phi(t,p), \end{aligned}$$

with initial conditions $p(0) = p_0$, $q(0) = q_0$, $p'(0) = q_0$, $q'(0) = q_0'$, $\lambda(0) = \lambda_0$, and $\lambda'(0) = \lambda_0'$.

The matrix K reads $\epsilon^2 \frac{M}{2} I_4$, where I_4 is the 4×4 identity matrix. The function $f : \mathbb{R}^9 \to \mathbb{R}^4$ is given by

$$f(t,p,\lambda) = \begin{pmatrix} (l_0 - l_l)\dfrac{x_l}{l_l} & +\lambda_1 x_b + 2\lambda_2(x_l - x_r) \\ (l_0 - l_l)\dfrac{y_l}{l_l} & +\lambda_1 y_b + 2\lambda_2(y_l - y_r) - \epsilon^2\dfrac{M}{2} \\ (l_0 - l_r)\dfrac{x_r - x_b}{l_r} & -2\lambda_2(x_l - x_r) \\ (l_0 - l_r)\dfrac{y_r - y_b}{l_r} & -2\lambda_2(y_l - y_r) - \epsilon^2\dfrac{M}{2}. \end{pmatrix}.$$

Here, $(x_l, y_l, x_r, y_r)^T := p$, and l_l and l_r are given by

$$\sqrt{x_l^2 + y_l^2} \quad \text{and} \quad \sqrt{(x_r - x_b)^2 + (y_r - y_b)^2}.$$

Furthermore, the functions $x_b(t)$ and $y_b(t)$ are defined by

$$\begin{aligned} x_b(t) &= \sqrt{l^2 - y_b^2(t)}, \\ y_b(t) &= r \sin(\omega t). \end{aligned}$$

The function $\phi : \mathbb{R}^5 \to \mathbb{R}^2$ reads

$$\phi(t,p) = \begin{pmatrix} x_l x_b + y_l y_b \\ (x_l - x_r)^2 + (y_l - y_r)^2 - l^2 \end{pmatrix}.$$

The constants are listed below.

l	$=$	2	ϵ	$=$	10^{-2}	τ	$=$	$\pi/5$
l_0	$=$	$1/2$	M	$=$	10	ω	$=$	10

Consistent initial values are

$$p_0 = \begin{pmatrix} 0 \\ 1/2 \\ 1 \\ 1/2 \end{pmatrix}, \quad q_0 = \begin{pmatrix} -1/2 \\ 0 \\ -1/2 \\ 0 \end{pmatrix},$$

$$q_0' = \frac{2}{Mc^2} f(0, p_0, \lambda_0), \quad \lambda_0 = \lambda_0' = (0,0)^T.$$

The index of the variables p, q, and λ is 1, 2, and 3, respectively.

5.4.3 Numerical results

For this problem, we use order = 15 except for the CPU time versus order diagram. We use FADBAD++ as the AD package.

Table 5.5 presents the reference solution at the end of the integration interval. The column with HIDAETS is the reference solution computed by HIDAETS with atol = rtol = 10^{-16} and dtol = 10^{-14} on the described setting above. The column with PSIDE is the reference solution from [LS98], which is computed on Cray C90, using PSIDE with Cray's double precision and atol = rtol = 10^{-16}. The column with GAMD is from [MI03], which is computed by GAMD with quadruple precision on an Alpha Server DS20E, with a 667 MHz EV67 processor and atol = rtol = 10^{-24}. The underlined digits are the same as the reference solution computed by HIDAETS.

Figure 5.10 shows the behavior of x_l, y_l, x_r, and y_r over the integration interval.

Var	HIDAETS	PSIDE	GAMD
x_1	$0.4934557842752397 \cdot 10^{-1}$	$0.4934557842755629 \cdot 10^{-1}$	$0.4934557842754028 \cdot 10^{-1}$
x_2	0.4969894602300073	0.4969894602303324	0.4969894602301711
x_3	$0.1041742524885424 \cdot 10$	$0.1041742524885400 \cdot 10$	$0.1041742524885421 \cdot 10$
x_4	0.3739110272653672	0.3739110272652214	0.3739110272653612
x_5	$-0.7705836840358462 \cdot 10^{-1}$	$-0.7705836840321485 \cdot 10^{-1}$	$-0.7705836840409723 \cdot 10^{-1}$
x_6	$0.7446866592147278 \cdot 10^{-2}$	$0.7446866596327776 \cdot 10^{-2}$	$0.7446866587237779 \cdot 10^{-2}$
x_7	$0.1755681575356589 \cdot 10^{-1}$	$0.1755681574942899 \cdot 10^{-1}$	$0.1755681575372322 \cdot 10^{-1}$
x_8	0.7703410437798304	0.7703410437794031	0.7703410437792519
x_9	$-0.4736886590853484 \cdot 10^{-2}$	$-0.4736886750784630 \cdot 10^{-2}$	$-0.4736886590848568 \cdot 10^{-2}$
x_{10}	$-0.1104680331259640 \cdot 10^{-2}$	$-0.1104680411345730 \cdot 10^{-2}$	$-0.1104680331257160 \cdot 10^{-2}$

Table 5.5: Reference solutions for the car axis problem.

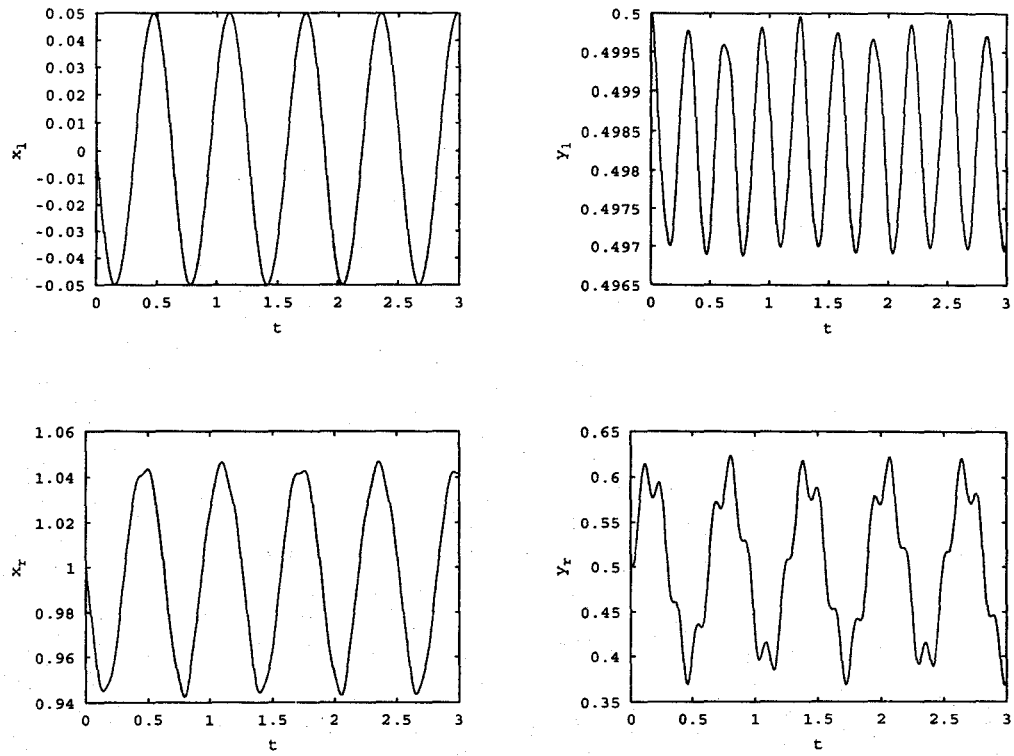


Figure 5.10: Plots of x_l, y_l, x_r, and y_r versus time for the car axis problem.

tol	steps	CPU time (sec)	CPU time per step
10^{-5}	86	1.53	0.018
10^{-7}	115	1.98	0.017
10^{-9}	157	2.70	0.017
10^{-11}	214	3.81	0.018
10^{-13}	289	4.99	0.017

Table 5.6: Run characteristics for the car axis problem.

Table 5.6 presents the run characteristics. We use atol = rtol = $10^{-(2m+1)}$, $m = 2, \ldots, 6$ and dtol = $0.5 \cdot$ atol.

Figure 5.11 shows the work-precision diagrams. We use atol = 10^{-m}, $m = 5, \ldots, 14$, rtol = atol, and dtol in (5.1).

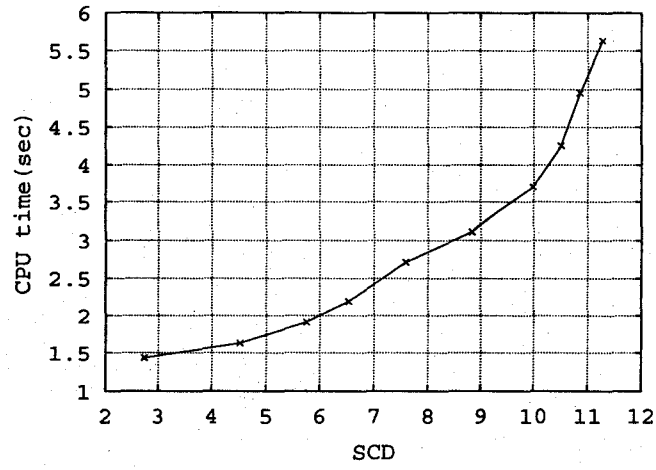


Figure 5.11: Work-precision diagram for the car axis problem.

Figure 5.12 illustrate error against different tolerances with reference solutions from HIDAETS, PSIDE, and GAMD. From this figure, we can conclude that with reference solution from PSIDE, we can obtain seven correct digits, and we can obtain at least nine correct digits compared to the reference solution from GAMD.

Figure 5.13 exhibits stepsize behavior over the integration interval. One of the plots is with tol = 10^{-7}, the other is with tol = 10^{-13}. The down-spikes near the end of the integration interval are because of the final stepsize algorithms in subsection 3.7.2.

Figure 5.14 displays the CPU time against order with different tolerances. From the figure, we can conclude that for the car axis problem, order $\approx$ 25 is optimal for different tolerances.

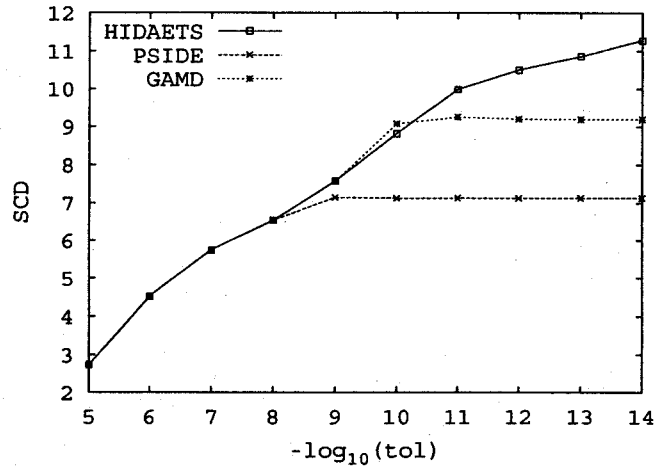


Figure 5.12: Error versus tolerance for the car axis problem.

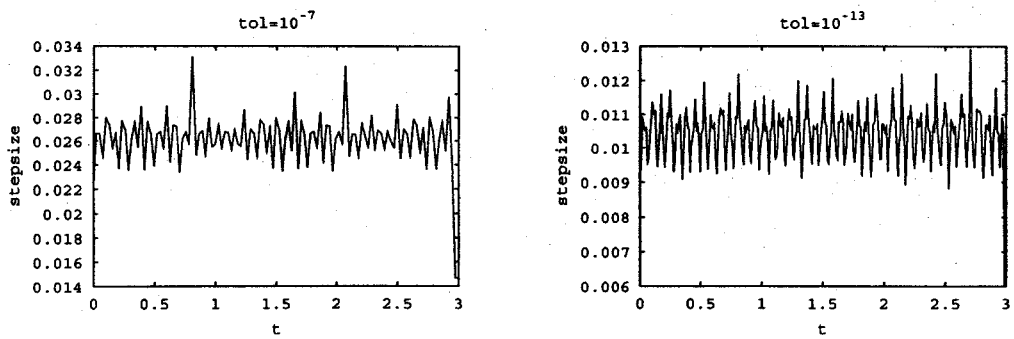


Figure 5.13: Stepsize versus time with tolerances 10^{-7} and 10^{-13}.

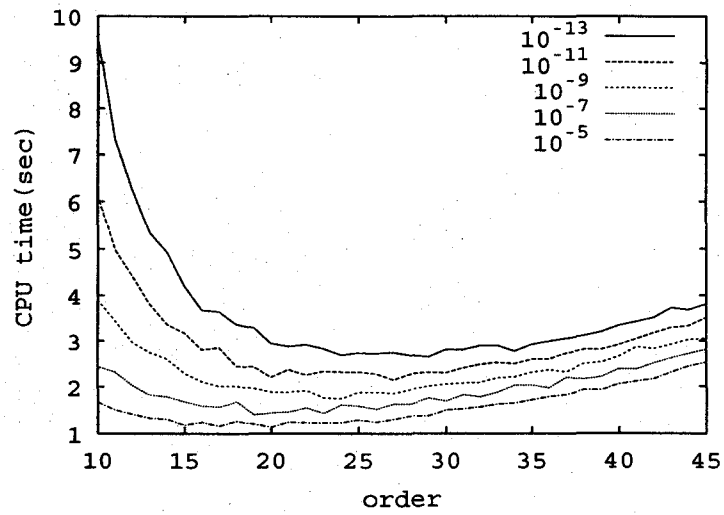


Figure 5.14: CPU time versus order with different tolerances.

5.5 Two-link robotic arm

5.5.1 General information

This problem is a slight simplification of the equations for the prescribed-path control of a two-link robotic arm [Pry98]. It is a DAE of index 5, consisting of 3 differential and 3 algebraic equations.

5.5.2 Mathematical description of the problem

The problem is of the form

$$
\begin{aligned}
0 &= x_1'' - [\nu + X(a(x_3) + 2b(x_3)) + a(x_3)\omega], \\
0 &= x_2'' - [-\nu + X(1 - 3a(x_3) - 2b(x_3)) - a(x_3)\omega + \mu_2], \\
0 &= x_3'' - [-\nu + X(a(x_3) - 9b(x_3)) - 2x_1'^2 c(x_3) - d(x_3)Y'^2 \\
&\qquad - (a(x_3) + b(x_3))\omega], \\
0 &= \cos x_1 + \cos(Y) - p_1(t), \\
0 &= \sin x_1 + \sin(Y) - p_2(t), \\
0 &= \omega - (\mu_1 - \mu_2),
\end{aligned}
$$

where

$$
\begin{aligned}
p_1(t) &= \cos(e^t - 1) + \cos(t - 1), \\
p_2(t) &= \sin(1 - e^t) + \sin(1 - t), \\
a(s) &= \frac{2}{2 - \cos^2 s}, \quad b(s) = \frac{\cos s}{2 - \cos^2 s}, \\
c(s) &= \frac{\sin s}{2 - \cos^2 s}, \quad d(s) = \frac{\cos s \sin s}{2 - \cos^2 s}, \\
X &= 2x_3 - x_2, \quad Y = x_1 + x_3, \\
\nu &= 2Y'^2 c(x_3) + x_1'^2 d(x_3).
\end{aligned}
$$

By construction, the solution has $x_1 = 1 - e^t$, $x_3 = e^t - t$. Here we integrate the problem from $t_0 = 0$ to $t_{\text{end}} = 1.3$, with initial conditions $x_1(0) = 0$, $x_3(0) = 1$.

5.5.3 Numerical solution of the problem

For this problem, we use order = 15 except for the CPU time versus order diagram. We use FADBAD++ as the AD package.

Table 5.7 presents the reference solution at the end of the integration interval. The reference solution is computed with atol = rtol = 10^{-16} and dtol = 10^{-14} on the described setting before.

x_1	-2.6692966676192422	x_2	2.6578533275805367
x_3	2.3692966676192442	μ_2	$2.2158319076934220 \cdot 10$
ω	$-6.5122431545546700 \cdot 10^{-1}$	μ_1	$2.1507094761478751 \cdot 10$

Table 5.7: Reference solution for the two-link robotic arm.

Figure 5.15 illustrates relative errors between solutions of x_1 and x_3 and their true solutions over the integration interval. In this figure, the relative error lines are discontinuous. That means the relative errors reach machine precision at the points without plotting. This figure is a strong evidence of the reference solution's accuracy.

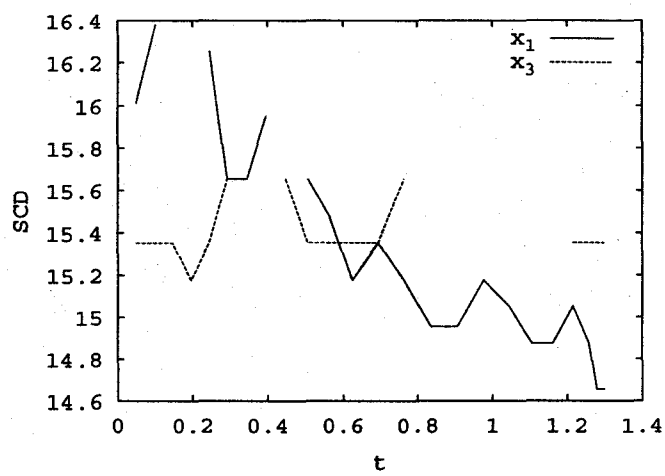


Figure 5.15: Comparison of reference solutions with true solutions x_1 and x_3.

Figure 5.16 shows the behavior of x_1, x_3, ω, x_2, μ_2, μ_1 over the integration interval.

Table 5.8 presents the run characteristics. We use atol = rtol = $10^{-(2m+1)}$, $m = 2, \ldots, 6$ and dtol = $0.5 \cdot$ atol for IPOPT.

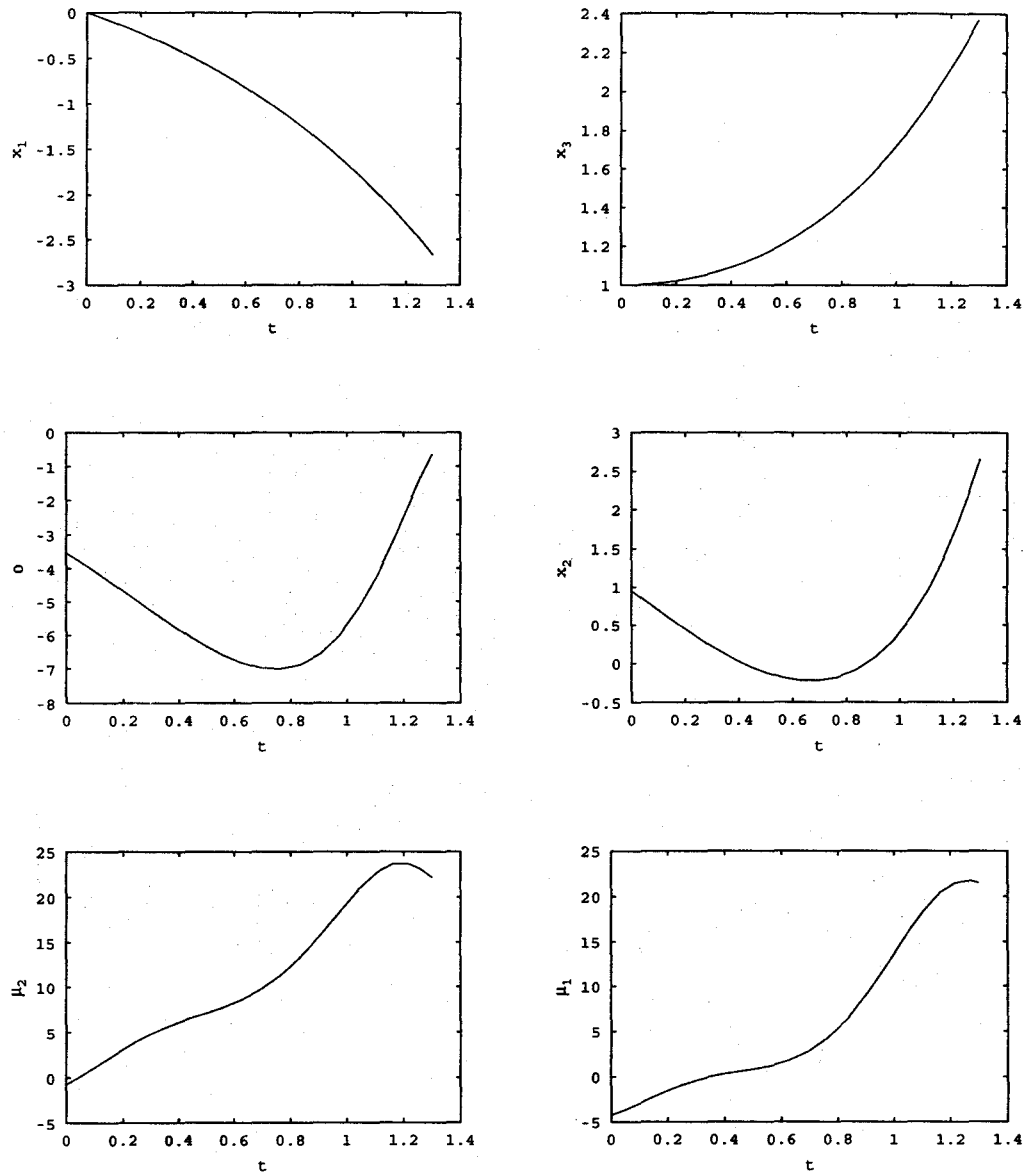


Figure 5.16: Plots of x_1, x_3, ω, x_2, μ_2, and μ_1 versus time for the two-link robotic arm.

tol	steps	CPU time (sec)	CPU time per step
10^{-5}	5	0.34	0.068
10^{-7}	6	0.40	0.067
10^{-9}	8	0.52	0.065
10^{-11}	11	0.63	0.057
10^{-13}	15	0.89	0.059

Table 5.8: Run characteristics for the two-link robotic arm.

Figure 5.17 shows the work-precision diagram. We use atol $= 10^{-m}$, $m = 5, \ldots, 14$, rtol = atol, and dtol in (5.1).

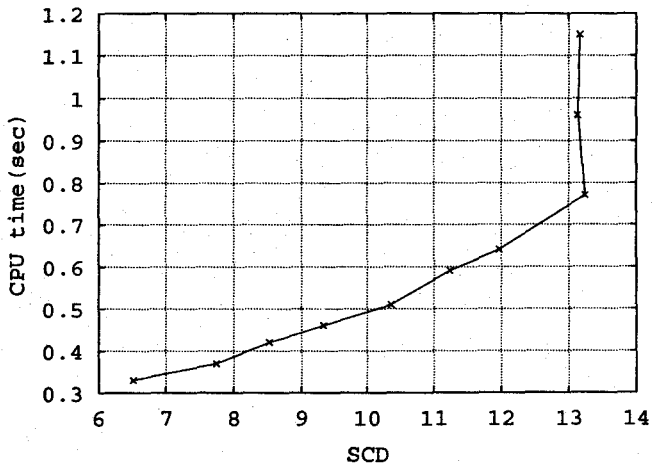


Figure 5.17: Work-precision diagram for the two-link robotic arm.

Figure 5.18 illustrates error against different tolerances.

Figure 5.19 exhibits stepsize behavior over the integration interval. One of the plots is with tol $= 10^{-7}$, the other is with tol $= 10^{-13}$.

Figure 5.20 displays the CPU time against order with different tolerances. From the figure, we can conclude that for the two-link robotic arm problem, order ≈ 20 is optimal for different tolerances.

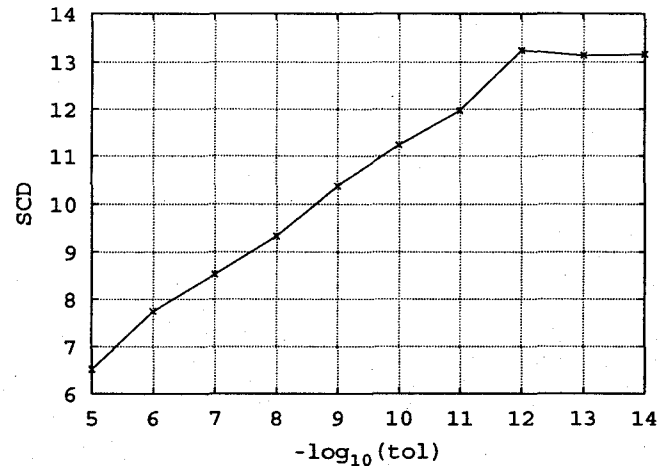


Figure 5.18: Error versus tolerance for the two-link robotic arm.

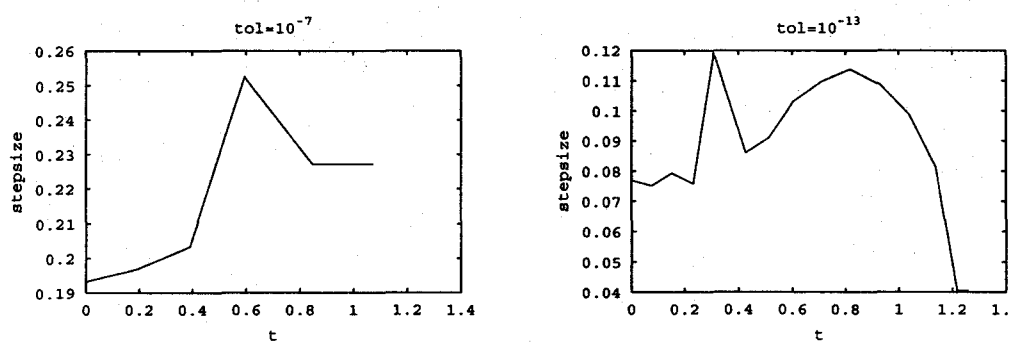


Figure 5.19: Stepsize versus time with tolerances 10^{-7} and 10^{-13}.

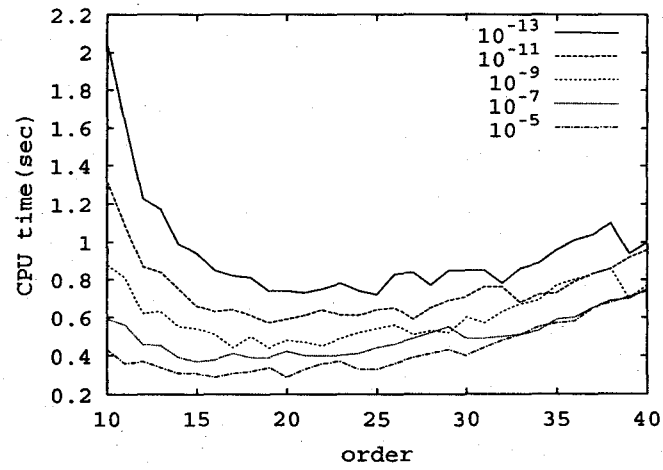


Figure 5.20: CPU time versus order with different tolerances.

5.6 Transistor amplifier

5.6.1 General information

The problem is a stiff DAE of index 1, consisting of 8 equations.

5.6.2 Mathematical description of the problem

The problem is of the form

$$M\frac{dy}{dt} = f(y), \quad y(0) = y_0, \quad y'(0) = y'_0,$$

with $y \in \mathbb{R}^8$, $0 \le t \le 0.2$.

The matrix M is of rank 5 and given by

$$M = \begin{pmatrix} -C_1 & C_1 & 0 & 0 & 0 & 0 & 0 & 0 \\ C_1 & -C_1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -C_2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -C_3 & C_3 & 0 & 0 & 0 \\ 0 & 0 & 0 & C_3 & -C_3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -C_4 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -C_5 & C_5 \\ 0 & 0 & 0 & 0 & 0 & 0 & C_5 & -C_5 \end{pmatrix},$$

and the function f by

$$\begin{pmatrix} -\frac{U_e(t)}{R_0} + \frac{y_1}{R_0} \\ -\frac{U_b}{R_2} + y_2(\frac{1}{R_1} + \frac{1}{R_2}) - (\alpha - 1)g(y_2 - y_3) \\ -g(y_2 - y_3) + \frac{y_3}{R_3} \\ -\frac{U_b}{R_4} + \frac{y_4}{R_4} + \alpha g(y_2 - y_3) \\ -\frac{U_b}{R_6} + y_5(\frac{1}{R_5} + \frac{1}{R_6}) - (\alpha - 1)g(y_5 - y_6) \\ -g(y_5 - y_6) + \frac{y_6}{R_7} \\ -\frac{U_b}{R_8} + \frac{y_7}{R_8} + \alpha g(y_5 - y_6) \\ \frac{y_8}{R_9} \end{pmatrix}.$$

Here g and U_e are auxiliary functions given by

$$g(x) = \beta(e^{\frac{x}{U_F}} - 1) \quad \text{and} \quad U_e(t) = 0.1\sin(200\pi t).$$

The values of the parameters are:

$U_b = 6,$		$R_0 = 1000,$			
$U_F = 0.026,$		$R_k = 9000$	for	$k = 1, \ldots, 9,$	
$\alpha = 0.99,$		$C_k = k \cdot 10^{-6}$	for	$k = 1, \ldots, 5.$	
$\beta = 10^{-6},$					

Consistent initial values at $t = 0$ are

$$y_0 = \begin{pmatrix} 0 \\ U_b/(\frac{R_2}{R_1}+1) \\ U_b/(\frac{R_2}{R_1}+1) \\ U_b \\ U_b/(\frac{R_6}{R_5}+1) \\ U_b/(\frac{R_6}{R_5}+1) \\ U_b \\ 0 \end{pmatrix}, \quad y_0' = \begin{pmatrix} 51.338775 \\ 51.338775 \\ -U_b/((\frac{R_2}{R_1}+1)(C_2 \cdot C_3)) \\ -24.9757667 \\ -24.9757667 \\ -U_b/((\frac{R_6}{R_5}+1)(C_4 \cdot C_7)) \\ -10.00564453 \\ -10.00564453 \end{pmatrix}.$$

5.6.3 Numerical results

For this problem, we use order = 15 except for the CPU time versus order diagram. We use FADBAD++ as the AD package.

Table 5.9 presents the reference solution at the end of the integration interval. The column with HIDAETS is the reference solution computed by HIDAETS with atol = rtol = 10^{-15} and dtol = 10^{-14} on the described setting before. The column with PSIDE is the reference solution from [MI03], which is computed on Cray C90, using PSIDE with Cray's double precision and atol = rtol = 10^{-14}. The underlined digits are the same as the reference solution computed by HIDAETS.

Figure 5.21 shows the behavior of the solution over the integration interval.

Table 5.10 presents the run characteristics. We use atol = rtol = $10^{-(2m+1)}$, $m = 2, \ldots, 6$ and dtol = $0.5 \cdot$ atol.

Figure 5.22 shows the work-precision diagrams. We use atol = 10^{-m}, $m = 5, \ldots, 14$, rtol = atol, and dtol in (5.1).

Figure 5.23 illustrate error against different tolerances with reference solutions from HIDAETS and PSIDE. From the figure, we can conclude that with reference solution from PSIDE, we can obtain at least 11 correct digits.

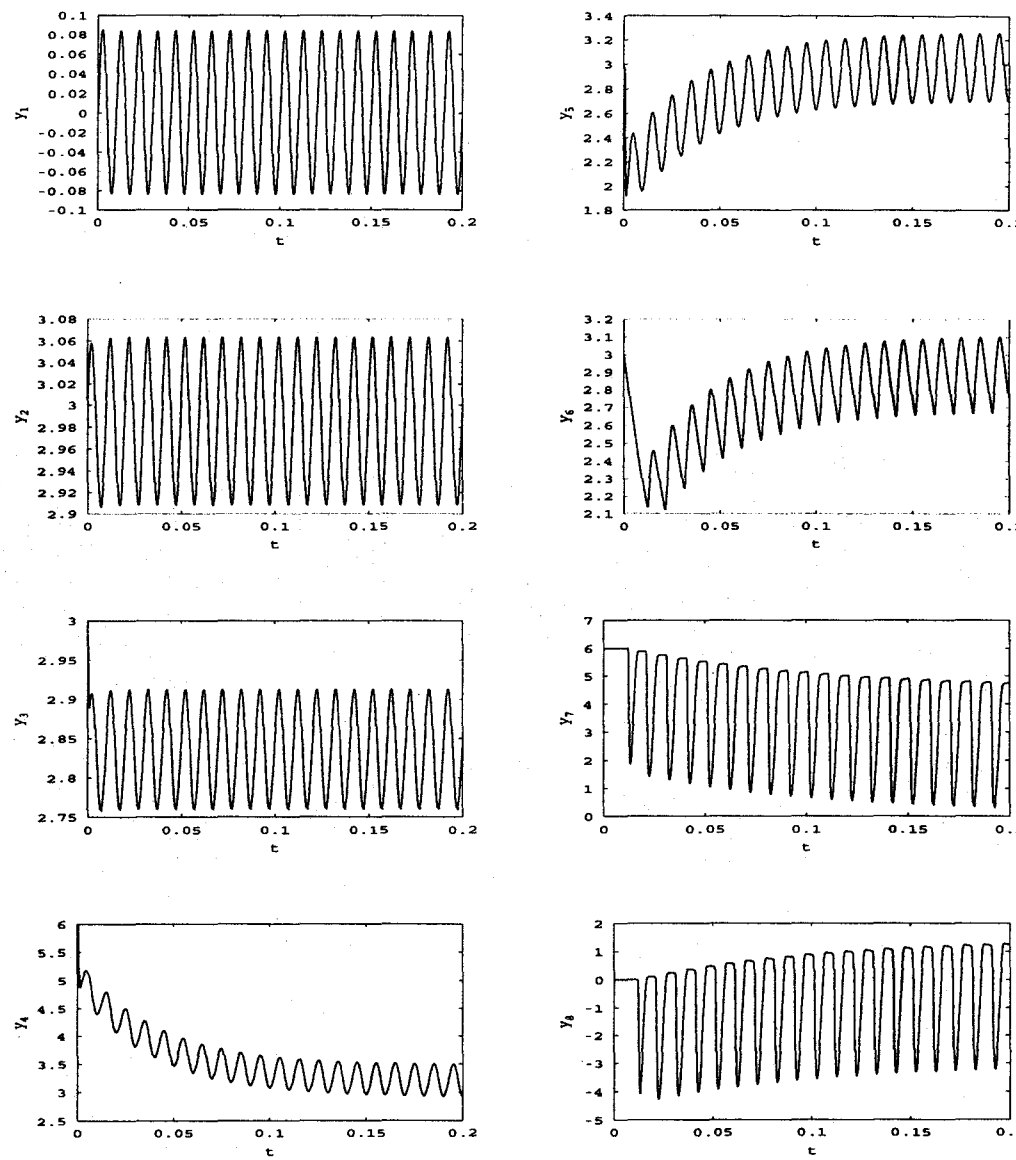


Figure 5.21: Plots of y_1, y_2, y_3, y_4, y_5, y_6, y_7, and y_8 versus time for the transistor amplifier problem.

Var	HIDAETS	PSIDE
y_1	$-0.5562145012259387 \cdot 10^{-2}$	$-0.5562145012262709 \cdot 10^{-2}$
y_2	$0.3006522471902849 \cdot 10$	$0.3006522471903042 \cdot 10$
y_3	$0.1041742524885424 \cdot 10$	$0.1041742524885400 \cdot 10$
y_4	0.2849958788607940	0.2849958788608128
y_5	$0.2704617865005508 \cdot 10$	$0.2704617865010554 \cdot 10$
y_6	$0.2761837778388728 \cdot 10$	$0.2761837778393145 \cdot 10$
y_7	$0.4770927631615038 \cdot 10$	$0.4770927631616772 \cdot 10$
y_8	$0.1236995868081448 \cdot 10$	$0.1236995868091548 \cdot 10$

Table 5.9: Reference solutions for the transistor amplifier problem.

tol	steps	CPU time (sec)	CPU time per step
10^{-5}	361	2.43	0.007
10^{-7}	496	3.27	0.007
10^{-9}	677	4.42	0.007
10^{-11}	901	6.17	0.007
10^{-13}	1244	8.10	0.007

Table 5.10: Run characteristics for the transistor amplifier problem.

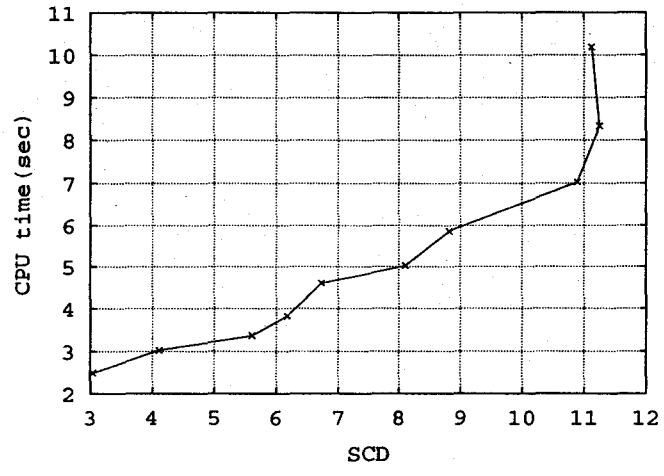


Figure 5.22: Work-precision diagram for the transistor amplifier problem.

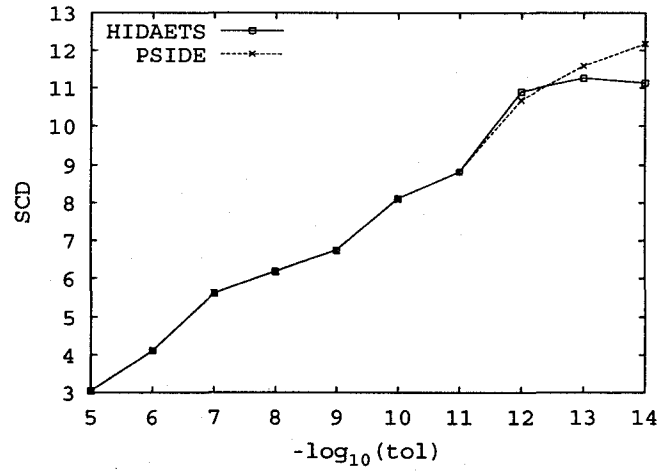


Figure 5.23: Error versus tolerance for the transistor amplifier problem.

Figure 5.24 exhibits stepsize behavior over the integration interval. One of the plots is with tol = 10^{-7}, the other is with tol = 10^{-13}.

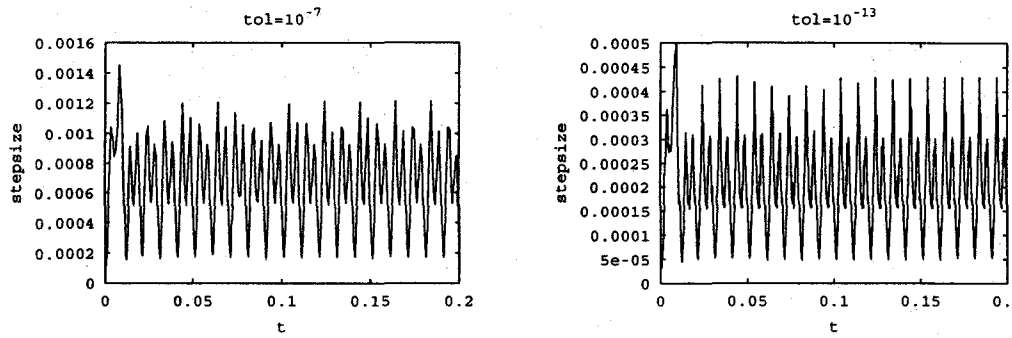


Figure 5.24: Stepsize versus time with tolerances 10^{-7} and 10^{-13}.

Figure 5.25 displays the CPU time against order with different tolerances. From the figure, we can conclude that for the transistor amplifier problem, order $\approx$ 25 is optimal for different tolerances.

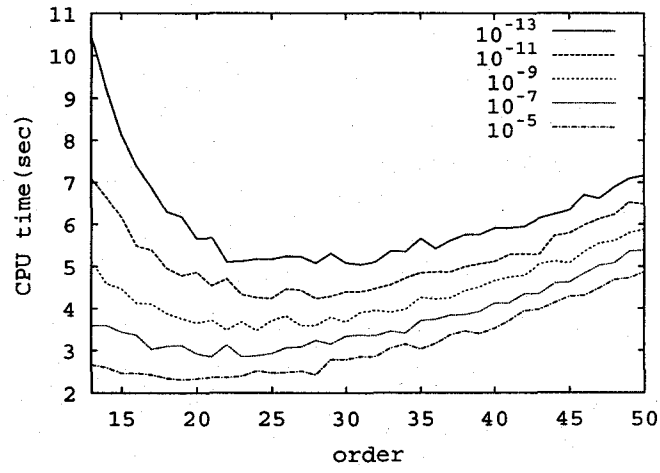


Figure 5.25: CPU time versus order with different tolerances.

5.7 Summary of numerical results

The above numerical results demonstrate that HIDAETS is efficient, accurate, and suitable for solving high-index DAE initial value problems. Here we present a brief summary of our numerical experience.

1. HIDAETS works well with a DAE problem whose index is too high for the existing solvers. The highest index we have solved is five (the double pendula problem and the two-link robotic arm problem). HIDAETS can also solve higher-order ODEs directly.

2. Due to the high order of Taylor series methods, HIDAETS can obtain much more accurate results than standard DAE solvers.

3. On nonstiff and mildly stiff problems, HIDAETS performs (very) well, especially at high accuracy. For stiff problems, HIDAETS currently cannot run fast due to the explicit nature of Taylor methods.

4. For all tested problems with range of tolerances from 10^{-5} to 10^{-13}, the running time of HIDAETS is close to optimal when the order is between 20 and 30. As we do not have variable order control at present, we recommend orders between 20 and 30.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

We have presented a numerical method for solving high-index DAEs. Given a DAE described by a computer program, the necessary structural analysis data is obtained via operator overloading. After that, we compute an initial consistent point based on the user's input. If we can obtain a consistent point, we continue the integration; if not, we cannot solve the given DAE problem.

After obtaining a consistent initial point, we compute TCs by automatic differentiation. We also compute an appropriate stepsize subject to the tolerance. With this stepsize, we compute a TS solution by summing the series.

To ensure that this numerical solution is a consistent one, we project it. If we cannot obtain a consistent point, we reduce the current stepsize, recompute the TS solution, and project it again. We iterate this process, until we obtain a consistent point.

We repeat the above process till $t = t_{\text{end}}$. In this case, our method succeeds in solving the given DAE problem.

We have presented the specification, design, implementation, and usage of HI-DAETS. We have covered most aspects of developing a numerical software package on using an object-oriented approach. In addition to the documentation in this thesis, we provide Web based documentation through the Doxygen [vH04] documentation system.

Beside these, we report detailed numerical results for five typical high-index DAE problems. Testing results show that HIDAETS is an efficient and accurate solver for IVPs in high-index DAEs.

6.2 Future works

The numerical results for the car axis problem [MI03] shows that HIDAETS is efficient for moderately stiff DAEs, however, because of their explicit nature, Taylor series methods are not efficient for integrating very stiff DAEs. For example, HIDAETS is very inefficient on the HIRES problem [MI03] as the stepsize is about 10^{-6} through the integration interval. Nedialkov [Ned99] presented a promising approach to generalize the Taylor series to stiff ODE problems by Hermite-Obreschkoff (HO) methods. We can enable HIDAETS to solve highly stiff DAEs if employing HO methods. Besides Taylor series, computing relevant Jacobians efficiently with the HO approach remains to be studied.

The examples in HIDAETS include ten DAEs and ODEs which are mostly from literature. More testing and practical engineering problems need to be performed.

Future work will include study of HO methods, enabling HIDAETS applicable to both non-stiff and stiff DAEs, developing an elaborate numerical IVP test set, and applying HIDAETS to engineering applications.

Bibliography

[AP98] U. M. Ascher and L. R. Petzold. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations.* Soc for Industrial & Applied Math, July 1998.

[BCC+92] C. H. Bischof, A. Carle, G. F. Corliss, A. Griewank, and P. Hovland. ADIFOR: Generating derivative codes from Fortran programs. *Scientific Programming*, 1(1):11–29, 1992.

[BCP96] K. E. Brenan, S. L. Campbell, and L. R. Petzold. *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations.* Soc for Industrial & Applied Math, January 1996.

[Ber97] M. Berz. COSY INFINITY version 8 reference manual. Technical Report MSUCL-1088, national Superconducting Cyclotron Lab., Michigan State University, East Lansing, Mich., 1997.

[BLV03] C. Bischof, B. Lang, and A. Vehreschild. Automatic Differentiation for MATLAB Programs. *Proc. Appl. Math. Mech.*, 2(1):50–53, 2003.

[BS96a] C. Bendtsen and O. Stauning. FADBAD, a flexible C++ package for automatic differentiation. Technical report, Department of Mathematical Modelling, Technical University of Denmark, Lyngby, Denmark, August 1996.

[BS96b] C. Bendtsen and O. Stauning. FADBAD, a flexible C++ package for automatic differentiation. Technical Report IMM-REP-1996-17, Department of Mathematical Modelling, Technical University of Denmark, Lyngby, Denmark, Aug 1996.

[Cam95] S. L. Campbell. High-index differential algebraic equations. *Mech. Struct. & Mach.*, 23(2):199–222, 1995.

[CC82] G. Corliss and Y. F. Chang. Solving ordinary differential equations using Taylor series. *ACM Transactions on Mathematical Software*, 8(2):114–144, 1982.

[CC94] Y. F. Chang and G. F. Corliss. ATOMFT: Solving ODEs and DAEs using Taylor series. *Computers and Mathematics with Application*, 28:209–233, 1994.

[CG95] S. L. Campbell and C. W. Gear. The index of general nonlinear DAEs. *Numerische Mathematik*, 72:173–196, 1995.

[CH96] S. L. Campbell and R. Hollenbeck. Automatic differentiation and implicit differential equations. In M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 215–227. SIAM, Philadelphia, PA, 1996.

[CL96] G. F. Corliss and W. Lodwick. Role of constraints in the validated solution of DAEs. Technical report, Marquette University Department of Mathematics, Statistics, and Computer Science, Milwaukee, Wisc., March 1996.

[GJU96] A. Griewank, D. Juedes, and J. Utke. ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. *ACM Transactions on Mathematical Software*, 22(2):131–167, June 1996.

[Gri93] A. Griewank. *Complexity in Nonlinear Optimization*, chapter Some Bounds on the Complexity of Gradients, Jacobians, and Hessians, pages 128–161. World Scientific, 1993.

[GW04] A. Griewank and A. Walther. On the efficient generation of Taylor expansions for DAE solutions by automatic differentiation. Technical report, Technische Universität Dresden, Department of Mathematics, Institute of Scientific Computing, 2004.

[HW96] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems.* Springer-Verlag, 1996.

[JV87] R. Jonker and A. Volgenant. A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computing*, 38:325–340, 1987.

[LS98] W. M. Lioen and J. B. Swart. Test set for initial value problem solvers. Technical report, Modeling, Analysis and Simulation (MAS), CWI, Amsterdam, Netherlands, December 1998.

[MI03] F. Mazzia and F. Iavernaro. Test set for initial value problem solvers. Technical report, Department of Mathematics, University of Bari, Italy, August 2003.

[MS93] S. E. Mattsson and G. Söderlind. Index reduction in differential-algebraic equations using dummy derivatives. *SIAM. J. Sci. Comput.*, 14:677–692, 1993.

[Ned99] N. S. Nedialkov. *Computing Rigorous Bounds on the Solution of an Initial Values Problems for an Ordinary Differential Equation.* PhD thesis, Department of Computer Science, University of Toronto, Toronto, Canada, M5S 3G4, February 1999.

[NJ02] N. S. Nedialkov and K. R. Jackson. The design and implementation of a validated object-oriented solver for IVPs for ODEs. Technical Report 6, Software Quality Research Laboratory, Department of Computing and Software, McMaster University, Hamilton, Canada, L8S 4L7, 2002.

[NP03] N. S. Nedialkov and J. D. Pryce. Solving differential-algebraic equations by Taylor series(I): computing Taylor coefficients. *BIT*, pages 001–038, 2003.

[Pan88] C. C. Pantelides. The consistent initialization of differential-algebraic systems. *SIAM. J. Sci. Stat. Comput*, 9:213–231, 1988.

[Pry98] J. D. Pryce. Solving high-index DAEs by Taylor series. *Numerical Algorithms*, 19:195–211, 1998.

[Pry01] J. D. Pryce. A simple structural analysis method for DAEs. *BIT*, 41(2):364–394, 2001.

[vH04] D. van Heesch. *User Manual for Doxygen 1.4.3*, 2004.

[WB04] A. Wächter and L. T. Biegler. On the implementation of a primal-dual interior point filter line search algorithm for large-scale nonlinear programming. Technical report, IBM T. J. Watson Research Center, Yorktown, USA, March 2004.

Appendix A

Symbols and Acronyms

Name	Definition
AD	Automatic Differentiation
AWA	Anfangswertaufgabe
BDF	Backward Differentiation Formula
DAE	Differential-Algebraic Equation
DAETS	DAE by Taylor Series
DOF	Degree of Freedom
GAM	Generalized Adams Methods
HIDAETS	High-Index DAE by Taylor Series
HVT	Highest Value Transversal
IRK	Implicit Runge-Kutta
IVP	Initial Value Problem
$\mathbf{J}$	System Jacobian
LP	Linear Programming
ODE	Ordinary Differential Equation
SA	Structural Analysis
SCD	Significant Correct Digits
TC	Taylor Coefficient
TS	Taylor Series
Σ	Signature Matrix
∇	Gradient
$\triangleright$	Comment

Appendix B

Automatic Differentiation

Automatic differentiation is a set of techniques based on the chain rule to obtain derivatives of a function given as a computer program [Gri93]. By applying the chain rule of derivative calculus repeatedly to a sequence of elementary arithmetic operations, derivatives of arbitrary order can be computed automatically and accurate to working precision.

The problem is defined by a computer program. The application of AD to this computer program results in the automatic generation of a new computer program, which computes the derivatives of the output variables with respect to the input variables.

AD methods are widely used for solving problems in which the output variables are computed "directly" from the input variables. It provides fast and accurate values of derivative objects, such as gradients, Jacobians, and Hessians, which are required by modern tools for optimization, nonlinear systems, differential equations, or sensitivity analysis.

Here, we first introduce basic AD algorithms, then discuss its implementation and existing AD tools briefly.

B.1 Basics of AD

Abstractly, a program for evaluating an m-vector y as a function of n-vector x has the form:

$$\begin{aligned} x &= (x_1, x_2, ..., x_n) \\ &\downarrow \\ z &= \{z_1, z_2, ..., z_p\}, \quad p \gg m+n \\ &\downarrow \\ y &= (y_1, y_2, ..., y_m), \end{aligned}$$

where the intermediate variables z are related through a series of elementary functions which may be unary,

$$z_k = f^k_{\text{elem}}(z_i), \quad i < k$$

consisting of operations such as ($-$, pow, sin, ...), or binary,

$$z_k = f^k_{\text{elem}}(z_i, z_j), \quad i, j < k$$

such as (+, /, ...).

AD has two basic approaches, the forward mode and the reverse mode. We illustrate them in the following two subsections.

B.1.1 Forward mode

In the forward mode, derivatives are propagated throughout the computation using the chain rule. For example, for the elementary step $z_k = f^k_{\text{elem}}(z_i, z_j)$, the intermediate derivative dz_k/dx, can be propagated in the forward mode as:

$$\frac{dz_k}{dx} = \frac{\partial f^k_{\text{elem}}}{\partial z_i}\frac{dz_i}{dx} + \frac{\partial f^k_{\text{elem}}}{\partial z_j}\frac{dz_j}{dx}.$$

This chain rule-based computation is done for all the intermediate variables z and for the output variables y, finally yielding the derivative dy/dx.

Example B.1. Consider $f(x) = (x+x^2)^2$. We want to compute df/dx by the forward mode of AD.

Let:

$$z_1 = x, \tag{B.1}$$
$$z_2 = z_1 * z_1, \tag{B.2}$$
$$z_3 = z_1 + z_2, \tag{B.3}$$
$$z_4 = z_3 * z_3. \tag{B.4}$$

Then, we have

$$\frac{dz_1}{dx} = 1, \quad \text{(from B.1)}$$
$$\frac{dz_2}{dx} = 2z_1\frac{dz_1}{dx} = 2x, \quad \text{(from B.2)}$$
$$\frac{dz_3}{dx} = \frac{dz_1}{dx} + \frac{dz_2}{dx} = 1 + 2x, \quad \text{(from B.3)}$$
$$\frac{dz_4}{dx} = 2z_3\frac{dz_3}{dx} = 2(x + xx)(1 + 2x). \quad \text{(from B.4)}$$

B.1.2 Reverse mode

The reverse mode computes the derivatives dy/dz_k for all intermediate variables in reverse order. For example, for the elementary step $z_k = f^k_{\text{elem}}(z_i, z_j)$, the derivatives are propagated as:

$$\frac{dy}{dz_i} = \frac{\partial f^k_{\text{elem}}}{\partial z_i}\frac{dy}{dz_k} \quad \text{and} \quad \frac{dy}{dz_j} = \frac{\partial f^k_{\text{elem}}}{\partial z_j}\frac{dy}{dz_k}. \tag{B.5}$$

Here is the explanation for (B.5). Suppose we have obtained dy/dz_k. For $z_k = f^k_{\text{elem}}(z_i, z_j)$, we want to compute dy/dz_i. We have

$$\frac{dy}{dz_i} = \frac{dy}{dz_k}\frac{\partial z_k}{\partial z_i} = \frac{\partial f^k_{\text{elem}}}{\partial z_i}\frac{dy}{dz_k}.$$

At the end of the computation, the derivative dy/dx is obtained. The key is that the derivative propagation is done in reverse manner, hence, we need dy/dz_k to compute derivatives dy/dz_i, dy/dz_j. At the beginning, dy/dy is initialized to 1.

Example B.2. For $f(x) = (x + x^2)^2$, we process the reverse mode as

$$\begin{aligned}
&\frac{dz_4}{dz_4} = 1, \\
&\frac{dz_4}{dz_3} = \frac{dz_4}{dz_3}\frac{dz_4}{dz_4} = 2z_3 = 2(x + xx), && \text{(from B.4)} \\
&\frac{dz_4}{dz_2} = \frac{dz_4}{dz_3}\frac{\partial z_3}{\partial z_2} = 1 \cdot 2(x + xx) = 2(x + xx), && \text{(from B.3)} \\
&\frac{dz_4}{dz_1} = \frac{dz_4}{dz_3}\frac{\partial z_3}{\partial z_1} + \frac{dz_4}{dz_2}\frac{dz_2}{dz_1} = 2(x + xx)(1 + 2x), && \text{(from B.2, B.3)} \\
&\frac{dz_4}{x} = \frac{dz_4}{dz_1}\frac{dz_1}{dx} = 2(x + xx)(1 + 2x). && \text{(from B.1)}
\end{aligned}$$

B.2 AD tools

Implementations of AD can be broadly classified into source transformation and operator overloading. Source transformation changes the semantics by explicitly rewriting the code, while operator overloading exploits the possibility of some modern programming languages to redefine the semantics of elementary operators.

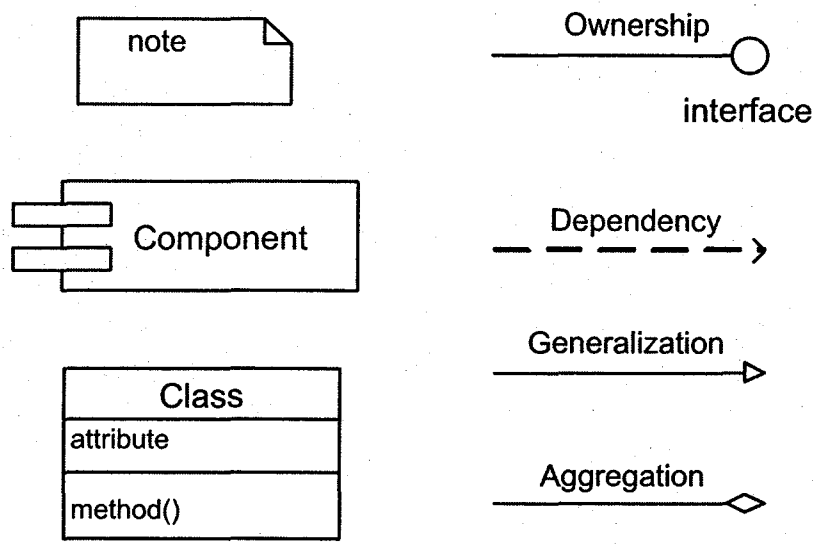
Table B.1 lists some existing and typical AD software packages. More details and packages can be found at `http://www.autodiff.org`. In Table B.1, O stands for operator overloading, and S stands for source transformation.

Name	Author(s)	Language	Mode	Method
ADOL-C	Griewank, Juedes, and Utke [GJU96]	C/C++	Forward, Reverse	O
FADBAD++	Stauning and Bendtsen [BS96b]	C/C++	Forward, Reverse	O
ADiMat	Vehreschild [BLV03]	MATLAB	Forward	S O
ADIFOR	Bischof, Carle, Corliss, Griewank, and Hovland [BCC+92]	Fortran77	Forward	S

Table B.1: Some existing AD tools.

Appendix C

Some UML Legends



Appendix D

Some Source Code

D.1 The integration process

```
void DAESolver :: integrate( double& t0, double& tend, InitPoint & x,
                             void *DAEParam )
{
  int p, ind, outputind;

  ind = ptrParam->getInd();
  if ( ind == normalexit )
    return;

  outputind = ptrParam->getOutput();
  p = ptrParam->getOrder();

  if ( firstentry == true )
    initialization( t0, x, DAEParam, ind, outputind );

  if ( ptrParam->getInd() == Jsingular )
    return;

  if( ind == tillend )
    {
      cout<<"\n\nINTEGRATION PROCESS";
      cout<<"\n\n   Integrated at     Local Error"<<endl;
    }
```

```
  while( t < tend )
    {

      compTSSol( t, tend, ind, p );

      compConsSol( t, p );

      fout<<h<<endl;

      if ( ind == onestep )
      {
        t0 = t + h;
        ptrAD->getInitialPoint( x );
        cout<<endl<<"\n\nSOLUTION WITH STEPSIZE "<<h<<endl<<endl;
        ptrAD->printInitialPoint( 0, ptrParam->getDigits() );
        cout<<endl<<endl;
        ptrParam->setInd( normalexit );
        break;
      }

      t = t + h;
      t0 = t;
      fout<<t<<" ";
      err = ptrErrorEst->estError( ptrAD, p, h );
      if( outputind >= process )
      PrintProgress( t, err );
      ptrAD->outputSolution( fout );
    }

  fout.close();
  ptrStats->statTime();

  if( outputind >= solution && ( ind == tillend ) )
    printSolution( ptrParam->getDigits() );
  if( outputind >= stat && ( ind == tillend ) )
    printStats();
  ptrParam->setInd( normalexit );
  firstentry = true;
}
```

```
void DAESolver :: initialization( double t0, InitPoint & x, void *DAEParam,
                                  int & ind, int & outputind )
{

  if( outputind >= offsets )
    ptrOffsets->printOffsets();

  t = t0;

  ptrStats->resetStats();

  OpenFile( filename );

  ptrAD->setInitialPoint( x );
  ptrAD->setIntegrationTime( t0 );

  if( ind == tillend || ind == onestep || ind == Jacobian )
    ptrAD->generateCompGraph( DAEParam );

  if( !ptrProjection->projectConsInitPoint( ) )
    {
      ptrParam->setInd( Jsingular );
      return;
    }

  if( ind == tillend || ind == onestep )
    {
      fout<<t<<" ";
      ptrAD->outputSolution( fout );
    }

  if( outputind >= initpoint )
    {
      cout<<"\n\nINITIAL POINT\n\n";
      ptrAD->printInitialPoint( 0, ptrParam->getDigits() );
    }

  firstentry = false;
```

```
}


void DAESolver :: compTSSol( double t, double tend, int & ind, int p )
{

  ptrAD->compJacobian( 0 );

  if( ind == Jacobian )
    {
      ptrAD->printJacobian( 0 );
      cout<<endl;
      ptrParam->setInd( normalexit );
      break;
    }

  ptrAD->compTCs( p );

  tol = ptrStepSize->compTol( ptrAD, ptrParam );

  h = ptrStepSize->compStepSize( ptrAD, ptrErrorEst, tol, p );
  h = ptrStepSize->compFinalStep( t, tend, h );
  if( !ptrStepSize->checkStepsize( h, ptrParam->getHmin() ) )
    {
      ptrParam->setInd( smallstepsize );
      break;
    }

  ptrAD->compSolution( p, h );

}

void DAESolver :: compConsSol( double t, int p )
{

  ptrAD->setIntegrationTime( t + h );

  double k = 0;
```

```
  while( !ptrProjection->projectConsInitPoint() && k < 11 )
    {

      ptrStats->statSteps(0);
      h = 0.5*h;

      if ( !ptrStepSize->checkStepsize( h, ptrParam->getHmin() ) )
        {
          ptrParam->setInd( smallstepsize );
          break;
        }

      if( k = 10 )
        {
          ptrParam->setInd( Jsingular );
          break;
        }
      ptrAD->compSolution( p, h );
      ptrAD->setIntegrationTime( t + h );

    }

    ptrStats->statSteps(1);

}
```

D.2 Computing signature matrix and offsets

```
void Offsets :: compOffsets( LAPSolver &solver )
{

  solver.compHVT( n, SignMatrix, rowsol, colsol );

  for( int i = 0; i < n; i++ )
    {
      c[i] = 0;
      d[i] = 0;
    }
```

```
  vector<int> d_old(n);

  int max;

  bool computed = false;
  while(!computed)
    {
      d_old = d;
      for ( int j = 0; j < n; j++ )
        {
          max = SignMatrix[0][j] + c[0];

          for ( int i = 1; i < n; i++ )
            {
              if ( SignMatrix[i][j] + c[i] > max )
              max = SignMatrix[i][j] + c[i];
            }

          d[j] = max;
        }

      computed = true;
      int k = 0;
      while ( k < n && computed )
        if ( d[k] != d_old[k] )
          computed = false;
        else
          k++;

      if ( !computed )
        for ( int j = 0; j < n; j++ )
          {
            int i = colsol[j];
            c[i] = d[j] - SignMatrix[i][j];
          }
    }
}
```

D.3 Computing Jacobians

D.3.1 Computing $\partial \mathrm{f}_{I_k} / \partial \mathrm{x}_{J_k}$

```
void FADBADTS :: compJacobian( int stage )
{
  for ( int j = 0; j < n; j++ )
    {
      int Lj = ptrOffsets->getDerivNoX( j, stage );
      if ( Lj >= 0 )
        {
          for ( int q = 0; q <= Lj; q++ )
            for ( int k = 0; k < n; k++ )
              Fin[j][q].d(k) = 0.0;
          TFin[j][Lj].d(j) = 1.0;
        }
    }

  for ( int i = 0; i < n; i++ )
    {
      int Li = ptrOffsets->getDerivNoF( i, stage );
      if ( Li >= 0 )
        TFout[i].eval( Li );
    }
}
```

D.3.2 Printing Jacobian

```
void FADBADTS :: printJacobian( int stage, ostream & s )
{
  s << endl << endl  << "SYSTEM JACOBIAN AT STAGE "
    << stage << ":" << endl;

  int p = s.precision();
  s.precision(4);

  int Li, Lj;
  for ( int i = 0; i < n; i++ )
    {
      Li = ptrOffsets->getDerivNoF( i, stage );
```

```
      if ( Li >= 0 )
        {
          for ( int j = 0; j < n; j++ )
            {
              Lj = ptrOffsets->getDerivNoX( j, stage );
              if( Lj>= 0 )
                {
                  s << std::setw(9) << TFout[i][Li].d(j)*factor[Li]/factor[Lj];
                }
              }
            s<<endl;
          }
    }
  s.precision(p);
}
```

D.4 TCs computation

D.4.1 Nonlinear case

```
void IpoptFunc :: get_F( long n, double *x, double *f )
{
  assert( x && f );
  double pp;
  f[0] = 0;
  for ( int i = 0; i < n; i ++ )
    {
      pp = x[i] - x0x0[i];   // x0x0 : the initial guess in AD class
      pp *= pp;
      f[0] += pp;
    }
  f[0] *= 0.5;
}

void IpoptFunc :: get_G( long n, double *x, double *g )
{
  assert( x && g );
  for ( int i = 0; i < n; i ++ )
    {
```

```
      g[i] = x[i] - x0x0[i];
    }
}

void IpoptFunc :: get_C( long n, double *x, long m, double *cc )
{
  assert( x && cc );

  get_C_AD( x, m, cc );
}


void IpoptFunc :: get_A( long task, long n, double *x, long *nz, double *A,
                long *Arow, long *Acol )
{
  assert( x && nz && A && Arow && Acol );
  assert(ptrAD);

  if ( task == 0 )
    {
      *nz = ptrAD -> getNoNonzeros( STAGE );
    }
  else
    {
      get_A_AD( x, nz, A, Arow, Acol );
    }
}

bool Projection :: projectSolution( int stage, double *xx )
{

  n = pOffsets->noVariables(stage);
  m = pOffsets->noEquations(stage);

  assert(parameters);

  double dtol = parameters->getDTol();

  pIF = new IpoptFunc( stage, n, m, xx, ad );
  assert( pIF && optpackage );
```

```
  bool temp = optpackage->compOptSolution( xx, n, m, &dtol, pIF );
  delete pIF;

  for( int i = 0; i < n; i++ )
    x[i] = xx[i];

  return temp;

}
```

D.4.2 Computing TCs

```
void FADBADTS :: compTCs( int p )
{

  for ( int i = 0; i < n; i++ )
    Tout[i].reset();

  for ( int j = 0; j < n; j++ )
    {
      for ( int i = 0; i < ptrOffsets->getDVector(j)+1; i++ )
        {
          Tin[j][i] = TFin[j][i].x();
        }
      for ( int i = ptrOffsets->getDVector(j)+1;
                i <= ptrOffsets->getDVector(j)+p; i++ )
        {
          Tin[j][i] = 0;
        }
    }

  getScaledJacobian( fjac );

  LU( n, fjac, ipiv );

  for ( int i = 0; i < n; i++ )
    {
      U[i] = 1;
      V[i] = 1;
```

```
    }

  for ( int k = 1; k <= p; k++ )
    {
      compTerm( k );
    }
}
```

D.4.3 Computing term

```
void FADBADTS :: compTerm( int stage )
{
  for ( int i = 0; i < n; i++ )
    {
      U[i] /= ptrOffsets->getDerivNoF( i, stage );
      V[i] /= ptrOffsets->getDerivNoX( i, stage );
    }

  int Li;
  for ( int i = 0; i < n; i++ )
    {
      Li = ptrOffsets->getDerivNoF( i, stage );
      Tout[i].eval(Li);
      Fvec[i] = Tout[i][Li];
    }

  for ( int i = 0; i < n; i++ )
    Fvec[i] = Fvec[i]/U[i]*U[indexmaxoffset];

  LSolve( n, fjac, ipiv, Fvec );

  for ( int j = 0; j < n; j++ )
    Tout[j].reset();

  for( int j = 0; j < n; j ++ )
    {
      Tin[j][ptrOffsets->getDerivNoX(j,stage)] -=
                  Fvec[j]*V[j]/U[indexmaxoffset];
    }
```

```
}
```

D.5 Error estimation

```
double ErrorEst :: estError( AD *ptrAD, int p, double h )
{
  assert( ptrAD );
  int l;
  double temp;
  double temppow = pow(h, (double) p);
  for( int j = 0; j < n; j ++ )
    {

      l = p + ptrOffsets->getDVector(j);
      temp = temppow * factor[l]/factor[p];

      e[j] = temp * ptrAD->getXjk( j, l );
    }

  double temperr = NORM :: norm1( e, n );
  return temperr;
}
```

D.6 Stepsize selection

D.6.1 Tolerance computation

```
double StepSize :: compTol( InitPoint *ptrInitPoint,
                            Parameters *ptrParam )
{
  double temp = ptrParam->getRTol()*ptrInitPoint->getNorm1X();
  temp += ptrParam->getATol();
  return temp;
}
```

D.6.2 Stepsize selection

```
double StepSize :: compStepSize( AD *ptrAD, ErrorEst *ptrErr,
                                 double tol, int p )
{
  double est = ptrErr->estError( ptrAD, p, 1 );

  assert( est != 0 );
  double h = pow( tol/est, 1.0/p );

  assert( h != 0 );
  return h;
}
```

D.6.3 Final stepsize selection

```
double StepSize :: compFinalStep( double t, double tend, double h )
{
  if( 2*fabs(h) < fabs(tend - t) )
    {
      if( tend > t )
        return h;
      else
        return -h;
    }
  else if( fabs(h) < fabs(tend - t) && 2*fabs(h) > fabs(tend - t) )
    {
      h = ( tend - t ) / 2;
      return h;
    }
  else
    {
      h = (tend - t);
      return h;
    }
}

bool StepSize :: compHmin( double h )
{
  if ( fabs(h) < ptrParam->getHmin() )
```

```
    {
      cerr<<"Stepsize is too small"<<endl;
      return false;
    }
  else
    return true;
}
```