VNODE-LP

A Validated Solver for Initial Value Problems in Ordinary Differential Equations

Nedialko S. Nedialkov

Department of Computing and Software
McMaster University
Hamilton, Ontario, Canada

Technical Report CAS-06-06-NN

Contents

List of Figures

Preface

We present VNODE-LP, a C++ solver for computing bounds on the solution of an initial-value problem (IVP) for an ordinary differential equation (ODE). In contrast to traditional ODE solvers, which compute approximate solutions, this solver proves that a unique solution to a problem exists and then computes rigorous bounds that are guaranteed to contain it. Such bounds can be used to help prove a theoretical result, check if a solution satisfies a condition in a safety-critical calculation, or simply to verify the results produced by a traditional ODE solver.

This package is a successor of the VNODE [25], Validated Numerical ODE, package of N. Nedialkov. A distinctive feature of the present solver is that it is developed entirely using Literate Programming (LP) [17]. As a result, the correctness of VNODE-LP's implementation can be examined much easier than the correctness of VNODE—the theory, documentation, and source code of VNODE-LP are interwoven in this manuscript, which can be verified for correctness by a human expert, like in a peer-review process.

Literate programming. With LP, a program (or function) is normally subdivided into pieces of code or *chunks*, and each of them may be subdivided into smaller chunks. How they are divided and put together should be clear from the exposition.

The present document is produced by `cweave` [18] on LaTeX-like *cweb* files, which contain both LaTeX text and C++ code. The C++ code for VNODE-LP and all the examples are generated by running `ctangle` [18] on those files; see Figure 1.

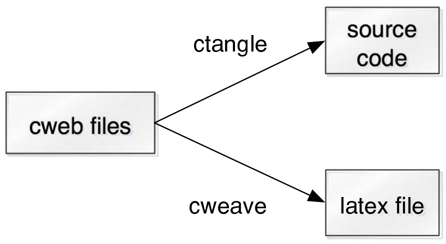


Figure 1. *Producing* C++ *and LaTeX files from cweb files*

Structure. Part I describes the problem VNODE-LP solves, shows how it can be installed, and illustrates on several examples how VNODE-LP can be used. Parts II–V contain the implementation of this package.

If a reader is interested only in using VNODE-LP, then studying Part I should provide sufficient knowledge for using this package.

This document is *open*: errors found by a reader will be fixed and suggestions on improving it will be incorporated. Such suggestions can be on both exposition and code.

Acknowledgments. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada.

George Corliss has made many valuable comments on this manuscript. Discusssions with George Corlis, Baker Kearfott, John Pryce, and Spencer Smith have resulted in various improvements of the presentation.

N. Nedialkov
July 27, 2006

Part I

Introduction, Installation, Use

Chapter 1

Introduction

1.1 The problem VNODE-LP solves

We consider the IVP

$$y'(t) = f(t, y), \quad y(t_0) = y_0, \qquad y \in \mathbb{R}^n,\ t \in \mathbb{R}. \tag{1.1}$$

We denote the set of closed (finite) intervals on $\mathbb{R}$ by

$$\mathbb{IR} = \left\{ \boldsymbol{a} = [\underline{\boldsymbol{a}}, \overline{\boldsymbol{a}}] \mid \underline{\boldsymbol{a}} \le x \le \overline{\boldsymbol{a}},\ \underline{\boldsymbol{a}}, \overline{\boldsymbol{a}} \in \mathbb{R} \right\}.$$

An interval vector is a vector with interval components. We denote the set of n-dimensional interval vectors by $\mathbb{IR}^n$.

Given a point $t_{end} \neq t_0$ $(t_{end} \in \mathbb{R})$ and $\boldsymbol{y}_0 \in \mathbb{IR}^n$, the goal of VNODE-LP *is to compute $\boldsymbol{y}_{end} \in \mathbb{IR}^n$ at t_{end} that contains the solution to (1.1) at t_{end} for all $y_0 \in \boldsymbol{y}_0$. If* VNODE-LP *cannot reach t_{end}, bounds on the solution at some t^* between t_0 and t_{end} are returned.*

This package is applicable to ODE problems for which derivatives of the solution $y(t)$ exist to some order; that is, $y(t)$ is sufficiently smooth. As a consequence, the code list of f should not contain functions such as branches, abs, or min.

In practice, t_0 or t_{end}, or both, may not be representable as floating-point numbers; for example the decimal 0.1 has an infinite binary representation. In this case, the user can set a machine-representable interval $\boldsymbol{t}_0$ [resp. $\boldsymbol{t}_{\text{end}}$] containing t_0 [resp. t_{end}].

1.2 On Literate Programming

The VNODE-LP package is a successor of VNODE [23, 25]. Both are written in C++. A major difference is that VNODE-LP is produced entirely, including this manuscript, using Literate Programming (LP) [17] and CWEB [18]. Why LP?

In general, interval methods produce results that can have the power of a mathematical proof. For example, when computing an enclosure of the solution of an IVP ODE, an interval method first proves that there exists a unique solution to the problem and then produces bounds that contain it. When solving a nonlinear equation, an interval method can prove that a region does not contain a solution or compute bounds that contain a unique solution to the problem.

However, if an interval method is not implemented correctly, it may not produce rigorous results. Furthermore, we cannot claim mathematical rigor if we miss to include even a single roundoff error in a computation. Therefore, it is of paramount importance to ensure that an interval algorithm is encoded correctly in a programming language.

In the author's opinion, interval software should be written such that it can be readily verified in a human peer-review process, like a mathematical proof is checked for correctness. The main goal of this work is to implement and document an interval solver for IVPs for ODEs such that its correctness can be verified by a reviewer.

To accomplish our goal, we have chosen the LP approach. The author has found LP particularly suitable for ensuring that an implementation of a numerical algorithm is a correct translation of its underlying theory into a programming language. Some of the benefits of employing LP follow.

- We can combine theory, source code, and documentation in a single document; we shall refer to it as an LP document.
- With LP, we can produce nearly "one-to-one" translation of the mathematical theory of a method into a computer program. In particular, we can split the theory into small pieces, translate each of them, and keep mathematical expressions and the corresponding code close together in a unified document. This facilitates verifying the correctness of smaller pieces and of a program as a whole.
- Since theory and implementation are in a single document, it is easier to keep them consistent, compared to having separate theory, source code, and documentation.

The user guide, theory, and source code of VNODE-LP are presented in the remainder of this document. The source code of VNODE-LP is extracted from source, *cweb* files using CWEB's [18] `ctangle`. This manuscript is produced by running `cweave` on these files and then calling LaTeX.

If the correctness of this manuscript is confirmed by reviewers in a peer-review-like process, we may trust the correctness of the implementation of VNODE-LP, and accept the bounds it computes as rigorous. When claiming rigor, however, we presume that the operating system, compiler, and the packages VNODE-LP uses do not contain errors.

1.3 Applications

Applications of validated integration include, for example, the solution of Smale's 14th problem [30] and rigorous computation of asteroid orbits [7]. The (previous)

VNODE package had been employed in applications such as rigorous multibody simulations [5], reliable surface intersection [22, 28], computing bounds on eigenvalues [8], parameter and state estimation [15], rigorous shadowing [11, 12], and theoretical computer science [3].

1.4 Limitations

Generally, VNODE-LP is suitable for computing bounds on the solution of an IVP ODE with point initial conditions, or interval initial conditions with a sufficiently small width, over not very long time intervals. If the initial condition set is not small enough and/or long time integration is desired, the reader is referred to the Taylor models approach of Berz and Makino, and their COSY package. Alternatively, one can subdivide the initial interval vector (box) $\boldsymbol{y}_0$ into smaller boxes, perform integrations with them as initial conditions, and build an enclosure of the solution at the desired t_{end}.

1.5 Prerequisites

A user of VNODE-LP does not need to know how the underlying methods work. It is sufficient to know that, if $\boldsymbol{a}$ and $\boldsymbol{b} \in \mathbb{IR}$ and $\bullet \in \{+, -, \times, /\}$, then VNODE-LP builds on the interval-arithmetic (IA) operations defined as

$$\boldsymbol{a} \bullet \boldsymbol{b} = \bigl\{x \bullet y \mid x \in \boldsymbol{a},\ y \in \boldsymbol{b}\bigr\}, \tag{1.2}$$

where division is undefined if $0 \in \boldsymbol{b}$. This definition can be implemented, for example, as

$$\begin{aligned}
\boldsymbol{a} + \boldsymbol{b} &= [\underline{\boldsymbol{a}} + \underline{\boldsymbol{b}}, \overline{\boldsymbol{a}} + \overline{\boldsymbol{b}}], \\
\boldsymbol{a} - \boldsymbol{b} &= [\underline{\boldsymbol{a}} - \overline{\boldsymbol{b}}, \overline{\boldsymbol{a}} - \underline{\boldsymbol{b}}], \\
\boldsymbol{a} \times \boldsymbol{b} &= [\min\{\underline{\boldsymbol{a}}\underline{\boldsymbol{b}}, \underline{\boldsymbol{a}}\overline{\boldsymbol{b}}, \overline{\boldsymbol{a}}\underline{\boldsymbol{b}}, \overline{\boldsymbol{a}}\overline{\boldsymbol{b}}\}, \max\{\underline{\boldsymbol{a}}\underline{\boldsymbol{b}}, \underline{\boldsymbol{a}}\overline{\boldsymbol{b}}, \overline{\boldsymbol{a}}\underline{\boldsymbol{b}}, \overline{\boldsymbol{a}}\overline{\boldsymbol{b}}\}], \quad \text{and} \\
\boldsymbol{a}/\boldsymbol{b} &= [\underline{\boldsymbol{a}}, \overline{\boldsymbol{a}}] \times [1/\overline{\boldsymbol{b}}, 1/\underline{\boldsymbol{b}}], \quad 0 \notin \boldsymbol{b}.
\end{aligned}$$

On a computer, $\boldsymbol{a}$ and $\boldsymbol{b}$ are representable machine intervals, and the computed result of an IA operation must contain (1.2), provided no exceptions occur. For example, if intervals are represented by their endpoints, when computing $\boldsymbol{a} + \boldsymbol{b}$, the true $\underline{\boldsymbol{a}} + \underline{\boldsymbol{b}}$ is rounded towards $-\infty$, and the true $\overline{\boldsymbol{a}} + \overline{\boldsymbol{b}}$ is rounded towards $+\infty$.

From a language perspective, we have tried to avoid using advanced C++ techniques; however, basic knowledge of C++ is required.

The installation of VNODE-LP is explained in Chapter 2. Chapter 3 presents various examples of how VNODE-LP can be used. Chapter 4 lists and describes the functions available to a user of VNODE-LP. Chapter 5 contains descriptions of test cases. Various listings are given in Chapter 6.

Chapter 2

Installation

In this Chapter, we list the utilities and packages necessary for installing VNODE-LP, list successful installations, and then describe the installation process.

2.1 Prerequisites

The following utilities are needed:

1. `gunzip` (GNU unzip)
2. `tar` (tape archiver)
3. `ar` (for creating a library archive)
4. C++ compiler
5. GNU `make`
6. `libg2c` run-time library, if the GNU C++ compiler is used

Normally, 1–5 are present on a Unix-based system, while `libg2c` may need to be installed.

The following packages are used by VNODE-LP and must be installed before VNODE-LP is installed:

interval arithmetic: FILIB++ [19] *or* PROFIL/BIAS [16]

linear algebra: LAPACK [2] and BLAS [1]

2.2 Successful installations

To date VNODE-LP has been successfully compiled and installed as follows.

IA package	Operating system	Architecture	Compiler
FILIB++	Linux	x86	gcc
	Solaris	Sparc	gcc
PROFIL	Linux	x86	gcc
	Solaris	Sparc	gcc
	Mac OSX	PowerPC	gcc
	Windows with Cygwin	x86	gcc

Remark. At the time of writing this manuscript, the author has not been able to install FILIB++ correctly on Mac OS X. However, VNODE-LP compiles on it.

2.3 Installation process

The installation process consists of the following steps:

1. extracting the source code
2. preparing a configuration file
3. building the VNODE-LP library, examples, and tests
4. installing the library files

2.3.1 Extracting the source code

VNODE-LP can be downloaded from `www.cas.mcmaster.ca~/nedialk/vnodelp`. The corresponding file is `vnodelp.tar.gz`. To extract the source files, type

```
tar -zxvf vnodelp.tar.gz
```

This will create the directory `vnodelp` and store the VNODE-LP files in it.

2.3.2 Preparing a configuration file

The user has to prepare a *configuration file*, which contains information such as compiler, options, libraries, and various directory paths. There are four such files used by the author: `MacOSXWithFilib`, `MacOSXWithProfil`, `LinuxWithFilib`, and `LinuxWithProfil`, located in `vnodelp/config`. One can modify any of these files or create his own configuration file, where the variables described in Figure 2.1 should be set appropriately. The files `MacOSXWithProfil` and `LinuxWithProfil` are given in Figures 2.2 and 2.3.

variable	stores
CXX	name of C++ compiler
CXXFLAGS	C++ compiler flags
GPP_LIBS	GNU C++ standard library `libstdc++` and the `libg2c` run-time library
LDFLAGS	linker flags
I_PACKAGE	FILIB_VNODE or PROFIL_VNODE
I_INCLUDE	name of the directory containing include files of the interval-arithmetic package
I_LIBDIR	name of the directory containing interval libraries
I_LIBS	names of interval libraries
MAX_ORDER	value for the maximum order VNODE-LP can use
L_LAPACK	name of the directory containing the LAPACK library
L_BLAS	name of the directory containing the BLAS library
LAPACK_LIB	name of the LAPACK library file
BLAS_LIB	name of the BLAS library file

Figure 2.1. *Variables of a* VNODE-LP *configuration file*

2.3.3 Building the VNODE-LP library and examples

The `makefile` in `vnodelp` (see Figure 2.4) contains two variables that need to be set appropriately:

`CONFIG_FILE` contains the name of the configuration file; and

`INSTALL_DIR` contains the directory, where VNODE-LP should be installed.

After these variables are set appropriately, type

```
make
```

The library `libvnode.a` will be created in subdirectory `vnodelp/lib`, and the examples will be created in `vnodelp/examples`. Then, several test programs in subdirectory `tests` will be compiled and executed. If VNODE-LP compiles successfully and the tests pass, the following message should appear.

```
*****************************************
***  VNODE-LP has compiled successfully
***  All tests have executed successfully
*****************************************
If you have set the install directory, type
make install
```

```
CXX      = g++
CXXFLAGS = -O2 -g -Wall -pedantic -Wno-deprecated
GPP_LIBS = -lstdc++ /sw/lib/libg2c.a
LDFLAGS += -bind_at_load -Wno

# interval package
I_PACKAGE = PROFIL_VNODE
I_INCLUDE =     $(HOME)/NUMLIB/Profil-2.1/src           \
                $(HOME)/NUMLIB/Profil-2.1/src/BIAS      \
                $(HOME)/NUMLIB/Profil-2.1/src/Base

I_LIBDIR  =     $(HOME)/NUMLIB/Profil-2.1/src/BIAS      \
                $(HOME)/NUMLIB/Profil-2.1/src/Base      \
                $(HOME)/NUMLIB/Profil-2.1/src/lr
I_LIBS    =     -lProfil -lBias -llr

MAX_ORDER = 50

# LAPACK and BLAS
L_LAPACK   = $(HOME)/NUMLIB/LAPACK
L_BLAS     = $(HOME)/NUMLIB/LAPACK
LAPACK_LIB = -llapack_MACOSX
BLAS_LIB   = -lblas_MACOSX

# --- DO NOT CHANGE BELOW ---
INCLUDES  = $(addprefix -I, $(I_INCLUDE))         \
        -I$(PWD)/FADBAD++
LIB_DIRS  = $(addprefix -L, $(I_LIBDIR)           \
        $(L_LAPACK) $(L_BLAS))
CXXFLAGS += -D${I_PACKAGE} \
        -DMAXORDER=$(MAX_ORDER) $(INCLUDES)
LDFLAGS += $(LIB_DIRS)
LIBS = $(I_LIBS) $(LAPACK_LIB) $(BLAS_LIB)        \
        $(GPP_LIBS)
```

Figure 2.2. *File* `config/MacOSXWithProfil`

2.3.4 Installing the library files

To install the library and the related include files, type

```
make install
```

This will create a subdirectory `vnodelp` of the directory stored in `INSTALL_DIR` and subdirectories of `vnodelp` as follows:

```
CXX = gcc
CXXFLAGS = -O2 -Wall -Wno-deprecated -DNDEBUG
GPP_LIBS = -lstdc++ -lg2c

# interval package
I_PACKAGE = PROFIL_VNODE
I_INCLUDE =                                          \
          $(HOME)/NUMLIB/Profil-2.0/include          \
          $(HOME)/NUMLIB/Profil-2.0/include/BIAS     \
          $(HOME)/NUMLIB/Profil-2.0/src/Base
I_LIBDIR  = $(HOME)/NUMLIB/Profil-2.0/lib
I_LIBS    = -lProfil -lBias -llr

MAX_ORDER = 50

# LAPACK and BLAS
L_LAPACK   =
L_BLAS     =
LAPACK_LIB = -llapack
BLAS_LIB   = -lblas

# --- DO NOT CHANGE BELOW ---
INCLUDES = $(addprefix -I, $(I_INCLUDE))             \
          -I$(PWD)/FADBAD++
LIB_DIRS = $(addprefix -L, $(I_LIBDIR)               \
          $(L_LAPACK) $(L_BLAS))
CXXFLAGS += -D${I_PACKAGE} \
          -DMAXORDER=$(MAX_ORDER) $(INCLUDES)
LDFLAGS += $(LIB_DIRS)
LIBS = $(I_LIBS) $(LAPACK_LIB) $(BLAS_LIB)           \
          $(GPP_LIBS)
```

Figure 2.3. *File* `config/LinuxWithProfil`

directory	contains
`lib`	`libvnode.a`
`include`	`libvnode.a`'s include files
`config`	configuration files
`doc`	documentation file `vnode.pdf`

Subsection 3.1.4 contains details about how to build user's programs.

```
# set CONFIG_FILE and INSTALL_DIR

CONFIG_FILE ?= MacOSXWithProfil
INSTALL_DIR ?= $(HOME)

#--- DO NOT CHANGE BELOW ---
```

Figure 2.4. *The first six lines of* `makefile` *in* `vnodelp`

Chapter 3

Examples

We start with an example showing a basic integration with VNODE-LP, Section 3.1. In Section 3.2 we examine how VNODE-LP does on a simple scalar ODE. Section 3.3 contains an example of integrating a time-dependent system of ODEs and illustrates how this package can be used to check the correctness of the numerical results produced by a standard ODE method.

Section 3.4 outlines how to integrate with interval initial condition and output intermediate results. We describe how VNODE-LP outputs results at given time points in Section 3.5, and how parameters can be passed to an ODE problem in Section 3.6.

In Section 3.7, we show how an integration can be controlled, and in Section 3.8, we perform a simple study of the computational work versus the order of the method implemented in VNODE-LP. Section 3.9 contains a study of the computational work versus the size of the problem. Section 3.10 illustrates the stepsize behavior when integrating an orbit problem. Finally, Section 3.11 shows the stepsize behavior of VNODE-LP as the stiffness in an ODE increases.

3.1 Basic usage

In VNODE-LP, the user has to specify the right side of an ODE problem and provide a main program.

3.1.1 Problem definition

An ODE must be specified by a template function for evaluating $y' = f(t, y)$ of the form

18 ⟨ template ODE function 18 ⟩ ≡

```
template⟨typename var_type⟩
void ODEName(int n, var_type *yp, const var_type *y, var_type t,
        void *param)
{
```

```
            /* body */
}
```

Here n is the size of the problem, t is the time variable, y is a pointer to input variables, *yp* is a pointer to output variables, and *param* is a pointer to additional parameters that can be passed to this function.

As an example, consider the Lorenz system

$$\begin{aligned} y_1' &= \sigma(y_2 - y_1) \\ y_2' &= y_1(\rho - y_3) - y_2 \\ y_3' &= y_1 y_2 - \beta y_3, \end{aligned}$$

where σ, ρ, and β are constants. This system is encoded in the *Lorenz* function below. The constants have values $\sigma = 10$, $\beta = 8/3$, and $\rho = 28$. We initialize *beta* with the interval containing 8/3: **interval**(8.0) creates an interval with endpoints 8.0, and **interval**(8.0)/3.0 is the interval containing 8/3.[1] The last parameter, *param*, is not used here, but its role is discussed in Section 3.6.

```
19 ⟨Lorenz 19⟩ ≡
  template⟨typename var_type⟩
  void Lorenz(int n, var_type *yp, const var_type *y, var_type t,
          void *param)
  {
    interval sigma(10.0), rho(28.0);
    interval beta = interval(8.0)/3.0;
    yp[0] = sigma * (y[1] - y[0]);
    yp[1] = y[0] * (rho - y[2]) - y[1];
    yp[2] = y[0] * y[1] - beta * y[2];
  }
```

This code is used in chunks 20, 45, 48, 61, and 70.

3.1.2 Main program

We give a simple main program and explain its parts.

```
20 ⟨simple main program 20⟩ ≡
  ⟨Lorenz 19⟩
  int main()
  {
    ⟨set initial condition and endpoint 21⟩
    ⟨create AD object 22⟩
    ⟨create a solver 23⟩
    ⟨integrate (basic) 24⟩
    ⟨check if success 25⟩
```

[1]The result of this division is the interval with endpoints 8/3 rounded toward $-\infty$ and 8/3 rounded towards $+\infty$.

```
  ⟨output results 26⟩
  return 0;
}
```
This code is used in chunk 27.

The initial condition and endpoint are represented as intervals in VNODE-LP. In this example, they are all point values stored as intervals. The components of **iVector** (interval vector) are accessed like a C/C++ array is accessed.

21 ⟨set initial condition and endpoint 21⟩ ≡
```
const int n = 3;
interval t = 0.0, tend = 20.0;
iVector y(n);
y[0] = 15.0;
y[1] = 15.0;
y[2] = 36.0;
```
This code is used in chunks 20, 42, 48, 58, and 61.

Then we create an automatic differentiation (AD) object of type **FADBAD_AD**. It is instantiated with data types for computing Taylor coefficients (TCs) of the ODE solution and TCs of the solution to its variational equation, respectively [23]. To compute these coefficients, we employ the FADBAD++ package [29]. The first parameter in the constructor of **FADBAD_AD** is the size of the problem. The second and third parameters are the name of the template function, here *Lorenz*.

22 ⟨create AD object 22⟩ ≡
```
AD *ad = new FADBAD_AD(n, Lorenz, Lorenz);
```
This code is used in chunks 20, 45, 48, 61, and 68.

Now, we create a solver:

23 ⟨create a solver 23⟩ ≡
```
VNODE *Solver = new VNODE(ad);
```
This code is used in chunks 20, 35, 40, 45, 48, 58, 61, and 68.

The integration is carried out by the *integrate* function. It attempts to compute bounds on the solution at *tend*. When *integrate* returns, either $t = tend$ or $t \neq tend$. In both cases, y contains the ODE solution at t.

24 ⟨integrate (basic) 24⟩ ≡
```
Solver→integrate(t, y, tend);
```
This code is used in chunks 20, 35, 40, and 68.

We check if an integration is successful by calling *Solver→successful*( ):

25 ⟨check if success 25⟩ ≡
```
if (¬Solver→successful())
      cout ≪ "VNODE-LP␣could␣not␣reach␣t␣=␣" ≪ tend ≪ endl;
```
This code is used in chunks 20, 35, 40, and 68.

Finally, we output the computed enclosure of the solution at t by

26 ⟨output results 26⟩ ≡
```
cout ≪ "Solution␣enclosure␣at␣t␣=␣" ≪ t ≪ endl;
printVector(y);
```
This code is used in chunks 20, 35, 40, and 47.

3.1.3 Files

The code of VNODE-LP is in the namespace **vnodelp**. The interface to VNODE-LP is stored in the file `vnode.h`, which must be included in any file using VNODE-LP. We store our program in the file `basic.cc`.

27 ⟨`basic.cc` 27⟩ ≡
```
#include <ostream>
#include "vnode.h"
  using namespace std;
  using namespace vnodelp;
  ⟨simple main program 20⟩
```

3.1.4 Building an executable

We describe how `basic.cc` is compiled and linked with the VNODE-LP library. The subdirectory `user_program` of `vnodelp` contains the files `basic.cc` and `makefile`, which is given in Figure 3.1. We consider this file here.

```
INSTALL_DIR = $(HOME)
CONFIG_FILE = $(INSTALL_DIR)/vnodelp/config/MacOSXWithProfil

include $(CONFIG_FILE)

CXXFLAGS += -I$(INSTALL_DIR)/vnodelp/include\
            -I$(INSTALL_DIR)/vnodelp/FADBAD++
LDFLAGS   += -L$(INSTALL_DIR)/vnodelp/lib

basic:   basic.o
         $(CXX) $(LDFLAGS) -o $@ basic.o -lvnode $(LIBS)

clean:
         @-$(RM) *.o core.* basic
```

Figure 3.1. `makefile` *in* `vnodelp/user_program`

The directory where `vnodelp` resides is set in `INSTALL_DIR`, and the configuration file is set in `CONFIG_FILE`. The variables `CXXFLAGS` and `LDFLAGS` need not be changed. Finally, the rule for building `basic` is given. To create the executable file `basic`, type `make` in `vnodelp/user_program`.

3.1.5 Output

The output of `basic` is

```
Solution enclosure at t = [20,20]
14.30[38159449608937,44694855332662]
9.5[785941360078012,801274302834650]
39.038[2373597549516,4111043348412]
```

These results are interpreted as

$$y(20) \in \begin{pmatrix} [14.3038159449608937, 14.3044694855332662] \\ 9.5785941360078012, \ \ 9.5801274302834650] \\ [39.0382373597549516, 39.0384111043348412] \end{pmatrix}. \tag{3.1}$$

For comparison, if we integrate this problem with MAPLE using `dsolve` with options `method=taylorseries` and `abserr=Float(1,-18)`, and with `Digits := 20`, we obtain

$$y(20) \approx \begin{pmatrix} 14.304146251277895001 \\ 9.5793690774871976695 \\ 39.038325167739731729 \end{pmatrix},$$

which is contained in the bounds (3.1).

Remark.

1. All numerical results in this manuscript are produced with PROFIL/BIAS on 1.25 GHz PowerPC G4 with MacOS X, 512 MB RAM, and 512KB L2 cache.
2. The output format is due to the PROFIL/BIAS [16] interval-arithmetic package.
3. On different architectures, or with different IA packages on the same architecture, the computed results are likely to differ, but they must contain the true results.

3.1.6 Standard coding

All source-code files in the VNODE-LP distribution, except the test programs in subdirectory `vnodelp/tests`, are generated with `ctangle` from `CWEB`. Since a user may not use LP, we also give the "standard" C++ code of `basic.cc` in Figure 3.2.

For the remaining examples, we do not explain how they are compiled. For details, see the `makefile` in Figure 6.1. This file is in the directory `vnodelp/examples`. Also, we do not provide "standard" code of the corresponding C++ files; if needed, it can be extracted from the `ctangle` generated files `*.cc` in `vnodelp/examples`.

```
#include <ostream>
#include "vnode.h"

using namespace std;
using namespace vnodelp;

template<typename var_type>
void Lorenz(int n, var_type*yp, const var_type*y, var_type t,
            void*param)
{
  interval sigma(10.0), rho(28.0);
  interval beta = interval(8.0)/3.0;

  yp[0] = sigma*(y[1]-y[0]);
  yp[1] = y[0]*(rho-y[2])-y[1];
  yp[2] = y[0]*y[1]-beta*y[2];
}

int main()
{
  const int n = 3;
  interval t = 0.0,tend = 20.0;
  iVector y(n);
  y[0] = 15.0;
  y[1] = 15.0;
  y[2] = 36.0;

  AD *ad= new FADBAD_AD(n,Lorenz,Lorenz);
  VNODE *Solver= new VNODE(ad);

  Solver->integrate(t,y,tend);
  if(!Solver->successful())
    cout<<"VNODE-LP could not reach t = "<<tend<<endl;

  cout<<"Solution enclosure at t = "<<t<<endl;
  printVector(y);

  return 0;
}
```

Figure 3.2. *The "standard"* C++ *code of* `basic.cc`

3.2 One-dimensional ODE

VNODE-LP is designed to be a general-purpose ODE interval solver. Nevertheless, it should handle scalar ODEs. This example illustrates how VNODE-LP deals with the simple problem

$$y' = -y, \quad y(0) = 1, \qquad t_{\text{end}} = 20.$$

We write

32 ⟨scalar ODE example 32⟩ ≡
```
template⟨typename var_type⟩
void ScalarExample(int n, var_type *yp, const var_type *y, var_type t,
        void *param)
{
  yp[0] = −y[0];
}
```
This code is used in chunk 35.

The initial condition and endpoint are set in

33 ⟨set scalar ODE initial condition and endpoint 33⟩ ≡
```
const int n = 1;
interval t = 0.0, tend = 20.0;
iVector y(n);    /* number of state variables is 1 */
y[0] = 1.0;
```
This code is used in chunk 35.

To create the necessary AD object, we call

34 ⟨create scalar AD object 34⟩ ≡
```
AD *ad = new FADBAD_AD(n, ScalarExample, ScalarExample);
```
This code is used in chunk 35.

The main program is

35 ⟨scalar.cc 35⟩ ≡
```
#include <ostream>
#include "vnode.h"
  using namespace std;
  using namespace vnodelp;
  ⟨scalar ODE example 32⟩
  int main()
  {
    ⟨set scalar ODE initial condition and endpoint 33⟩
    ⟨create scalar AD object 34⟩
    ⟨create a solver 23⟩
  ⟨ integrate (basic) 24 ⟩
  ⟨ check if success 25 ⟩
  ⟨ output results 26 ⟩
  return 0;
}
```

The output of this program is `0.000000002061153[6,7]`, which must enclose e^{-20}. Using MAPLE with a 30-digit computation, we obtain for e^{-20}

$$\underline{0.2061153}622438557827965940380 16 \cdot 10^{-8}$$

(the digits that coincide are underlined), which is contained in the interval computed by VNODE-LP.

3.3 Time-dependent ODE

We show an example of integrating a time-dependent ODE. We choose the E1 problem from the DETEST test set [14]. This problem is

$$\begin{aligned}
y_1' &= y_2 \\
y_2' &= -\left(\frac{y_2}{t+1} + \left(1 - \frac{0.25}{(t+1)^2}\right) y_1\right) \\
y_1(0) &= 0.6713967071418030, \\
y_2(0) &= 0.09540051444747446, \\
t_{\text{end}} &= 20.
\end{aligned}$$

37 ⟨ DETEST E1 37 ⟩ ≡
```
template⟨typename var_type⟩
void DETEST_E1(int n, var_type *yp, const var_type *y, var_type t,
        void *param)
{
  var_type t1 = t + 1.0;
  yp[0] = y[1];
  yp[1] = -(y[1]/t1 + (1.0 - 0.25/(t1 * t1)) * y[0]);
}
```
This code is used in chunk 40.

We store the initial condition as intervals containing the corresponding decimal values. The function *string_to_interval* converts a decimal string to a machine interval that contains the decimal value stored in the string.

38 ⟨ set E1 initial condition and endpoint 38 ⟩ ≡
```
const int n = 2;
interval t = 0.0, tend = 20.0;
iVector y(n);
```

```
  y[0] = string_to_interval("0.6713967071418030");
  y[1] = string_to_interval("0.09540051444747446");
```
This code is used in chunk 40.

To create an AD object, we call

```
39 ⟨ create E1 39 ⟩ ≡
  AD *ad = new FADBAD_AD(n, DETEST_E1, DETEST_E1);
```
This code is used in chunk 40.

The main program is

```
40 ⟨ E1.cc 40 ⟩ ≡
#include <ostream>
#include "vnode.h"
  using namespace std;
  using namespace vnodelp;
  ⟨ DETEST E1 37 ⟩
  int main( )
  {
    ⟨ set E1 initial condition and endpoint 38 ⟩
    ⟨ create E1 39 ⟩
    ⟨ create a solver 23 ⟩
    ⟨ integrate (basic) 24 ⟩
    ⟨ check if success 25 ⟩
    ⟨ output results 26 ⟩
    return 0;
  }
```

The output of this program is

```
Solution enclosure at t = [20,20]
0.14567236007282[02,87]
-0.0988350019557[410,507]
```

We have also computed a numerical solution using MATLAB's `ode45`. The corresponding programs are in Figure 6.3, and the output is

$$\begin{array}{r}\underline{\texttt{0.14567235}}\texttt{996177}\\ -\underline{\texttt{0.0988350018}}\texttt{2770}\end{array}$$

We have underlined the digits that coincide in the VNODE-LP and MATLAB output.

Since VNODE-LP includes all possible errors in its computation, including conversion errors in the input from decimal to binary, we have bounds on the true solution at $t = 20$. Using these bounds, we can check the accuracy of the numerical solution found by MATLAB's `ode45`.

Remark. At the time of writing this manuscript, FILIB++ does not provide facilities for rigorous conversion of a decimal string to an interval containing its value. Hence, *string_to_interval* works only when VNODE-LP is compiled with PROFIL/BIAS.

3.4 Interval initial conditions

Suppose we want to compute bounds on the solution of the Lorenz problem for all

$$y(0) \in \begin{pmatrix} 15 + [-10^{-4}, 10^{-4}] \\ 15 + [-10^{-4}, 10^{-4}] \\ 36 + [-10^{-4}, 10^{-4}] \end{pmatrix}.$$

We set an interval initial condition by

42 ⟨ set interval initial condition and endpoint 42 ⟩ ≡
  ⟨ set initial condition and endpoint 21 ⟩
  interval *eps* = **interval**$(-1, 1)/1 \cdot 10^4$;
  $y[0]$ += *eps*;
  $y[1]$ += *eps*;
  $y[2]$ += *eps*;
This code is used in chunk 45.

Before presenting the rest of the code, we introduce some notation. We denote the radius of an interval $\boldsymbol{a}$ by

$$\mathrm{r}(\boldsymbol{a}) = (\overline{\boldsymbol{a}} - \underline{\boldsymbol{a}})/2.$$

The midpoint of $\boldsymbol{a}$ is denoted by

$$\mathrm{m}(\boldsymbol{a}) = (\overline{\boldsymbol{a}} + \underline{\boldsymbol{a}})/2.$$

(In machine arithmetic, the true $\overline{\boldsymbol{a}} - \underline{\boldsymbol{a}}$ is rounded upward, and the true $(\overline{\boldsymbol{a}} + \underline{\boldsymbol{a}})/2$ is rounded to the nearest.) Radius and midpoint are defined componentwise for interval vectors.

Informally, the *global excess* at a point t^* is the overestimation in the computed bounds over the true solution set at t^* [23]. VNODE-LP computes an estimate, which is a nonnegative number, of the global excess in the computed bounds on each solution component, and also the max norm of these estimates.

When stepping in time from $\boldsymbol{t}_0$ to $\boldsymbol{t}_{\text{end}}$, VNODE-LP computes bounds on the solution at points that are machine numbers, except possibly at $\boldsymbol{t}_{\text{end}}$, which may be a machine interval with a nonzero radius. In this example, we output intermediate results during the integration. We tell the integrator to return after each step is completed by calling *setOneStep*(*on*). This is convenient when we want to access intermediate results, for example, for plotting solutions, stepsize, etc. In the code below, we record such results in a file, which is later used by `gnuplot` to generate the plots in Figure 3.3.

43 ⟨indicate single step 43⟩ ≡
```
Solver→setOneStep(on);
```
This code is used in chunks 44 and 61.

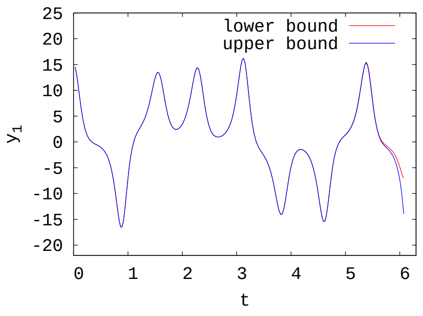
Now we integrate and record in the file `lorenzi.out` the midpoint and the radius of the computed bounds on $y_1(t)$ for each t selected by the solver. We also output an estimate on the *global excess*. The function *getGlobalExcess* returns the max norm of a vector with estimates on the global excess for each solution component. We exit the **while** loop below if this estimate exceeds 15 (This number is chosen so we can visualize the divergence of the computed bounds; see Figure 3.3(b).)

44 ⟨integrate with interval initial condition 44⟩ ≡
```
  ⟨indicate single step 43⟩
  ofstream outFile("lorenzi.out", ios::out);
  while (t ≠ tend) {
    Solver→integrate(t, y, tend);
    if (Solver→successful() ∧ Solver→getGlobalExcess() ≤ 15.0) {
      outFile ≪ midpoint(t) ≪ "\t"
            ≪ midpoint(y[0]) ≪ "\t"
            ≪ rad(y[0]) ≪ "\t"
            ≪ Solver→getGlobalExcess() ≪ endl;
    }
    else break;
  }
  outFile.close();
```
This code is used in chunk 45.

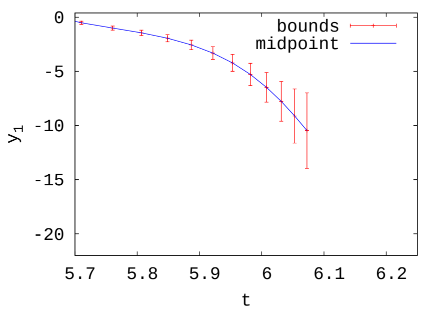
The main program is

45 ⟨integi.cc 45⟩ ≡
```
#include <fstream>
#include "vnode.h"
  using namespace std;
  using namespace vnodelp;
  ⟨Lorenz 19⟩
  int main()
  {
    ⟨set interval initial condition and endpoint 42⟩
    ⟨create AD object 22⟩
    ⟨create a solver 23⟩
    ⟨integrate with interval initial condition 44⟩
    return 0;
  }
```

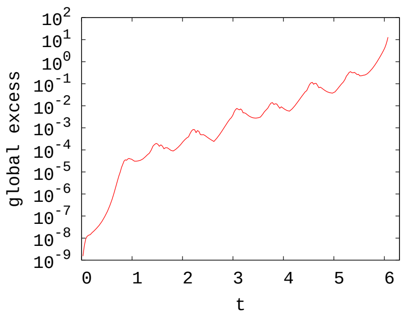
In Figure 3.3(a) and (b), we plot the lower and upper bounds on y_1. Their divergence is clearly seen in (b). In (c) we plot the logarithm of the estimate on



(a) Bounds on $y_1(t)$ versus t



(b) Bounds on $y_1(t)$ versus t; "midpoint" lines connect the midpoints of the computed intervals



(c) log 10(global excess) versus t

Figure 3.3. *Plots generated using* `integi.cc`

the global excess versus t. On this problem, the bounds become too wide as t goes beyond $\approx 6.0.7$.

The `gnuplot` file employed to generate this plot is given in Figure 6.2.

3.5 Producing intermediate results

We show how we can output enclosures on the solution at given points, for example, at $t = 0.1, 0.2, 0.3$, and $t = 10$. The decimal value 0.1 cannot be stored exactly as an IEEE floating-point number—we store 0.1 as an interval.

```
47 ⟨integrate with intermediate output 47⟩ ≡
  interval step = string_to_interval("0.1");

  tend = 0.0;
  for (int i = 1; i ≤ 3; i++) {
    tend += step;
    Solver→integrate(t, y, tend);
    ⟨ output results 26 ⟩
  }
  tend = 10;
  Solver→integrate(t, y, tend);
  ⟨ output results 26 ⟩
```

This code is used in chunk 48.

The main program is

48 ⟨ `intermediate.cc` 48 ⟩ ≡

```
#include <iostream>
#include "vnode.h"
  using namespace std;
  using namespace vnodelp;
  ⟨ Lorenz 19 ⟩
  int main( )
  {
    ⟨ set initial condition and endpoint 21 ⟩
    ⟨ create AD object 22 ⟩
    ⟨ create a solver 23 ⟩
    ⟨ integrate with intermediate output 47 ⟩
    return 0;
  }
```

The output is

```
Solution enclosure at t = 0.099999999999999[9,11]
9.5199890775031[033,833]
1.172296185059[1790,3086]
36.286934318697[3267,4618]

Solution enclosure at t = 0.199999999999999[9,11]
2.870955583196[2392,4120]
-1.446088378503[6309,8119]
27.476193552015[1645,3422]

Solution enclosure at t = 0.299999999999999[9,11]
0.339508665582[1531,3150]
-0.925287772052[5888,8462]
20.901865632568[5157,7041]

Solution enclosure at t = [10,10]
-5.909806[3819893544,7254843408]
-11.34140[28468979033,34573302108]
9.08017[76297270730,80129492258]
```

In this output, `0.099999999999999[9,11]` is interpreted as the interval with left point

$$0.099999999999999 + 0000000000000009 = 0.0999999999999999$$

and right point

$$0.099999999999999 + 0000000000000011 = 1.0000000000000001.$$

Remark. One has to be careful when interpreting the (decimal) output. For example consider the first output point for t, `t = 0.099999999999999[9,11]`. VNODE-LP computes bounds for all t in the **binary interval containing 0.1**, but prints the rounded out decimal interval, which in general contains the binary one (but it is not necessarily the same).

3.6 ODE control

3.6.1 Passing data to an ODE

Suppose that we want to pass the constants to the Lorenz system. We can encapsulate them in the structure

51 ⟨ constants Lorenz 51 ⟩ ≡
```
struct LorenzConsts {
  interval beta;
  double rho, sigma;
};
```
This code is used in chunk 58.

We can set σ, β, and ρ in the main programs as

52 ⟨ set ODE parameters 52 ⟩ ≡
```
LorenzConsts p;
p.sigma = 10.0;
p.beta = interval(8.0)/3.0;
p.rho = 28.0;
```
This code is used in chunk 58.

We can access parameters for the ODE through the **void** pointer *param*. The user has to ensure that such parameters are properly stored and later extracted through *param*.

53 ⟨ passing parameters to Lorenz 53 ⟩ ≡
```
template⟨typename var_type⟩
void Lorenz2(int n, var_type *yp, const var_type *y, var_type t,
        void *param)
{
  LorenzConsts *p = (LorenzConsts *) param;
  interval beta = p→beta;
  double sigma = p→sigma;
  double rho = p→rho;
```

```
    yp[0] = sigma * (y[1] − y[0]);
    yp[1] = y[0] * (rho − y[2]) − y[1];
    yp[2] = y[0] * y[1] − beta * y[2];
  }
```

This code is used in chunk 58.

To pass parameters to the ODE, we create an AD object *with fourth parameter the address of p*:

54 ⟨create problem object with parameters 54⟩ ≡

```
  AD *ad = new FADBAD_AD(n, Lorenz2, Lorenz2, &p);
```

This code is used in chunk 58.

3.6.2 Integration with parameter change

We illustrate how to integrate with changing β. First, we integrate from *t0* to *tend* and output $\mathrm{m}(\boldsymbol{y}_j)$ into a file; $\boldsymbol{y}_j$ is the computed enclosure at t_j.

55 ⟨simple integration 55⟩ ≡

```
  Solver→setOneStep(on);
  ofstream outFile1("odeparam1.out", ios::out);
  while (t ≠ tend) {
    Solver→integrate(t, y, tend);
    outFile1 ≪ midpoint(y[0]) ≪ "\t"
        ≪ midpoint(y[1]) ≪ "\t"
        ≪ midpoint(y[2]) ≪ endl;
  }
  outFile1.close();
```

This code is used in chunk 58.

Now, we

1. integrate from *t0* to *tend*/2,
2. change β, and
3. integrate to *tend*.

Before calling *integrate* again, we call *setFirstEntry*. This call ensures that certain internal data structures are initialized. If *setFirstEntry* is not called, *integrate* would use data corresponding to the last computed solution from the most recent call to *integrate*.

56 ⟨integrate from *t* to *tend*/2 56⟩ ≡

```
  t = 0.0;
  y[0] = 15;
  y[1] = 15;
  y[2] = 36;
  interval tend2 = tend/2.0;
  Solver→setFirstEntry( );
  while (t ≠ tend2)
          Solver→integrate(t, y, tend2);
```
This code is used in chunk 57.

When changing β, the integration continues with the last computed solution, which is computed with the previous value for β. After a parameter is changed, *ad→eval(&p) must be called.* This ensures that some internal data structures are updated, to reflect the change in the ODE.

```
57 ⟨ integrate with resetting constants 57 ⟩ ≡
  ⟨ integrate from t to tend/2 56 ⟩
  p.beta = 5.0;      /* change β */
  ad→eval(&p);       /* must be called to update internal data structures */
  ofstream outFile2("odeparam2.out", ios::out);
  while (t ≠ tend) {
    Solver→integrate(t, y, tend);
    outFile2 ≪ midpoint(y[0]) ≪ "\t"
         ≪ midpoint(y[1]) ≪ "\t"
         ≪ midpoint(y[2]) ≪ endl;
  }
  outFile2.close( );
```
This code is used in chunk 58.

The main program is

```
58 ⟨ odeparam.cc 58 ⟩ ≡
#include <fstream>
#include "vnode.h"
  using namespace std;
  using namespace vnodelp;
  ⟨ constants Lorenz 51 ⟩
  ⟨ passing parameters to Lorenz 53 ⟩
  int main( )
  {
    ⟨ set initial condition and endpoint 21 ⟩
    ⟨ set ODE parameters 52 ⟩
    ⟨ create problem object with parameters 54 ⟩
    ⟨ create a solver 23 ⟩
    ⟨ simple integration 55 ⟩
    ⟨ integrate with resetting constants 57 ⟩
    return 0;
  }
```

In Figure 3.4, we plot $\mathrm{m}(\boldsymbol{y}_j)$ in (y_1, y_2, y_3) coordinates corresponding to $\beta = 8/3$ and $\beta = 5$. The `gnuplot` file for producing this plot is in Figure 6.4.

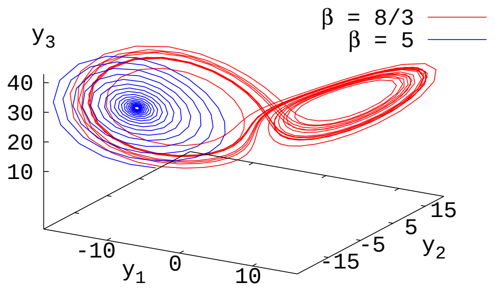


Figure 3.4. *Midpoints of the computed bounds with $\beta = 8/3$ from 0 to 20; and with $\beta = 8/3$ changed to 5 at $t = 10$*

.

3.7 Integration control

We start by introducing various facts related to the integration process of VNODE-LP. Then we show ways of controlling it.

The method implemented in VNODE-LP is a one-step method based on Taylor series and the Hermite-Obreschkoff scheme [23]. These are high-order methods, where typical values for the order, denoted here by p, can be in the range of 20 to 30; see Section 3.8.

VNODE-LP steps in time from t_0 to t_{end}, where the jth integration step, $j \geq 1$, is from t_{j-1} to t_j. The associated stepsize is $h_j = t_j - t_{j-1}$.[2] Each of these t_j is a representable machine number, except t_0 and t_{end}, which can be machine intervals $\boldsymbol{t}_0$ and $\boldsymbol{t}_{\mathrm{end}}$ containing the true t_0 and t_{end}, respectively. For simplicity of the exposition, we assume point values t_0 and t_{end}.

In addition to computing bounds $\boldsymbol{y}_j$ on the solution at each t_j, VNODE-LP also computes bounds for all $t \in \boldsymbol{T}_j = [t_{j-1}, t_j]$, or $\boldsymbol{T}_j = [t_j, t_{j-1}]$ if the integration is in negative direction. We denote such bounds by $\widetilde{\boldsymbol{y}}_j$, and refer to them as a priori bounds [23]; we shall also refer to $\boldsymbol{y}_j$ as tight bounds.

On the first integration step, VNODE-LP determines initial stepsize and the magnitude of the smallest stepsize that is allowed, $h_{\min}$. Then, on each step, VNODE-LP automatically selects a stepsize subject to absolute and relative error tolerances, atol and rtol. If the selected stepsize h_j is such that $|h_j| < h_{\min}$, the integration cannot continue, and VNODE-LP returns.

[2]VNODE-LP selects h_j and then finds t_j, where in computer arithmetic the true $t_{j-1} + h_j$ is rounded toward zero when computing t_j.

The following parameters can be changed by the user.

parameter	default value
p	20
atol	10^{-12}
rtol	10^{-12}
$h_{\min}$	computed by VNODE-LP

The functions for changing them are given in Section 4.4. The order p does not vary during an integration. By default $p = 20$, but its value can be changed at the beginning of an integration. As pointed out earlier, p must be between 3 and the value set to `MAX_ORDER` in the `makefile` for building the library; cf. Figures 2.2 and 2.3. If the user sets a value for $h_{\min}$, then this value is used by *integrate*.

If we wish to set, for example,

$$\begin{aligned} \text{rtol} = \text{atol} &= 10^{-14}, \\ p &= 40, \quad \text{and} \\ h_{\min} &= 10^{-5}, \end{aligned}$$

we proceed with (`cweave` typesets `1e-14` as $1 \cdot 10^{-14}$)

60 ⟨ set control data for the solver 60 ⟩ ≡

```
  Solver→setTols(1 · 10^−14, 1 · 10^−14);
  Solver→setOrder(40);
  Solver→setHmin(1 · 10^−5);
```

This code is used in chunk 61.

We write the main program

61 ⟨ `integctrl.cc` 61 ⟩ ≡

```
#include <fstream>
#include "vnode.h"
  using namespace std;
  using namespace vnodelp;
  ⟨ Lorenz 19 ⟩
  int main( )
  {
    ⟨ set initial condition and endpoint 21 ⟩
    ⟨ create AD object 22 ⟩
    ⟨ create a solver 23 ⟩
    ⟨ set control data for the solver 60 ⟩
    ⟨ open files 62 ⟩
    ⟨ indicate single step 43 ⟩
    ⟨ output initial condition 64 ⟩
    while (t ≠ tend) {
      Solver→integrate(t, y, tend);
      ⟨ output solution 65 ⟩
    }
    ⟨ close files 63 ⟩
    return 0;
  }
```

Writing into files

We output data into three files, `lorenz.tight`, `lorenz.apriori` and `lorenz.step`. In `lorenz.tight`, each line contains (rounded in decimal to output precision)

$$\mathrm{m}(\boldsymbol{t}_j) \quad \underline{\boldsymbol{y}}_{1,j} \quad \overline{\boldsymbol{y}}_{1,j} \quad \mathrm{w}(\boldsymbol{y}_{1,j})$$

where the subscripts $1, j$ refers to the jth computed solution for component y_1; $\mathrm{w}(\boldsymbol{y}_{1,j}) = 2\mathrm{r}(\boldsymbol{y}_{1,j})$ is the width, or diameter of $\boldsymbol{y}_{1,j}$. This width can be viewed as the *global excess* in the computed $\boldsymbol{y}_{1,j}$.

In `lorenz.apriori`, we output the a apriori enclosures and the corresponding time intervals in a form suitable for `gnuplot` to produce boxes denoting these a priori bounds; see Figure 3.5(b). The function *getAprioriEncl* returns $\widetilde{\boldsymbol{y}}_j$, and *getT* returns $\boldsymbol{T}_j$.

In `lorenz.step`, each line is

$$\mathrm{m}(\boldsymbol{t}_j) \quad h_j$$

The function *getStepsize* returns h_j.

```
62 ⟨ open files 62 ⟩ ≡
     ofstream outFile1("lorenz.tight", ios::out);
     ofstream outFile2("lorenz.step", ios::out);
     ofstream outFile3("lorenz.apriori", ios::out);
   This code is used in chunk 61.
```

```
63 ⟨ close files 63 ⟩ ≡
     outFile1.close();
     outFile2.close();
     outFile3.close();
   This code is used in chunk 61.
```

In the code below, *inf* returns the left point of an interval, and *sup* returns the right point of an interval. (The output goes through C++'s stream output, so the endpoints are not rounded outward.)

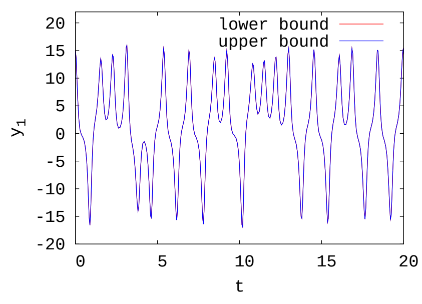
```
64 ⟨ output initial condition 64 ⟩ ≡
     outFile1 ≪ midpoint(t) ≪ "\t"
         ≪ inf(y[0]) ≪ "\t" ≪ sup(y[0]) ≪ "\t" ≪ width(y[0]) ≪ endl;
   This code is used in chunk 61.
```

```
65 ⟨ output solution 65 ⟩ ≡
     outFile1 ≪ midpoint(t) ≪ "\t"
         ≪ inf(y[0]) ≪ "\t" ≪ sup(y[0]) ≪ "\t" ≪ width(y[0]) ≪ endl;
```

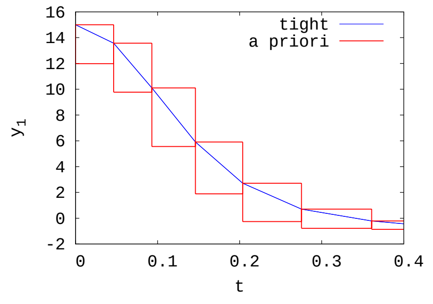
```
outFile2 << midpoint(t) << "\t" << Solver->getStepsize() << endl;
iVector Y = Solver->getAprioriEncl();
interval Tj = Solver->getT();
outFile3 << inf(Tj) << "\t" << inf(Y[0]) << endl;
outFile3 << inf(Tj) << "\t" << sup(Y[0]) << endl << endl;
outFile3 << sup(Tj) << "\t" << inf(Y[0]) << endl;
outFile3 << sup(Tj) << "\t" << sup(Y[0]) << endl << endl;
outFile3 << inf(Tj) << "\t" << inf(Y[0]) << endl;
outFile3 << sup(Tj) << "\t" << inf(Y[0]) << endl << endl;
outFile3 << inf(Tj) << "\t" << sup(Y[0]) << endl;
outFile3 << sup(Tj) << "\t" << sup(Y[0]) << endl << endl;
```
This code is used in chunk 61.

Plots

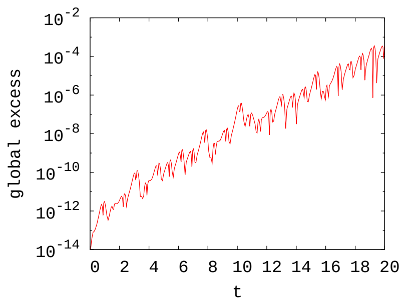
To visualize the results in these files, we produce the plots in Figure 3.5. In Figure 3.5(a) and (b), the upper and lower tight bounds cannot be distinguished when plotted. In (b), the a priori bounds are shown as boxes. The **gnuplot** file for generating this figure is displayed in Figure 6.5



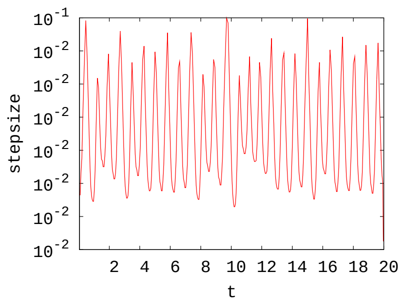
(a) Tight bounds on $y_1(t)$ versus t



(b) Tight and a priori bounds on $y_1(t)$ versus t



(c) $\log_{10}(\mathrm{w}(\boldsymbol{y}_{1,j}))$ versus t

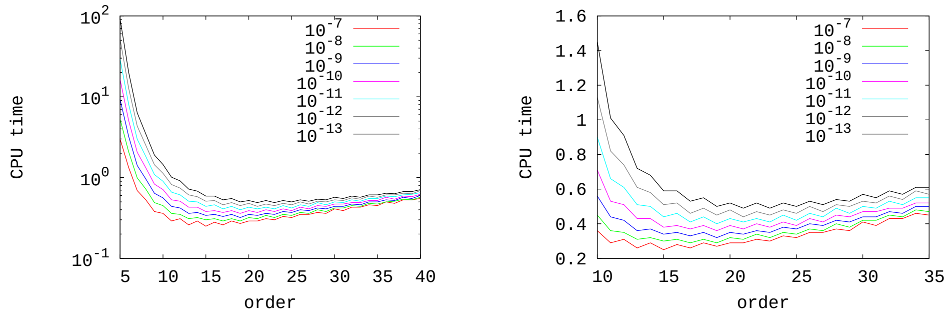


(d) Stepsize versus t

Figure 3.5. *Plots generated using* `integctrl.cc`

3.8 Work versus order

We show in Figure 3.6 how the computing time depends on the order for various tolerance when integrating the Lorenz system. We consider orders $p = 5, 6, \ldots, 40$ and tolerances atol = rtol = $10^{-7}, 10^{-8}, \ldots, 10^{-13}$.



(a) $\log_{10}$(CPU time) versus order (b) CPU time versus order

Figure 3.6. *Plots generated using* `orderstudy.cc`

The `gnuplot` file employed to generate these plots is given in Figure 6.6.

The main program is

```
68 ⟨ main program for order study 68 ⟩ ≡
  static double tol[ ] = {$1 \cdot 10^{-7}, 1 \cdot 10^{-8}, 1 \cdot 10^{-9}, 1 \cdot 10^{-10}, 1 \cdot 10^{-11}, 1 \cdot 10^{-12}$,
      $1 \cdot 10^{-13}$};
  int main( )
  {
    const int $n = 3$;
    ⟨ create AD object 22 ⟩
    ⟨ create a solver 23 ⟩
    iVector y(n);
    for (int $i = 0$; $i < 7$; i++)
    {
      Solver→setTols(tol[i]);
      ⟨ create file name 69 ⟩
      ofstream outFile(file_name.c_str( ), ios :: out);
      cout ≪ "␣tol␣=␣" ≪ tol[i] ≪
          ":␣writing␣into␣" ≪ file_name ≪ "␣..." ≪ endl;
      for (int $p = 5$; $p \le 40$; p++)
      {
        Solver→setOrder(p);
        interval $t = 0.0$, $tend = 10.0$;
        y[0] = 15.0;
        y[1] = 15.0;
        y[2] = 36.0;
        Solver→setFirstEntry( );

        double time = getTime( );

        ⟨ integrate (basic) 24 ⟩
        ⟨ check if success 25 ⟩
        time = getTotalTime(time, getTime( ));
        outFile ≪ p ≪ "\t" ≪ time ≪ endl;
      }
      outFile.close( );
    }
    return 0;
  }
```

This code is used in chunk 70.

We create a file name for each tolerance value by

69 ⟨ create file name 69 ⟩ ≡

```
  string prefix("order");

  std::stringstream num(std::stringstream::out);

  num ≪ tol[i];

  string file_name = prefix + num.str( ) + ".out";
```

This code is used in chunk 68.

We store all this into

70 ⟨ orderstudy.cc 70 ⟩ ≡

```
#include <fstream>
#include <sstream>
#include <string>
#include <cstdlib>
#include "vnode.h"
  using namespace std;

  using namespace vnodelp;

  ⟨ Lorenz 19 ⟩
  ⟨ main program for order study 68 ⟩
```

3.9 Work versus problem size

We investigate how the computing time depends on the size of the problem. We consider the DETEST problem C3 [14]

$$y' = \begin{pmatrix} -2 & 1 & 0 & 0 & \cdots & 0 \\ 1 & -2 & 1 & 0 & \cdots & 0 \\ 0 & 1 & -2 & 1 & \cdots & 0 \\ & & \vdots & & & \\ 0 & \cdots & & 1 & -2 & 1 \\ 0 & \cdots & & 0 & 1 & -2 \end{pmatrix} y \tag{3.2}$$

with $y(0) = (1, 0, \ldots, 0)^T$. We integrate with problem sizes $n = 40, 60, \ldots, 200$ for $t = 0$ to $t = 5$.

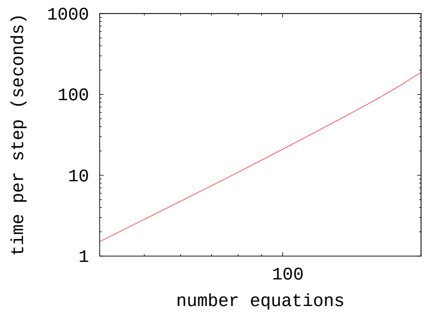
The C++ program is

71 ⟨ detest_c3.cc 71 ⟩ ≡

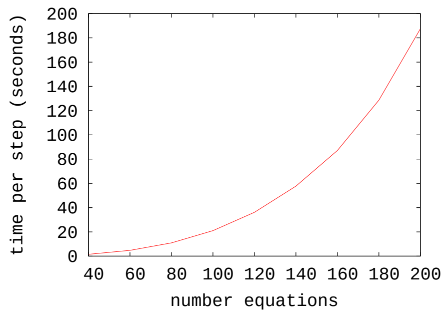
```
#include <ostream>
#include "vnode.h"
  using namespace std;
  using namespace vnodelp;

  template⟨typename var_type⟩
  void DETEST_C3(int n, var_type *yp, const var_type *y, var_type t,
          void *param)
  {
    yp[0] = −2.0 * y[0] + y[1];
    for (int i = 1; i < n − 1; i++) {
      yp[i] = y[i − 1] − 2.0 * y[i] + y[i + 1];
    }
    yp[n − 1] = y[n − 2] − 2.0 * y[n − 1];
  }

  int main()
  {
    for (int n = 40; n ≤ 200; n += 20) {
      cout ≪ n;

      interval t = 0.0, tend = 5;
      iVector y(n);

      for (int i = 0; i < n; i++) y[i] = 0;
      y[0] = 1;

      AD *ad = new FADBAD_AD(n, DETEST_C3, DETEST_C3);
      VNODE *Solver = new VNODE(ad);
      double time_start = getTime();

      Solver→integrate(t, y, tend);

      double time_end = getTime();
```

```
        cout ≪ "␣␣" ≪ getTotalTime(time_start,
            time_end) ≪ "␣␣␣" ≪ Solver→steps ≪ endl;
        delete Solver;
        delete ad;
    }
    return 0;
}
```

In Figure 3.7(a) and (b), we display the CPU time versus n. The `gnuplot` file for generating this figure is in Figure 6.7. It is not difficult to see that the computing



(a) $\log_{10}$(CPU time) versus n



(b) CPU time versus n

Figure 3.7. *CPU time versus n for Problem 3.2.* VNODE-LP *takes 8 steps for each n.*

time grows like n^3.

3.10 Stepsize behavior

We consider the orbit problem (see for example [4])

$$\begin{aligned} y_1'' &= y_1 + 2y_2' - \widehat{\mu}\frac{y_1+\mu}{D_1} - \mu\frac{y_1-\widehat{\mu}}{D_2}, \\ y_2'' &= y_2 - 2u_1' - \widehat{\mu}\frac{y_2}{D_1} - \mu\frac{y_2}{D_2}, \end{aligned} \tag{3.3}$$

where

$$\mu = 0.012277471, \quad \widehat{\mu} = 1-\mu, \tag{3.4}$$

$$D_1 = \left((y_1+\mu)^2 + y_2^2\right)^{3/2}, \quad \text{and} \tag{3.5}$$

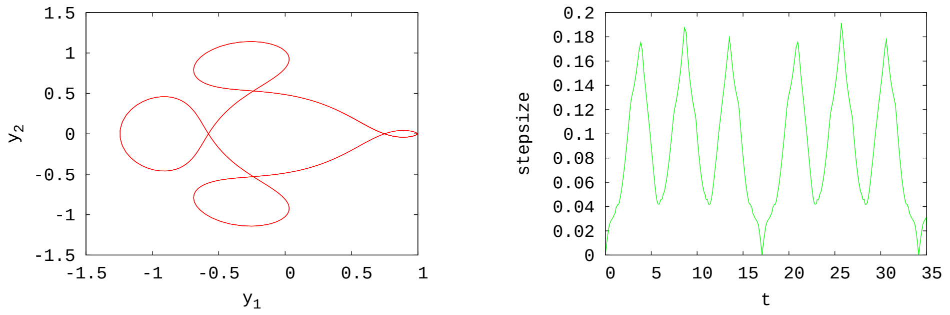
$$D_2 = \left((y_1-\widehat{\mu})^2 + y_2^2\right)^{3/2}. \tag{3.6}$$

We integrate this problem with

$$\begin{aligned} y_1(0) &= 0.994, \\ y_2(0) &= 0, \\ y_1'(0) &= 0, \\ y_2'(0) &= -2.00158510637908252240537862224, \end{aligned} \tag{3.7}$$

and $t_{\text{end}} = 35$, which corresponds to slightly more than two periods.

In Figure 3.8, we plot y_2 versus y_1 and the stepsize versus t. (The `gnuplot` file is in Figure 6.8.)



(a) Midpoints of the bounds on $y_2(t)$ versus $y_1(t)$

(b) Stepsize versus t

Figure 3.8. *Plots generated using* `orbit.cc`

The C++ program follows.

```
73 ⟨orbit.cc 73⟩ ≡
#include <fstream>
#include <sstream>
#include <string>
#include <cstdlib>
#include "vnode.h"
  using namespace std;
  using namespace vnodelp;
  template⟨typename var_type⟩
  void Orbit(int n, var_type *yp, const var_type *y, var_type t,
          void *param)
  {
    interval mu = string_to_interval("0.012277471");
    interval mu_h = 1.0 − mu;
    var_type D1 = pow(sqr(y[0] + mu) + sqr(y[1]), interval(1.5));
    var_type D2 = pow(sqr(y[0] − mu_h) + sqr(y[1]), interval(1.5));
    yp[0] = y[2];
    yp[1] = y[3];
    yp[2] = y[0] + 2.0 * y[3] - mu_h * (y[0] + mu)/D1 - mu * (y[0] - mu_h)/D2;
    yp[3] = y[1] - 2.0 * y[2] - mu_h * y[1]/D1 - mu * y[1]/D2;
  }
  int main()
  {
    const int n = 4;
    iVector y(n);
    y[0] = string_to_interval("0.994");
    y[1] = 0;
    y[2] = 0;
    y[3] = string_to_interval("-2.001585106379082522405378622224");
    interval t = 0.0, tend = 35;
    AD *ad = new FADBAD_AD(n, Orbit, Orbit);
    VNODE *Solver = new VNODE(ad);
    ofstream outFileSol("orbit_sol.out", ios::out);
    ofstream outFileStep("orbit_step.out", ios::out);
    outFileSol << midpoint(y[0]) << "\t" << midpoint(y[1]) << endl;
    Solver->setOneStep(on);
    while (t != tend) {
      Solver->integrate(t, y, tend);
      outFileSol << midpoint(y[0]) << "\t" << midpoint(y[1]) << endl;
      outFileStep << midpoint(t) << "\t"
          << Solver->getStepsize() << endl;
    }
    outFileSol.close();
    outFileStep.close();
    return 0;
  }
```

3.11 Stiff problems

We illustrate how VNODE-LP behaves when integrating a stiff problem. We integrate Van der Pol's equation (written as a first-order system)

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= \mu(1 - y_1^2)y_2 - y_1 \end{aligned} \tag{3.8}$$

with

$$y(0) = (2, 0)^T, \quad t_{\text{end}} = 200. \tag{3.9}$$

We perform integrations with $\mu = 10, 10^2, 10^3$, and 10^4. In Table 3.1, we show the number of steps and CPU time used by VNODE-LP. In Figure 3.9, we plot the stepsizes versus t. As can be seen from this table and figure, VNODE-LP is not efficient when this problem becomes stiff. In general, VNODE-LP works well on

μ	steps	time (secs)
10^1	2377	2.4
10^2	11697	11.6
10^3	126459	124.2
10^4	1180844	1173.2

Table 3.1. *Number of steps and CPU time used by* VNODE-LP *on (3.8–3.9)*

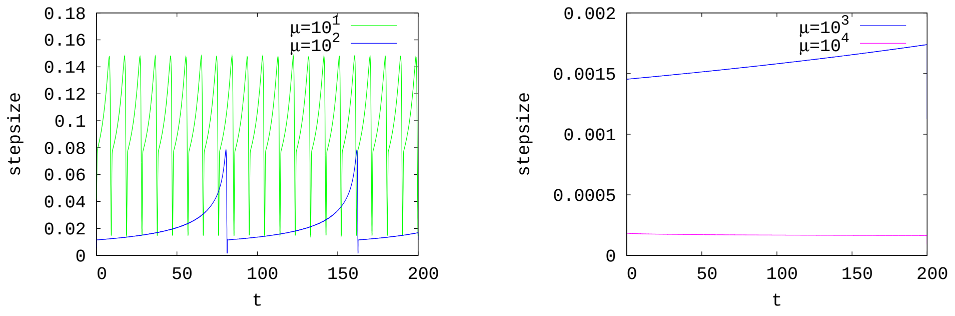


Figure 3.9. *Stepsize versus t on (3.8–3.9) for* $\mu = 10, 10^2, 10^3, 10^4$

non-stiff and mildly stiff problems. A more detailed study regarding stiff problems can be found in [23].

The program used to produce the numerical results for this problem follows. The `gnuplot` file for generating the plots is in Figure 6.9.

```
74 ⟨ vanderpol.cc 74 ⟩ ≡
#include <fstream>
#include <iomanip>
#include <sstream>
#include <string>
#include "vnode.h"
  using namespace std;
  using namespace vnodelp;
  template⟨typename var_type⟩
  void VDP(int n, var_type *yp, const var_type *y, var_type t,
           void *param)
  {
    double *MU = (double *) param;
    yp[0] = y[1];
    yp[1] = (*MU) * (1 − sqr(y[0])) * y[1] − y[0];
  }

  int main()
  {
```

```
  const int n = 2;
  double MU = 10.0;
  AD *ad = new FADBAD_AD(n, VDP, VDP, &MU);
  VNODE *Solver = new VNODE(ad);
     /* file for storing Table data */
  ofstream outSteps("vdp_nosteps.out", ios::out);
  outSteps ≪ fixed ≪ showpoint ≪ setprecision(1);
  for (int i = 1; i ≤ 4; i++) {
              /* file for storing stepsizes */
    string prefix("vdp_step");
    std::stringstream num(std::stringstream::out);
    num ≪ i;
    string file_name = prefix + num.str( ) + ".out";
    ofstream outStepSizes(file_name.c_str( ), ios::out);
    cout ≪ "␣MU␣=␣" ≪ MU ≪
        "␣writing␣into␣" ≪ file_name ≪ "..." ≪ endl;
    Solver→setFirstEntry( );
    Solver→setOneStep(on);
    interval t = 0, tend = 200;
    iVector y(n);
    y[0] = 2.0, y[1] = 0;
    interval t_prev = t;
    double time = getTime( );
    while (t ≠ tend) {
      Solver→integrate(t, y, tend);
      if (midpoint(t − t_prev) ≥ 0.01 ∨ t ≡ tend) {
        outStepSizes ≪ midpoint(t) ≪ "\t" ≪ Solver→getStepsize( ) ≪ endl;
        t_prev = t;
      }
    }
    outStepSizes.close( );
    time = getTotalTime(time, getTime( ));
    outSteps ≪ "$10^{" ≪ i ≪ "}$" ≪ "\t␣&␣"
        ≪ Solver→getNoSteps( ) ≪ "\t␣&␣"
        ≪ time ≪ "\t" ≪ "\\\\" ≪ endl;
    MU *= 10.0;
    ad→eval(&MU);
  }
  outSteps.close( );
  return 0;
}
```

Chapter 4

Interface

First, we list the data types used by VNODE-LP that are of interest to the user. Then we list and describe briefly the public functions of the VNODE-LP solver.

4.1 Interval data type

If PROFIL/BIAS [16] is employed, **interval** is defined as

76 ⟨interval data type (PROFIL) 76⟩ ≡
```
  typedef INTERVAL interval;
```
This code is used in chunk 115.

If FILIB++ [19] is employed, **interval** is defined as

77 ⟨interval data type (FILIB++) 77⟩ ≡
```
  typedef filib::interval⟨double⟩ interval;
```
This code is used in chunk 115.

VNODE-LP does not call the functions of these packages directly. Instead, it implements wrapper functions; see Subsection 4.3. To build VNODE-LP on a new IA package, only these functions need to be implemented using this new package.

4.2 Interval vector

VNODE-LP uses interval vector, **iVector**, defined as

79 ⟨interval vector 79⟩ ≡
```
#include <vector>
#include "vnodeinterval.h"
  using namespace std;
  using namespace v_bias;
  typedef vector < interval > iVector;
```
This code is used in chunk 122.

4.3 Wrapper functions

In the descriptions that follow, $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{c}$ denote corresponding intervals in mathematical notation. We assume that the endpoints of the input intervals are representable machine numbers and no exceptions occur when a result is computed.

double *inf*(**const interval** &*a*)
returns $\underline{\boldsymbol{a}}$.

double *sup*(**const interval** &*a*)
returns $\overline{\boldsymbol{a}}$.

double *midpoint*(**const interval** &*a*)
returns $(\underline{\boldsymbol{a}} + \overline{\boldsymbol{a}})/2$ rounded to the nearest.

double *width*(**const interval** &*a*)
returns $\overline{\boldsymbol{a}} - \underline{\boldsymbol{a}}$ rounded to $+\infty$.

double *mag*(**const interval** &*a*)
returns $\max\{|\underline{\boldsymbol{a}}|, |\overline{\boldsymbol{a}}|\}$.

bool *subseteq*(**const interval** &*a*, **const interval** &*b*)
returns *true* if $\boldsymbol{a} \subseteq \boldsymbol{b}$; *false* otherwise.

bool *interior*(**const interval** &*a*, **const interval** &*b*)
returns *true* if $\boldsymbol{a}$ is in the interior of $\boldsymbol{b}$; *false* otherwise.

bool *disjoint*(**const interval** &*a*, **const interval** &*b*)
returns *true* if $\boldsymbol{a} \cap \boldsymbol{b} = \emptyset$; *false* otherwise.

bool *intersect*(**interval** &*c*, **const interval** &*a*, **const interval** &*b*)
returns *true* if $\boldsymbol{a} \cap \boldsymbol{b} \neq \emptyset$ and sets $\boldsymbol{c} = \boldsymbol{a} \cap \boldsymbol{b}$; *false* if $\boldsymbol{a} \cap \boldsymbol{b} = \emptyset$ and leaves the input c unchanged.

interval *pi*( )
returns an interval containing π.

interval *pow*(**const interval** &*a*, **const interval** &*b*)
returns an interval containing $\{x^y \mid x \in \boldsymbol{a}, y \in \boldsymbol{b}\}$.

interval *pow*(**const interval** &*a*, **int** *k*)
returns an interval containing $\{x^k \mid x \in \boldsymbol{a}\}$.

Each of the functions e that follows returns $\{e(x) \mid x \in \boldsymbol{a}\}$.

interval *exp*(**const interval** &*a*)

interval *log*(**const interval** &*a*)

interval *sqr*(**const interval** &*a*)

interval *sqrt*(**const interval** &*a*)

interval *sin*(**const interval** &*a*)

interval *cos*(**const interval** &*a*)

interval *tan*(**const interval** &*a*)

interval *asin*(**const interval** &*a*)

interval *acos*(**const interval** &*a*)

interval *atan*(**const interval** &*a*)

Finally,

interval *string_to_interval*(**const char** ∗*s*)
returns an interval that contains the decimal number that is stored in the character string input.

4.4 Solver's public functions

The present solver is implemented by the class **VNODE**. We explain briefly its constructor and public member functions.

4.4.1 Constructor

VNODE(**AD** ∗*ad*)
constructs a **VNODE** object. Here *ad* is a pointer to an object of a class derived from the **AD** class; see Subsection 4.5. Currently, there is only one such class, **FADBAD_AD**, and *ad* is a pointer to an object of this class.

4.4.2 Integrator

void *integrate*(**interval** &*t*, **iVector** &*y*, **interval** *tend*)
By default, *integrate*($t, y, tend$) tries to compute an enclosure of the solution to an ODE problem at *tend*. If successful, y contains such an enclosure at $t = tend$. If an integration is not successful, y is an enclosure of the computed solution at $t \neq tend$.

The initial and end points, t_0 and t_{end}, can be stored as intervals containing their true values. Normally, the corresponding interval for t_0 [resp. t_{end}] should be of the width of (at most) a few machine numbers. If we denote these intervals by $\boldsymbol{t}_0$ and $\boldsymbol{t}_{\text{end}}$, *integrate* requires that $\boldsymbol{t}_0 \cap \boldsymbol{t}_{\text{end}} = \emptyset$. If $\boldsymbol{t}_0 \cap \boldsymbol{t}_{\text{end}} \neq \emptyset$, *integrate* returns without performing an integration.

4.4.3 Set functions

void *setTols*(**double** a, **double** $r = 0$)
sets atol to a and rtol to r. The latter parameter has a default value of 0. For example, *setTols*$(1 \cdot 10^{-10})$ sets atol $= 10^{-10}$ and rtol $= 0$.

The code roughly controls the "drift" away from the true solution at each integration step to be of size atol $+ \|\boldsymbol{y}\|_\infty \cdot$ rtol, where $\|\boldsymbol{y}\|$ is a measure of the size of current solution. This drift is accounted for by the enclosure, thus the size of the enclosure at the end point being roughly proportional to atol $+ \|\boldsymbol{y}\|_\infty \cdot$ rtol.

Default values are atol $= 10^{-12}$ and rtol $= 10^{-12}$.

void *setOrder*(**int** p)
sets the order to p. It must be between 3 and the value set to `MAX_ORDER` in the `makefile` for building the library; cf. Figures 2.2 and 2.3. If p is not in this range, *integrate* returns without performing an integration.

Experience suggests that order in the range of about 20 to 30 results in efficient integration.

Default value is 20.

void *setHmin*(**double** h)
sets the value of the magnitude of minimum stepsize allowed to h. If minimum stepsize is not set, or *setHmin*(0) is called, *integrate* computes such.

void *setFirstEntry*( )
indicates to *integrate* that this is a first entry into it. If this function is called before *integrate*, the latter will perform various initializations before time-stepping. When *integrate* is called for the first time, *setFirstEntry*( ) need not be called before *integrate*.

void *setOneStep*(**stepAction** *action*)
tells the integrator whether to stop, *action* $\equiv$ *on* or continue, *action* $\equiv$ *off*, after each step it takes. If *setOneStep*(*on*) is called before *integrate*, the latter will return after each step it takes. To turn off this feature, call *setOneStep*(*off*). In this case, if *integrate* is re-entered, it will not stop after each step it takes (except the last one).

4.4.4 Get functions

bool *successful*( ) **const**
returns *true* if an integration is successful and *false* otherwise. If *integrate* has not been called, *successful*( ) returns *true* by default.

unsigned int *getMaxOrder*( ) **const** returns the maximum order allowed in an integration. This is the value set in `MAX_ORDER` in the configuration file.

double *getStepsize*( ) **const**
returns the value of the most recent stepsize.

unsigned int *getNoSteps*( ) **const**
returns the number of successful steps taken by VNODE-LP.

const iVector &*getAprioriEncl*( ) **const**
returns the last computed $\widetilde{\boldsymbol{y}}_j$.

const interval &*getT*( ) **const**
returns the last computed $\boldsymbol{T}_j$.

double *getGlobalExcess*( )
returns an estimate of the global excess in the most recent computed enclosure.

double *getGlobalExcess*(**unsigned int** *i*)
returns an estimate of the global excess in the ith component of the most recent computed enclosure.

4.5 Constructing an AD object

VNODE-LP computes Taylor coefficients for the ODE and its variational equation. The user has to construct an object for computing such coefficients by

AD $*ad$ = **new FADBAD_AD**(*n*, *function_name*, *function_name*)
or

AD $*ad$ = **new FADBAD_AD**(*n*, *function_name*, *function_name*, *param*)

Here n is the size of the problem, *function_name* is the name of the template function, as described earlier, and *param* is a pointer to parameters that need to be passed to the ODE problem. After a parameter is changed, **void** *eval*(**void** $*p$) *must be called* to update internal structures in the **AD** object.

4.6 Some helpful functions

template⟨**class T**⟩ **void** *printVector*(**const T** &*v*, **const char** $*s = 0$)
prints the components of a vector v on the standard output. If the second parameter is given, *printVector* prints it before the content of v.

double *getTime*( )
returns the current time measured as user time.

double *getTotalTime*(**double** *start_time*, **double** *end_time*)
returns *end_time* − *start_time* rounded to the nearest.

Chapter 5

Testing

The code for the test cases is located in subdirectory `tests`. We give a brief description of each test.

5.1 General tests

File `test0.cc`

With the tests in this file, we check if the functions described in Subsection 4.3 compile and execute.

File `test01.cc`

We check if the elementary functions FADBAD++ uses compile and execute.

5.2 Linear problems

5.2.1 Constant coefficient problems

We consider

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= -y_1 \end{aligned} \tag{5.1}$$

with

$$y(0) = (1,1)^T. \tag{5.2}$$

The true solution is

$$y(t) = \begin{pmatrix} \cos(t) & \sin(t) \\ -\sin(t) & \cos(t) \end{pmatrix} y(0). \tag{5.3}$$

File `test1.cc`

We integrate (5.1, 5.2) from $t = 0$ to $t_{\text{end}} = 10000$. At each integration point t_j, selected by the solver, we evaluate (5.3) in interval arithmetic and check if the resulting enclosure intersects with the computed bounds. This test succeeds if they intersect for all t_j.

File `test2.cc`

We integrate (5.1, 5.2) and check that at $t = 2k\pi$, for $k = 2, 4, \dots, 1000$, $y(0)$ is contained in the computed bounds. If so, this test is successful.

File `test3.cc`

We integrate (5.1, 5.2), but with $t_{\text{end}} = -10000$. This test is successful if the true solution (5.3), evaluated in interval arithmetic, and the computed bounds intersect at each point selected by the solver.

File `test4.cc`

We integrate (5.1, 5.2) and check that at $-2k\pi$, for $k = 2, 4, \dots, 1000$, $y(0)$ is contained in the computed bounds. If so, this test is successful.

File `test5.cc`

We consider

$$\begin{aligned} y_1' &= y_1 - 2y_2 \\ y_2' &= 3y_1 - 4y_2 \end{aligned} \tag{5.4}$$

with

$$y(0) = (1, -1)^T. \tag{5.5}$$

The true solution is

$$y(t) = \begin{cases} 5e^{-t} - 4e^{-2t} \\ 5e^{-t} - 6e^{-2t}. \end{cases} \tag{5.6}$$

We integrate (5.4, 5.5) and check that, at each point selected by the solver, the true (evaluated in interval arithmetic (5.6)) and computed solutions intersect. If they do, we accept this test as successful.

5.2.2 Time-dependent problems

File `test6.cc`

The problem is

$$\begin{aligned}
y_1' &= \sin(t+10)y_1 - 2y_2 - y_3 - \cos(t)\\
y_2' &= 3y_1 - 4\cos(t^2)y_2 - \cos(t)\\
y_3' &= e^{-t^2}y_1 - e^{-t^2}y_2 - \sin(t)\\
y(0) &\in ([0,5],[-2,6],[5,12])^T, \quad t_{\text{end}} = 20.
\end{aligned}$$

We compute an enclosure at $t_{\text{end}} = 20$ with the above initial condition set. Then we compute bounds at t_{end} for each corner of the initial box. These bounds must intersect with the enclosure resulting from $([0,5],[-2,6],[5,12])^T$.

We have also computed (accurate) approximate solutions using MAPLE for each corner of the initial box. At t_{end}, each approximate solution must be inside the bounds computed with the same corner point.

5.3 Nonlinear problems

File `test_n1.cc`

We integrate the Lorenz system with $y(0) = (15,15,36)^T$ from 0 to 1. Denote the enclosure at $t=1$ by $\boldsymbol{y}_{(1)}$. We have also computed an approximate solution with MAPLE. Denote it by $\widehat{y}_{(1)}$. We check first if

$$\widehat{y}_{(1)} \in \boldsymbol{y}_{(1)}. \tag{5.7}$$

Then we integrate this system with $y(1) \in \boldsymbol{y}_{(1)}$ and $t_{\text{end}} = 0$. Denote the computed enclosure at $t=0$ by $\boldsymbol{y}_{(0)}$. We check if

$$y(0) \in \boldsymbol{y}_{(0)}. \tag{5.8}$$

Finally, we integrate with $y(0) \in \boldsymbol{y}_{(0)}$ and $t_{\text{end}} = 1$. Denote the computed enclosure by $\boldsymbol{y}^*_{(1)}$. We check if

$$\widehat{y}_{(1)} \in \boldsymbol{y}^*_{(1)} \quad \text{and} \quad \boldsymbol{y}^*_{(1)} \cap \boldsymbol{y}_{(1)} \neq 0. \tag{5.9}$$

This test is successful if (5.7), (5.8), and (5.9) hold.

File `test_n2.cc`

We integrate a three-body problem from 0 to 1 and then from 1 to 0. The initial condition at $t=0$ must be contained in the computed bounds.

File `test_n3.cc`

The same three-body problem is integrated from 0 to 1 with orders from 5 to the value set in `MAX_ORDER`. If all the computed enclosures with these orders intersect, we accept this test as successful.

File `test_n4.cc`

We integrate [20]

$$y'' + cy' + \sin(y) = b\cos(t),$$

written as a first-order system

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= b\cos(t) - cy_2 - \sin(y_1) \end{aligned}$$

with

$$y(0) \in ([0, 0], [1.9999, 2.0001])^T, \quad t_{\text{end}} = 8,$$

and $c = 0$ and $b = 0$.

We compare the computed enclosure by VNODE-LP and AWA [20]. If these enclosure intersect, we accept this test as successful.

File `test_n5.cc`

This is the restricted three-body test problem from AWA:

$$\begin{aligned} x'' &= x + 2y' - l\frac{x+m}{((x+m)^2+y^2)^{3/2}} - m\frac{(x-1)}{\big((x-1)^2+y^2\big)^{3/2}} \\ x'' &= y - 2x' - l\frac{y}{((x+m)^2+y^2)^{3/2}} - m\frac{y}{\big((x-1)^2+y^2\big)^{3/2}}, \end{aligned}$$

where $m = 1/82.45$ and $l = 1 - m$. The initial values are

$$\begin{aligned} x(0) &= 1.2, \\ x'(0) &= 0, \\ y(0) &= 0, \quad \text{and} \\ y'(0) &= -1.04935750983. \end{aligned}$$

We integrate from 0 to 6.192169331396. Again, this test is successful if the computed enclosures by VNODE-LP and AWA intersect.

File `test_n6.cc`

We integrate the Pleiades problem from the Test Set for IVP Solvers [21]. At the end point, we subtract the reference solution, given in [21], from the computed bounds. If the max norm of the resulting interval vector is $<= 10^{-2}$, we accept this test as successful.

Chapter 6

Listings

```
# CONFIG_FILE is set in vnodelp/makefile and exported in this
# file. vnodelp/makefile calls this makefile.

include ../config/$(CONFIG_FILE)

CXXFLAGS += -I../include          # compiler flags
LDFLAGS  += -L../lib              # library flags
LIBS      = -lvnode $(I_LIBS) $(LAPACK_LIB) \
         $(BLAS_LIB) $(GPP_LIBS) # libraries

EXAMPLES = orbit vanderpol basic E1 scalar basic         \
         intermediate integctrl odeparam integi          \
         order detest_c3

examples: $(EXAMPLES)

E1:  E1.o
         $(CXX) $(LDFLAGS) -o $@ E1.o $(LIBS)
scalar:  scalar.o
         $(CXX) $(LDFLAGS) -o $@ scalar.o $(LIBS)
basic:  basic.o
         $(CXX) $(LDFLAGS) -o $@ basic.o $(LIBS)
intermediate: intermediate.o
         $(CXX) $(LDFLAGS) -o $@ intermediate.o $(LIBS)
integctrl: integctrl.o
         $(CXX) $(LDFLAGS) -o $@ integctrl.o $(LIBS)
odeparam: odeparam.o
         $(CXX) $(LDFLAGS) -o $@ odeparam.o $(LIBS)
integi: integi.o
         $(CXX) $(LDFLAGS) -o $@ integi.o $(LIBS)
order: orderstudy.o
         $(CXX) $(LDFLAGS) -o $@ orderstudy.o $(LIBS)
detest_c3: detest_c3.o
         $(CXX) $(LDFLAGS) -o $@ detest_c3.o $(LIBS)
vanderpol: vanderpol.o
         $(CXX) $(LDFLAGS) -o $@ vanderpol.o $(LIBS)
orbit: orbit.o
         $(CXX) $(LDFLAGS) -o $@ orbit.o $(LIBS)
clean:
         @-$(RM) -rf pchdir tca.* *.o *.out core.* $(EXAMPLES)
cleanall:
         @-$(RM) -rf pchdir tca.* *.o *.cc *.out core.* $(EXAMPLES)
```

Figure 6.1. *The* `makefile` *in the* `examples` *directory*

```
# file basici.gp
set terminal postscript eps enh color solid "Courier" 28

set xlabel "t"
set ylabel "y_1"

set output 'lorenzi1.eps'
plot [0:6.3][-22:25]\
'lorenzi.out' u 1:($2+$3) \
        title 'lower bound' w l lt 1 lw 2,\
'lorenzi.out' u 1:($2-$3)\
        title 'upper bound' w l lt 3 lw 2

set output 'lorenzi2.eps'
set format y "%g"
set xtics 5.7,0.1,6.3
set ylabel "y_1" 0
plot [5.7:6.25][-22:0.4]\
'lorenzi.out' u 1:2:3 \
        title 'bounds'   w errorbars lw 2,\
'lorenzi.out' u 1:2   \
        title 'midpoint' w lines lt 3 lw 2

set output 'lorenzi_excess.eps'
set ylabel "global excess" 2
set logscale y
set xtics 1
set format y "10^{%L}"
plot [0:6.3] 'lorenzi.out' u 1:4 \
        notitle w l lt 1 lw 2
```

Figure 6.2. *The* `gnuplot` *file for generating the plot in Figure 3.3*

```
function dy = E1(t,y)
dy = zeros(2,1);
t1 = t+1;
dy(1) = y(2);
dy(2) = -(y(2)/t1 + (1.0- 0.25/(t1*t1))*y(1));
```

```
clear;
options = odeset('RelTol',1e-10,'AbsTol',1e-10);

y(1)= 0.6713967071418030;
y(2)= 0.09540051444747446;

[T,Y] = ode45(@E1,[0 20],y,options);
format long
Y(end,1:2)'
```

Figure 6.3. *The* MATLAB *code for the DETEST E1 problem*

```
# file odeparam.gp
set terminal postscript eps enh color solid "Courier" 28

set xlabel "y_1"
set ylabel "y_2"
set zlabel "y_3"

set xtics -20,10,20
set ytics -25,10,25
set ztics  0,10,45

set output 'odeparam.eps'
splot 'odeparam1.out' u 1:2:3 title '{/Symbol␣b}␣=␣8/3'\
                w l lt 1 lw 2,\
      'odeparam2.out' u 1:2:3 title '{/Symbol␣b}␣=␣5'\
                w l lt 3 lw 2
```

Figure 6.4. *The* `gnuplot` *file for generating the plots in Figure 3.4*

```
# file integctrl.gp
set terminal postscript eps enh color solid "Courier" 28

set ylabel "y_1"
set xlabel "t"

set output 'lorenz.eps'
plot [][-20:22]\
 'lorenz.tight' u 1:2 title 'lower bound' w l lw 2,\
 'lorenz.tight' u 1:3 title 'upper bound' w l lt 3 lw 2

set output 'lorenz2.eps'
set xtics 0, 0.1, 0.4
plot [0:0.4]\
 'lorenz.tight'   u 1:2 title 'tight'     w l lt 3 lw 2,\
 'lorenz.tight'   u 1:3 notitle           w l lt 3 lw 2,\
 'lorenz.apriori' u 1:2 title   'a priori' w l lt 1 lw 3

set output 'lorenz_err.eps'
set xtics 0,2,20
set ylabel "global excess" 2
set logscale y
set format y "10^{%L}"
plot 'lorenz.tight' u 1:4 notitle w l lw 2

set output 'lorenz_step.eps'
set nologscale
set ylabel "stepsize"
plot 'lorenz.step' u 1:2 notitle w l lw 2
```

Figure 6.5. *The* `gnuplot` *file for generating the plots in Figure 3.5*

```
# file orderstudy.gp
set terminal postscript eps enh color solid "Courier" 28

set ylabel "CPU time"
set xlabel "order"

set output 'order.eps'
plot  [10:35]\
       'order1e-07.out' u 1:2 title '10^{-7}'
w l lt 1 lw 2,\
       'order1e-08.out' u 1:2 title '10^{-8}'
w l lt 2 lw 2,\
       'order1e-09.out' u 1:2 title '10^{-9}'
w l lt 3 lw 2,\
       'order1e-10.out' u 1:2 title '10^{-10}' w l lt 4 lw 2,\
       'order1e-11.out' u 1:2 title '10^{-11}' w l lt 5 lw 2,\
       'order1e-12.out' u 1:2 title '10^{-12}' w l lt 9 lw 2,\
       'order1e-13.out' u 1:2 title '10^{-13}' w l lt 7 lw 2


set logscale y
set format y "10^{%L}"
set output 'timeorder.eps'
plot 'order1e-07.out' u 1:2 title '10^{-7}'  w l lt 1 lw 2,\
     'order1e-08.out' u 1:2 title '10^{-8}'  w l lt 2 lw 2,\
     'order1e-09.out' u 1:2 title '10^{-9}'  w l lt 3 lw 2,\
     'order1e-10.out' u 1:2 title '10^{-10}' w l lt 4 lw 2,\
     'order1e-11.out' u 1:2 title '10^{-11}' w l lt 5 lw 2,\
     'order1e-12.out' u 1:2 title '10^{-12}' w l lt 9 lw 2,\
     'order1e-13.out' u 1:2 title '10^{-13}' w l lt 7 lw 2
```

Figure 6.6. *The* `gnuplot` *file for generating the plots in Figure 3.6*

```
# file work.gp
set terminal postscript eps enh color solid "Courier" 28

# model of the work
f(x) = a + b*x

fit f(x) "work.out" using (log($1)):(log($2)) via a,b

set xlabel "number equations"
set ylabel "time per step (seconds)"
set xrange [40:200]

set output 'work.eps'
plot 'work.out' using 1:2 notitle with lines

set logscale
set output 'worklog.eps'
plot 'work.out' using 1:2 notitle with lines
```

Figure 6.7. *The* `gnuplot` *file for generating the plots in Figure 3.7*

```
# file orbit.gp
set terminal postscript eps enh color solid "Courier" 28

set xlabel "y_1"
set ylabel "y_2"


set output 'orbit_sol.eps'
plot  'orbit_sol.out' u 1:2 notitle w l lt 1 lw 2

set xlabel "t"
set ylabel "stepsize"
set output 'orbit_step.eps'
plot  'orbit_step.out' u 1:2 notitle w l lt 2 lw 2
```

Figure 6.8. *The* `gnuplot` *file for generating the plots in Figure 3.8*

```
# file vanderpol.gp
set terminal postscript eps enh color solid "Courier" 28

set xlabel 't'
set ylabel 'stepsize'

set output 'vdp_step1.eps'
plot [0:200][0:0.18] 'vdp_step1.out' u 1:2 \
       title '{/Symbol_m}=10^1' w l lt 2 lw 2,\
     'vdp_step2.out' u 1:2 \
       title '{/Symbol_m}=10^2' w l lt 3 lw 2

set output 'vdp_step2.eps'
plot [0:200][0:0.002] 'vdp_step3.out' u 1:2 \
       title '{/Symbol_m}=10^3' w l lt 3 lw 2,\
     'vdp_step4.out' u 1:2 \
       title '{/Symbol_m}=10^4' w l lt 4 lw 2
```

Figure 6.9. *The* `gnuplot` *file for generating the plots in Figure 3.9*

Part II

Third-party Components

Chapter 7
Packages

The VNODE-LP package builds on

- LAPACK [2] and BLAS [1],
- interval-arithmetic (IA) package FILIB++ [19] *or* PROFIL/BIAS [16], and
- automatic differentiation (AD) package FADBAD++ [29].

The interfaces to the IA package are kept as small as possible, which allows a new package to be introduced without substantial programming effort.

Chapter 8 presents the implementation of interfaces to FILIB++ and PROFIL/BIAS. Chapter 9 discusses functions for changing the rounding mode. The AD in VNODE-LP is implemented through abstract classes, which are described in Chapter 17. Implementation of these classes using FADBAD++ is given in Chapter 22.

Chapter 8

IA package

The basic data type in VNODE-LP is **interval**. This package can be built on top of FILIB++ [19] or PROFIL [16], or potentially other packages. The user can select which of these packages to use; for more details see the installation instructions in Section 2.3.

Each of these IA packages can be replaced, provided that the new package supplies an interval data type with overloaded arithmetic operations and elementary functions working with interval arguments. To incorporate a new IA package, the body of the functions described in Section 4, and implemented below, need to be implemented using the new package.

8.1 Functions calling FILIB++

113 ⟨functions calling FILIB++ 113⟩ ≡

```
inline double inf(const interval &a) {
  return a.inf();
}
inline double sup(const interval &a) {
  return a.sup();
}
inline double midpoint(const interval &a) {
  return a.mid();
}
inline double width(const interval &a) {
  return a.diam();
}
inline double mag(const interval &a) {
  return a.mag();
}
```

```
inline bool subseteq(const interval &a, const interval &b) {
  return filib::subset(a, b);
}
inline bool interior(const interval &a, const interval &b) {
  return filib::interior(a, b);
}
inline bool disjoint(const interval &a, const interval &b) {
  return filib::disjoint(a, b);
}
inline bool intersect(interval &c, const interval &a, const interval &b)
{
  if (filib::disjoint(a, b)) return false;
  c = filib::intersect(a, b);
  return true;
}
inline interval pi() {
  return filib::interval<double>::PI();
}
inline interval pow(const interval &a, const interval &b) {
  return filib::pow(a, b);
}
inline interval pow(const interval &a, int b) {
  return filib::power(a, b);
}
inline interval exp(const interval &a) {
  return filib::exp(a);
}
inline interval log(const interval &a) {
  return filib::log(a);
}
inline interval sqr(const interval &a) {
  return filib::sqr(a);
}
inline interval sqrt(const interval &a) {
  return filib::sqrt(a);
}
inline interval sin(const interval &a) {
  return filib::sin(a);
}
inline interval cos(const interval &a) {
  return filib::cos(a);
}
```

```
inline interval tan(const interval &a) {
  return filib::tan(a);
}
inline interval asin(const interval &a) {
  return filib::asin(a);
}
inline interval acos(const interval &a) {
  return filib::acos(a);
}
inline interval atan(const interval &a) {
  return filib::atan(a);
}
inline interval string_to_interval(const char *s)
{
  std::cerr << "\n\n␣***␣WARNING␣***␣\n";
  std::cerr << "␣␣␣␣␣Conversion␣from␣a␣string␣to␣an␣int\
      erval␣containing␣it␣\n";
  std::cerr << "␣␣␣␣␣has␣not␣been␣implemented␣in␣filib++.\n";
  std::cerr << "␣␣␣␣␣For␣this␣feature,␣consider␣using␣PROFIL/BIAS.\n";
  std::cerr << "␣␣␣␣␣The␣input␣string␣" << s <<
      "␣is␣converted␣to␣double.\n";
  std::cerr << "␣␣␣␣␣␣␣␣␣␣***␣THE␣COMPUTED␣BOUNDS␣MAY␣N\
      OT␣BE␣CORRECT␣***␣\n\n";
  double a;
  std::istringstream iss(s);
  iss >> a;
  return interval(a);
}
```

This code is used in chunk 115.

8.2 Functions calling PROFIL

114 ⟨functions calling PROFIL 114⟩ ≡

```
inline double inf(const interval &a) {
  return Inf(a);
}
inline double sup(const interval &a) {
  return Sup(a);
}
inline double midpoint(const interval &a) {
  return Mid(a);
}
inline double width(const interval &b) {
  return Diam(b);
}
inline double mag(const interval &a) {
  return Abs(a);
}
inline bool subseteq(const interval &a, const interval &b) {
  return a ≤ b;
}
inline bool interior(const interval &a, const interval &b) {
  return (Inf(a)) > (Inf(b)) ∧ ((Sup(a)) < (Sup(b)));
}
inline bool disjoint(const interval &a, const interval &b) {
  interval c;
  return ¬Intersection(c, a, b);
}
inline bool intersect(interval &c, const interval &a, const interval &b)
{
  return Intersection(c, a, b);
}
inline interval pi( ) {
  return ArcCos(−1.0);
}
inline interval pow(const interval &a, int b) {
  return Power(a, b);
}
inline interval pow(const interval &a, const interval &b) {
  return Power(a, b);
}
inline interval exp(const interval &a) {
  return Exp(a);
}
inline interval log(const interval &a) {
  return Log(a);
}
inline interval sqr(const interval &a) {
  return Sqr(a);
}
inline interval sqrt(const interval &a) {
  return Sqrt(a);
}
inline interval sin(const interval &a) {
  return Sin(a);
}
```

```
  inline interval cos(const interval &a) {
    return Cos(a);
  }
  inline interval tan(const interval &a) {
    return Tan(a);
  }
  inline interval asin(const interval &a) {
    return ArcSin(a);
  }
  inline interval acos(const interval &a) {
    return ArcCos(a);
  }
  inline interval atan(const interval &a) {
    return ArcTan(a);
  }
  inline interval string_to_interval(const char *s)
  {
    return Enclosure(s);
  }
```

This code is used in chunk 115.

Files

The interface to the IA package is stored in

115 ⟨vnodeinterval.h 115⟩ ≡

```
#ifndef VNODEINTERVAL_H
#define VNODEINTERVAL_H
#ifdef PROFIL_VNODE
#include <Interval.h>
#include <Functions.h>
#include <LongReal.h>
#include <LongInterval.h>
  namespace v_bias {
    ⟨interval data type (PROFIL) 76⟩
    ⟨functions calling PROFIL 114⟩
  }
#endif
#ifdef FILIB_VNODE
#include <interval/interval.hpp>
#include <iostream>
#include <sstream>
#include <string>
  namespace v_bias {
    ⟨interval data type (FILIB++) 77⟩
    ⟨functions calling FILIB++ 113⟩
  }
#endif
#endif
```

Chapter 9

Changing the rounding mode

For changing the rounding mode, VNODE-LP calls the functions below.

void *round_nearest*( )
sets the rounding mode to the nearest.

void *round_down*( )
sets the rounding mode to $-\infty$.

void *round_up*( )
sets the rounding mode to ∞.

Depending on the selected IA package, they call corresponding functions either from PROFIL/BIAS or FILIB++. A particular implementation is selected by the value of the variable `I_PACKAGE` in the configuration file; see Subsection 2.3.2.

9.1 Changing the rounding mode using FILIB++

Changing the rounding mode using FILIB++ is done through **round_control** defined as

117 ⟨rounding control type (FILIB++) 117⟩ ≡
```
typedef filib::rounding_control < double , true > round_control;
```
This code is used in chunk 120.

The necessary functions are implemented as follows.

118 ⟨changing the rounding mode (FILIB++) 118⟩ ≡
```
inline void round_nearest( ) {
  round_control::tonearest( );
}
inline void round_down( ) {
  round_control::downward( );
}
```

```
  inline void round_up( ) {
    round_control::upward( );
  }
```

This code is used in chunk 120.

9.2 Changing the rounding mode using BIAS

Similarly, we implement functions for changing the rounding mode using BIAS functions.

```
119 ⟨changing the rounding mode (BIAS) 119⟩ ≡
  inline void round_nearest( ) {
    BiasRoundNear( );
  }
  inline void round_down( ) {
    BiasRoundDown( );
  }
  inline void round_up( ) {
    BiasRoundUp( );
  }
```

This code is used in chunk 120.

Files

The above functions are stored in

```
120 ⟨vnoderound.h 120⟩ ≡
#ifndef VNODEROUND_H
#define VNODEROUND_H
#ifdef FILIB_VNODE
#include <rounding_control/rounding_control_double.hpp>
  namespace v_bias {
    ⟨rounding control type (FILIB++) 117⟩
    ⟨changing the rounding mode (FILIB++) 118⟩
  }
#endif
#ifdef PROFIL_VNODE
#include "Bias0.h"
  namespace v_bias {
    ⟨changing the rounding mode (BIAS) 119⟩
  }
#endif
#endif
```

Part III

Linear Algebra and Related Functions

Chapter 10

Vectors and matrices

The VNODE-LP package works with point and interval vectors and matrices, namely **pVector**, **iVector**, **pMatrix**, and **iMatrix**. Here "**p**" is for point and "**i**" is for interval. The **iVector** type was defined in Subsection 4.2; **pVector**, **pMatrix**, and **iMatrix** are defined as

122 ⟨ vector and matrix types 122 ⟩ ≡

```
⟨ interval vector 79 ⟩
typedef vector⟨double⟩ pVector;
typedef vector⟨vector⟨double⟩⟩ pMatrix;
typedef vector⟨vector⟨interval⟩⟩ iMatrix;
```

This code is used in chunk 127.

Size and memory allocation

template⟨class Matrix⟩ void *sizeM* (**Matrix** &A, **unsigned int** n)
allocates space for an $n \times n$ matrix.

template⟨class Matrix⟩ *sizeM* (**const Matrix** &A)
returns the size of a square matrix A.

template⟨class Vector⟩ unsigned int *sizeV* (**const Vector** &a)
allocate space for an n vector.

template⟨class Vector⟩ unsigned int *sizeV* (**const Vector** &a)
returns the size of a vector.

124 ⟨ size/allocation 124 ⟩ ≡

```
template⟨class Matrix⟩ inline void sizeM(Matrix &A, unsigned int n)
{
  A.resize(n);
  for (unsigned int i = 0; i < A.size( ); i++) A[i].resize(n);
}
```

```
  template⟨class Matrix⟩ inline unsigned int sizeM(const Matrix &A)
  {
    return A.size( );
  }
  template⟨class Vector⟩ inline void sizeV(Vector &a, unsigned int n)
  {
    a.resize(n);
  }
  template⟨class Vector⟩ inline unsigned int sizeV(const Vector &a)
  {
    return a.size( );
  }
```

This code is used in chunk 127.

Files

127 ⟨vector_matrix.h 127⟩ ≡

```
#ifndef VECTOR_MATRIX
#define VECTOR_MATRIX
#include <vector>
#include "vnodeinterval.h"
  using namespace std;
  using namespace v_bias;
  namespace v_blas {
    ⟨vector and matrix types 122⟩
    ⟨size/allocation 124⟩
  }
#endif
```

Chapter 11

Basic functions

We provide "generic" functions for operations involving matrices and vectors. Below, A, B, and C are $n \times n$ matrices, x, y, and z are n vectors, and a is a scalar.

11.1 Vector operations

template$\langle$**class Vector**, **class scalar**$\rangle$
void *setV*(**Vector** &z, **scalar** a)
sets each component of z to a.

template$\langle$**class Vector**, **class Vector**$\rangle$
void *assignV*(**Vector** &z, **const Vector** &x)
copies x to z

template$\langle$**class Vector**, **class scalar**$\rangle$
void *scaleV*(**Vector** &z, **scalar** a)
multiplies each element of z by a.

void *addViVi*(**iVector** &z, **const iVector** &x)
adds z and x and stores the result in z.

void *addViVp*(**iVector** &z, **const pVector** &x)
adds z and x and stores the result in z.

void *subViVp*(**iVector** &z, **const pVector** &x)
subtracts x from z and stores the result in z.

void *subViVi*(**iVector** &z, **const iVector** &x)
subtracts x from z and stores the result in z.

void *addViVi*(**iVector** &z, **const iVector** &x, **const iVector** &y)
adds x and y and stores the result in z.

void *addViVp*(**iVector** &z, **const iVector** &x, **const pVector** &y)
adds x and y and stores the result in z.

void *subViVp*(**iVector** &z, **const iVector** &x, **const pVector** &y)
subtracts y from x and stores the result in z.

double *inf_normV*(**const iVector** &z)
returns $\max |\boldsymbol{z}_i|$, where $|\boldsymbol{a}_i| = \max\{|\underline{a}_i|, |\overline{a}_i|\}$.

double *inf_normV*(**const pVector** &z)
returns $\|z\|$.

template⟨**class scalar**, **class Vector1**, **class Vector2**⟩
inline void *dot_product*(**scalar** &r, **const Vector1** &a, **const Vector2** &b)
computes the dot product of a and b and stores it in r. For input point vectors, this dot product is computed in the current rounding mode.

double *norm2*(**const pVector** &v)
returns the two norm of a point vector. For a point vector, this norm is computed in the current rounding mode.

129 ⟨ vector operations 129 ⟩ ≡

```
template⟨class Vector, class scalar⟩
        inline void setV(Vector &z, scalar a)
{
  fill(z.begin(), z.end(), a);
}
template⟨class Vector1, class Vector2⟩
        inline void assignV(Vector1 &z, const Vector2 &x)
{
  for (unsigned int i = 0; i < sizeV(z); i++) z[i] = x[i];
}
template⟨class Vector, class scalar⟩
        inline void scaleV(Vector &z, scalar a)
{
  for (unsigned int i = 0; i < sizeV(z); i++) z[i] *= a;
}
inline void addViVi(iVector &z, const iVector &x)
{
  transform(z.begin(), z.end(), x.begin(), z.begin(), plus⟨v_bias::interval⟩());
}
inline void addViVp(iVector &z, const pVector &x)
{
  transform(z.begin(), z.end(), x.begin(), z.begin(), plus⟨v_bias::interval⟩());
}
```

```
inline void subViVp(iVector &z, const pVector &x)
{
  transform(z.begin( ), z.end( ), x.begin( ), z.begin( ),
      minus⟨v_bias::interval⟩( ));
}
inline void subViVi(iVector &z, const iVector &x)
{
  transform(z.begin( ), z.end( ), x.begin( ), z.begin( ),
      minus⟨v_bias::interval⟩( ));
}
inline void addViVi(iVector &z, const iVector &x, const iVector &y)
{
  transform(x.begin( ), x.end( ), y.begin( ), z.begin( ), plus⟨v_bias::interval⟩( ));
}
inline void addViVp(iVector &z, const iVector &x, const pVector &y)
{
  transform(x.begin( ), x.end( ), y.begin( ), z.begin( ), plus⟨v_bias::interval⟩( ));
}
inline void subViVp(iVector &z, const iVector &x, const pVector &y)
{
  transform(x.begin( ), x.end( ), y.begin( ), z.begin( ),
      minus⟨v_bias::interval⟩( ));
}
inline double inf_normV(const iVector &z)
{
  double s = 0;
  for (unsigned int i = 0; i < sizeV(z); i++)
    if (v_bias::mag(z[i]) > s)  s = v_bias::mag(z[i]);
  return s;
}
inline double inf_normV(const pVector &z)
{
  double s = 0;
  for (unsigned int i = 0; i < sizeV(z); i++)
    if (fabs(z[i]) > s)  s = fabs(z[i]);
  return s;
}
template⟨class scalar, class Vector1, class Vector2⟩
          inline void dot_product(scalar &r, const Vector1 &a, const
         Vector2 &b)
{
  r = 0.0;
  for (unsigned int i = 0; i < a.size( ); i++)  r += a[i] * b[i];
}
```

```
inline double norm2(const pVector &v) {
   double s;
   dot_product(s, v, v);
   return sqrt(s);
}
```

This code is used in chunk 137.

11.2 Matrix/vector operations

void *multMiVi*(**iVector** &z, **const iMatrix** &A, **const iVector** &x)
multiplies A and x and stores the result in z.

void *multMpVi*(**iVector** &z, **const pMatrix** &A, **const iVector** &x)
multiplies A and x and stores the result in z.

void *multMiVp*(**iVector** &z, **const pMatrix** &A, **const iVector** &x)
multiplies A and x and stores the result in z.

131 ⟨ matrix times vector 131 ⟩ ≡

```
inline void multMiVi(iVector &z, const iMatrix &A, const iVector &x)
{
   for (unsigned int i = 0; i < A.size( ); i++) dot_product(z[i], A[i], x);
}
inline void multMpVi(iVector &z, const pMatrix &A, const iVector &x)
{
   for (unsigned int i = 0; i < A.size( ); i++) dot_product(z[i], A[i], x);
}
inline void multMiVp(iVector &z, const iMatrix &A, const pVector &x)
{
   for (unsigned int i = 0; i < A.size( ); i++) dot_product(z[i], A[i], x);
}
```

This code is used in chunk 137.

11.3 Matrix operations

template⟨**class Matrix**, **class scalar**⟩
void *setM*(**Matrix** &C, **scalar** a)
sets each component of C to a.

template⟨**class Matrix**⟩ **void** *setId*(**Matrix** &C)
sets C to the identity matrix.

template⟨**class Matrix1**, **class Matrix2**⟩
void *assignM*(**Matrix1** &C, **const Matrix2** &A)
copies A to C.

template⟨**class Matrix**, **class scalar**⟩ **void** *scaleM*(**Matrix** &C, **scalar** a)
multiplies each component of C by a.

template⟨**class Matrix**⟩ **void** *transpose*(**Matrix** &C, **const Matrix** &A)
stores the transpose of A in C.

template⟨**class Matrix**⟩ **void** *addId*(**Matrix** &C)
adds the identity matrix to C.

template⟨**class Matrix**⟩ **void** *subFromId*(**Matrix** &C)
subtracts C from the identity matrix and stores the result in C.

void *addMiMi*(**iMatrix** &C, **const iMatrix** &A)
adds C and A and stores the result in C.

void *subMiMp*(**iMatrix** &C, **const pMatrix** &A)
subtracts A from C and stores the result in C.

void *multMiMi*(**iMatrix** &C, **const iMatrix** &A, **const iMatrix** &B)
multiplies A and B and stored the result in C.

void *multMiMp*(**iMatrix** &C, **const iMatrix** &A, **const pMatrix** &B)
multiplies A and B and stored the result in C.

double *inf_normM*(**const iMatrix** &C)
computes $\|C\|_\infty$. The computation is in the current rounding mode. For example, if an upper bound on $\|C\|_\infty$ is desired, the rounding mode must be set to $+\infty$ before calling *inf_normM*.

132 ⟨ matrix operations 132 ⟩ ≡

```
template⟨class Matrix, class scalar⟩
        inline void setM(Matrix &C, scalar a)
{
  for (unsigned int i = 0; i < C.size( ); i++) setV(C[i], a);
}
template⟨class Matrix⟩
        inline void setId(Matrix &C)
{
  setM(C, 0.0);
  for (unsigned int i = 0; i < sizeM(C); i++) C[i][i] = 1.0;
}
template⟨class Matrix1, class Matrix2⟩
        inline void assignM(Matrix1 &C, const Matrix2 &A)
{
  for (unsigned int i = 0; i < C.size( ); i++) assignV(C[i], A[i]);
}
```

```
template⟨class Matrix, class scalar⟩
        inline void scaleM(Matrix &C, scalar a)
{
  for (unsigned int i = 0; i < C.size(); i++) scaleV(C[i], a);
}
template⟨class Matrix⟩
        inline void transpose(Matrix &C, const Matrix &A)
{
  for (unsigned int i = 0; i < C.size(); i++) setColumn(C, A[i], i);
}
template⟨class Matrix⟩
        inline void addId(Matrix &C)
{
  for (unsigned int i = 0; i < sizeM(C); i++) C[i][i] += 1.0;
}
template⟨class Matrix⟩
        inline void subFromId(Matrix &C)
{
  unsigned int n = sizeM(C);

  for (unsigned int i = 0; i < n; i++)
    for (unsigned int j = 0; j < n; j++) C[i][j] = −C[i][j];
  for (unsigned int i = 0; i < n; i++) C[i][i] += 1.0;
}
inline void addMiMi(iMatrix &C, const iMatrix &A)
{
  for (unsigned int i = 0; i < C.size(); i++) addViVi(C[i], A[i]);
}
inline void subMiMp(iMatrix &C, const pMatrix &A)
{
  for (unsigned int i = 0; i < C.size(); i++) subViVp(C[i], A[i]);
}
inline void multMiMi(iMatrix &C, const iMatrix &A, const iMatrix &B)
{
  unsigned int n = sizeM(A);

  for (unsigned int i = 0; i < n; i++)
    for (unsigned int j = 0; j < n; j++) {
      C[i][j] = 0.0;
      for (unsigned int k = 0; k < n; k++) C[i][j] += A[i][k] ∗ B[k][j];
    }
}
inline void multMiMp(iMatrix &C, const iMatrix &A, const pMatrix &B)
{
  unsigned int n = sizeM(A);
  for (unsigned int $i = 0$; $i < n$; $i$++)
    for (unsigned int $j = 0$; $j < n$; $j$++) {
      $C[i][j] = 0.0$;
      for (unsigned int $k = 0$; $k < n$; $k$++) $C[i][j]$ += $A[i][k] * B[k][j]$;
    }
}
template⟨class Matrix⟩ inline double inf_normM(const Matrix &C)
{
  unsigned int $n$ = sizeM(C);
  double $m = 0$;
  for (unsigned int $i = 0$; $i < n$; $i$++) {
    double $s = 0$;
    for (unsigned int $j = 0$; $j < n$; $j$++) $s$ += v_bias::mag(C[i][j]);
    if ($s > m$) $m = s$;
  }
  return $m$;
}
```

This code is used in chunk 137.

11.4 Get/set column

template⟨class Vector, class Matrix⟩
void *getColumn*(**Vector** &z, **const Matrix** &C, **unsigned int** j)
stores the jth column of C in z.

template⟨class Matrix, class Vector⟩
void *setColumn*(**Matrix** &C, **const Vector** &z, **unsigned int** j)
sets the jth column of C to z.

134 ⟨ get/set column 134 ⟩ ≡

```
template⟨class Vector, class Matrix⟩
void getColumn(Vector &z, const Matrix &C, unsigned int j)
{
  for (unsigned int $i = 0$; $i <$ sizeM(C); $i$++) $z[i] = C[i][j]$;
}
template⟨class Matrix, class Vector⟩
void setColumn(Matrix &C, const Vector &z, unsigned int j)
{
  for (unsigned int $i = 0$; $i <$ sizeM(C); $i$++) $C[i][j] = z[i]$;
}
```

This code is used in chunk 137.

11.5 Conversions

The following two functions are convenient when calling LAPACK.

template⟨**class scalar**, **class Matrix**⟩
void *matrix2pointer*(**scalar** $*M$, **const Matrix** $\&C$)
copies C into an array pointed to by M. The matrix at M is in a column-major form. That is, the (i, j) element of the matrix in the array at M is at $jn + i$, where n is the size of the matrix.

template⟨**class Matrix**, **class scalar**⟩
void *pointer2matrix*(**Matrix** $\&C$, **const scalar** $*M$)
copies a matrix stored in an array pointed to by M into a matrix C. The matrix at M is in a column-major form. That is, the (i, j) element of the matrix in the array at M is at $jn + i$, where n is the size of the matrix.

```
135 ⟨ matrix2pointer 135 ⟩ ≡
  template⟨class scalar, class Matrix⟩
          inline void matrix2pointer(scalar *M, const Matrix &C)
  {
    unsigned int n = sizeM(C);
    for (unsigned int j = 0; j < n; j++)
      for (unsigned int i = 0; i < n; i++) M[j * n + i] = C[i][j];
  }
```
This code is used in chunk 137.

```
136 ⟨ pointer2matrix 136 ⟩ ≡
  template⟨class Matrix, class scalar⟩
          inline void pointer2matrix(Matrix &C, const scalar *M)
  {
    unsigned int n = sizeM(C);
    for (unsigned int j = 0; j < n; j++)
      for (unsigned int i = 0; i < n; i++) C[i][j] = M[j * n + i];
  }
```
This code is used in chunk 137.

Files

We store the above functions in

```
137 ⟨ basiclinalg.h 137 ⟩ ≡
#ifndef BASICLINALG_H
#define BASICLINALG_H
#include <algorithm>
#include "vector_matrix.h"
  namespace v_blas {
    ⟨ vector operations 129 ⟩
    ⟨ matrix times vector 131 ⟩
    ⟨ matrix operations 132 ⟩
    ⟨ get/set column 134 ⟩
    ⟨ matrix2pointer 135 ⟩
    ⟨ pointer2matrix 136 ⟩
    ⟨ print vector 366 ⟩
  }
#endif
```

Chapter 12

Interval functions

12.1 Inclusion

This *subseteq* function returns *true* if, for two interval vectors $\boldsymbol{a}$ and $\boldsymbol{b}$, $\boldsymbol{a} \subseteq \boldsymbol{b}$, and returns *false* otherwise.

139 ⟨check vector inclusion 139⟩ ≡

```
inline bool subseteq(const iVector &a, const iVector &b)
{
  return equal(a.begin( ), a.end( ), b.begin( ), v_bias :: subseteq);
}
```

See also chunk 141.

This code is used in chunk 151.

12.2 Interior

The *interior* function returns *true* if, for two interval vectors $\boldsymbol{a}$ and $\boldsymbol{b}$, $\boldsymbol{a}$ is componentwise in the interior of $\boldsymbol{b}$, and returns *false* otherwise.

140 ⟨check if in the interior 140⟩ ≡

```
inline bool interior(const iVector &a, const iVector &b)
{
  return equal(a.begin( ), a.end( ), b.begin( ), v_bias :: interior);
}
```

This code is used in chunk 151.

The *disjoint* function returns *true* if, for two interval vectors $\boldsymbol{a}$ and $\boldsymbol{b}$, $\boldsymbol{a} \cap \boldsymbol{b} = \emptyset$, and returns *false* otherwise.

141 ⟨check vector inclusion 139⟩ +≡

```
inline bool disjoint(const iVector &a, const iVector &b)
{
  return equal(a.begin( ), a.end( ), b.begin( ), v_bias :: disjoint);
}
```

12.3 Radius

The radius of an interval is

142 ⟨rad (interval) 142⟩ ≡
```
inline double rad(const interval &a)
{
  return 0.5 * v_bias::width(a);
}
```
This code is used in chunk 151.

We compute the radius of an interval vector by

143 ⟨rad (vector) 143⟩ ≡
```
inline void rad(pVector &r, const iVector &v)
{
  transform(v.begin(), v.end(), r.begin(), v_bias::rad);
}
```
This code is used in chunk 151.

12.4 Width

We compute the width of an interval vector by

144 ⟨width 144⟩ ≡
```
inline void width(pVector &r, const iVector &a)
{
  transform(a.begin(), a.end(), r.begin(), v_bias::width);
}
```
This code is used in chunk 151.

12.5 Midpoints

145 ⟨midpoint of an interval vector 145⟩ ≡
```
inline void midpoint(pVector &r, const iVector &a)
{
  transform(a.begin(), a.end(), r.begin(), v_bias::midpoint);
}
```
This code is used in chunk 151.

146 ⟨midpoint of an interval matrix 146⟩ ≡
```
inline void midpoint(pMatrix &R, const iMatrix &A)
{
  for (unsigned int i = 0; i < A.size(); i++) midpoint(R[i], A[i]);
}
```
This code is used in chunk 151.

12.6 Intersection

If interval vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ intersect, we store their intersection in z and return *true*. Otherwise, we return *false*.

```
147 ⟨ intersection of interval vectors 147 ⟩ ≡
  inline bool intersect(iVector &z, const iVector &x, const iVector &y)
  {
    interval c;
    for (unsigned int i = 0; i < sizeV(y); i++) {
      bool b = v_bias::intersect(c, x[i], y[i]);
      if (¬b) return false;
      else z[i] = c;
    }
    return true;
  }
```

This code is used in chunk 151.

12.7 Computing h such that $[0,h]\boldsymbol{a} \subseteq \boldsymbol{b}$

12.7.1 The interval case

Given intervals $\boldsymbol{a}$ and $\boldsymbol{b}$, where $0 \in \boldsymbol{b}$, we wish to compute the largest machine-representable $h \geq 0$ such that

$$[0,h]\,\boldsymbol{a} \subseteq \boldsymbol{b}.$$

We consider the following cases.

1. $\underline{\boldsymbol{a}} = \overline{\boldsymbol{a}}$.
   (a) If $\underline{\boldsymbol{a}} = \overline{\boldsymbol{a}} = 0$, then we set h to be the largest representable machine number.
   (b) If $\underline{\boldsymbol{a}} = \overline{\boldsymbol{a}} > 0$, then $[0,h]\boldsymbol{a} = [0,h\overline{\boldsymbol{a}}] \subseteq \boldsymbol{b}$ iff $h \leq \overline{\boldsymbol{b}}/\overline{\boldsymbol{a}}$.
   (c) If $\underline{\boldsymbol{a}} = \overline{\boldsymbol{a}} < 0$, then $[0,h]\boldsymbol{a} = [\underline{\boldsymbol{a}}h, 0] \subseteq \boldsymbol{b}$ iff $h \leq \underline{\boldsymbol{b}}/\underline{\boldsymbol{a}}$.
2. $\underline{\boldsymbol{a}} \neq \overline{\boldsymbol{a}}$.
   (a) $\underline{\boldsymbol{a}} \geq 0$. Then $\overline{\boldsymbol{a}} > 0$ and $[0,h]\boldsymbol{a} = [0,h\overline{\boldsymbol{a}}] \subseteq \boldsymbol{b}$ iff $h \leq \overline{\boldsymbol{b}}/\overline{\boldsymbol{a}}$.
   (b) $\overline{\boldsymbol{a}} \leq 0$. Then $\underline{\boldsymbol{a}} < 0$ and $[0,h]\boldsymbol{a} = [h\underline{\boldsymbol{a}}, 0] \subseteq \boldsymbol{b}$ iff $h \leq \underline{\boldsymbol{b}}/\underline{\boldsymbol{a}}$.

We summarize the above cases in the following procedure:

1. If $\underline{\boldsymbol{a}} = \overline{\boldsymbol{a}} = 0$ then set h to be the largest representable machine number;
2. else
   (a) if $\underline{\boldsymbol{a}} \geq 0$ then $h = \nabla(\overline{\boldsymbol{b}}/\overline{\boldsymbol{a}})$
   (b) else $h = \nabla(\underline{\boldsymbol{b}}/\underline{\boldsymbol{a}})$.

149 $\langle$ h such that $[0,h]\boldsymbol{a} \subseteq \boldsymbol{b}$ (intervals) 149 $\rangle \equiv$

```
#include <climits>
  using namespace std;
  using namespace v_bias;
  inline double compH(const v_bias::interval &a, const v_bias::interval
      &b)
  {
    if (inf(a) ≡ 0 ∧ sup(a) ≡ 0) return numeric_limits⟨double⟩::max();
    round_down();
    if (inf(a) ≥ 0) return sup(b)/sup(a);
    return inf(b)/inf(a);
  }
```

This code is used in chunk 150.

12.7.2 The interval vector case

Given two interval vectors $\boldsymbol{a}$ and $\boldsymbol{b}$, where b contains the vector with zero component, we compute the largest machine-representable $h \geq 0$ such that

$$[0,h]\boldsymbol{a} \subseteq \boldsymbol{b}.$$

150 $\langle$ h such that $[0,h]\boldsymbol{a} \subseteq \boldsymbol{b}$ (interval vectors) 150 $\rangle \equiv$

```
  ⟨ h such that [0,h]a ⊆ b (intervals) 149 ⟩
  double compH(const iVector &a, const iVector &b)
  {
    double hmin = compH(a[0], b[0]);
    for (unsigned int i = 1; i < sizeV(a); i++) {
      double h = compH(a[i], b[i]);
      if (h < hmin) hmin = h;
    }
    return hmin;
  }
```

This code is used in chunk 152.

Files

151 $\langle$ `intvfuncs.h` 151 $\rangle \equiv$

```
#ifndef INTVFUNC_H
#define INTVFUNC_H
#include "vector_matrix.h"
  namespace v_bias {
    ⟨ rad (interval) 142 ⟩
  } namespace v_blas {
    using namespace v_bias;
    ⟨ check if in the interior 140 ⟩
    ⟨ check vector inclusion 139 ⟩
    ⟨ width 144 ⟩
    ⟨ midpoint of an interval vector 145 ⟩
    ⟨ midpoint of an interval matrix 146 ⟩
    ⟨ intersection of interval vectors 147 ⟩
    ⟨ rad (vector) 143 ⟩
    double compH(const iVector &a, const iVector &b);
  }
#endif
```

152 ⟨ intvfuncs.cc 152 ⟩ ≡

```
#include <limits>
#include <algorithm>
#include "vnodeinterval.h"
#include "vnoderound.h"
#include "vector_matrix.h"
  namespace v_blas {
    ⟨ h such that [0, h]a ⊆ b (interval vectors) 150 ⟩
  }
```

Chapter 13

QR factorization

In our implementation of VNODE-LP, we need to compute the QR factorization of an $n \times n$ matrix. We employ the routines `DGEQRF` and `DORGQR` from LAPACK. `DGEQRF` computes a QR factorization of a real $m \times n$ matrix A. `DORGQR` generates an $m \times n$ real matrix Q with (approximately) orthonormal columns, which is defined from the elementary reflectors returned by `DGEQRF`. If *computeQR* is successful, *true* is returned; otherwise *false* is returned.

153 ⟨ compute QR factorization 153 ⟩ ≡

```
extern "C"
{
  void dgeqrf_(int *m, int *n, double *A, int *lda, double *tau, double
       *work, int *lwork, int *info);
  void dorgqr_(int *m, int *n, int *k, double *A, int *lda, double
       *tau, double *work, int *lwork, int *info);
}

bool computeQR(pMatrix &Q, const pMatrix &A)
{
  int n = sizeM(A);
  int m = n;
  int lda = n;
  int k = n;
  int info;
  int lwork = 10 * n;     /* lwork has to be n *(optimal block size). The value
       10 is somewhat random. */
  double *tau = new double[n];
  double *work = new double[lwork];
  double *M = new double[n * n];
  v_bias::round_nearest( );
  matrix2pointer(M, A);
  dgeqrf_(&m, &n, M, &lda, tau, work, &lwork, &info);
```

```
    if (info ≡ 0) {
      dorgqr_(&m, &n, &k, M, &lda, tau, work, &lwork, &info);
      if (info ≡ 0) pointer2matrix(Q, M);
    }
    delete[] M;
    delete[] work;
    delete[] tau;
    if (info ≡ 0) return true;
    return false;
  }
```

This code is used in chunk 154.

Files

154 ⟨qr.cc 154⟩ ≡

```
#include "basiclinalg.h"
#include "vnoderound.h"
  namespace vnodelp {
    using namespace v_blas;
    ⟨compute QR factorization 153⟩
  }
```

Chapter 14

Matrix inverse

The **MatrixInverse** class provides functions for computing an approximate inverse of a point matrix, enclosing the inverse of a point matrix and enclosing the inverse of a floating-point approximation to an orthogonal matrix.

14.1 Matrix inverse class

```
156 ⟨class MatrixInverse 156⟩ ≡
  class MatrixInverse {
  public:
    MatrixInverse(int n);

    bool invertMatrix(pMatrix &Ainv, const pMatrix &A);
    bool encloseMatrixInverse(iMatrix &Ainv, const pMatrix &A);
    bool orthogonalInverse(iMatrix &Ainv, const pMatrix &A);

    ~MatrixInverse( );

    int iterations;
  private:
    bool encloseLS(iVector &x, const pMatrix &A, const iVector &b0,
        const iMatrix &B, double beta);
    pVector radx;
    iVector b0, x1, x;
    iMatrix B, Ci;
    pMatrix C;
    double *M;
    int *ipiv;
    double *work;
    int lwork;
  };
```
This code is used in chunk 172.

invertMatrix tries to compute a floating-point approximation to the inverse of A (if it exists). If successful, *invertMatrix* stores the result in *Ainv* and returns *true*; otherwise, it returns *false*.

encloseMatrixInverse tries to enclose the inverse of A (if it exists). If successful, *encloseMatrixInverse* stores the result in *Ainv* and returns *true*; otherwise, it returns *false*.

orthogonalInverse tries to enclose the inverse of a floating-point approximation to an orthogonal matrix. If successful, *orthogonalInverse* stores the result in *Ainv* and returns *true*; otherwise it returns *false*.

14.2 Computing A^{-1}

First, we compute the LU factorization of A using LAPACK's *dgetrf*. Then, using this LU factorization, we try to compute the inverse of A using LAPACK's *dgetri*.

158 ⟨compute A^{-1} 158⟩ ≡

```
  extern "C"
  {
    void dgetrf_(int *m, int *n, double *A, int *lda,
        int *ipiv, int *info);
    void dgetri_(int *n, double *A, int *lda,
        int *ipiv, double *work, int *lwork, int *info);
  }
  bool MatrixInverse::invertMatrix(pMatrix &Ainv, const pMatrix &A)
  {
    int n = sizeM(A);
    int lda = n;
    int info;
    matrix2pointer(M, A);      /*
```

- n number of rows and columns
- M the matrix to be factored on input
- *lda* the leading dimension of M, $lda \geq max(1, n)$. We set $lda = n$
- *ipiv* pivot indices
- *info* success if $info \equiv 0$

```
      */
    v_bias::round_nearest();
    dgetrf_(&n, &n, M, &lda, ipiv, &info);
    if (info ≠ 0) {
#ifdef VNODE_DEBUG
```

```
        printMessage("Could␣not␣invert␣a␣matrix");
#endif
        return false;
      }    /*
```

- *lwork* size of *work*. We set $lwork = 2 * n$ in the constructor
- *work* array of size *lwork*
- *info* success if $info \equiv 0$

```
       */
      dgetri_(&n, M, &lda, ipiv, work, &lwork, &info);
      if (info ≠ 0) {
#ifdef VNODE_DEBUG
        printMessage("Could␣not␣invert␣a␣matrix");
#endif
        return false;
      }
      pointer2matrix(Ainv, M);
      return true;
  }
```

This code is used in chunk 173.

14.3 Enclosing the solution of a linear system

We try to enclose the solution to the linear system $Ax = b$, $A \in \mathbb{R}^{n\times n}$, $b \in \mathbb{R}^n$ in *encloseLS*. It implements Krawczyk's method; see for example [10]. By $C \in \mathbb{R}^{n\times n}$ we denote a preconditioner. This method works if $\|I - CA\|_\infty = \beta < 1$. Let

$$\alpha = \frac{\|Cb\|_\infty}{1-\beta}.$$

Then

$$x \in ([-\alpha, \alpha], \ldots, [-\alpha, \alpha])^T.$$

This shows how to compute an initial box that is guaranteed to contain x. Then Krawczyk's iteration is

$$\boldsymbol{x}^{(i+1)} = \bigl(Cb + (I - CA)\boldsymbol{x}^{(i)}\bigr) \cap \boldsymbol{x}^{(i)}.$$

The *encloseLS* function follows. In the comment inside it, before the horizontal line, we list the input variables and what they contain. The output is described after this line.

When describing code in this manuscript, we shall mix variable names and math symbols, where appropriate. Such a mixture may be viewed as an abuse of notation, but the author has found it helpful. For example $b0 \ni Cb$ below means that the interval vector *b0* contains the true Cb. Similarly, $beta \geq \|I - CA\|_\infty$ denotes that the value of the input variable *beta* must be greater or equal the true value of $\|I - CA\|_\infty$.

160 ⟨ enclose the solution to $Ax = b$ 160 ⟩ ≡
```
bool MatrixInverse::encloseLS(iVector &x,
            const pMatrix &A, const iVector &b0,
            const iMatrix &B, double beta)
{     /*
```

$$\frac{\begin{array}{r l} A & \text{is the matrix from } Ax = b \\ b0 & \ni Cb \\ B & \ni I - CA \\ beta & \geq \|I - CA\|_\infty, \quad beta \text{ must be} < 1 \end{array}}{x \text{ contains the solution to } Ax = b \text{ if } true \text{ is returned}}$$

```
    */
  ⟨ compute initial box 161 ⟩
  ⟨ do Krawczyk's iteration 165 ⟩
  return true;
}
```
This code is used in chunk 173.

14.3.1 Initial box

The input *b0* contains Cb, and the input B contains $I - CA$. Also, $beta \geq \|I - CA\|_\infty$. We compute $a \geq \|Cb\|_\infty/(1-\beta)$ and set the initial box containing the solution.

161 ⟨ compute initial box 161 ⟩ ≡
```
v_bias::round_down( );
double a = 1 − beta;      /* a ≤ 1 − β */
double a1 = inf_normV(b0);      /* a1 ≥ ‖Cb‖∞ */
v_bias::round_up( );
a = a1/a;      /* a ≥ ‖Cb‖∞/(1 − β) */
setV(x, v_bias::interval(−a, a));
```
This code is used in chunk 160.

14.3.2 Krawczyk's iteration

We implement $Cb + (I - CA)\boldsymbol{x}^{(i)}$. That is, we program the expression $x1 = b0 + B * x$.

162 ⟨ Krawczyk's iteration 162 ⟩ ≡
```
multMiVi(x1, B, x);
addViVi(x1, b0);
```
See also chunk 163.
This code is used in chunk 165.

Now we can intersect *x1* and x. The result is stored in x, if $b \equiv true$. Otherwise, we return *false*.

```
163 ⟨Krawczyk's iteration 162⟩ +≡
  bool b = intersect(x, x, x1);
  if (b ≡ false) {
#ifdef VNODE_DEBUG
    printMessage("x␣and␣x1␣do␣not␣intersect");
#endif
    return false;
  }
```

For the **while** loop that follows, we need to compute the sum of the radii of the components of x.

```
164 ⟨sum radii 164⟩ ≡
  rad(radx, x);
  v_bias::round_nearest();
  sum_radii = accumulate(radx.begin(), radx.end(), 0.0);
```

This code is used in chunk 165.

Now, we compose the whole iteration. Initially, we set *sum_old_radii* to the largest machine number. The **while** loop below iterates as far as $sum\_radii < mult * sum\_old\_radii$. The factor $mult = (1 + beta)/2$ is the same as in [10].

```
165 ⟨do Krawczyk's iteration 165⟩ ≡
  double sum_old_radii = numeric_limits⟨double⟩::max();
  double sum_radii;
  ⟨sum radii 164⟩
  round_up();
  int max_iterations = 20;
  int counter = 0;
  double mult = (1 + beta)/2;
  while (sum_radii < mult * sum_old_radii ∧ counter < max_iterations) {
    ⟨Krawczyk's iteration 162⟩
    sum_old_radii = sum_radii;
    ⟨sum radii 164⟩
    counter++;
  }
  iterations = counter;
```

This code is used in chunk 160.

14.4 Enclosing the inverse of a general point matrix

To enclose the inverse of a point matrix A, we enclose the solution to

$$Ax_i = e_i \quad \text{for } i = 1, \ldots, n,$$

where e_i is the ith unit vector. Then, the ith column of the inverse is $\boldsymbol{x}_i$.

First we compute a floating-point inverse of A. If *invertMatrix* fails, *encloseMatrixInverse* returns *false*. Then we compute *beta* that is needed for *encloseLS* and call this function for each e_i.

```
166 ⟨ enclose the inverse of a matrix 166 ⟩ ≡
  bool MatrixInverse::encloseMatrixInverse(iMatrix &Ainv,
          const pMatrix &A)
  {
    bool b = invertMatrix(C, A);
    if (b ≡ false) return false;
    ⟨ find beta 167 ⟩
    ⟨ enclose each column 169 ⟩
#ifdef VNODE_DEBUG
    iMatrix B = Ainv;
    multMiMp(B, Ainv, A);
    int n = sizeM(A);
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++) {
        interval b = B[i][j];
        if (i ≡ j) assert(v_bias::subseteq(interval(1.0), b));
        else assert(v_bias::subseteq(interval(0.0), b));
      }
#endif
    return true;
  }
This code is used in chunk 173.
```

We compute B such that it contains $I - CA$.

```
167 ⟨ find beta 167 ⟩ ≡
  assignM(Ci, C);
  multMiMp(B, Ci, A);
  subFromId(B);
See also chunk 168.
This code is used in chunks 166 and 170.
```

Now we find *beta* $\geq \beta$. If *beta* ≥ 1, we return *false*, as Krawczyk's iteration cannot proceed.

```
168 ⟨ find beta 167 ⟩ +≡
  v_bias::round_up();
  double beta = inf_normM(B);
  if (beta ≥ 1) return false;
```

Finally, we can call *encloseLS* for each right side e_i. *getColumn* extracts the ith column of C in *b0*. *setColumn* sets the interval vector containing the corresponding solution to $Ax = e_i$.

169 ⟨ enclose each column 169 ⟩ ≡
```
for (unsigned int $i = 0$; $i < sizeM(A)$; $i{+}{+}$) {
   $getColumn(b0, C, i)$;
   bool $b = encloseLS(x, A, b0, B, beta)$;
   if ($b \equiv false$) {
#ifdef VNODE_DEBUG
      $printMessage$("Could␣not␣enclose␣the␣solution␣to␣a␣linear␣system");
#endif
      return $false$;
   }
   $setColumn(Ainv, x, i)$;
}
```
This code is used in chunks 166 and 170.

14.5 Enclosing the inverse of an orthogonal matrix

We have a floating-point approximation for an orthogonal matrix. Normally, its transpose is not the same as the inverse of this matrix. Hence, we need to enclose it.

170 ⟨ enclose the inverse of an orthogonal matrix 170 ⟩ ≡
```
bool MatrixInverse :: $orthogonalInverse$(iMatrix &$Ainv$,
         const pMatrix &$A$)
{
   $transpose(C, A)$;
   ⟨ find beta 167 ⟩
   ⟨ enclose each column 169 ⟩
   return $true$;
}
```
This code is used in chunk 173.

14.6 Constructor and destructor

171 ⟨ **MatrixInverse** constructor/destructor 171 ⟩ ≡
```
MatrixInverse :: MatrixInverse(int $n$)
{
   $M$ = new double[$n * n$];
   $ipiv$ = new int[$n$];
   $lwork = 2 * n$;
   $work$ = new double[$lwork$];
   $sizeV(radx, n)$;
   $sizeV(b0, n)$;
   $sizeV(x1, n)$;
   $sizeV(x, n)$;
   $sizeM(B, n)$;
   $sizeM(Ci, n)$;
```

```
    sizeM(C, n);
  }
  MatrixInverse::~MatrixInverse( )
  {
    delete[] work;
    delete[] ipiv;
    delete[] M;
  }
```

This code is used in chunk 173.

Files

172 ⟨matrixinverse.h 172⟩ ≡

```
#ifndef MATRIXINVERSE_H
#define MATRIXINVERSE_H
#include <numeric>
#include "vnodeinterval.h"
#include "vnoderound.h"
#include "vector_matrix.h"
  namespace v_blas {
    using namespace v_bias;
    ⟨class MatrixInverse 156⟩
  }
#endif
```

173 ⟨matrixinverse.cc 173⟩ ≡

```
#include <climits>
#include "matrixinverse.h"
#include "basiclinalg.h"
#include "intvfuncs.h"
#include "debug.h"
  using namespace std;
        namespace v_blas {
    ⟨MatrixInverse constructor/destructor 171⟩
    ⟨compute A^{-1} 158⟩
    ⟨enclose the solution to Ax = b 160⟩
    ⟨enclose the inverse of a matrix 166⟩
    ⟨enclose the inverse of an orthogonal matrix 170⟩
  }
```

Part IV

Solver Implementation

Chapter 15

Structure

We list the classes in VNODE-LP along with brief descriptions.These classes are depicted in Figure 15.1.

Solution contains a representation of the solution at each time point.

Apriori contains a representation of an enclosure of the solution over an integration step.

Control stores various data for controlling an integration.

AD_ODE provides functions for generating Taylor coefficients for the solution to an ODE.

AD_VAR provides functions for generating Taylor coefficients for the solution to the variational equation.

AD aggregates objects of **AD_ODE** and **AD_VAR**.

FadbadODE, **FadbadVarODE**, and **FADBAD_AD** are implementations of **AD_ODE**, **AD_VAR**, and **AD**, respectively.

HOE implements the High-Order Enclosure method [27] for enclosing the solution to an ODE and computing a priori bounds.

IHO implements the Interval Hermite-Obreschkoff method [23] for computing tight bounds on the solution.

VNODE implements the overall integrator.

MatrixInverse provides functions for computing the inverse of a matrix.

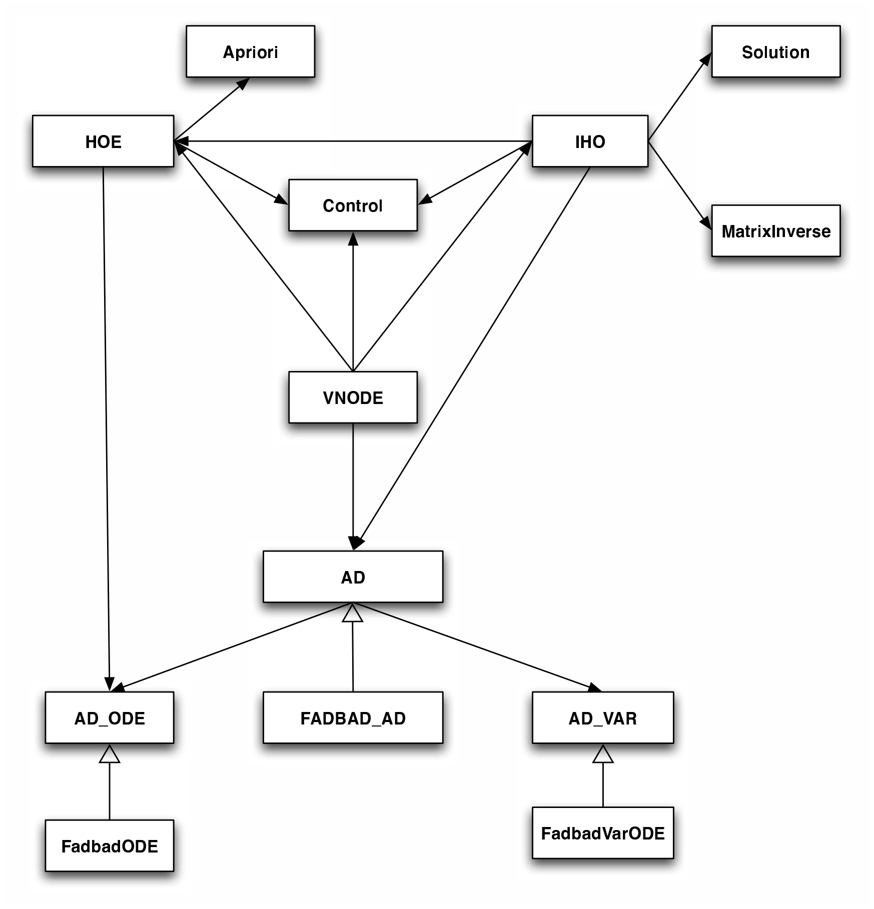


Figure 15.1. *Classes in* VNODE-LP. *The triangle arrows denote* inheritance *relations; the normal arrows denote* uses *relations.*

Chapter 16

Solution enclosure representation

The present solver implements a one-step method. The initial point t_0 and endpoint t_{end} are stored as intervals. For example, if t_0 is representable, the interval is $\boldsymbol{t}_0 = [t_0, t_0]$; otherwise, $\boldsymbol{t}_0$ is an interval containing t_0. The interval containing t_{end} is denoted by $\boldsymbol{t}_{\text{end}}$.

Internally, VNODE-LP select points that are machine-representable numbers. For simplicity in the considerations that follow, we denote such a point by the interval $\boldsymbol{t}_j$, which contains t_j. At $\boldsymbol{t}_0$, the user provides $\boldsymbol{y}_0$. On each integration step, we maintain two types of representations of an enclosure of an ODE solution: tight enclosure at $\boldsymbol{t}_j$ and an a priori enclosure over $[\underline{\boldsymbol{t}}_j, \overline{\boldsymbol{t}}_{j+1}]$ (or $[\underline{\boldsymbol{t}}_{j+1}, \overline{\boldsymbol{t}}_j]$ if the integration is in negative direction).

16.1 Tight enclosure

We maintain three representations as enclosures on the solution at $\boldsymbol{t}_j$:

$$\begin{aligned}&\{u_j + S_j\alpha + A_j r \mid \alpha \in \boldsymbol{\alpha},\, r \in r_j\}\\&\{u_j + S_j\alpha + Q_j r \mid \alpha \in \boldsymbol{\alpha},\, r \in r_{\text{QR},j}\}, \quad \text{and}\\&\boldsymbol{y}_j.\end{aligned}$$

Here, $u_j, \alpha, r \in \mathbb{R}^n$; $S_j, A_j, Q_j \in \mathbb{R}^{n\times n}$; and $\boldsymbol{r}_j, \boldsymbol{r}_{\text{QR},j}, \boldsymbol{\alpha}, \boldsymbol{y}_j \in \mathbb{IR}^n$. The first set corresponds to the P (parallelepiped) method, and the second set corresponds to the QR method [24]. The use of these sets is discussed in detail in Chapter 20.

178 ⟨ tight enclosure representation 178 ⟩ ≡
```
class Solution {
public:
  Solution(int n);
  void init(const v_bias::interval &t0, const iVector &y0);
  v_bias::interval t;
  pVector u;
```

```
    iVector y;
    iVector alpha, r, rQR;
    pMatrix S, A, Q;
  };
```

This code is used in chunk 184.

The constructor of the **Solution** class allocates memory for the vector and matrix members of this class.

179 ⟨create **Solution** 179⟩ ≡

```
  Solution::Solution(int n)
  {
    sizeV(u, n);
    sizeV(alpha, n);
    sizeV(r, n);
    sizeV(rQR, n);
    sizeV(y, n);
    sizeM(S, n);
    sizeM(A, n);
    sizeM(Q, n);
  }
```

This code is used in chunk 185.

Initially, when $j = 0$, we set

$$\begin{aligned} u_0 &= \mathrm{m}(\boldsymbol{y}_0), \\ \boldsymbol{\alpha} &= \boldsymbol{y}_0 - u_0, \\ \boldsymbol{r}_0 &= 0, \\ \boldsymbol{r}_{\mathrm{QR},0} &= 0, \\ S_0 &= I, \\ A_0 &= I, \quad \text{and} \\ Q_0 &= I. \end{aligned}$$

When computing the midpoint of *y0* $= \boldsymbol{y}_0$, we find a point vector u that is the rounded to the nearest true midpoint of *y0*.

180 ⟨initialize **Solution** 180⟩ ≡

```
  void Solution::init(const v_bias::interval &t0, const iVector &y0)
  {
    t = t0;
    y = y0;
    midpoint(u, y0);      /* u ≈ m(y_0) */
    subViVp(alpha, y0, u);      /* alpha ⊇ y_0 − u */
    setV(r, 0.0);
    setV(rQR, 0.0);
    setId(S);
    setId(Q);
    setId(A);
  }
```
This code is used in chunk 185.

16.2 A priori enclosure

On each integration step, we validate existence and uniqueness of a solution and compute a priori bounds $\widetilde{\boldsymbol{y}}_j$ such that

$$y(t; t_j, y_j) \in \widetilde{\boldsymbol{y}}_j$$

for all $t \in \widetilde{\boldsymbol{t}}_j := \boldsymbol{t}_j + [0,1]h_j$ and all $y_j \in \boldsymbol{y}_j$. Here, $h_j \in \mathbb{R}$ is a stepsize selected by VNODE-LP.

The interval $\widetilde{\boldsymbol{t}}_j$ and the interval vector $\widetilde{\boldsymbol{y}}_j$ are stored in

181 ⟨ a priori enclosure representation 181 ⟩ ≡
```
  class Apriori {
  public:
    Apriori(int n);
    void init(const v_bias::interval t0, const iVector y0);
    v_bias::interval t;
    iVector y;
  };
```
This code is used in chunk 184.

The constructor allocates a vector y

182 ⟨ create **Apriori** 182 ⟩ ≡
```
  Apriori::Apriori(int n) {
    sizeV(y, n);
  }
```
This code is used in chunk 185.

When initializing an **Apriori** object, we set t and y.

183 ⟨ initialize **Apriori** 183 ⟩ ≡
```
  void Apriori::init(const v_bias::interval t0, const iVector y0) {
    t = t0;
    y = y0;
  }
```
This code is used in chunk 185.

Files

184 ⟨ solution.h 184 ⟩ ≡
```
#ifndef SOLUTION_H
```

```
#define SOLUTION_H
  namespace vnodelp {
    using namespace v_bias;
    using namespace v_blas;
    ⟨ tight enclosure representation 178 ⟩
    ⟨ a priori enclosure representation 181 ⟩
  }
#endif
```

185 ⟨ solution.cc 185 ⟩ ≡

```
#include "vnodeinterval.h"
#include "basiclinalg.h"
#include "solution.h"
#include "intvfuncs.h"
  namespace vnodelp {
    ⟨ create Solution 179 ⟩
    ⟨ initialize Solution 180 ⟩
    ⟨ create Apriori 182 ⟩
    ⟨ initialize Apriori 183 ⟩
  }
```

Chapter 17

Taylor coefficient computation

We provide an abstract class, **AD_ODE**, for generating Taylor coefficients (TCs) for the solution to an ODE and an abstract class, **AD_VAR**, for generating TCs for the solution to the ODE's variational equation. Concrete implementations of these classes using FADBAD++ [6] are discussed in Chapter 22.

17.1 Taylor coefficients for an ODE solution

187 ⟨ AD_ODE 187 ⟩ ≡

```
class AD_ODE {
public:
  virtual void set(const v_bias::interval &t0, const iVector &y0,
      const v_bias::interval &h, int k) = 0;
  virtual void compTerms( ) = 0;
  virtual void sumTerms(iVector &sum, int m) = 0;
  virtual void getTerm(iVector &term, int i) const = 0;
  virtual v_bias::interval getStepsize( ) const = 0;
  virtual void eval(void *param) = 0;
  virtual ~AD_ODE( ) { }
};
```

This code is used in chunk 194.

Brief descriptions of the above functions follow. The kth TC of the solution to (1.1) at (t^*, y^*) is denoted by $f^{[i]}(t^*, y^*)$; cf. [23].

set initializes a TC computation by setting a point of expansion $\boldsymbol{t}_0$, $\boldsymbol{y}_0$, enclosure on the stepsize $\boldsymbol{h}$, and order k.

compTerms encloses

$$\boldsymbol{h}f^{[1]}(\boldsymbol{t}_0, \boldsymbol{y}_0),\ \boldsymbol{h}^2 f^{[2]}(\boldsymbol{t}_0, \boldsymbol{y_0}), \dots, \boldsymbol{h}^k f^{[k]}(\boldsymbol{t}_0, \boldsymbol{y_0}).$$

sumTerms encloses $\sum_{i=0}^{m} \boldsymbol{h}^i f^{[i]}(\boldsymbol{t}_0, \boldsymbol{y}_0)$, where $m \le k$. The result is stored in the parameter *sum*.

getTerm obtains the ith term, $\boldsymbol{h}^i f^{[i]}(t_0, \boldsymbol{y}_0)$, where $i \le k$.

getStepsize returns $\boldsymbol{h}$.

eval evaluates $f(t, y)$ and rebuilds the computational graph. *param* is a pointer to parameters that can be passed to f.

17.2 Taylor coefficients for the solution of the variational equation

189 ⟨AD_VAR 189⟩ ≡
```
class AD_VAR {
public:
  virtual void set(const v_bias::interval &t0, const iVector &y0,
      const v_bias::interval &h, int k) = 0;
  virtual void compTerms( ) = 0;
  virtual void sumTerms(iMatrix &sum, int m) = 0;
  virtual void getTerm(iMatrix &term, int i) const = 0;
  virtual void eval(void *param) = 0;
  virtual ~AD_VAR( ) { }
};
```
This code is used in chunk 195.

Brief descriptions of the above functions follow.

set initializes a TC computation by setting a point of expansion $\boldsymbol{t}_0$, $\boldsymbol{y}_0$, stepsize $\boldsymbol{h}$, and order k.

compTerms encloses

$$\boldsymbol{h}\frac{\partial f^{[1]}}{\partial y}(\boldsymbol{t}_0, \boldsymbol{y}_0),\ \boldsymbol{h}^2\frac{\partial f^{[2]}}{\partial y}(\boldsymbol{t}_0, \boldsymbol{y_0}), \ldots, \boldsymbol{h}^k\frac{\partial f^{[k]}}{\partial y}(\boldsymbol{t}_0, \boldsymbol{y_0}).$$

sumTerms encloses

$$\sum_{i=0}^{m} \boldsymbol{h}^i \frac{\partial f^{[i]}}{\partial y}(\boldsymbol{t}_0, \boldsymbol{y}_0),$$

where $m \le k$. The result is stored in the parameter *sum*.

getTerm obtains the ith term, $\boldsymbol{h}^i \frac{\partial f^{[i]}}{\partial y}(\boldsymbol{t}_0, \boldsymbol{y}_0)$, where $i \le k$.

eval evaluates $f(t, y)$ and rebuilds the computational graph. *param* is a pointer to parameters that can be passed to f.

17.3 AD class

It is convenient to encapsulate the above classes into

191 ⟨encapsulated AD 191⟩ ≡

```
  class AD {
  public:
    AD(int n, AD_ODE *a, AD_VAR *av);
    void eval(void *p);
    virtual int getMaxOrder( ) const = 0;
       /* maximum order that is allowed */
  public:
    int size;
    AD_ODE *tayl_coeff_ode;
    AD_VAR *tayl_coeff_var;
  };
```

This code is used in chunk 196.

The constructor of **AD** sets the size of the problem and pointers to objects of **AD_ODE** and **AD_VAR**.

192 ⟨implementation of encapsulated AD 192⟩ ≡

```
  inline AD::AD(int n, AD_ODE *a, AD_VAR *av)
  : size(n), tayl_coeff_ode(a), tayl_coeff_var(av) { }
```

See also chunk 193.

This code is used in chunk 196.

The *eval* function calls the corresponding *eval* functions of **AD_ODE** and **AD_VAR**

193 ⟨implementation of encapsulated AD 192⟩ +≡

```
  inline void AD::eval(void *p)
  {
    tayl_coeff_ode→eval(p);
    tayl_coeff_var→eval(p);
  }
```

Files

The above classes are stored in `ad_ode.h`, `ad_var.h`, and `allad.h`, respectively.

194 ⟨`ad_ode.h` 194⟩ ≡

```
#ifndef AD_ODE_H
#define AD_ODE_H
#include "vnodeinterval.h"
#include "vector_matrix.h"
  namespace vnodelp {
    using namespace v_bias;
    using namespace v_blas;
    ⟨ AD_ODE 187 ⟩
  }
#endif
```

195 ⟨ ad_var.h 195 ⟩ ≡

```
#ifndef AD_VAR_H
#define AD_VAR_H
#include "vnodeinterval.h"
#include "vector_matrix.h"
  namespace vnodelp {
    using namespace v_bias;
    using namespace v_blas;
    ⟨ AD_VAR 189 ⟩
  }
#endif
```

196 ⟨ allad.h 196 ⟩ ≡

```
#ifndef ALLAD_H
#define ALLAD_H
#include "ad_ode.h"
#include "ad_var.h"
  namespace vnodelp {
    ⟨ encapsulated AD 191 ⟩
    ⟨ implementation of encapsulated AD 192 ⟩
  }
#endif
```

Chapter 18

Control data

We store various data needed to control an integration in a **Control** class. An integration in VNODE-LP is carried out by the *integrate* function of **VNODE**; see Chapter 21.

18.1 Indicator type

First, we introduce an enumerated **Ind** data type, where a variable of this type can take the following values:

value	description
first_entry	indicates a first entry into *integrate*
success	*integrate* has reached $\boldsymbol{t}_{\text{end}}$ successfully
failure	an error has occurred in *integrate*

198 ⟨indicator type 198⟩ ≡
```
typedef enum {
  first_entry, success, failure
} Ind;
```
This code is used in chunk 201.

18.2 Interrupt type

Similarly, we have an **Interrupt** type, where a variable can take values as described below.

value	description
no	*integrate* tries to reach $\boldsymbol{t}_{\text{end}}$
before_accept	*integrate* takes a step and returns before accepting this step

199 ⟨interrupt type 199⟩ ≡
```
typedef enum {
   no, before_accept
} Interrupt;
```
This code is used in chunk 201.

18.3 Control data

In the **Control** class, we store the following data:

name	default value	description
ind	*first_entry*	indicator variable
interrupt	*no*	indicates if interrupts are requested
order	20	order of the method
atol	10^{-12}	absolute error tolerance
rtol	10^{-12}	relative error tolerance
hmin	0	magnitude of the minimum stepsize allowed. If a positive value is set by the user, this value for *hmin* will be used in an integration. Otherwise, the solver computes a minimum stepsize as discussed in Subsection 21.2.3.

200 ⟨control class 200⟩ ≡
```
class Control {
public:
   Ind ind;
   Interrupt interrupt;
   unsigned int order;
   double atol, rtol;
   double hmin;
   Control( ) :
         ind(first_entry),
          interrupt(no),
          order(20),
          atol(1 · 10^−12), rtol(1 · 10^−12),
          hmin(0) { }
};
```
This code is used in chunk 201.

Files

We store **Ind**, **Interrupt**, and **Control** in

201 ⟨ `control.h` 201 ⟩ ≡

```
#ifndef CONTROL_H
#define CONTROL_H
  namespace vnodelp {
    ⟨ indicator type 198 ⟩
    ⟨ interrupt type 199 ⟩
    ⟨ control class 200 ⟩
  }
#endif
```

Chapter 19

Computing a priori bounds

We have implemented the HOE method [27] to compute a priori bounds on the solution of an ODE problem. In Section 19.1 we summarize the relevant theory. The **HOE** class is given in Section 19.2. The implementation of the HOE method is in Section 19.3. The rest of the functions of this class is in Section 19.4.

19.1 Theory background

The HOE method is based on the following two results; cf. [9, 27].

1. If h_j and $\widetilde{\boldsymbol{y}}_j$ are such that $\boldsymbol{y}_j \subseteq \operatorname{int}(\widetilde{\boldsymbol{y}}_j)$ and

   $$\sum_{i=0}^{k-1}(t-t_j)^i f^{[i]}(t_j, y_j) + (t-t_j)^k f^{[k]}(t_j + [0,1]h_j, \widetilde{\boldsymbol{y}}_j) \subseteq \widetilde{\boldsymbol{y}}_j$$

   for all $t \in t_j + [0,1]h_j$ and all $y_j \in \boldsymbol{y}_j$, then

   $$y' = f(t,y), \quad y(t_j) = y_j \in \boldsymbol{y}_j \tag{19.1}$$

   has a unique solution

   $$y(t; t_j, y_j) \in \widetilde{\boldsymbol{y}}_j$$

   for all $t \in t_j + [0,1]h_j$ and all $y_j \in \boldsymbol{y}_j$.

2. Let $h_{j,0} \neq 0$ and let $\boldsymbol{p}_j$ be an interval vector enclosing the set

   $$\mathcal{P}_j = \left\{ \sum_{i=0}^{k-1}(t-t_j)^i f^{[i]}(t_j, y_j) \mid t \in t_j + [0,1]h_{j,0},\ y_j \in \boldsymbol{y}_j \right\}. \tag{19.2}$$

   That is, $\mathcal{P}_j \subseteq \boldsymbol{p}_j$.

Let $\boldsymbol{u}_j$ be such that

$$\begin{aligned}&\widetilde{\boldsymbol{y}}_j = \boldsymbol{p}_j + \boldsymbol{u}_j, \quad \text{and}\\ &\boldsymbol{y}_j \subseteq \text{int}(\widetilde{\boldsymbol{y}}_j).\end{aligned} \tag{19.3}$$

If $h_{j,1} \neq 0$ is such that

$$[0,1]h_{j,1}^k f^{[k]}(t_j + [0,1]h_{j,0}, \widetilde{\boldsymbol{y}}_j) \subseteq \boldsymbol{u}_j, \tag{19.4}$$

and

$$h_j = \text{sign}(h_{j,0}) \cdot \min\{|h_{j,0}|, |h_{j,1}|\}, \tag{19.5}$$

then there exists a unique solution $y(t; t_j, y_j)$ to (19.1) for all $t \in t_j + [0,1]h_j$ and all $y_j \in \boldsymbol{y}_j$. Moreover,

$$y(t; t_j, y_j) \in \widetilde{\boldsymbol{y}}_j \quad \text{for all} \quad t \in t_j + [0,1]h_j \quad \text{and all} \quad y_j \in \boldsymbol{y}_j.$$

19.2 The HOE class

The class implementing the HOE method is

```
204 ⟨class HOE 204⟩ ≡
  class HOE {
  public:
    HOE(int n);

    void compAprioriEnclosure(const interval &t0, const iVector &y0,
        bool &info);
    void acceptSolution();

    ⟨set functions HOE 221⟩
    ⟨get functions HOE 222⟩
    ~HOE();

  private:
    Apriori *apriori_trial, *apriori;
    Control *control;
    AD_ODE *tayl_coeff;

    double h, h_next, h_trial, t_trial;
    int order_trial;

    iVector term, p, u, v;
    const interval one;      /* one = [0,1] */

    interval comp_beta(const iVector &v, const iVector &u, int k);
  };
```

This code is used in chunk 226.

19.3 Implementation of the HOE method

19.3.1 Computing $\boldsymbol{p}_j$

We have to enclose $\mathcal{P}_j$ in (19.2). On the first step, if t_0 is not a representable machine number, an interval $\boldsymbol{t}_0$ containing t_0 would be given. We assume that, in general, $t_j \in \boldsymbol{t}_j$. Then

$$t \in \boldsymbol{t}_j + [0,1]h_{j,0} \quad \text{and} \quad t - t_j \in \boldsymbol{t}_j - \boldsymbol{t}_j + [0,1]h_{j,0}.$$

The width of an interval $\boldsymbol{a}$ is $\mathrm{w}(\boldsymbol{a}) = \overline{\boldsymbol{a}} - \underline{\boldsymbol{a}}$. (Width of an interval vector is defined componentwise.) Denote $\boldsymbol{t}_j^* = (\boldsymbol{t_j} - \boldsymbol{t_j})/h_{j,0} + [0,1]$. Then

$$\boldsymbol{t}_j^* = \begin{cases} (\boldsymbol{t_j} - \boldsymbol{t_j})/h_{j,0} + [0,1] & \text{if } \mathrm{w}(\boldsymbol{t}_j) > 0, \\ [0,1] & \text{if } \mathrm{w}(\boldsymbol{t}_j) = 0. \end{cases}$$

Hence $t - t_j \in \boldsymbol{t}_j^*\, h_{j,0}$, and we can bound $\mathcal{P}_j$ as

$$\mathcal{P}_j \subseteq \sum_{i=0}^{k-1} \boldsymbol{t}_j^{*i}\, h_{j,0}^i f^{[i]}(\boldsymbol{t}_j, \boldsymbol{y}_j).$$

We enclose first $h_{j,0}^i f^{[i]}(\boldsymbol{t}_j, \boldsymbol{y}_j)$ for $i = 1, \ldots, k-1$. Then, we use an interval form of Horner's rule to compute

$$\boldsymbol{p}_j := \boldsymbol{y}_j + \boldsymbol{t}_j^*\Big(h_{j,0} f^{[1]}(\boldsymbol{t}_j, \boldsymbol{y}_j) + \cdots + \boldsymbol{t}_j^*\big(h_{j,0}^{k-2} f^{[k-2]}(\boldsymbol{t}_j, \boldsymbol{y}_j) + \boldsymbol{t}_j^* h_{j,0}^{k-1} f^{[k-1]}(\boldsymbol{t}_j, \boldsymbol{y}_j)\big)\cdots\Big).$$

206 $\langle$compute $\boldsymbol{p}_j$ 206$\rangle \equiv$ /*

$$\begin{aligned} \mathit{t\_0} &= \boldsymbol{t}_j \\ \mathit{y0} &\supseteq \boldsymbol{y}_j \\ \mathit{h\_trial} &= h_{j,0} \\ \mathit{order\_trial} &= k \end{aligned}$$

$$\begin{aligned} \mathit{tayl\_coeff} &\ \text{contains enclosures on } h_{j,0}^i f^{[i]}(\boldsymbol{t}_j, \boldsymbol{y}_j) \text{ for } i = 0, \ldots, k-1 \\ \mathit{t\_enc} &\supseteq \boldsymbol{t}_j^* = (\boldsymbol{t_j} - \boldsymbol{t_j})/h_{j,0} + [0,1] \\ p &\supseteq \boldsymbol{p}_j \end{aligned}$$

```
      */
  tayl_coeff→set(t0, y0, h_trial, order_trial − 1);
  tayl_coeff→compTerms( );
  interval t_enc = (t0 − t0)/h_trial + one;
  tayl_coeff→getTerm(p, order_trial − 1);
  for (int i = order_trial − 2; i ≥ 0; i−−) {
    scaleV(p, t_enc);
    tayl_coeff→getTerm(term, i);
    addViVi(p, term);
  }
```
This code is used in chunk 217.

19.3.2 Computing $\boldsymbol{u}_j$ and $\widetilde{\boldsymbol{y}}_j$

If $\boldsymbol{u}_j$ is symmetric with no component equal to $[0,0]$ then (19.3) holds. Denote

$$\mathrm{tol}_j = \mathrm{rtol} \cdot \|\boldsymbol{y}_j\| + \mathrm{atol}.$$

Throughout this manuscript, we shall assume the infinity norm, unless stated otherwise. For an interval vector $\boldsymbol{a}$,

$$\|\boldsymbol{a}\| = \max_i \{|\underline{\boldsymbol{a}}_i|, |\overline{\boldsymbol{a}}_i|\}.$$

Let $\boldsymbol{u}_j$ be the n-vector with each component $h_{j,0}[-\mathrm{tol}_j/2, \mathrm{tol}_j/2]$. Then, we form

$$\widetilde{\boldsymbol{y}}_j = \boldsymbol{p}_j + \boldsymbol{u}_j.$$

207 $\langle$compute $\boldsymbol{u}_j$ and $\widetilde{\boldsymbol{y}}_j$ 207$\rangle \equiv$ /*

$$\begin{array}{rl} y0 & \supseteq \boldsymbol{y}_j \\ control\text{→}atol & = \mathrm{atol} \\ control\text{→}rtol & = \mathrm{rtol} \\ h\_trial & = h_{j,0} \\ p & \supseteq \boldsymbol{p}_j \\ \hline tol & \geq \mathrm{rtol} \cdot \|\boldsymbol{y}_j\| + \mathrm{atol} \\ u & \supseteq \boldsymbol{u}_j \\ apriori\_trial & \supseteq \widetilde{\boldsymbol{y}}_j = \boldsymbol{p}_j + \boldsymbol{u}_j. \end{array}$$

```
  */
round_up();
double tol = inf_normV(y0) * control→rtol + control→atol;
setV(u, h_trial * interval(−tol/2, tol/2));
addViVi(apriori_trial→y, p, u);
assert(interior(y0, apriori_trial→y));
```

This code is used in chunk 217.

Motivation for the choice of $\boldsymbol{u}_j$

Denote

$$\boldsymbol{z}_j = f^{[k]}(t_j + [0,1]h_{j,0}, \widetilde{\boldsymbol{y}}_j).$$

We can consider $|h_{j,1}|^k \|\mathrm{w}(\boldsymbol{z}_j)\|$ as an estimate of the *local excess* at $\boldsymbol{t}_{j+1}$ that we introduce on the step from $\boldsymbol{t}_j$ to $\boldsymbol{t}_{j+1}$.

The magnitude of an interval $\boldsymbol{a}$ is $|\boldsymbol{a}| = \max\{|\underline{\boldsymbol{a}}|, |\overline{\boldsymbol{a}}|\}$. Magnitude of an interval vector is defined componentwise. From (19.4), (19.5), and the choice for $\boldsymbol{u}_j$,

$$\begin{aligned} |h_{j,1}|^k \, \mathrm{w}\big(\boldsymbol{z}_j\big) \le \mathrm{w}\big([0,1]h_{j,1}^k \, \boldsymbol{z}_j\big) &= |h_{j,1}|^k \, |\boldsymbol{z}_j| \\ &\le \mathrm{w}(\boldsymbol{u}_j) = h_{j,0} \, (\mathrm{tol}_j, \mathrm{tol}_j, \cdots, \mathrm{tol}_j)^T. \end{aligned}$$

Hence,

$$|h_{j,1}|^k \, \|\mathrm{w}(\boldsymbol{z}_j)\| \le h_{j,0} \cdot \mathrm{tol}_j.$$

If $h_{j,1} = h_{j,0}$, we have a local excess per unit step (LEPUS) control. If $h_{j,1} < h_{j,0}$, then usually $h_{j,1}$ is not much smaller than $h_{j,0}$, and we have almost LEPUS.

19.3.3 Computing a stepsize

We enclose first

$$\boldsymbol{v}_j = h_{j,0}^k f^{[k]}(\boldsymbol{t}_j + [0,1]h_{j,0}, \widetilde{\boldsymbol{y}}_j).$$

210 ⟨compute stepsize 210⟩ ≡ /*

$$\begin{array}{rl} \mathit{t0} &= \boldsymbol{t}_j \\ \mathit{apriori\_trial} &\supseteq \widetilde{\boldsymbol{y}}_j \\ \mathit{h\_trial} &= h_{j,0} \\ \mathit{order\_trial} &= k \\ \hline \mathit{tayl\_coeff} & \text{contains enclosures on } h_{j,0}^i f^{[i]}(\boldsymbol{t}_j + [0,1]h_{j,0}, \widetilde{\boldsymbol{y}}_j) \text{ for } i = 0, \dots, k \\ v &\supseteq h_{j,0}^k f^{[k]}(\boldsymbol{t}_j + [0,1]h_{j,0}, \widetilde{\boldsymbol{y}}_j) \end{array}$$

*/

```
tayl_coeff→set(t0 + one * h_trial, apriori_trial→y, h_trial, order_trial);
tayl_coeff→compTerms( );
tayl_coeff→getTerm(v, order_trial);
```

See also chunks 212 and 214.
This code is used in chunk 217.

Now we consider two cases: $\underline{\boldsymbol{t}}_j = \overline{\boldsymbol{t}}_j$ and $\underline{\boldsymbol{t}}_j < \overline{\boldsymbol{t}}_j$.

The case $\underline{\boldsymbol{t}}_j = \overline{\boldsymbol{t}}_j$

Let $\gamma > 0$ be the largest number such that

$$\begin{aligned} [0,1]h_{j,1}^k f^{[k]}(\boldsymbol{t}_j + [0,1]h_{j,0}, \widetilde{\boldsymbol{y}}_j) &= [0,1](\gamma h_{j,0}^k) f^{[k]}(\boldsymbol{t}_j + [0,1]h_{j,0}, \widetilde{\boldsymbol{y}}_j) \\ &= [0,\gamma]\boldsymbol{v}_j \\ &\subseteq \boldsymbol{u}_j. \end{aligned}$$

Such a γ exists since $\boldsymbol{u}_j$ is symmetric. Then, we write $\beta = \gamma^{1/k}$ and set

$$h_j = \begin{cases} \beta h_{j,0} & \text{if } \beta < 1 \\ h_{j,0} & \text{if } \beta \geq 1. \end{cases}$$

In practice, we compute $\boldsymbol{\beta} \ni \beta > 0$, with $\underline{\boldsymbol{\beta}} \geq 0$ and find

$$h_j = \begin{cases} \downarrow(\underline{\boldsymbol{\beta}}\, h_{j,0}) & \text{if } \underline{\boldsymbol{\beta}} < 1,\ h_{j,0} > 0 \\ \uparrow(\underline{\boldsymbol{\beta}}\, h_{j,0}) & \text{if } \underline{\boldsymbol{\beta}} < 1,\ h_{j,0} < 0 \\ h_{j,0} & \text{if } \underline{\boldsymbol{\beta}} \geq 1. \end{cases} \tag{19.6}$$

Here $\uparrow$ denotes rounding up, and $\downarrow$ denotes rounding down.

212 $\langle$compute stepsize 210$\rangle$ $+\equiv$ /*

$$\frac{\begin{array}{l} u \supseteq \boldsymbol{u}_j \\ v \supseteq \boldsymbol{v}_j = h_{j,0}^k f^{[k]}(\boldsymbol{t}_j + [0,1]h_{j,0}, \widetilde{\boldsymbol{y}}_j) \end{array}}{h = h_j}$$

```
    */
interval beta = comp_beta(v, u, order_trial);
assert(inf(beta) ≥ 0);
if (inf(beta) < 1) {
   if (h_trial > 0) round_down();
   else round_up();
   h = inf(beta) * h_trial;
}
else h = h_trial;
```

The case $\underline{\boldsymbol{t}}_j < \overline{\boldsymbol{t}}_j$

Similar to (19.4), we try to find (the largest) $|h_{j,1}| \leq |h_{j,0}|$ such that

$$\begin{aligned}(t - t_j)^k f^{[k]}(t_j + [0,1]h_{j,0} &\subseteq (\boldsymbol{t}_j - \boldsymbol{t}_j + [0,1]h_{j,1})^k f^{[k]}(\boldsymbol{t}_j + [0,1]h_{j,0}, \widetilde{\boldsymbol{y}}_j) \\ &\subseteq \boldsymbol{u}_j. \end{aligned} \tag{19.7}$$

Normally $|h_{j,0}| \gg |\boldsymbol{t}_j - \boldsymbol{t}_j|$, and therefore,

$$(\boldsymbol{t}_j - \boldsymbol{t}_j + [0,1]h_{j,0})^k \approx [0,1]h_{j,0}^k.$$

Hence, we may consider setting h_j as in (19.6). However, (19.7) may not hold. To find h_j such that it is likely to hold, we set

$$h_j = 0.9h_{j,0}$$

and check if

$$\begin{aligned}\big((\boldsymbol{t}_j-\boldsymbol{t}_j)+[0,1]h_j\big)^k f^{[k]}(\boldsymbol{t}_j+[0,1]h_{j,0},\widetilde{\boldsymbol{y}}_j)\\ =\left(\frac{\boldsymbol{t}_j-\boldsymbol{t}_j+[0,1]h_j}{h_{j,0}}\right)^k h_{j,0}^k f^{[k]}(\boldsymbol{t}_j+[0,1]h_{j,0},\widetilde{\boldsymbol{y}}_j)\\ =\left(\frac{\boldsymbol{t}_j-\boldsymbol{t}_j+[0,1]h_j}{h_{j,0}}\right)^k \boldsymbol{v}_j\subseteq\boldsymbol{u}_j.\end{aligned}$$

If the above inclusion test fails, we reduce h_j and repeat as shown below.

214 ⟨ compute stepsize 210 ⟩ +≡

```
if (inf(t0) < sup(t0)) {     /*
```

$$\begin{aligned}v &\supseteq \boldsymbol{v}_j\\ h &= h_j\\ h\_trial &= h_{j,0}\\ t0 &= \boldsymbol{t}_j\\ control{\to}hmin &= h_{\min}\\ \hline tt &\supseteq \boldsymbol{t}_j-\boldsymbol{t}_j\\ t\_enc &\supseteq \frac{\boldsymbol{t}_j-\boldsymbol{t}_j+[0,1]h_j}{h_{j,0}}\\ v &\supseteq \left(\frac{\boldsymbol{t}_j-\boldsymbol{t}_j+[0,1]h_j}{h_{j,0}}\right)^k h_{j,0}^k f^{[k]}(\boldsymbol{t}_j+[0,1]h_{j,0},\widetilde{\boldsymbol{y}}_j)\\ \hline h &= h_j\end{aligned}$$

```
    */
  interval tt = t0 − t0;
  while (fabs(h) > control→hmin) {
    t_enc = (tt + one ∗ h)/h_trial;
    scaleV(v, pow(t_enc, order_trial));
    if (subseteq(v, u)) break;
    h = 0.9 ∗ h;
  }
}
```

19.3.4 Forming the time interval

Once h_j is found, we need to determine the next representable integration point and the machine interval over which the a priori bounds hold.

1. If $h_j > 0$, we compute

$$t_{j+1} = \downarrow(\underline{\boldsymbol{t}}_j + h_j) \quad \text{and}$$
$$\boldsymbol{T}_j = [\underline{\boldsymbol{t}}_j, t_{j+1}].$$

2. If $h_j < 0$, we compute

$$\begin{aligned} t_{j+1} &= \uparrow(\overline{\boldsymbol{t}}_j + h_j) \quad \text{and} \\ \boldsymbol{T}_j &= [t_{j+1}, \overline{\boldsymbol{t}}_j]. \end{aligned}$$

Proposition 19.1.

(i) $\boldsymbol{T}_j \subseteq \boldsymbol{t}_j + [0,1]h_{j,0}$

(ii) *If* $|h_j| \geq \mathrm{w}(\boldsymbol{t}_j)$, *then* $\boldsymbol{t}_j \subseteq \boldsymbol{T}_j$

Proof. $h_{j,0} > 0$. Then $0 < h_j \leq h_{j,0}$ and

$$\begin{aligned} \boldsymbol{T}_j &= [\underline{\boldsymbol{t}}_j, t_{j+1}] \subseteq [\underline{\boldsymbol{t}}_j, \underline{\boldsymbol{t}}_j + h_j] \subseteq [\underline{\boldsymbol{t}}_j, \underline{\boldsymbol{t}}_j + h_{j,0}] = \underline{\boldsymbol{t}}_j + [0,1]h_{j,0} \\ &\subseteq \boldsymbol{t}_j + [0,1]h_{j,0}. \end{aligned}$$

If $h_j \geq \mathrm{w}(\boldsymbol{t}_j) = \overline{\boldsymbol{t}}_j - \underline{\boldsymbol{t}}_j$, then $\overline{\boldsymbol{t}}_j \leq \downarrow(\underline{\boldsymbol{t}}_j + h_j) \leq \underline{\boldsymbol{t}}_j + h_j$, and

$$[\underline{\boldsymbol{t}}_j, \overline{\boldsymbol{t}}_j] \subseteq [\underline{\boldsymbol{t}}_j, \downarrow(\underline{\boldsymbol{t}}_j + h_j)] = \boldsymbol{T}_j.$$

$h_{j,0} < 0$. Then $0 > h_j \geq h_{j,0}$ and

$$\begin{aligned} \boldsymbol{T}_j &= [t_{j+1}, \overline{\boldsymbol{t}}_j] \subseteq [\overline{\boldsymbol{t}}_j + h_j, \overline{\boldsymbol{t}}_j] \subseteq [\overline{\boldsymbol{t}}_j + h_{j,0}, \overline{\boldsymbol{t}}_j] = \overline{\boldsymbol{t}}_j + [0,1]h_{j,0} \\ &\subseteq \boldsymbol{t}_j + [0,1]h_{j,0}. \end{aligned}$$

If $-h_j \geq \overline{\boldsymbol{t}}_j - \underline{\boldsymbol{t}}_j$, then $\underline{\boldsymbol{t}}_j \geq \uparrow(\overline{\boldsymbol{t}}_j + h_j) \geq \overline{\boldsymbol{t}}_j + h_j$, and

$$\boldsymbol{t}_j = [\underline{\boldsymbol{t}}_j, \overline{\boldsymbol{t}}_j] \subseteq [\uparrow(\underline{\boldsymbol{t}}_j + h_j), \overline{\boldsymbol{t}}_j] = \boldsymbol{T}_j.$$

□

215 ⟨form time interval 215⟩ ≡ /*

$$\begin{aligned} h &= h_j \\ t0 &= \boldsymbol{t}_j \end{aligned}$$

$$\begin{aligned} t\_trial &= t_{j+1} \\ apriori\_trial \to t &= \boldsymbol{T}_j \end{aligned}$$

```
    */
  double td;
  if (h > 0) {
    td = inf(t0);
    round_down();
    t_trial = td + h;
    $apriori\_trial\text{→}t = \textbf{interval}(td, t\_trial);$
  }
  else {
    $td = sup(t0);$
    $round\_up();$
    $t\_trial = td + h;$
    $apriori\_trial\text{→}t = \textbf{interval}(t\_trial, td);$
  }
  $assert(subseteq(t0, apriori\_trial\text{→}t));$
```

This code is used in chunk 217.

19.3.5 Selecting a trial stepsize for the next step

We can select for the next step $h_{j+1,0} = \beta h_{j,0}$. This is reasonable since, if $|h_{j+1,0}| > |h_{j,0}|$, we have computed an a priori enclosure with $h_{j,0}$, but we could have possibly done this with a stepsize between $h_{j,0}$ and $h_{j+1,0}$. That is, we assume that we might be successful on the next step with $|h_{j+1,0}| = \beta\,|h_{j,0}| > |h_{j,0}|$. Similar considerations apply when $|h_{j+1,0}| < |h_{j,0}|$. Hence, we select

$$h_{j+1,0} = \beta h_{j,0}$$

for the next step.

216 ⟨ select stepsize 216 ⟩ ≡ /*

$$\frac{\begin{array}{c} h\_trial = h_{j,0} \\ beta \supseteq \boldsymbol{\beta} \end{array}}{h\_next = h_{j+1,0}}$$

```
    */
  $h\_next = inf(beta) * h\_trial;$
```

This code is used in chunk 217.

19.3.6 Computing a priori bounds

First, we check if the stepsize is not very small. Then, we compute $\boldsymbol{p}_j$, $\widetilde{\boldsymbol{y}}_j$, and h_j. If $|h_j| \le h_{\min}$, we cannot validate existence and uniqueness and return *false*. Otherwise, we form $\boldsymbol{T}_j$ and select $h_{j+1,0}$ for the next step.

217 ⟨ validate existence and uniqueness 217 ⟩ ≡

```
  void HOE::compAprioriEnclosure(const interval &t0, const iVector &y0,
          bool &info)
```

```
{    /*
```

$$
\begin{array}{rl}
\mathit{t0} &= \boldsymbol{t}_j \\
\mathit{y0} &\supseteq \boldsymbol{y}_j \\
\mathit{h\_trial} &= h_{j,0} \\
\mathit{order\_trial} &= k \\
\hline
\mathit{apriori\_trial}\to t &\supseteq \boldsymbol{T}_j \\
\mathit{apriori\_trial}\to y &\supseteq \widetilde{\boldsymbol{y}}_j \\
\mathit{tayl\_coeff} & \text{contains enclosures on } h_{j,0}^i f^{[i]}(\boldsymbol{t}_j + [0,1]h_{j,0}, \widetilde{\boldsymbol{y}}_j) \text{ for } i = 0, \ldots, k \\
\mathit{h\_trial} &= h_{j+1,0}
\end{array}
$$

```
      */
   if (fabs(h_trial) ≤ control→hmin) {
      info = false;
      return;
   }
   ⟨compute $\boldsymbol{p}_j$ 206⟩;
   ⟨compute $\boldsymbol{u}_j$ and $\widetilde{\boldsymbol{y}}_j$ 207⟩;
   ⟨compute stepsize 210⟩;
   if (fabs(h) ≤ control→hmin) {
      info = false;
      return;
   }
   ⟨form time interval 215⟩;
   ⟨select stepsize 216⟩;
   info = true;
}
```

This code is used in chunk 227.

19.4 Other functions

19.4.1 Constructor and destructor

219 ⟨constructor-destructor HOE 219⟩ ≡

```
HOE::HOE(int n)
: one(interval(0, 1)) {
   sizeV(term, n);
   sizeV(p, n);
   sizeV(u, n);
   sizeV(v, n);
   apriori_trial = new Apriori(n);
   apriori = new Apriori(n);
   assert(apriori ∧ apriori_trial);
   tayl_coeff = 0;
```

```
    control = 0;
  }
  HOE::~HOE()
  {
    delete apriori;
    delete apriori_trial;
  }
```

This code is used in chunk 227.

19.4.2 Accept a solution

To accept a trial enclosure, we store it in the $*apriori$ object.

220 ⟨ accept solution (HOE) 220 ⟩ ≡

```
  void HOE::acceptSolution() {     /*
```

$$\begin{array}{rl} apriori\_trial & \text{contains } \widetilde{\boldsymbol{y}}_j \text{ and } \boldsymbol{T}_j \\ apriori & \text{contains } \widetilde{\boldsymbol{y}}_{j-1} \text{ and } \boldsymbol{T}_{j-1} \\ h\_trial & = h_{j,0} \\ h\_next & = h_{j+1,0} \\ \hline apriori & \text{contains } \widetilde{\boldsymbol{y}}_j \text{ and } \boldsymbol{T}_j \\ h\_trial & = h_{j+1,0} \end{array}$$

```
        */
    apriori→t = apriori_trial→t;
    assignV(apriori→y, apriori_trial→y);
    h_trial = h_next;
  }
```

This code is used in chunk 227.

19.4.3 Set functions

221 ⟨ set functions HOE 221 ⟩ ≡

```
  void set(Control *ctrl, AD *ad)
  {
    control = ctrl;
    tayl_coeff = ad→tayl_coeff_ode;
  }
  void init(const interval &t0, const iVector &y0) {
    apriori→init(t0, y0);
  }
  void setTrialStepsize(double h0) {
    h_trial = h0;
  }
```

```
void setTrialOrder(int order0) {
  order_trial = order0;
}
```

This code is used in chunk 204.

19.4.4 Get functions

222 ⟨get functions HOE 222⟩ ≡

```
double getStepsize() const {
  return h;
}
double getTrialStepsize() const {
  return h_trial;
}
const interval &getT() const {
  return apriori→t;
}
interval getTrialT() const {
  return apriori_trial→t;
}
const iVector &getApriori() const {
  return apriori→y;
}
const iVector &getTrialApriori() const {
  return apriori_trial→y;
}
```

See also chunk 223.

This code is used in chunk 204.

Obtain error term

We obtain $h_{j,0}^i f^{[i]}(\boldsymbol{t}_j + [0,1]h_{j,0}, \widetilde{\boldsymbol{y}}_j)$ for given $i = 0, 1, \ldots, k$ by

223 ⟨get functions HOE 222⟩ +≡ /*

$$e \supseteq h_{j,0}^i f^{[i]}(\boldsymbol{t}_j + [0,1]h_{j,0}, \widetilde{\boldsymbol{y}}_j)$$

```
    */
void getErrorTerm(iVector &e, int i) const {
  tayl_coeff→getTerm(e, i);
}
```

19.4.5 Enclosing β

The function *comp_beta* returns an enclosure of β.

224 ⟨compute β 224⟩ ≡ /*

$$gamma = \gamma,\ \text{the largest representable } \gamma > 0 \text{ such that } \gamma \boldsymbol{v}_j \subseteq \boldsymbol{u}_j$$
$$beta \supseteq \beta = \gamma^{1/k}$$

```
      */
  interval HOE::comp_beta(const iVector &v, const iVector &u, int k)
  {
    double gamma = v_blas::compH(v, u);
    interval i_gamma = gamma;
    interval i_pw = interval(1.0)/interval(double(k));
    interval beta = pow(i_gamma, i_pw);
    return beta;
  }
```

This code is used in chunk 227.

Files

226 ⟨hoe.h 226⟩ ≡

```
#ifndef HOE_H
#define HOE_H
  namespace vnodelp {
    ⟨class HOE 204⟩
  }
#endif
```

227 ⟨hoe.cc 227⟩ ≡

```
#include <cmath>
#include <cassert>
#include <algorithm>
#include "vnodeinterval.h"
#include "vnoderound.h"
#include "basiclinalg.h"
#include "intvfuncs.h"
#include "control.h"
#include "solution.h"
#include "allad.h"
#include "hoe.h"
  using namespace v_blas;
  namespace vnodelp {
    ⟨constructor-destructor HOE 219⟩
    ⟨validate existence and uniqueness 217⟩
    ⟨accept solution (HOE) 220⟩
    ⟨compute β 224⟩
  }
```

Chapter 20

Computing tight bounds on the solution

To compute tight bounds on the solution, we employ the interval Hermite-Obreschkoff (IHO) method developed in [23]. It consists of two phases: a predictor and a corrector. We present the relevant theory first and then describe its implementation.

20.1 Theory background

20.1.1 Predictor

For $y_j \in \boldsymbol{y}_j \subseteq \widetilde{\boldsymbol{y}}_j$, we have

$$y(t_{j+1}; t_j, y_j) \in y_j + \sum_{i=1}^{q} h_j^i f^{[i]}(t_j, y_j) + h_j^{q+1} f^{[q+1]}(\boldsymbol{T}_j, \widetilde{\boldsymbol{y}}_j), \tag{20.1}$$

where $h_j = t_{j+1} - t_j$ and t_j, $t_{j+1} \in \boldsymbol{T}_j$.

Let $J(f^{[i]}; \boldsymbol{y}_j)$ be the Jacobian of $f^{[i]}$ evaluated at $\boldsymbol{y}_j$ and denote

$$\boldsymbol{U}_{j+1} = I + \sum_{i=1}^{q} h_j^i \, J(f^{[i]}; \boldsymbol{y}_j).$$

These Jacobians are computed by generating TCs for the solution of the associated variational equation

$$Y' = \frac{\partial f}{\partial y} Y, \quad Y(t_j) = I,$$

where I is the $n \times n$ identity matrix.

We assume that at t_j the solution $y(t_j; t_0, y_0)$ is contained in

$$\begin{aligned}
&\boldsymbol{y}_j \\
&\{u_j + S_j\alpha + A_j r \mid \alpha \in \boldsymbol{\alpha},\, r \in \boldsymbol{r}_j\}, \quad \text{and} \\
&\{u_j + S_j\alpha + Q_j r \mid \alpha \in \boldsymbol{\alpha},\, r \in \boldsymbol{r}_{\text{QR},j}\},
\end{aligned}$$

where $u_j \in \boldsymbol{y}_j$.

If we apply the mean-value theorem to the $f^{[i]}$ in (20.1), we obtain that for any

$$\begin{aligned} y_j \in \{&u_j + S_j\alpha + A_j r \mid \alpha \in \boldsymbol{\alpha},\, r \in \boldsymbol{r}_j\} \\ &\cap \{u_j + S_j\alpha + Q_j r \mid \alpha \in \boldsymbol{\alpha},\, r \in \boldsymbol{r}_{\mathrm{QR},j}\} \\ &\cap \boldsymbol{y}_j, \end{aligned}$$

$$\begin{aligned} y(t_{j+1}; t_j, y_j) \in u_j &+ \sum_{i=1}^{q} h_j^i f^{[i]}(t_j, u_j) + h_j^{q+1} f^{[q+1]}(\boldsymbol{T}_j, \widetilde{\boldsymbol{y}}_j) \\ &+ (\boldsymbol{U}_{j+1}S_j)\boldsymbol{\alpha} + \big\{(\boldsymbol{U}_{j+1}A_j)\boldsymbol{r}_j \cap (\boldsymbol{U}_{j+1}Q_j)\boldsymbol{r}_{\mathrm{QR},j}\big\}. \end{aligned} \tag{20.2}$$

For brevity, denote

$$\begin{aligned} \widehat{u}_{j+1} &= u_j + \sum_{i=1}^{q} h_j^i f^{[i]}(t_j, u_j) \\ \boldsymbol{z}_{j+1} &= h_j^{q+1} f^{[q+1]}(\boldsymbol{T}_j, \widetilde{\boldsymbol{y}}_j), \quad \text{and} \\ \boldsymbol{x}_{j+1} &= (\boldsymbol{U}_{j+1}S_j)\boldsymbol{\alpha} + \big\{(\boldsymbol{U}_{j+1}A_j)\boldsymbol{r}_j \cap (\boldsymbol{U}_{j+1}Q_j)\boldsymbol{r}_{\mathrm{QR},j}\big\}. \end{aligned}$$

Then

$$y(t_{j+1}; t_0, y_0) \in \boldsymbol{y}_{j+1}^* := (\widehat{u}_{j+1} + \boldsymbol{z}_{j+1} + \boldsymbol{x}_{j+1}) \cap \widetilde{\boldsymbol{y}}_j.$$

20.1.2 Corrector

In the corrector, we compute $\boldsymbol{y}_{j+1} \subseteq \boldsymbol{y}_{j+1}^*$. Usually, $\boldsymbol{y}_{j+1}$ is much tighter than $\boldsymbol{y}_{j+1}^*$.

Let

$$y_j = y(t_j; t_0, y_0) \quad \text{and} \quad y_{j+1} = y(t_{j+1}; t_0, y_0).$$

Denote $k = p + q + 1$ $(p, q \geq 0)$ and

$$c_i^{q,p} = \frac{q!\,(q+p-i)!}{(p+q)!\,(q-i)!} \qquad (q, p, \text{ and } i \geq 0). \tag{20.3}$$

Denote also

$$\gamma_{p,q} = \frac{q!p!}{(p+q)!} \tag{20.4}$$

Then [23]

$$\begin{aligned} \sum_{i=0}^{q} (-1)^i c_i^{q,p} h_j^i f^{[i]}(t_{j+1}, y_{j+1}) \in &\sum_{i=0}^{p} c_i^{p,q} h_j^i f^{[i]}(t_j, y_j) \\ &+ (-1)^q \gamma_{p,q} h_j^{p+q+1} f^{[p+q+1]}(\boldsymbol{T}_j, \widetilde{\boldsymbol{y}}_j). \end{aligned} \tag{20.5}$$

Denote

$$
\begin{aligned}
y^*_{j+1} &= \mathrm{m}(\boldsymbol{y}^*_{j+1}), && (20.6)\\
\boldsymbol{B}_{j+1} &= \sum_{i=0}^{q} (-1)^i c_i^{q,p} h_j^i J(f^{[i]}; \boldsymbol{y}^*_{j+1}), && (20.7)\\
\boldsymbol{F}_j &= \sum_{i=0}^{p} c_i^{p,q} h_j^i J(f^{[i]}; \boldsymbol{y}_j), && (20.8)\\
C_{j+1} &= \mathrm{m}(\boldsymbol{B}_{j+1}), && (20.9)\\
\boldsymbol{S}_{j+1} &= (C_{j+1}^{-1}\boldsymbol{F}_j) S_j, && (20.10)\\
\boldsymbol{A}_{j+1} &= (C_{j+1}^{-1}\boldsymbol{F}_j) A_j, && (20.11)\\
\boldsymbol{Q}_{j+1} &= (C_{j+1}^{-1}\boldsymbol{F}_j) Q_j, && (20.12)\\
\boldsymbol{e}_{j+1} &= (-1)^q \gamma_{p,q}\, h_j^{p+q+1}\, f^{[p+q+1]}(\boldsymbol{T}_j, \widetilde{\boldsymbol{y}}_j), && (20.13)\\
g_{j+1} &= \sum_{i=0}^{p} c_i^{p,q} h_j^i f^{[i]}(u_j) - \sum_{i=0}^{q} (-1)^i c_i^{q,p} h_j^i f^{[i]}(y^*_{j+1}), && (20.14)\\
\boldsymbol{d}_{j+1} &= g_{j+1} + \boldsymbol{e}_{j+1}, \quad \text{and} && (20.15)\\
\boldsymbol{w}_{j+1} &= C_{j+1}^{-1}\boldsymbol{d}_{j+1} + (I - C_{j+1}^{-1}\boldsymbol{B}_{j+1})(\boldsymbol{y}^*_{j+1} - y^*_{j+1}). && (20.16)
\end{aligned}
$$

(For sufficiently small h_j, we can enclose the inverse of C_{j+1}.)

Since

$$y_{j+1},\ y^*_{j+1} \in \boldsymbol{y}^*_{j+1} \quad \text{and} \quad y_j,\, u_j \in \boldsymbol{y}_j,$$

we can apply the mean-value theorem to the two sums in (20.5), and using the above notation derive that

$$
\begin{aligned}
y_{j+1} &\in y^*_{j+1} + \boldsymbol{S}_{j+1}\boldsymbol{\alpha} + \boldsymbol{A}_{j+1}\boldsymbol{r}_j + \boldsymbol{w}_{j+1} \quad \text{and} && (20.17)\\
y_{j+1} &\in y^*_{j+1} + \boldsymbol{S}_{j+1}\boldsymbol{\alpha} + \boldsymbol{Q}_{j+1}\boldsymbol{r}_{\mathrm{QR},j} + \boldsymbol{w}_{j+1}. && (20.18)
\end{aligned}
$$

Denoting

$$\boldsymbol{s_{j+1}} = (\boldsymbol{A}_{j+1}\boldsymbol{r}_j) \cap (\boldsymbol{Q}_{j+1}\boldsymbol{r}_{\mathrm{QR},j}),$$

we have

$$y(t_{j+1}; t_0, y_0) \in \boldsymbol{y}_{j+1} := (y^*_{j+1} + \boldsymbol{S}_{j+1}\boldsymbol{\alpha} + \boldsymbol{s}_{j+1} + \boldsymbol{w}_{j+1}) \cap \boldsymbol{y}^*_{j+1}.$$

20.1.3 Computing a solution representation

We have to determine u_{j+1}, S_{j+1}, A_{j+1}, Q_{j+1}, $\boldsymbol{r}_{j+1}$, and $\boldsymbol{r}_{\mathrm{QR},j+1}$ for the next step.

Let

$$u_{j+1} = \mathrm{m}(\boldsymbol{y_{j+1}}), \tag{20.19}$$
$$S_{j+1} = \mathrm{m}(\boldsymbol{S}_{j+1}), \tag{20.20}$$
$$\boldsymbol{v}_{j+1} = y^*_{j+1} - u_{j+1} + (\boldsymbol{S}_{j+1} - S_{j+1})\boldsymbol{\alpha} + \boldsymbol{w}_{j+1}. \tag{20.21}$$
$$A_{j+1} = \mathrm{m}(\boldsymbol{A}_{j+1}), \tag{20.22}$$
$$\boldsymbol{r}_{j+1} = (A^{-1}_{j+1}\boldsymbol{A}_{j+1})\boldsymbol{r}_j + A^{-1}_{j+1}\boldsymbol{v}_{j+1}, \tag{20.23}$$
$$Q_{j+1} \quad \text{the orthonormal matrix described in Subsection 20.1.4,} \quad \text{and} \tag{20.24}$$
$$\boldsymbol{r}_{\mathrm{QR},j+1} = (Q^{-1}_{j+1}\boldsymbol{Q}_{j+1})\boldsymbol{r}_{\mathrm{QR},j} + Q^{-1}_{j+1}\boldsymbol{v}_{j+1}. \tag{20.25}$$

Using (20.19–20.25) in (20.17–20.18), we derive that

$$y_{j+1} \in \left\{u_{j+1} + S_{j+1}\alpha + A_{j+1}r \mid \alpha \in \boldsymbol{\alpha},\, r \in \boldsymbol{r}_{j+1}\right\} \quad \text{and}$$
$$y_{j+1} \in \left\{u_{j+1} + S_{j+1}\alpha + Q_{j+1}r \mid \alpha \in \boldsymbol{\alpha},\, r \in \boldsymbol{r}_{\mathrm{QR},j+1}\right\}.$$

We assumed above that we can enclose A^{-1}_{j+1}. If we cannot enclose A^{-1}_{j+1}, or if

$$Q_{j+1}\boldsymbol{r}_{\mathrm{QR},j+1} \subseteq A_{j+1}\boldsymbol{r}_{j+1},$$

we set

$$A_{j+1} = Q_{j+1} \quad \text{and} \quad \boldsymbol{r}_{j+1} = \boldsymbol{r}_{\mathrm{QR},j+1}.$$

20.1.4 Computing Q_{j+1}

Let

$$\widetilde{A}_{j+1} = \mathrm{m}(\boldsymbol{Q}_{j+1}) \quad \text{and} \quad D = \mathrm{diag}\big(\mathrm{w}(\boldsymbol{r}_{\mathrm{QR},j})\big).$$

Let also P_{j+1} be a permutation matrix such that the columns of $\widetilde{A}_{j+1}DP_{j+1}$ are sorted in non-increasing order in the Euclidean norm. Then, we perform the QR factorization of $\widetilde{A}_{j+1}DP_{j+1} = Q_{j+1}R_{j+1}$ and use Q_{j+1} in our method.

20.2 Implementation

20.2.1 The IHO class

235 ⟨class IHO 235⟩ ≡
```
class IHO {
public:
  IHO(int n);
  ⟨set and get functions 277⟩
  void compCoeffs();
  void compTightEnclosure(interval &t_next);
  void acceptSolution();
  virtual ~IHO();
private:
  void compCpq(int p, int q);
  void compCqp(int p, int q);
  interval compErrorConstant(int p, int q);
private:
  unsigned int p, q, order_trial;
  interval h_trial;
  pVector y_pred_point;
  iVector y, y_pred, globalExcess, temp, temp2, x, u_next, predictor_excess,
      corrector_excess, z, w, gj, term, d, s;
  iMatrix Fj, M, Cinv, G, B, S, A, Q, U, V, Ainv;
  pMatrix C, A_point;
  MatrixInverse *matrix_inverse;
  Solution *solution, *trial_solution;
  interval *C_pq, *C_qp;
  HOE *hoe;
  AD *ad;
  Control *control;
  interval errorConstant;
};
```

This code is used in chunk 292.

20.2.2 Computing a tight enclosure

The main functions is

```
236 ⟨ compute tight enclosure 236 ⟩ ≡
  void IHO::compTightEnclosure(interval &t_next)
  {
    ⟨ initialize IHO method 238 ⟩
    ⟨ predictor: compute $\boldsymbol{y}^*_{j+1}$ 242 ⟩
    ⟨ corrector: compute $\boldsymbol{y}_{j+1}$ 247 ⟩
    ⟨ set $\boldsymbol{t}_{j+1}$ 264 ⟩
    ⟨ find solution representation for next step 265 ⟩
  }
```

This code is used in chunk 293.

20.2.3 Initialization

Stepsize

We assume we have enclosures on the solution at $t_j \in \boldsymbol{t}_j$ and now wish to compute enclosures at $t_{j+1} \in \boldsymbol{t}_{j+1}$. Hence, we have for the stepsize

$$h_j = t_{j+1} - t_j \in \boldsymbol{h}_j = \boldsymbol{t}_{j+1} - \boldsymbol{t}_j.$$

238 $\langle$ initialize IHO method 238 $\rangle \equiv$ /*

$$\begin{array}{r} t\_next = \boldsymbol{t}_{j+1} \\ solution\to t = \boldsymbol{t}_j \\ \hline h\_trial \supseteq \boldsymbol{h}_j = \boldsymbol{t}_{j+1} - \boldsymbol{t}_j \end{array}$$

```
     */
  $h\_trial = t\_next - solution\to t$;
```

See also chunks 240 and 241.

This code is used in chunk 236.

Order and method coefficients

The order is in *order_trial*. Initially, $order\_trial = 0$. If $control\to order \neq order\_trial$, we set $order\_trial = control\to order$ and compute the coefficients (20.3) and (20.4) of the method. If the value for the order is k, we require that $p + q + 1 = k$ and $p \leq q$. We set

$$p = \lfloor (k-1)/2 \rfloor \quad \text{and} \quad q = \lceil (k-1)/2 \rceil.$$

239 $\langle$ compute IHO method coefficients 239 $\rangle \equiv$

```
void IHO :: compCoeffs ( )
{
  if (order_trial ≠ control→order) {
            /* deal with order */
    order_trial = control→order;
    double pq = (order_trial − 1)/2.0;
    p = int(floor((pq)));
    q = int(ceil((pq)));
    assert(p + q + 1 ≡ order_trial);
        /* reallocate memory if necessary */
    if (C_pq) delete[] C_pq;
    C_pq = new interval[p + 1];
    if (C_qp) delete[] C_qp;
    C_qp = new interval[q + 1];
        /* compute coefficients */
    compCpq(p, q);
    compCqp(p, q);
    errorConstant = compErrorConstant(p, q);
  }
}
```

This code is used in chunk 293.

240 $\langle$ initialize IHO method 238 $\rangle +\equiv$

```
  compCoeffs( );
```

Local excess

In the HOE method, we enclose

$$h_{j,0}^{q+1} f^{[q+1]}(\boldsymbol{t}_j + [0,1]h_{j,0}, \widetilde{\boldsymbol{y}}_j)$$

and

$$h_{j,0}^{p+q+1} f^{[p+q+1]}(\boldsymbol{t}_j + [0,1]h_{j,0}, \widetilde{\boldsymbol{y}}_j) = h_{j,0}^{k} f^{[k]}(\boldsymbol{t}_j + [0,1]h_j, \widetilde{\boldsymbol{y}}_j).$$

In the IHO method, we have to enclose

$$h_j^{q+1} f^{[q+1]}(\boldsymbol{T}_j, \widetilde{\boldsymbol{y}}_j) \quad \text{and} \quad h_j^{p+q+1} f^{[p+q+1]}(\boldsymbol{T}_j, \widetilde{\boldsymbol{y}}_j),$$

where $\boldsymbol{T}_j \subseteq \boldsymbol{t}_j + [0,1]h_{j,0}$, cf. Proposition 19.1.

We have

$$\begin{aligned} h_j^{q+1} f^{[q+1]}(\boldsymbol{T}_j, \widetilde{\boldsymbol{y}}_j) &= \left(\frac{h_j}{h_{j,0}}\right)^{q+1} h_{j,0}^{q+1} f^{[q+1]}(\boldsymbol{T}_j, \widetilde{\boldsymbol{y}}_j) \\ &\subseteq \left(\frac{h_j}{h_{j,0}}\right)^{q+1} h_{j,0}^{q+1} f^{[q+1]}(\boldsymbol{t}_j + [0, h_{j,0}], \widetilde{\boldsymbol{y}}_j). \end{aligned}$$

Similarly, we have

$$\begin{aligned} h_j^{k} f^{[k]}(\boldsymbol{T}_j, \widetilde{\boldsymbol{y}}_j) &= \left(\frac{h_j}{h_{j,0}}\right)^{k} h_{j,0}^{k} f^{[k]}(\boldsymbol{T}_j, \widetilde{\boldsymbol{y}}_j) \\ &\subseteq \left(\frac{h_j}{h_{j,0}}\right)^{k} h_{j,0}^{k} f^{[k]}(\boldsymbol{t}_j + [0, h_{j,0}], \widetilde{\boldsymbol{y}}_j). \end{aligned}$$

Hence, we just re-scale.

241 ⟨ initialize IHO method 238 ⟩ +≡ /*

$$\begin{array}{rl} \mathit{h\_trial} & \ni h_j \\ \mathit{tayl\_coeff}\to\mathit{getStepsize}(\,) & \text{returns } h_{j,0} \\ \mathit{tayl\_coeff} & \text{contains enclosures on } h_{j,0}^{i} f^{[i]}(\boldsymbol{t}_j + [0,1]h_{j,0}, \widetilde{\boldsymbol{y}}_j) \\ & \quad \text{for } i = 0, \dots, k = p+q+1 \\ \hline \mathit{rescale} & \ni h_j / h_{j,0} \\ \mathit{predictor\_excess} & \supseteq h_{j,0}^{q+1} f^{[q+1]}(\boldsymbol{t}_j + [0,1]h_{j,0}, \widetilde{\boldsymbol{y}}_j) \\ \mathit{predictor\_excess} & \supseteq h_{j}^{q+1} f^{[q+1]}(\boldsymbol{t}_j + [0,1]h_{j,0}, \widetilde{\boldsymbol{y}}_j) \\ \mathit{corrector\_excess} & \supseteq h_{j,0}^{p+q+1} f^{[p+q+1]}(\boldsymbol{t}_j + [0,1]h_{j,0}\widetilde{\boldsymbol{y}}_j) \\ \mathit{corrector\_excess} & \supseteq h_{j}^{p+q+1} f^{[p+q+1]}(\boldsymbol{t}_j + [0,1]h_{j,0}, \widetilde{\boldsymbol{y}}_j) \end{array}$$

*/

```
interval rescale = h_trial / ad→tayl_coeff_ode→getStepsize( );
hoe→getErrorTerm(predictor_excess, q + 1);
scaleV(predictor_excess, v_bias::pow(rescale, q + 1));
hoe→getErrorTerm(corrector_excess, p + q + 1);
scaleV(corrector_excess, pow(rescale, order_trial));
```

20.2.4 Predictor

We enclose $\widehat{u}_{j+1}$, $\boldsymbol{U}_{j+1}$, $\boldsymbol{x}_{j+1}$, and $\boldsymbol{y}^*_{j+1}$. Below, $t_j \in \boldsymbol{t}_j$.

242 $\langle$ predictor: compute $\boldsymbol{y}^*_{j+1}$ 242 $\rangle \equiv$

```
  ⟨ $\widehat{u}_{j+1} = u_j + \sum_{i=1}^q h_j^i f^{[i]}(t_j, u_j)$ 243 ⟩
  ⟨ $\boldsymbol{U}_{j+1} = I + \sum_{i=1}^q h_j^i \, J(f^{[i]}; \boldsymbol{y}_j)$ 244 ⟩
  ⟨ $\boldsymbol{x}_{j+1} = (\boldsymbol{U}_{j+1} S_j)\boldsymbol{\alpha} + \{(\boldsymbol{U}_{j+1} A_j)\boldsymbol{r}_j \cap (\boldsymbol{U}_{j+1} Q_j)\boldsymbol{r}_{\mathrm{QR},j}\}$ 245 ⟩
  ⟨ $\boldsymbol{y}^*_{j+1} = (\widehat{u}_{j+1} + \boldsymbol{z}_{j+1} + \boldsymbol{x}_{j+1}) \cap \widetilde{\boldsymbol{y}}_j$ 246 ⟩
```

This code is used in chunk 236.

243 $\langle\, \widehat{u}_{j+1} = u_j + \sum_{i=1}^q h_j^i f^{[i]}(t_j, u_j)$ 243 $\rangle \equiv$ /*

$$\begin{array}{rl} solution\text{→}t & = \boldsymbol{t}_j \\ solution\text{→}u & = u_j \\ h\_trial & \supseteq \boldsymbol{h}_j \\ q & \text{order} \\ \hline tayl\_coeff & \text{contains enclosures on } h_j^i f^{[i]}(t_j, u_j) \text{ for } i = 0, \ldots, q \\ u\_next & \ni \widehat{u}_{j+1} \end{array}$$

```
  */
  $assignV(temp, solution\text{→}u)$;      /* $temp$ stores $u_j$ as an interval */
  $ad\text{→}tayl\_coeff\_ode\text{→}set(solution\text{→}t, temp, h\_trial, q)$;
  $ad\text{→}tayl\_coeff\_ode\text{→}compTerms(\,)$;
  $ad\text{→}tayl\_coeff\_ode\text{→}sumTerms(u\_next, q)$;
```

This code is used in chunk 242.

244 $\langle\, \boldsymbol{U}_{j+1} = I + \sum_{i=1}^q h_j^i \, J(f^{[i]}; \boldsymbol{y}_j)$ 244 $\rangle \equiv$ /*

$$\begin{array}{rl} solution\text{→}t & = \boldsymbol{t}_j \\ solution\text{→}y & \supseteq \boldsymbol{y}_j \\ h\_trial & \supseteq \boldsymbol{h}_j \\ q & \text{order} \\ \hline tayl\_coeff\_var & \text{contains enclosures on } h_j^i \, J(f^{[i]}; \boldsymbol{y}_j) \\ & \text{for } i = 0, 1, \ldots, q \\ & U \supseteq \boldsymbol{U}_{j+1} = I + \displaystyle\sum_{i=1}^q h_j^i \, J(f^{[i]}; \boldsymbol{y}_j) \end{array}$$

```
  */
  $ad\text{→}tayl\_coeff\_var\text{→}set(solution\text{→}t, solution\text{→}y, h\_trial, q)$;
  $ad\text{→}tayl\_coeff\_var\text{→}compTerms(\,)$;
  $ad\text{→}tayl\_coeff\_var\text{→}sumTerms(U, q)$;
```

This code is used in chunk 242.

Now we evaluate

245 $\langle\, \boldsymbol{x}_{j+1} = (\boldsymbol{U}_{j+1}S_j)\boldsymbol{\alpha} + \big\{(\boldsymbol{U}_{j+1}A_j)\boldsymbol{r}_j \cap (\boldsymbol{U}_{j+1}Q_j)\boldsymbol{r}_{\mathrm{QR},j}\big\}$ 245 $\rangle \equiv$ /*

$$\begin{array}{rl}
\textit{solution} & \text{contains } u_j, S_j, \boldsymbol{\alpha}, A_j, Q_j, \boldsymbol{r}_j, \boldsymbol{r}_{\mathrm{QR},j}, \boldsymbol{y}_j \\
U & \supseteq \boldsymbol{U}_{j+1} \\
\hline
M & \supseteq \boldsymbol{U}_{j+1}A_j \\
\textit{temp} & \supseteq (U_{j+1}A_j)\boldsymbol{r}_j \\
M & \supseteq \boldsymbol{U}_{j+1}Q_j \\
\textit{temp2} & \supseteq (\boldsymbol{U}_{j+1}Q_j)\boldsymbol{r}_{\mathrm{QR},j} \\
\textit{temp} & \supseteq (\boldsymbol{U}_{j+1}A_j)\boldsymbol{r}_j \cap (\boldsymbol{U}_{j+1}Q_j)\boldsymbol{r}_{\mathrm{QR},j} \\
M & \supseteq \boldsymbol{U}_{j+1}S_j \\
\textit{tem2} & \supseteq (\boldsymbol{U}_{j+1}S_j)\boldsymbol{\alpha} \\
x & \supseteq \boldsymbol{x}_{j+1} = (\boldsymbol{U}_{j+1}S_j)\boldsymbol{\alpha} + \big\{(\boldsymbol{U}_{j+1}A_j)\boldsymbol{r}_j \cap (\boldsymbol{U}_{j+1}Q_j)\boldsymbol{r}_{\mathrm{QR},j}\big\}
\end{array}$$

```
  */
multMiMp(M, U, solution→A);
multMiVi(temp, M, solution→r);
multMiMp(M, U, solution→Q);
multMiVi(temp2, M, solution→rQR);
bool finite_temp = finite_interval(temp);
bool finite_temp2 = finite_interval(temp2);
if (¬finite_temp ∧ ¬finite_temp2) {
  control→ind = failure;
  return;
}
else {
  if (finite_temp ∧ finite_temp2) {
    bool b = intersect(temp, temp2, temp);
    assert(b);
  }
  else {
    if (¬finite_temp) assignV(temp, temp2);
        /* else temp is finite and temp2 is not. We do not do anything. */
  }
}
multMiMp(M, U, solution→S);
multMiVi(temp2, M, solution→alpha);
addViVi(x, temp, temp2);
```

This code is used in chunk 242.

Finally, we compute

246 $\langle\, \boldsymbol{y}^*_{j+1} = (\widehat{u}_{j+1} + \boldsymbol{z}_{j+1} + \boldsymbol{x}_{j+1}) \cap \widetilde{\boldsymbol{y}}_j$ 246 $\rangle \equiv$ /*

$$\begin{array}{r}
\textit{u\_next} \ni \widehat{u}_{j+1} \\
\textit{predictor\_excess} \supseteq \boldsymbol{z}_{j+1} \\
x \supseteq \boldsymbol{x}_{j+1} \\
\textit{hoe}\rightarrow\textit{getTrialApriori}(\,) \supseteq \widetilde{\boldsymbol{y}}_j \\
\hline
\textit{y\_pred} \supseteq u_{j+1} + \boldsymbol{z}_{j+1} \\
\textit{y\_pred} \supseteq \widehat{u}_{j+1} + \boldsymbol{z}_{j+1} + \boldsymbol{x}_{j+1} \\
\textit{y\_pred} \supseteq (\widehat{u}_{j+1} + \boldsymbol{z}_{j+1} + \boldsymbol{x}_{j+1}) \cap \widetilde{\boldsymbol{y}}_j
\end{array}$$

*/

```
addViVi(y_pred, u_next, predictor_excess);
addViVi(y_pred, x);
bool b = intersect(y_pred, hoe→getTrialApriori( ), y_pred);
assert(b);
```

This code is used in chunk 242.

20.2.5 Corrector

The computation of $\boldsymbol{y}_{j+1}$ is given by

247 $\langle$ corrector: compute $\boldsymbol{y}_{j+1}$ 247 $\rangle \equiv$

$\langle\, \boldsymbol{F}_j = \sum_{i=0}^{p} c_i^{p,q} h_j^i J(f^{[i]}; \boldsymbol{y}_j)$ 248 $\rangle$
$\langle\, \boldsymbol{B}_{j+1} = \sum_{i=0}^{q} (-1)^i c_i^{q,p} h_j^i J(f^{[i]}; \boldsymbol{y}^*_{j+1})$ 249 $\rangle$
$\langle\, C_{j+1} = \mathrm{m}(\boldsymbol{B}_{j+1})$ 250 $\rangle$
$\langle\, \boldsymbol{G}_{j+1} = C_{j+1}^{-1} \boldsymbol{F}_j$ 251 $\rangle$
$\langle\, \boldsymbol{S}_{j+1} = \boldsymbol{G}_{j+1} S_j$ 252 $\rangle$
$\langle\, \boldsymbol{A}_{j+1} = \boldsymbol{G}_{j+1} A_j$ 253 $\rangle$
$\langle\, \boldsymbol{Q}_{j+1} = \boldsymbol{G}_{j+1} Q_j$ 254 $\rangle$
$\langle\, \boldsymbol{e}_{j+1} = (-1)^q \gamma_{p,q}\, \boldsymbol{h}_j^{p+q+1}\, f^{[p+q+1]}(\boldsymbol{T}_j, \widetilde{\boldsymbol{y}}_j)$ 255 $\rangle$
$\langle\, g_{j+1} = \sum_{i=0}^{p} c_i^{p,q} h_j^i f^{[i]}(u_j) - \sum_{i=0}^{q} (-1)^i c_i^{q,p} h_j^i f^{[i]}(y^*_{j+1})$ 256 $\rangle$
$\langle\, \boldsymbol{d}_{j+1} = g_{j+1} + \boldsymbol{e}_{j+1}$ 260 $\rangle$
$\langle\, \boldsymbol{w}_{j+1} = C_{j+1}^{-1} \boldsymbol{d}_{j+1} + (I - C_{j+1}^{-1} \boldsymbol{B}_{j+1})(\boldsymbol{y}^*_{j+1} - y^*_{j+1})$ 261 $\rangle$
$\langle\, \boldsymbol{s_{j+1}} = (\boldsymbol{A}_{j+1} \boldsymbol{r}_j) \cap (\boldsymbol{Q}_{j+1} \boldsymbol{r}_{\mathrm{QR},j})$ 262 $\rangle$
$\langle\, \boldsymbol{y}_{j+1} = (y^*_{j+1} + \boldsymbol{S}_{j+1} \boldsymbol{\alpha} + \boldsymbol{s}_{j+1} + \boldsymbol{w}_{j+1}) \cap \boldsymbol{y}^*_{j+1}$ 263 $\rangle$

This code is used in chunk 236.

Computing $\boldsymbol{F}_j$

The matrices $h_j^i J(f^{[i]}; \boldsymbol{y}_j)$ for $i = 1, \ldots, q$ have been already enclosed in the predictor. Since $p \le q$, we have the terms that we need to enclose $\boldsymbol{F}_j$ in (20.8). In the code below, we obtain the enclosure on $h_j^i J(f^{[i]}; \boldsymbol{y}_j)$ in M and then scale M such that it contains $c_i^{p,q} h_j^i J(f^{[i]}; \boldsymbol{y}_j)$.

248 $\langle \boldsymbol{F}_j = \sum_{i=0}^{p} c_i^{p,q} h_j^i J(f^{[i]}; \boldsymbol{y}_j)$ 248 $\rangle \equiv$ /*

tayl_coeff_var	contains enclosures on $h_j^i J(f^{[i]}; \boldsymbol{y}_j)$ for $i = 0, 1, \ldots q$
p	order
M	$\supseteq h_j^i J(f^{[i]}; \boldsymbol{y}_j)$
$C\_pq[i]$	$\ni c_i^{p,q}$
M	$\supseteq c_i^{p,q} h_j^i J(f^{[i]}; \boldsymbol{y}_j)$
Fj	$\supseteq \boldsymbol{F}_j$

```
    */
setM(Fj, 0.0);
for (int i = p; i ≥ 1; i−−) {
   ad→tayl_coeff_var→getTerm(M, i);
   scaleM(M, C_pq[i]);
   addMiMi(Fj, M);
}
addId(Fj);
```

This code is used in chunk 247.

Computing $\boldsymbol{B}_{j+1}$

We enclose $h_j^i J(f^{[i]}; \boldsymbol{y}_{j+1}^*)$ for $i = 1, \ldots, q$. We obtain these enclosures in M, which is scaled such that it encloses $(-1)^i c_i^{p,q} h_j^i J(f^{[i]}; \boldsymbol{y}_{j+1}^*)$.

249 $\langle \boldsymbol{B}_{j+1} = \sum_{i=0}^{q} (-1)^i c_i^{q,p} h_j^i J(f^{[i]}; \boldsymbol{y}_{j+1}^*)$ 249 $\rangle \equiv$ /*

t_next	$= \boldsymbol{t}_{j+1}$
y_pred	$\supseteq \boldsymbol{y}_{j+1}^*$
h_trial	$\supseteq \boldsymbol{h}_j$
q	order
tayl_coeff_var	contains $h_j^i J(f^{[i]}; \boldsymbol{y}_{j+1}^*)$ for $i = 0, 1, \ldots q$
$C\_qp[i]$	$\ni (-1)^i c_i^{q,p}$
M	$\supseteq h_j^i J(f^{[i]}; \boldsymbol{y}_{j+1}^i)$
M	$\supseteq (-1)^i c_i^{q,p} h_j^i J(f^{[i]}; \boldsymbol{y}_{j+1}^*)$
B	$\supseteq \boldsymbol{B}_{j+1}$

```
    */
ad→tayl_coeff_var→set(t_next, y_pred, h_trial, q);
ad→tayl_coeff_var→compTerms();
setM(B, 0.0);
for (int i = q; i ≥ 1; i−−) {
    ad→tayl_coeff_var→getTerm(M, i);
    scaleM(M, C_qp[i]);
    addMiMi(B, M);
  }
  addId(B);
```
This code is used in chunk 247.

250 $\langle\, C_{j+1} = \mathrm{m}(\boldsymbol{B}_{j+1})$ 250 $\rangle \equiv$ /*

$$\frac{B \supseteq \boldsymbol{B}_{j+1}}{C = C_{j+1} = \mathrm{m}(\boldsymbol{B}_{j+1})}$$

```
      */
  midpoint(C, B);
```
This code is used in chunk 247.

The function *encloseInverse* below encloses the inverse of a point matrix. If this function fails, it returns *false*; otherwise, it returns *true*.

251 $\langle\, \boldsymbol{G}_{j+1} = C_{j+1}^{-1}\boldsymbol{F}_j$ 251 $\rangle \equiv$ /*

$$\frac{\begin{array}{c} C = C_{j+1} \\ Fj \supseteq \boldsymbol{F}_j \end{array}}{\begin{array}{c} Cinv \ni C_{j+1}^{-1} \text{ if } ok \\ G \supseteq \boldsymbol{G}_{j+1} = C_{j+1}^{-1}\boldsymbol{F}_j \end{array}}$$

```
      */
  bool ok = matrix_inverse→encloseMatrixInverse(Cinv, C);
  if (¬ok)
  {
    control→ind = failure;
#ifdef VNODE_DEBUG
    printMessage("Could␣not␣invert␣the␣C␣matrix.");
#endif
    return;
  }
  multMiMi(G, Cinv, Fj);
```
This code is used in chunk 247.

252 $\langle\, \boldsymbol{S}_{j+1} = \boldsymbol{G}_{j+1}S_j$ 252 $\rangle \equiv$ /*

$$\frac{\begin{array}{c} solution \rightarrow S = S_j \\ G \supseteq \boldsymbol{G}_{j+1} \end{array}}{S \supseteq \boldsymbol{S}_{j+1} = \boldsymbol{G}_{j+1}S_j}$$

$*/$
$multMiMp(S, G, solution\text{->}S);$

This code is used in chunk 247.

253 $\langle\, \boldsymbol{A}_{j+1} = \boldsymbol{G}_{j+1}A_j$ 253 $\rangle \equiv$ $/*$

$$\frac{\begin{array}{r}solution\text{->}A = A_j\\ G \supseteq \boldsymbol{G}_{j+1}\end{array}}{A \supseteq \boldsymbol{A}_{j+1} = \boldsymbol{G}_{j+1}A_j}$$

$*/$
$multMiMp(A, G, solution\text{->}A);$

This code is used in chunk 247.

254 $\langle\, \boldsymbol{Q}_{j+1} = \boldsymbol{G}_{j+1}Q_j$ 254 $\rangle \equiv$ $/*$

$$\frac{\begin{array}{r}solution\text{->}Q = Q_j\\ G \supseteq \boldsymbol{G}_{j+1}\end{array}}{Q \supseteq \boldsymbol{Q}_{j+1} = \boldsymbol{G}_{j+1}Q_j}$$

$*/$
$multMiMp(Q, G, solution\text{->}Q);$

This code is used in chunk 247.

Computing $\boldsymbol{e}_{j+1}$

corrector_excess encloses $\boldsymbol{h}_j^{p+q+1} f^{[p+q+1]}(\boldsymbol{T}_j, \widetilde{\boldsymbol{y}}_j)$. We multiply it by $(-1)^q\gamma_{p,q}$.

255 $\langle\, \boldsymbol{e}_{j+1} = (-1)^q\gamma_{p,q}\, \boldsymbol{h}_j^{p+q+1} f^{[p+q+1]}(\boldsymbol{T}_j, \widetilde{\boldsymbol{y}}_j)$ 255 $\rangle \equiv$ $/*$

$$\frac{\begin{array}{r}corrector\_excess \supseteq \boldsymbol{h}_j^{p+q+1} f^{[p+q+1]}(\boldsymbol{T}_j, \widetilde{\boldsymbol{y}}_j)\\ errorConstant \supseteq (-1)^q\gamma_{p,q}\end{array}}{corrector\_excess \supseteq (-1)^q\gamma_{p,q}\boldsymbol{h}_j^{p+q+1} f^{[p+q+1]}(\boldsymbol{T}_j, \widetilde{\boldsymbol{y}}_j)}$$

$*/$
$scaleV(corrector\_excess, errorConstant);$

This code is used in chunk 247.

Enclosing g_{j+1}

256 $\langle\, g_{j+1} = \sum_{i=0}^{p} c_i^{p,q} h_j^i f^{[i]}(u_j) - \sum_{i=0}^{q} (-1)^i c_i^{q,p} h_j^i f^{[i]}(y_{j+1}^*)$ 256 $\rangle \equiv$

$\langle\, g_{j+1}^{\mathrm{f}} = \sum_{i=0}^{p} c_i^{p,q} h_j^i f^{[i]}(u_j)$ 257 $\rangle$
$\langle\, g_{j+1}^{\mathrm{b}} = \sum_{i=0}^{q} (-1)^i c_i^{q,p} h_j^i f^{[i]}(y_{j+1}^*)$ 258 $\rangle$
$\langle\, g_{j+1} = g_{j+1}^{\mathrm{f}} - g_{j+1}^{\mathrm{b}}$ 259 $\rangle$

This code is used in chunk 247.

We have the terms $h_j^i f^{[i]}(u_j)$ for $i = 1, \dots q$ computed in the predictor.

257 $\langle\, g^{\mathrm{f}}_{j+1} = \sum_{i=0}^{p} c_i^{p,q} h_j^i f^{[i]}(u_j)$ 257 $\rangle \equiv$ /*

$$\begin{array}{rl} \textit{tayl\_coeff} & \text{contains } h_j^i f^{[i]}(u_j) \text{ for } i = 0, 1, \dots q \\ p & \text{order} \\ \hline gj & = 0 \\ \textit{term} & \ni h_j^i f^{[i]}(u_j) \\ C\_pq[i] & \ni c_i^{p,q} \\ \textit{term} & \ni c_i^{p,q} h_j^i f^{[i]}(u_j) \\ \hline gj & \ni g^{\mathrm{f}}_{j+1} = \displaystyle\sum_{i=0}^{p} c_i^{p,q} h_j^i f^{[i]}(u_j) \end{array}$$

```
    */
setV(gj, 0.0);
for (int i = p; i ≥ 0; i−−) {
    ad→tayl_coeff_ode→getTerm(term, i);
    scaleV(term, C_pq[i]);
    addViVi(gj, term);
}
```

This code is used in chunk 256.

258 $\langle\, g^{\mathrm{b}}_{j+1} = \sum_{i=0}^{q} (-1)^i c_i^{q,p} h_j^i f^{[i]}(y^*_{j+1})$ 258 $\rangle \equiv$ /*

$$\begin{array}{rl} t\_next & = \boldsymbol{t}_{j+1} \\ y\_pred & \supseteq \boldsymbol{y}^*_{j+1} \\ h\_trial & \supseteq \boldsymbol{h}_j \\ q & \text{order} \\ \hline \textit{temp2} & = 0 \\ y\_pred\_point & = y^*_{j+1} = \mathrm{m}(\boldsymbol{y}^*_{j+1}) \\ \textit{temp} & \text{interval vector storing } y^*_{j+1} \\ \textit{tayl\_coeff} & \text{contains } h_j^i f^{[i]}(y^*_{j+1}) \text{ after } \textit{compTerms} \text{ is called} \\ \textit{term} & \ni h_j^i f^{[i]}(y^*_{j+1}) \\ C\_qp[i] & \ni (-1)^i c_i^{q,p} \\ \textit{term} & \ni (-1)^i c_i^{q,p} h_j^i f^{[i]}(y^*_{j+1}) \\ \hline \textit{temp2} & \ni g^{\mathrm{b}}_{j+1} = \displaystyle\sum_{i=0}^{q} (-1)^i c_i^{q,p} h_j^i f^{[i]}(y^*_{j+1}) \end{array}$$

```
    */
```

```
midpoint(y_pred_point, y_pred);
assignV(temp, y_pred_point);
ad→tayl_coeff_ode→set(t_next, temp, h_trial, q);
ad→tayl_coeff_ode→compTerms();

setV(temp2, 0.0);
for (int i = q; i ≥ 0; i−−) {
   ad→tayl_coeff_ode→getTerm(term, i);
   scaleV(term, C_qp[i]);
   addViVi(temp2, term);
}
```

This code is used in chunk 256.

259 $\langle\, g_{j+1} = g^{\mathrm{f}}_{j+1} - g^{\mathrm{b}}_{j+1}$ 259 $\rangle \equiv$ /*

$$\frac{\begin{array}{r} gj \ni g^{\mathrm{f}}_{j+1} \\ temp2 \ni g^{\mathrm{b}}_{j+1} \end{array}}{gj \ni g_{j+1} = g^{\mathrm{f}}_{j+1} - g^{\mathrm{b}}_{j+1}}$$

```
    */
subViVi(gj, temp2);
```

This code is used in chunk 256.

Computing $\boldsymbol{d}_{j+1}$

260 $\langle\, \boldsymbol{d}_{j+1} = g_{j+1} + \boldsymbol{e}_{j+1}$ 260 $\rangle \equiv$ /*

$$\frac{\begin{array}{r} corrector\_excess \supseteq \boldsymbol{e}_{j+1} \\ gj \ni g_{j+1} \end{array}}{d \supseteq \boldsymbol{d}_{j+1} = g_{j+1} + \boldsymbol{e}_{j+1}}$$

```
    */
addViVi(d, gj, corrector_excess);
```

This code is used in chunk 247.

Computing $\boldsymbol{w}_{j+1}$

Now, we compute

261 $\langle\, \boldsymbol{w}_{j+1} = C_{j+1}^{-1}\boldsymbol{d}_{j+1} + (I - C_{j+1}^{-1}\boldsymbol{B}_{j+1})(\boldsymbol{y}_{j+1}^{*} - y_{j+1}^{*})$ 261 $\rangle \equiv$ /*

$$\begin{array}{rl} \textit{y\_pred\_point} & = y_{j+1}^{*} \\ \textit{y\_pred} & \supseteq \boldsymbol{y}_{j+1}^{*} \\ B & \supseteq \boldsymbol{B}_{j+1} \\ \textit{Cinv} & \ni C_{j}^{-1} \\ d & \supseteq \boldsymbol{d}_{j+1} \\ \hline \textit{term} & \supseteq \boldsymbol{y}_{j+1}^{*} - y_{j+1}^{*} \\ M & \supseteq C_{j}^{-1}\boldsymbol{B}_{j+1} \\ M & \supseteq I - C_{j}^{-1}\boldsymbol{B}_{j+1} \\ \textit{temp} & \supseteq (I - C_{j+1}^{-1}\boldsymbol{B}_{j+1})(\boldsymbol{y}_{j+1}^{*} - y_{j+1}^{*}) \\ w & \supseteq \boldsymbol{w}_{j+1} = C_{j+1}^{-1}\boldsymbol{d}_{j+1} + (I - C_{j+1}^{-1}\boldsymbol{B}_{j+1})(\boldsymbol{y}_{j+1}^{*} - y_{j+1}^{*}) \end{array}$$

```
    */
subViVp(term, y_pred, y_pred_point);
multMiMi(M, Cinv, B);
subFromId(M);
multMiVi(temp, M, term);
multMiVi(w, Cinv, d);      /* w = Cinv * d + temp */
addViVi(w, temp);
```

This code is used in chunk 247.

262 $\langle\, \boldsymbol{s_{j+1}} = (\boldsymbol{A}_{j+1}\boldsymbol{r}_{j}) \cap (\boldsymbol{Q}_{j+1}\boldsymbol{r}_{\mathrm{QR},j})$ 262 $\rangle \equiv$ /*

$$\begin{array}{rl} \textit{solution}\to r & \supseteq \boldsymbol{r}_{j} \\ A & \supseteq \boldsymbol{A}_{j+1} \\ \textit{solution}\to rQR & \supseteq \boldsymbol{r}_{\mathrm{QR},j} \\ Q & \supseteq \boldsymbol{Q}_{j+1} \\ \hline \textit{temp} & \supseteq \boldsymbol{A}_{j+1}\boldsymbol{r}_{j} \\ s & \supseteq \boldsymbol{Q}_{j+1}\boldsymbol{r}_{\mathrm{QR},j} \\ s & \supseteq \boldsymbol{s_{j+1}} = (\boldsymbol{A}_{j+1}\boldsymbol{r}_{j}) \cap (\boldsymbol{Q}_{j+1}\boldsymbol{r}_{\mathrm{QR},j}) \end{array}$$

```
    */
multMiVi(temp, A, solution→r);
multMiVi(s, Q, solution→rQR);
b = intersect(s, temp, s);
assert(b);
```

This code is used in chunk 247.

263 $\langle\, \boldsymbol{y}_{j+1} = (y^*_{j+1} + \boldsymbol{S}_{j+1}\boldsymbol{\alpha} + \boldsymbol{s}_{j+1} + \boldsymbol{w}_{j+1}) \cap \boldsymbol{y}^*_{j+1}$ 263 $\rangle \equiv$ /*

$$\begin{array}{rl}
\textit{trial\_solution}\to\textit{alpha} & \supseteq \boldsymbol{\alpha} \\
S & \supseteq \boldsymbol{S}_{j+1} \\
\textit{y\_pred} & \supseteq \boldsymbol{y}^*_{j+1} \\
\textit{y\_pred\_point} & = y^*_{j+1} \\
s & \supseteq \boldsymbol{s}_{j+1} \\
w & \supseteq \boldsymbol{w}_{j+1} \\
\hline
\textit{globalExcess} & \supseteq \boldsymbol{s}_{j+1} + \boldsymbol{w}_{j+1} \\
\textit{temp} & \supseteq \boldsymbol{S}_{j+1}\boldsymbol{\alpha} \\
\textit{temp} & \supseteq \boldsymbol{S}_{j+1}\boldsymbol{\alpha} + \boldsymbol{s}_{j+1} + \boldsymbol{w}_{j+1} \\
\textit{temp} & \supseteq y^*_{j+1} + \boldsymbol{S}_{j+1}\boldsymbol{\alpha} + \boldsymbol{s}_{j+1} + \boldsymbol{w}_{j+1} \\
\textit{trial\_solution}\to y & \supseteq \boldsymbol{y}_{j+1} = (y^*_{j+1} + \boldsymbol{S}_{j+1}\boldsymbol{\alpha} + \boldsymbol{x}_{j+1}) \cap \boldsymbol{y}^*_{j+1}
\end{array}$$

```
    */
  addViVi(globalExcess, s, w);
  multMiVi(temp, S, trial_solution→alpha);
  addViVi(temp, globalExcess);
  addViVp(temp, y_pred_point);
  b = intersect(trial_solution→y, y_pred, temp);
  assert(b);
```

This code is used in chunk 247.

264 $\langle$ set $\boldsymbol{t}_{j+1}$ 264 $\rangle \equiv$

```
  trial_solution→t = t_next;
```

This code is used in chunk 236.

20.2.6 Enclosure representation

265 $\langle$ find solution representation for next step 265 $\rangle \equiv$

$\langle\, u_{j+1} = \mathrm{m}(\boldsymbol{y_{j+1}})$ 266 $\rangle$
$\langle\, S_{j+1} = \mathrm{m}(\boldsymbol{S}_{j+1})$ 267 $\rangle$
$\langle\, \boldsymbol{v}_{j+1} = y^*_{j+1} - u_{j+1} + (\boldsymbol{S}_{j+1} - S_{j+1})\boldsymbol{\alpha} + \boldsymbol{w}_{j+1}$ 268 $\rangle$
$\langle\, A_{j+1} = \mathrm{m}(\boldsymbol{A}_{j+1})$ 269 $\rangle$
$\langle\, \boldsymbol{r}_{j+1} = (A^{-1}_{j+1}\boldsymbol{A}_{j+1})\boldsymbol{r}_j + A^{-1}_{j+1}\boldsymbol{v}_{j+1}$ 270 $\rangle$
$\langle$ compute Q_{j+1} 273 $\rangle$
$\langle\, \boldsymbol{r}_{\mathrm{QR},j+1} = (Q^{-1}_{j+1}\boldsymbol{Q}_{j+1})\boldsymbol{r}_{\mathrm{QR},j} + Q^{-1}_{j+1}\boldsymbol{v}_{j+1}$ 271 $\rangle$
$\langle$ reset if needed 272 $\rangle$

This code is used in chunk 236.

266 $\langle\, u_{j+1} = \mathrm{m}(\boldsymbol{y_{j+1}})$ 266 $\rangle \equiv$ /*

$$\frac{\textit{trial\_solution}\!\to\! y \supseteq \boldsymbol{y}_{j+1}}{\textit{trial\_solution}\!\to\! u = u_{j+1}}$$

*/
midpoint(*trial_solution*→*u*, *trial_solution*→*y*);
This code is used in chunk 265.

267 $\langle\, S_{j+1} = \mathrm{m}(\boldsymbol{S}_{j+1})$ 267 $\rangle \equiv$ /*

$$\textit{trial\_solution}\!\to\! S = S_{j+1} = \mathrm{m}(\boldsymbol{S}_{j+1})$$

*/
midpoint(*trial_solution*→*S*, *S*);
This code is used in chunk 265.

268 $\langle\, \boldsymbol{v}_{j+1} = y^*_{j+1} - u_{j+1} + (\boldsymbol{S}_{j+1} - S_{j+1})\boldsymbol{\alpha} + \boldsymbol{w}_{j+1}$ 268 $\rangle \equiv$ /*

$$\frac{\begin{aligned} \textit{y\_pred\_point} &= y^*_{j+1} \\ \textit{trial\_solution}\!\to\! u &= u_{j+1} \\ S &\supseteq \boldsymbol{S}_{j+1} \\ \textit{trial\_solution}\!\to\! S &= S_{j+1} \\ \textit{trial\_solution}\!\to\! \textit{alpha} &\supseteq \boldsymbol{\alpha} \\ w &\supseteq \boldsymbol{w}_{j+1} \end{aligned}}{\begin{aligned} S &\supseteq \boldsymbol{S}_{j+1} - S_{j+1} \\ z &\supseteq (\boldsymbol{S}_{j+1} - S_{j+1})\boldsymbol{\alpha} \\ z &\supseteq (\boldsymbol{S}_{j+1} - S_{j+1})\boldsymbol{\alpha} + \boldsymbol{w}_{j+1} \\ z &\supseteq -u_{j+1} + (\boldsymbol{S}_{j+1} - S_{j+1})\boldsymbol{\alpha} + \boldsymbol{w}_{j+1} \\ z &\supseteq \boldsymbol{v}_{j+1} = y^*_{j+1} - u_{j+1} + (\boldsymbol{S}_{j+1} - S_{j+1})\boldsymbol{\alpha} + \boldsymbol{w}_{j+1} \end{aligned}}$$

*/
subMiMp(*S*, *trial_solution*→*S*);
multMiVi(*z*, *S*, *trial_solution*→*alpha*);
addViVi(*z*, *w*);
subViVp(*z*, *trial_solution*→*u*);
addViVp(*z*, *y_pred_point*);
This code is used in chunk 265.

269 $\langle\, A_{j+1} = \mathrm{m}(\boldsymbol{A}_{j+1})$ 269 $\rangle \equiv$ /*

$$\frac{A \supseteq \boldsymbol{A}_{j+1}}{\textit{trial\_solution}\!\to\! A = A_{j+1}}$$

```
      */
   midpoint(trial_solution→A, A);
```
This code is used in chunk 265.

270 $\langle\, \boldsymbol{r}_{j+1} = (A_{j+1}^{-1}\boldsymbol{A}_{j+1})\boldsymbol{r}_j + A_{j+1}^{-1}\boldsymbol{v}_{j+1}$ 270 $\rangle \equiv$ /*

$$\frac{\begin{aligned} trial\_solution\to A &= A_{j+1} \\ A &\supseteq \boldsymbol{A}_{j+1} \\ z &\supseteq \boldsymbol{v}_{j+1} \\ solution\to r &\supseteq \boldsymbol{r}_j \end{aligned}}{\begin{aligned} Ainv &\ni A_{j+1}^{-1} \text{ if } ok \\ temp &\supseteq A_{j+1}^{-1}\boldsymbol{v}_{j+1} \\ M &\supseteq A_{j+1}^{-1}\boldsymbol{A}_{j+1} \\ trial\_solution\to r &\supseteq (A_{j+1}^{-1}\boldsymbol{A}_{j+1})\boldsymbol{r}_j \\ trial\_solution\to r &\supseteq \boldsymbol{r}_{j+1} = (A_{j+1}^{-1}\boldsymbol{A}_{j+1})\boldsymbol{r}_j + A_{j+1}^{-1}\boldsymbol{v}_{j+1} \end{aligned}}$$

```
      */
   ok = matrix_inverse→encloseMatrixInverse(Ainv, trial_solution→A);
   if (ok) {
      multMiVi(temp, Ainv, z);
      multMiMi(M, Ainv, A);
      multMiVi(trial_solution→r, M, solution→r);
      addViVi(trial_solution→r, temp);
   }
```
This code is used in chunk 265.

271 $\langle\, \boldsymbol{r}_{\mathrm{QR},j+1} = (Q_{j+1}^{-1}\boldsymbol{Q}_{j+1})\boldsymbol{r}_{\mathrm{QR},j} + Q_{j+1}^{-1}\boldsymbol{v}_{j+1}$ 271 $\rangle \equiv$ /*

$$\frac{\begin{aligned} trial\_solution\to Q &= Q_{j+1} \\ z &\supseteq \boldsymbol{v}_{j+1} \\ solution\to rQR &\supseteq \boldsymbol{r}_{\mathrm{QR},j} \\ Q &\supseteq \boldsymbol{Q}_{j+1} \end{aligned}}{\begin{aligned} Ainv &\ni Q_{j+1}^{-1} \text{ if } ok \\ temp &\supseteq Q_{j+1}^{-1}\boldsymbol{v}_{j+1} \\ M &\supseteq Q_{j+1}^{-1}\boldsymbol{Q}_{j+1} \\ trial\_solution\to rQR &\supseteq (Q_{j+1}^{-1}\boldsymbol{Q}_{j+1})\boldsymbol{r}_{\mathrm{QR},j} \\ trial\_solution\to rQR &\supseteq \boldsymbol{r}_{\mathrm{QR},j+1} = (Q_{j+1}^{-1}\boldsymbol{Q}_{j+1})\boldsymbol{r}_{\mathrm{QR},j} + Q_{j+1}^{-1}\boldsymbol{v}_{j+1} \end{aligned}}$$

```
      */
```

```
  $b = matrix\_inverse \to orthogonalInverse(Ainv, trial\_solution \to Q)$;
  if $(b \equiv false)$ {
    $control \to ind = failure$;
#ifdef VNODE_DEBUG
    $printMessage($"Could␣not␣invert␣the␣Q␣matrix."$)$;
#endif
    return;
  }
  $multMiVi(temp, Ainv, z)$;
  $multMiMi(M, Ainv, Q)$;
  $multMiVi(trial\_solution \to rQR, M, solution \to rQR)$;
  $addViVi(trial\_solution \to rQR, temp)$;
```

This code is used in chunk 265.

272 ⟨ reset if needed 272 ⟩ ≡ /*

$$\begin{array}{r}
trial\_solution \to Q = Q_{j+1} \\
trial\_solution \to rQR \supseteq \boldsymbol{r}_{\mathrm{QR},j+1} \\
trial\_solution \to A = A_{j+1} \\
trial\_solution \to r \supseteq \boldsymbol{r}_{j+1} \\
\hline
temp \supseteq Q_{j+1}\boldsymbol{r}_{\mathrm{QR},j+1} \\
temp2 \supseteq A_{j+1}\boldsymbol{r}_{j+1} \\
\textbf{if}\ (\ subseteq(temp, temp2) \vee \neg ok\ ) \qquad\qquad\qquad\qquad \\
trial\_solution \to A = Q_{j+1} \\
trial\_solution \to r \supseteq \boldsymbol{r}_{\mathrm{QR},j+1}
\end{array}$$

```
      */
  $multMpVi(temp, trial\_solution \to Q, trial\_solution \to rQR)$;
  $multMpVi(temp2, trial\_solution \to A, trial\_solution \to r)$;
  if $(subseteq(temp, temp2) \vee \neg ok)$ {
    $assignM(trial\_solution \to A, trial\_solution \to Q)$;
    $assignV(trial\_solution \to r, trial\_solution \to rQR)$;
  }
```

This code is used in chunk 265.

Computing Q_{j+1}

273 ⟨compute Q_{j+1} 273⟩ ≡ /*

$$\frac{Q \supseteq \boldsymbol{Q}_{j+1}}{\begin{aligned} A\_point &= \mathrm{m}(\boldsymbol{Q}_{j+1}) \\ A\_point &= \mathrm{m}(\boldsymbol{Q}_{j+1})\,\mathrm{diag}\,\mathrm{w}(\boldsymbol{r}_{\mathrm{QR},j}) \\ C &= A\_point \text{ with columns sorted in non-increasing order in } \|\cdot\|_2 \\ C &= Q_{j+1}R_{j+1} \\ trial\_solution \to Q &= Q_{j+1} \end{aligned}}$$

```
    */
  midpoint(A_point, Q);
  int n = sizeM(Q);
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
      A_point[i][j] *= v_bias::width(solution→rQR[j]);
  if (¬(inf_normM(A_point) < numeric_limits⟨double⟩::max())) {
#ifdef VNODE_DEBUG
    printMessage("The␣computed␣enclosures␣are␣too␣wide");
#endif
    control→ind = failure;
    return;
  }
  sortColumns(C, A_point);
  b = computeQR(trial_solution→Q, C);
  assert(b);
```

This code is used in chunk 265.

20.2.7 Constructor

274 ⟨constructor IHO 274⟩ ≡

```
  IHO::IHO(int n)
  {
    order_trial = 0;
    solution = new Solution(n);
    trial_solution = new Solution(n);
    matrix_inverse = new MatrixInverse(n);
    C_pq = C_qp = 0;
    sizeV(y, n);
    sizeV(y_pred, n);
    sizeV(globalExcess, n);
    sizeV(y_pred_point, n);
    sizeV(temp, n);
    sizeV(temp2, n);
    sizeV(x, n);
    sizeV(u_next, n);
    sizeV(predictor_excess, n);
    sizeV(corrector_excess, n);
    sizeV(z, n);
    sizeV(w, n);
    sizeV(gj, n);
    sizeV(term, n);
    sizeV(d, n);
    sizeV(s, n);
    sizeM(Fj, n);
    sizeM(M, n);
    sizeM(Cinv, n);
    sizeM(G, n);
    sizeM(B, n);
    sizeM(S, n);
    sizeM(A, n);
    sizeM(Q, n);
    sizeM(U, n);
    sizeM(V, n);
    sizeM(Ainv, n);
    sizeM(C, n);
    sizeM(A_point, n);
  }
```

This code is used in chunk 293.

20.2.8 Destructor

275 ⟨ destructor IHO 275 ⟩ ≡

```
  IHO::~IHO()
  {
    delete matrix_inverse;
    delete trial_solution;
    delete solution;
    delete[] C_pq;
    delete[] C_qp;
  }
```

This code is used in chunk 293.

20.2.9 Accepting a solution

276 ⟨ accept solution (IHO) 276 ⟩ ≡

```
  void IHO::acceptSolution() {
    solution→t = trial_solution→t;
    assignV(solution→y, trial_solution→y);
    assignV(solution→u, trial_solution→u);
    assignM(solution→S, trial_solution→S);
    assignV(solution→alpha, trial_solution→alpha);
    assignM(solution→A, trial_solution→A);
    assignV(solution→r, trial_solution→r);
    assignM(solution→Q, trial_solution→Q);
    assignV(solution→rQR, trial_solution→rQR);
  }
```

This code is used in chunk 293.

20.2.10 Set and get functions

277 ⟨set and get functions 277⟩ ≡

```
  void set(Control *c, HOE *hoem, AD *ad0)
  {
    assert(c ∧ hoem ∧ ad0);
    control = c;
    hoe = hoem;
    ad = ad0;
  }
  void getTightEnclosure(iVector &y) const
  {
    y = solution→y;
  }
  void init(const interval &t, const iVector &y)
  {
    trial_solution→init(t, y);
    solution→init(t, y);
  }
  const iVector &getGlobalExcess() const
  {
    return globalExcess;
  }
```

This code is used in chunk 235.

20.2.11 Constants

Computing $c_i^{p,q}$

We show how to compute

$$c_i^{p,q} = \frac{p!}{(p+q)!}\frac{(q+p-i)!}{(p-i)!}.$$

From

$$c_{i-1}^{p,q} = \frac{p!}{(p+q)!}\frac{(q+p-i+1)!}{(p-i+1)!} = \frac{p!}{(p+q)!}\frac{(q+p-i)!}{(p-i)!}\frac{q+p-i+1}{p-i+1}$$
$$= \frac{q+p-i+1}{p-i+1}c_i^{p,q}$$

we compute

$$c_i^{p,q} = \frac{p-i+1}{q+p-i+1} c_{i-1}^{p,q}, \quad \text{where} \quad c_0^{p,q} = 1.$$

279 $\langle\, c_i^{p,q} = \frac{p!}{(p+q)!}\frac{(q+p-i)!}{(p-i)!}$ 279 $\rangle \equiv$

```
void IHO :: compCpq(int p, int q)
{    /*

          C_pq[i] ∋ $c_i^{p,q}$    for $i = 1, \dots, p$

     */
  int tmp = q + p + 1;
  C_pq[0] = 1.0;
  for (int i = 1; i ≤ p; i++) {
    C_pq[i] = (C_pq[i − 1] ∗ double(p − i + 1))/double(tmp − i);
  }
}
```

This code is used in chunk 293.

Computing $(-1)^i c_i^{q,p}$

When computing $c_i^{q,p}$, we multiply them by $(-1)^i$, that is $(-1)^i c_i^{q,p}$.
From

$$c_{i-1}^{q,p} = \frac{q!}{(q+p)!}\frac{(p+q-i+1)!}{(q-i+1)!} = \frac{q!}{(q+p)!}\frac{(p+q-i)!}{(q-i)!}\frac{p+q-i+1}{q-i+1}$$
$$= \frac{p+q-i+1}{q-i+1} c_i^{q,p}$$

we compute

$$(-1)^i c_i^{q,p} = (-1)\frac{q-i+1}{p+q-i+1}\left((-1)^{i-1} c_{i-1}^{q,p}\right), \quad \text{where} \quad c_0^{q,p} = 1.$$

280 $\langle\, (-1)^i c_i^{q,p} = (-1)^i \frac{q!}{(p+q)!}\frac{(q+p-i)!}{(q-i)!}$ 280 $\rangle \equiv$

```
void IHO :: compCqp(int p, int q)
{    /*

          C_qp[i] ∋ $(-1)^i c_i^{q,p}$    for $i = 1, \dots, q$

     */
  int tmp = q + p + 1;
  C_qp[0] = 1.0;
  for (int i = 1; i ≤ q; i++)
    C_qp[i] = (−C_qp[i − 1] ∗ double(q − i + 1))/double(tmp − i);
}
```

This code is used in chunk 293.

Even number

This function returns *true* if its arguments ($k \geq 0$) is even and *false* otherwise. We check if the last bit of k is 1.

```
281 ⟨check if a number is even 281⟩ ≡
  inline bool isEven(unsigned int k)
  {
    if ($k \equiv 0$) return true;
    if ($k$ & #01) return false;
    return true;
  }
```

This code is used in chunk 293.

Computing $(-1)^q q!p!/(p+q)!$

For given p and q, *compErrorConst* computes $(-1)^q q!p!/(p+q)!$. We use the relation

$$\frac{q!p!}{(p+q)!} = \frac{p!}{(q+1)(q+2)\cdots(p+q)} = \frac{1}{q+1}\frac{2}{q+2}\cdots\frac{p}{q+p}.$$

```
282 ⟨compute $(-1)^q q!p!/(p+q)!$ 282⟩ ≡        /*

            $err\_const \supseteq (-1)^q q!p!/(p+q)!$

     */
  interval IHO::compErrorConstant(int p, int q)
  {
    interval err_const(1.0);
    for (int $i = 1$; $i \leq p$; $i$++) {
      err_const *= double(i);
      err_const /= (q + i);
    }
    if (¬isEven(q)) err_const = −err_const;
    return err_const;
  }
```

This code is used in chunk 293.

20.2.12 Sorting columns of a matrix

Given an $n \times n$ point matrix A, we sort its columns such that, in the 2-norm, they are in non-increasing order. In *sortColumns* the columns of the input matrix A are sorted, and the result is stored in B.

284 ⟨sort columns 284⟩ ≡
```
void sortColumns(pMatrix &B, const pMatrix &A)
{
  ⟨compute column norms of A 286⟩
  ⟨check if sorting is needed 287⟩
  ⟨perform sorting 289⟩
}
```
This code is used in chunk 291.

First, we create a structure to store the index and the norm of the column corresponding to this index.

285 ⟨index-norm structure 285⟩ ≡
```
struct index_norm {
  int index;
  double norm;
};
```
This code is used in chunk 291.

We store in an array b in $b[i].norm$ the 2-norm of column i of A.

286 ⟨compute column norms of A 286⟩ ≡
```
int n = v_blas::sizeM(B);
index_norm *b = new index_norm[n];
pVector tmp;
v_blas::sizeV(tmp, n);
for (int j = 0; j < n; j++) {
  b[j].index = j;
  getColumn(tmp, A, j);
  b[j].norm = v_blas::norm2(tmp);
}
```
This code is used in chunk 284.

287 ⟨check if sorting is needed 287⟩ ≡
```
bool sorting_needed = false;
for (int i = 0; i < n - 1; i++) {
  if (b[i].norm < b[i + 1].norm) {
    sorting_needed = true;
    break;
  }
}
```
This code is used in chunk 284.

We use the standard *qsort* function. We need a *compare* function for it.

288 ⟨compare function 288⟩ ≡

```
inline int v_compare(const void *a1, const void *b1)
{
   index_norm *a = (index_norm *) a1;
   index_norm *b = (index_norm *) b1;
   if (a->norm > b->norm) return −1;
   if (a->norm < b->norm) return 1;
   return 0;
}
```

This code is used in chunk 291.

We sort the b array based on $b[i].norm$. Then the jth column of B is the $b[j].index$ column of A.

289 ⟨ perform sorting 289 ⟩ ≡

```
if (sorting_needed) std::qsort((void *) b, n, sizeof(index_norm), v_compare);
for (int j = 0; j < n; j++)
{
   getColumn(tmp, A, b[j].index);
   setColumn(B, tmp, j);
}
delete[] b;
```

This code is used in chunk 284.

Files

291 ⟨ sortcolumns.cc 291 ⟩ ≡

```
#include <cstdlib>
#include <cassert>
#include "vnodeinterval.h"
#include "basiclinalg.h"
   namespace vnodelp {
      using namespace v_bias;
      using namespace v_blas;
      ⟨ index-norm structure 285 ⟩
      ⟨ compare function 288 ⟩
      ⟨ sort columns 284 ⟩
   }
```

292 ⟨ iho.h 292 ⟩ ≡

```
#ifndef IHO_H
#define IHO_H
#include <cassert>
#include <cmath>
#include "ad_ode.h"
#include "ad_var.h"
```

```
#include "hoe.h"
#include "intvfuncs.h"
#include "basiclinalg.h"
#include "matrixinverse.h"
  namespace vnodelp {
    using namespace v_bias;
    using namespace v_blas;
    ⟨ class IHO 235 ⟩
  }
#endif
```

293 ⟨ iho.cc 293 ⟩ ≡

```
#include <cmath>
#include "vnodeinterval.h"
#include "basiclinalg.h"
#include "solution.h"
#include "control.h"
#include "allad.h"
#include "matrixinverse.h"
#include "iho.h"
#include "debug.h"
#include "miscfuns.h"
  namespace vnodelp {
  using namespace v_bias;
  using namespace v_blas;
  extern void sortColumns(pMatrix &B, const pMatrix &A);
  extern bool computeQR(pMatrix &B, const pMatrix &A);
  ⟨ constructor IHO 274 ⟩
  ⟨ destructor IHO 275 ⟩
  ⟨ check if a number is even 281 ⟩
```

⟨ $c_i^{p,q} = \frac{p!}{(p+q)!}\frac{(q+p-i)!}{(p-i)!}$ 279 ⟩

⟨ $(-1)^i c_i^{q,p} = (-1)^i \frac{q!}{(p+q)!}\frac{(q+p-i)!}{(q-i)!}$ 280 ⟩

⟨ compute $(-1)^q q!p!/(p+q)!$ 282 ⟩

⟨ accept solution (IHO) 276 ⟩

⟨ compute tight enclosure 236 ⟩

⟨ compute IHO method coefficients 239 ⟩}

Chapter 21

The VNODE class

21.1 Declaration

295 ⟨class VNODE 295⟩ ≡

```
typedef enum {
  on, off
} stepAction;
class VNODE {
public:
  int steps;
  VNODE(AD *ad);
  void integrate(interval &t0, iVector &y0, const interval &t_end);
  ⟨set VNODE parameters 331⟩
  ⟨get functions (VNODE) 329⟩
  ~VNODE();
private:
  void acceptSolution(interval &t0, iVector &y0);
  double compHstart(const interval &t0, const iVector &y0);
private:
  int direction;
  interval t_trial, Tj, h_accepted;
  iVector temp;
  double h_start, h_min;
  pVector tp;
  Control *control;
  HOE *hoe;
  IHO *iho;
  AD *ad;
};
```

This code is used in chunk 333.

21.2 The integrator function

The integrator function consists of the following steps:

1. check input correctness and determine the direction of the integration
2. initialize
3. validate existence and uniqueness and select stepsize
4. check if the end point is reached
5. compute a tight enclosure
6. decide how to proceed

```
296 ⟨integrator 296⟩ ≡
  void VNODE::integrate(interval &t0, iVector &y0, const interval &t_end)
  {
    ⟨check input correctness 298⟩
    ⟨determine direction 305⟩
    if (control→ind ≡ first_entry) {
      ⟨initialize integration 308⟩
    }
    while (t0 ≠ t_end) {
      if (control→ind ≡ first_entry ∨ control→ind ≡ success) {
        ⟨validate and select stepsize 317⟩
        ⟨check if last step 318⟩
      }
      ⟨compute enclosure 324⟩
      ⟨decide 325⟩
    }     /* restore hmin */
    control→hmin = h_min;
  }
```

This code is used in chunk 334.

21.2.1 Input correctness

control→ind must not be *failure*

```
298 ⟨check input correctness 298⟩ ≡
  if (control→ind ≡ failure) {
    vnodeMessage("Previous␣call␣to␣integrate()␣failed");
    vnodeMessage("Call␣setFirstEntry()␣before␣calling␣integrate");
    return;
  }
```

See also chunks 299, 300, 301, 302, 303, and 304.

This code is used in chunk 296.

The initial condition and the final points must be representable machine intervals.

```
299 ⟨ check input correctness 298 ⟩ +≡
  if (¬v_bias::finite_interval(t0) ∨ ¬v_bias::finite_interval(t_end)) {
    vnodeMessage("t0␣and␣t_end␣must␣be␣finite");
    control→ind = failure;
    return;
  }
  for (unsigned int i = 0; i < sizeV(y0); i++)
    if (¬v_bias::finite_interval(y0[i])) {
      vnodeMessage("y0␣must␣contain␣finite␣intervals");
      control→ind = failure;
      return;
    }
```

If the integrator is re-entered with the same end point, *t_end*, as the *t_end* from the most recent call, then it should return.

```
300 ⟨ check input correctness 298 ⟩ +≡
  t_trial = t0;
  if (t_trial ≡ t_end ∧ control→ind ≠ first_entry) {
    vnodeMessage("Set␣different␣t_end");
    return;
  }
```

At least one of the tolerances must be positive.

```
301 ⟨ check input correctness 298 ⟩ +≡
  if (control→atol ≤ 0 ∧ control→rtol ≤ 0) {
    vnodeMessage("Set␣nonnegative␣tolerances␣and␣at␣least\
        ␣one␣positive␣tolerance");
    return;
  }
```

The order must be between 3 and *getMaxOrder*( ).

```
302 ⟨ check input correctness 298 ⟩ +≡
  if (control→order < 3 | control→order > getMaxOrder()) {
    vnodeMessage("Set␣order␣>=␣3␣and␣order␣<=␣MAX_ORDER");
    control→ind = failure;
    return;
  }
```

The value for the minimum stepsize must be non-negative

```
303 ⟨ check input correctness 298 ⟩ +≡
  if (control→hmin < 0) {
    vnodeMessage("Set␣minimum␣stepsize␣>=␣0");
    return;
  }
```

The initial and final time intervals may not intersect.

```
304 ⟨check input correctness 298⟩ +≡
  if (¬v_bias::disjoint(t0, t_end)) {
    control→ind = failure;
    vnodeMessage("t0␣and␣t_end␣must␣be␣disjoint");
    return;
  }
```

21.2.2 Determine direction

The user provides $\boldsymbol{t}_0$ and $\boldsymbol{t}_{\mathrm{end}}$. The solver decides if the integration is from left to right or right to left.

If a first entry into *integrate*, we initialize *direction* with 0. We also save the direction from the most recent integration.

```
305 ⟨determine direction 305⟩ ≡
  if (control→ind ≡ first_entry) direction = 0;
  int last_direction = direction;
```

See also chunks 306 and 307.

This code is used in chunk 296.

1. If $\overline{\boldsymbol{t}}_0 < \underline{\boldsymbol{t}}_{\mathrm{end}}$, the integration is in positive direction.
2. If $\underline{\boldsymbol{t}}_0 > \overline{\boldsymbol{t}}_{\mathrm{end}}$, the integration is in negative direction.

```
306 ⟨determine direction 305⟩ +≡
  direction = 1;
  if (v_bias::sup(t_end) < v_bias::inf(t0))
    direction = −1;
```

If the direction of the integration is reversed without calling first *setFirstEntry*( ), then print a message and return.

```
307 ⟨determine direction 305⟩ +≡
  if (last_direction ≠ 0 ∧ last_direction ≡ −direction) {
    vnodeMessage("Integration␣direction␣changed␣without␣r\
        esetting.\n""␣␣␣␣Call␣setFirstEntry()␣before␣integrating.");
    control→ind = failure;
    return;
  }
```

21.2.3 Initialization

Initializing an integration includes the following parts discussed below.

308 ⟨ initialize integration 308 ⟩ ≡
```
steps = 0;
⟨ find minimum stepsize 309 ⟩
⟨ find initial stepsize 310 ⟩
⟨ set IHO method 312 ⟩
⟨ set HOE method 311 ⟩
```
This code is used in chunk 296.

Find minimum stepsize

- If *control→hmin* ≡ 0, the solver computes minimum stepsize.
- If *contol→ind* ≡ *success*, this is a re-entry to the integrator; it computes minimum stepsize.
- If *control→hmin* > 0∧*contol→ind* ≡ *first_entry*, the user has specified minimum stepsize. We still have to check if this value is not smaller than what the solver finds as a minimum stepsize.

309 ⟨ find minimum stepsize 309 ⟩ ≡
```
h_min = control→hmin;
if (control→hmin ≡ 0 ∨ control→ind ≡ success)
  control→hmin = compHmin(t0, t_end);
if (control→hmin > 0 ∧ control→ind ≡ first_entry) {
  double h = compHmin(t0, t_end);
  if (control→hmin < h) {
    control→ind = failure;
    vnodeMessage("Set␣larger␣value␣for␣hmin");
    return;
  }
  assert(control→hmin > 0);
}
```
This code is used in chunk 308.

Find initial stepsize

If *control→ind* ≡ *first_entry*, the solver computes initial stepsize.

310 ⟨ find initial stepsize 310 ⟩ ≡
```
if (control→ind ≡ first_entry) {
  h_start = compHstart(t0, y0);
}
if (h_start < control→hmin) {
  control→ind = failure;
  vnodeMessage("Minimum␣stepsize␣reached.");
  return;
}
```

```
if (direction ≡ −1) {
  if (control→ind ≡ first_entry) h_start = −h_start;
}
```
This code is used in chunk 308.

Setting the HOE method

We set in the HOE method the value for the initial stepsize, order, control structure, and an AD object.

311 ⟨ set HOE method 311 ⟩ ≡
```
hoe→setTrialStepsize(h_start);
hoe→setTrialOrder(control→order);
hoe→set(control, ad);
hoe→init(t0, y0);
```
This code is used in chunk 308.

Setting the IHO method

In the IHO method, we set the control object, the HOE method, and an AD object. We also compute the coefficients of the method and set the initial point.

312 ⟨ set IHO method 312 ⟩ ≡
```
iho→set(control, hoe, ad);
iho→compCoeffs( );
iho→init(t0, y0);
```
This code is used in chunk 308.

21.2.4 Methods involved in the initialization

Minimum stepsize

We determine the minimum magnitude, h_{min}, for the stepsize allowed as follows:

$$\begin{aligned} \widehat{t} &= \max\{|\boldsymbol{t}_0|, |\boldsymbol{t}_{\text{end}}|\}, \\ h_{\text{min}} &= \max\left\{\triangle\big(\text{next}(\widehat{t}) - \widehat{t}\big), \text{w}(\boldsymbol{t}_{\text{end}})\right\}. \end{aligned}$$

Here next(a) returns the next representable machine number to the right of a.

314 ⟨ compute h_{min} 314 ⟩ ≡
```
double compHmin(const interval &t0, const interval &t_end)
{    /*
```

$$\begin{aligned} t0 &= \boldsymbol{t}_0 \\ t\_end &= \boldsymbol{t}_{\text{end}} \\ \hline t &= \max\{|\boldsymbol{t}_0|, |\boldsymbol{t}_{\text{end}}|\} \\ t\_next &= \text{ next machine number to the right of } t \\ t &= \triangle\big(\text{next}(\widehat{t}) - \widehat{t}\big) \\ t &= \max\left\{\triangle\big(\text{next}(\widehat{t}) - \widehat{t}\big), \text{w}(\boldsymbol{t}_{\text{end}})\right\} \end{aligned}$$

```
        */
    double t = std::max(v_bias::mag(t0), v_bias::mag(t_end));
    double t_next = nextafter(t, t + 1);
    v_bias::round_up();
    t = t_next − t;
    t = std::max(t, v_bias::width(t_end));
    return t;
  }
```

This code is used in chunk 334.

Initial stepsize

On the first step we compute

$$h_{0,0} = \left(\frac{\text{tol}}{\|(k+1) f^{[k+1]}(\boldsymbol{y}_0)\|} \right)^{1/k},$$

(see [26]) where

$$\text{tol} = \text{rtol} \cdot \|\boldsymbol{y}_0\| + \text{atol}.$$

315 ⟨compute initial stepsize 315⟩ ≡

```
  double VNODE::compHstart(const interval &t0, const iVector &y0)
  {     /*
```

$$\begin{aligned} y0 &= \boldsymbol{y}_0 \\ k &\quad \text{order} \end{aligned}$$

$$\textit{tayl\_coeff\_ode} \text{ contains } f^{[i]}(\boldsymbol{t}_0, \boldsymbol{y}_0) \text{ for } i = 0, 1, \ldots, k+1$$
$$\text{after } \textit{compTerm} \text{ is called}$$

$$\begin{aligned} \textit{temp} &\supseteq f^{[k+1]}(\boldsymbol{t}_0, \boldsymbol{y}_0) \\ \textit{tol} &= \text{tol} = \text{rtol} \cdot \|\boldsymbol{y}_0\| + \text{atol} \\ t &= (k+1)\|f^{[k+1]}(\boldsymbol{t}_0, \boldsymbol{y}_0)\| \\ t &= \frac{\text{tol}}{(k+1)\|f^{[k+1]}(\boldsymbol{t}_0, \boldsymbol{y}_0)\|} \\ t &= \left(\frac{\text{tol}}{(k+1)\|f^{[k+1]}(\boldsymbol{t}_0, \boldsymbol{y}_0)\|} \right)^{1/k} \end{aligned}$$

```
        */
    int k = control→order;
    ad→tayl_coeff_ode→set(t0, y0, 1, k + 1);
    ad→tayl_coeff_ode→compTerms();
    ad→tayl_coeff_ode→getTerm(temp, k + 1);
    double tol = control→rtol ∗ inf_normV(y0) + control→atol;
    double t = (k + 1) ∗ inf_normV(temp);
    ⟨ check $t$ 316 ⟩
    $t = tol/t$;
    $t =$ **std**::$pow(t, 1.0/k)$;
    **return** $t$;
  }
```
This code is used in chunk 334.

The t that we compute above is normally $t > 0$. To prevent the case that it might be zero, we set t to be the largest of *control→atol* and *control→rtol*.

316 ⟨ check t 316 ⟩ ≡
```
  **if** $(t \equiv 0)$ $t =$ **std**::$max(control\rightarrow atol, control\rightarrow rtol)$;
```
This code is used in chunk 315.

21.2.5 Validate existence and uniqueness

We try to validate existence and uniqueness with $\boldsymbol{t}_j$ and $\boldsymbol{y}_j$. If *validate* returns *false*, we set *control→ind* = *failure*.

317 ⟨ validate and select stepsize 317 ⟩ ≡ /*

$$t0 = \boldsymbol{t}_j$$
$$y0 = \boldsymbol{y}_j$$

```
      */
  **bool** *info*;
  *hoe→compAprioriEnclosure*(*t0*, *y0*, *info*);
  **if** (*info* ≡ *false*) {
    *control→ind* = *failure*;
    *vnodeMessage*("Could␣not␣validate␣solution");
    **return**;
  }
```
This code is used in chunk 296.

21.2.6 Check last step

If *validate* returns successfully, we obtain an interval $[t_j, t_{j+1}]$, or $[t_{j+1}, t_j]$ if the integration is backwards, over which we have verified existence and uniqueness. Denote this interval by $\boldsymbol{T}_j$.

We want to prevent the solver from taking a very small last stepsize. To accomplish this, we adapt the idea for taking a last step from [13]. We consider three cases:

1. $\boldsymbol{t}_{\text{end}} \subseteq \boldsymbol{T}_j$
2. $\boldsymbol{t}_{\text{end}} \cap \boldsymbol{T}_j = \emptyset$
3. $t_{j+1} \in \boldsymbol{t}_{\text{end}}$

```
318 ⟨check if last step 318⟩ ≡        /*
                $Tj \supseteq \boldsymbol{T}_j$
                $tend = \boldsymbol{t}_{\text{end}}$

        */
    $Tj = hoe \rightarrow getTrialT()$;
    if (v_bias::subseteq(t_end, Tj)) {
        ⟨case 1 320⟩
    }
    else {
        if (v_bias::disjoint(t_end, Tj)) {
            ⟨case 2 321⟩
        }
        else {
            interval tmp;
            if (direction ≡ 1) tmp = v_bias::sup(Tj);
            else tmp = inf(Tj);      /* $tmp \ni \boldsymbol{t}_{j+1}$ */
            if (v_bias::subseteq(tmp, t_end)) {
                ⟨case 3 323⟩
            }
            else assert(0);
        }
    }
This code is used in chunk 296.
```

Case 1: $\boldsymbol{t}_{\mathbf{end}} \subseteq \boldsymbol{T}_j$. We set $\boldsymbol{t}_{j+1} = \boldsymbol{t}_{\text{end}}$; Figure 21.1.

```
320 ⟨case 1 320⟩ ≡
    t_trial = t_end;      /* t_end is inside Tj */
This code is used in chunk 318.
```

Case 2: $\boldsymbol{t}_{\mathbf{end}} \cap \boldsymbol{T}_j = \emptyset$. Denote

$$\Theta = |\boldsymbol{t}_{\text{end}} - t_j| = \max\{|\underline{\boldsymbol{t}}_{\text{end}} - t_j|, |\overline{\boldsymbol{t}}_{\text{end}} - t_j|\}.$$

(i) If $\mathrm{w}(\boldsymbol{T}_j) < \Theta/2$, then the next point we chose is the already selected t_{j+1}.

(ii) Otherwise, $\mathrm{w}(\boldsymbol{T}_j) \geq \Theta/2$; see Figure 21.2.

We compute

$$\boldsymbol{T}_j^* = \begin{cases} \boldsymbol{T}_j \cap (t_j + [0, \Theta/2]) & \text{if } t_j < t_{j+1} \\ \boldsymbol{T}_j \cap (t_j - [0, \Theta/2]) & \text{if } t_j > t_{j+1} \end{cases} \tag{21.1}$$

and select for the next step

$$t_{j+1} \leftarrow \begin{cases} \overline{\boldsymbol{T}}_{j+1}^* & \text{if } t_j < t_{j+1} \\ \underline{\boldsymbol{T}}_{j+1}^* & \text{if } t_j > t_{j+1} \end{cases} \tag{21.2}$$

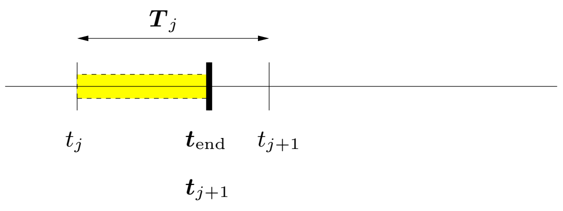


Figure 21.1. *The case $\boldsymbol{t}_{end} \subseteq \boldsymbol{T}_j$. We set $\boldsymbol{t}_{j+1} = \boldsymbol{t}_{tend}$.*

```
321  ⟨case 2 321⟩ ≡      /*
         $theta \geq \Theta = |\boldsymbol{t}_{\text{end}} - t_j|$
             $d \geq \mathrm{w}(\boldsymbol{T}_j)$

       */
     double theta = v_bias::mag(t_end − t0);
     if (v_bias::width(Tj) ≥ theta/2) {
        double d = theta/2.0;      /* $d = \Theta/2$ */
        interval tmp;
        if (direction ≡ 1) tmp = interval(0, d);
        else tmp = interval(−d, 0);
        bool b;
        b = v_bias::intersect(Tj, Tj, t0 + tmp);
        assert(b);
     }
     if (direction ≡ 1) t_trial = v_bias::sup(Tj);
     else t_trial = v_bias::inf(Tj);
```

This code is used in chunk 318.

Case 3: $t_{j+1} \in \boldsymbol{t}_{\mathbf{end}}$. We expect this situation to occur very rarely. We set

$$t_{j+1} \leftarrow \begin{cases} \underline{\boldsymbol{t}}_{\text{end}} & \text{if } t_j < t_{j+1} \\ \overline{\boldsymbol{t}}_{\text{tend}} & \text{if } t_j > t_{j+1}. \end{cases}$$

```
323  ⟨case 3 323⟩ ≡
     if (direction ≡ 1) t_trial = v_bias::inf(t_end);
     else t_trial = v_bias::sup(t_end);
```

This code is used in chunk 318.

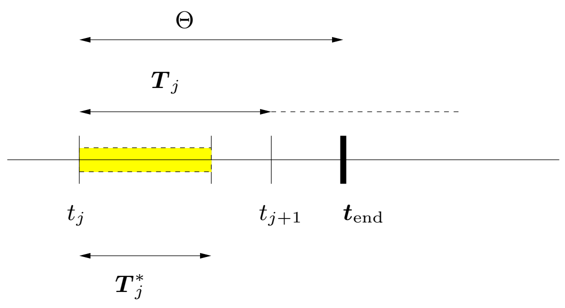


Figure 21.2. *When close to $\boldsymbol{t}_{end}$, we take the "middle" as the next integration point.*

21.2.7 Compute a tight enclosure

To compute a tight enclosure at $\boldsymbol{t}_{j+1}$, we call

```
324 ⟨compute enclosure 324⟩ ≡
      iho→compTightEnclosure(t_trial);
```
This code is used in chunk 296.

21.2.8 Decide

We decide how to proceed as shown below.

1. If *control→ind* ≡ *first_entry* we set *control→ind* ≡ *success*.
2. If *control→ind* ≡ *success*, we consider two cases.
   (a) If *control→interrupt* ≡ *no*, we continue the integration.
   (b) If *control→interrupt* ≡ *before_accept*, we return from the integration.
3. If *control→ind* ≡ *failure*, we return from the integration.

```
325 ⟨decide 325⟩ ≡
      bool ret = false;
      switch (control→ind) {
      case first_entry:
        control→ind = success;
      case success:
        if (control→interrupt ≡ before_accept) ret = true;
        break;
      case failure: return;
      }
      acceptSolution(t0, y0);
      if (ret) return;
```
This code is used in chunk 296.

Accept solution

We accept the solution computed in the *hoe* and *iho* objects. We also update $t0$ and $y0$.

326 ⟨VNODE accept solution 326⟩ ≡
```
void VNODE::acceptSolution(interval &t0, iVector &y0)
{
  hoe→acceptSolution( );
  iho→acceptSolution( );
  h_accepted = t_trial − t0;
  t0 = t_trial;
  iho→getTightEnclosure(y0);
  steps++;
}
```
This code is used in chunk 334.

21.3 Constructor/destructor

327 ⟨constructor (VNODE) 327⟩ ≡
```
VNODE::VNODE(AD *a)
{
  steps = 0;
  direction = 0;
  int n = a→size;
  sizeV(temp, n);
  sizeV(tp, n);
  hoe = new HOE(n);
  iho = new IHO(n);
  control = new Control;
  assert(a);
  ad = a;
}
```
This code is used in chunk 334.

328 ⟨destructor (VNODE) 328⟩ ≡
```
VNODE::~VNODE( )
{
  delete control;
  delete iho;
  delete hoe;
}
```
This code is used in chunk 334.

21.4 Get functions

To check if an integration is successful, we call

329 ⟨get functions (VNODE) 329⟩ ≡
```
unsigned int getMaxOrder( ) const
{
  return ad→getMaxOrder( );
}
bool successful( ) const
{
  if (control→ind ≡ success ∨ control→ind ≡ first_entry) return true;
  return false;
}
```
See also chunk 330.

This code is used in chunk 295.

330 ⟨get functions (VNODE) 329⟩ +≡
```
double getStepsize( ) const {      /* Returns the last stepsize taken. */
  return midpoint(h_accepted);
}
const iVector &getAprioriEncl( ) const {
      /* Obtains a reference to the a priori enclosure */
  return hoe→getApriori( );
}
const interval &getT( ) const {      /* Returns T_j. */
  return hoe→getT( );
}
double getGlobalExcess( ) {      /* Returns an estimate of the global excess. */
  width(tp, iho→getGlobalExcess( ));
  return inf_normV(tp);
}
double getGlobalExcess(unsigned int i) {
      /* Returns an estimate of the global excess in component i. */
  width(tp, iho→getGlobalExcess( ));
  return tp[i];
}
unsigned int getNoSteps( ) const {
      /* Returns the number of accepted steps during an integration. */
  return steps;
}
```

21.5 Set parameters

We set integration parameters by the following functions.

331 ⟨set VNODE parameters 331⟩ ≡

```
  void setTols(double a, double r = 0) {
        /* Sets atol and rtol. By default, rtol is set to 0 */
    control→atol = a;
    control→rtol = r;
  }
  void setOrder(int p) {      /* Sets order. */
    control→order = p;
  }
  void setHmin(double h) {      /* Sets a value for the minimum stepsize. */
    control→hmin = h;
  }
  void setOneStep(stepAction action) {
        /* Indicates an interrupt after each computed solution. */
    if (action ≡ on) control→interrupt = before_accept;
    else      /* off */
      control→interrupt = no;
  }
  void setFirstEntry() {      /* Indicates first entry */
    control→ind = first_entry;
  }
```

This code is used in chunk 295.

21.6 Files

21.6.1 Interface

```
333 ⟨vnodeint.h 333⟩ ≡
#ifndef VNODEINT_H
#define VNODEINT_H
#include <algorithm>
#include "miscfuns.h"
#include "vnodeinterval.h"
#include "vnoderound.h"
#include "vector_matrix.h"
#include "ad_ode.h"
#include "ad_var.h"
#include "allad.h"
#include "solution.h"
#include "control.h"
#include "matrixinverse.h"
#include "iho.h"
#include "hoe.h"
#include "vtiming.h"
#include "debug.h"
  namespace vnodelp {
    ⟨class VNODE 295⟩
  }
#endif
```

21.6.2 Implementation

```
334 ⟨integ.cc 334⟩ ≡
#include <cmath>
#include <algorithm>
#include <ostream>
  using namespace std;
#include "vnodeinterval.h"
#include "vector_matrix.h"
#include "matrixinverse.h"
#include "basiclinalg.h"
#include "vnodeint.h"
  namespace vnodelp {
    ⟨constructor (VNODE) 327⟩
    ⟨destructor (VNODE) 328⟩
    ⟨compute h_min 314⟩
    ⟨compute initial stepsize 315⟩
    ⟨integrator 296⟩
    ⟨VNODE accept solution 326⟩
  }
```

21.7 Interface to the VNODE-LP package

The interface to the VNODE-LP package is stored in

```
335 ⟨vnode.h 335⟩ ≡
#ifndef VNODE_H
#define VNODE_H
#include <algorithm>
#include "vnodeinterval.h"
#include "vnoderound.h"
#include "vector_matrix.h"
#include "ad_ode.h"
#include "ad_var.h"
#include "allad.h"
#include "vnodeint.h"
#include "solution.h"
#include "control.h"
#include "iho.h"
#include "hoe.h"
#include "matrixinverse.h"
#include "basiclinalg.h"
```

```
#include "fadbad_ad.h"
#include "fadbad_advar.h"
#include "fadbadad.h"
#endif
```

Part V

AD Implementation

Chapter 22

Using FADBAD++

Currently, VNODE employs the FADBAD++ package [29].

22.1 Computing ODE Taylor coefficients

22.1.1 FadbadODE class

339 ⟨ class FadbadODE 339 ⟩ ≡

```
typedef T⟨interval⟩ Tinterval;
typedef void(*Tfunction)(int n, Tinterval *yp, const Tinterval
      *y, Tinterval t, void *param);
class FadbadODE : public AD_ODE {
public:
   FadbadODE(int n, Tfunction, void *param = 0);
   void set(const interval &t0, const iVector &y0, const interval &h, int
         k);
   void compTerms( );
   void sumTerms(iVector &sum, int m);
   void getTerm(iVector &term, int i) const;
   interval getStepsize( ) const;
   void eval(void *param);
   ~FadbadODE( );
private:
   Tfunctionfcn;
   Tinterval *y_coeff, *f_coeff, t;
   int size;
   int order;
   interval stepsize;
};
```

This code is used in chunk 349.

22.1.2 Function description

In the constructor, we allocate the necessary memory, set the ODE problem, and generate the computational graph by calling *fcn*.

```
340 ⟨constructor (FadbadODE) 340⟩ ≡
  FadbadODE::FadbadODE(int n, Tfunction f, void *param)
  : AD_ODE() {
    size = n;
    y_coeff = new Tinterval[2 * n];
    f_coeff = y_coeff + n;
    fcn = f;
    fcn(size, f_coeff, y_coeff, t, param);
  }
```
This code is used in chunk 350.

```
341 ⟨FadbadODE destructor 341⟩ ≡
  FadbadODE::~FadbadODE()
  {
    delete y_coeff;
  }
```
This code is used in chunk 350.

```
342 ⟨initialize Taylor coefficients (FadbadODE) 342⟩ ≡
  void FadbadODE::set(const interval &t0, const iVector &y0, const
          interval &h, int k)
  {
    t[0] = t0;
    t[1] = h;
    stepsize = h;
    order = k;
    for (int eqn = 0; eqn < size; eqn++) y_coeff[eqn][0] = y0[eqn];
  }
```
This code is used in chunk 350.

The Taylor coefficients for the right side of $y' = f(t, y)$ are evaluated by *eval* and stored in *f_coeff*. Then, the ith coefficient for y is $h/i\times$ (the $(i-1)$st coefficient for f).

```
343 ⟨compute Taylor coefficients (FadbadODE) 343⟩ ≡
  void FadbadODE::compTerms()
  {
    for (int eqn = 0; eqn < size; eqn++)
      f_coeff[eqn].reset();      /* reset previously created terms */
    for (int coeff = 1; coeff ≤ order; coeff++)      /* compute coefficients */
      for (int eqn = 0; eqn < size; eqn++) {
        f_coeff[eqn].eval(coeff - 1);
        y_coeff[eqn][coeff] = stepsize * f_coeff[eqn][coeff - 1]/double(coeff);
      }
  }
```

This code is used in chunk 350.

344 ⟨ sum terms (FadbadODE) 344 ⟩ ≡

```
  void FadbadODE::sumTerms(iVector &sum, int m)
  {
    interval s;
    for (int eqn = 0; eqn < size; eqn++) {
      s = 0.0;
      for (int coeff = m; coeff ≥ 0; coeff--) s += y_coeff[eqn][coeff];
      sum[eqn] = s;
    }
  }
```

This code is used in chunk 350.

345 ⟨ get term (FadbadODE) 345 ⟩ ≡

```
  void FadbadODE::getTerm(iVector &term, int i) const
  {
    for (int eqn = 0; eqn < size; eqn++) term[eqn] = y_coeff[eqn][i];
  }
```

This code is used in chunk 350.

346 ⟨ obtain stepsize 346 ⟩ ≡

```
  interval FadbadODE::getStepsize( ) const
  {
    return stepsize;
  }
```

This code is used in chunk 350.

To evaluate the function and rebuild the computational graph, we do

347 ⟨ rebuild computational graph 347 ⟩ ≡

```
  void FadbadODE::eval(void *param)
  {
    fcn(size, f_coeff, y_coeff, t, param);
  }
```

This code is used in chunk 350.

22.1.3 Files

349 ⟨ fadbad_ad.h 349 ⟩ ≡

```
#ifndef Fadbad_ODE
#define Fadbad_ODE
```

```
#include "vnodeinterval.h"
#include "basiclinalg.h"
#include "ad_ode.h"
#include "ffadiff.h"
#include "fadbad_intv.inc"
  namespace vnodelp {
    ⟨ class FadbadODE 339 ⟩
  }
#endif
```

350 ⟨ fadbad_ad.cc 350 ⟩ ≡

```
#include "fadbad_ad.h"
  namespace vnodelp {
    ⟨ constructor (FadbadODE) 340 ⟩
    ⟨ FadbadODE destructor 341 ⟩
    ⟨ initialize Taylor coefficients (FadbadODE) 342 ⟩
    ⟨ compute Taylor coefficients (FadbadODE) 343 ⟩
    ⟨ sum terms (FadbadODE) 344 ⟩
    ⟨ get term (FadbadODE) 345 ⟩
    ⟨ obtain stepsize 346 ⟩
    ⟨ rebuild computational graph 347 ⟩
  }
```

22.2 Computing Taylor coefficients for the variational equation

22.2.1 FadbadVarODE class

352 ⟨ class FadbadVarODE 352 ⟩ ≡

```
  typedef T⟨F⟨interval⟩⟩ TFinterval;
  typedef void(*TFfunction)(int n, TFinterval *yp, const TFinterval
      *y, TFinterval tf, void *param);
  class FadbadVarODE : public AD_VAR {
  public:
    FadbadVarODE(int n, TFfunction f, void *param = 0);
    void set(const interval &t0, const iVector &y0, const interval &h, int
        k);
    void compTerms( );
    void sumTerms(iMatrix &sum, int m);
    void getTerm(iMatrix &term, int i) const;
    void eval(void *param) { fcn(size, tf_out, tf_in, tf, param); }
    ~FadbadVarODE( );
  private:
    TFinterval *tf_in, *tf_out, tf;
    TFfunction fcn;
    int size;
    int order;
    interval stepsize;
  };
```
This code is used in chunk 360.

22.2.2 Function description

353 ⟨ constructor (FadbadVarODE) 353 ⟩ ≡
```
  FadbadVarODE::FadbadVarODE(int n, TFfunction f, void *param)
  {
    size = n;
    tf_in = new TFinterval[2 * n];
    tf_out = tf_in + n;
    fcn = f;
    fcn(size, tf_out, tf_in, tf, param);
  }
```
This code is used in chunk 361.

354 ⟨ destructor (FadbadVarODE) 354 ⟩ ≡
```
  FadbadVarODE::~FadbadVarODE( )
  {
    delete[ ] tf_in;
  }
```
This code is used in chunk 361.

355 ⟨ initialize Taylor coefficients (FadbadVarODE) 355 ⟩ ≡
```
  void FadbadVarODE::set(const interval &t0, const iVector &y0, const
          interval &h, int k)
  {
    stepsize = h;
    order = k;
    tf[0].x( ) = t0;
    tf[1].x( ) = h;
    for (int eqn = 0; eqn < size; eqn++) {
      tf_in[eqn][0] = y0[eqn];
    }
  }
```
This code is used in chunk 361.

356 ⟨ compute Taylor coefficients (FadbadVarODE) 356 ⟩ ≡
```
  void FadbadVarODE::compTerms( )
  {
    for (int eqn = 0; eqn < size; eqn++) tf_out[eqn].reset();
    for (int eqn = 0; eqn < size; eqn++) tf_in[eqn][0].diff(eqn, size);
    for (int coeff = 0; coeff < order; coeff++) {
      for (int eqn = 0; eqn < size; eqn++) {
        tf_out[eqn].eval(coeff);
        tf_in[eqn][coeff + 1] = stepsize * (tf_out[eqn][coeff]/double(coeff + 1));
      }
    }
  }
```
This code is used in chunk 361.

357 ⟨sum terms (FadbadVarODE) 357⟩ ≡
```
  void FadbadVarODE::sumTerms(iMatrix &Sum, int k)
  {
    for (int row = 0; row < size; row++)
      for (int col = 0; col < size; col++) {
        interval s = 0.0;
        for (int coeff = k; coeff ≥ 1; coeff--) s += tf_in[row][coeff].d(col);
        Sum[row][col] = s;
      }
    for (int row = 0; row < size; row++) Sum[row][row] += 1.0;
  }
```
This code is used in chunk 361.

358 ⟨get term (FadbadVarODE) 358⟩ ≡
```
  void FadbadVarODE::getTerm(iMatrix &Term, int i) const
  {
    for (int row = 0; row < size; row++)
      for (int col = 0; col < size; col++) Term[row][col] = tf_in[row][i].d(col);
  }
```
This code is used in chunk 361.

22.3 Files

360 ⟨fadbad_advar.h 360⟩ ≡
```
#ifndef Fadbad_Var_ODE
#define Fadbad_Var_ODE
#include "vnodeinterval.h"
#include "vector_matrix.h"
#include "ad_var.h"
#include "ffadiff.h"
#include "fadbad_intv.inc"
  namespace vnodelp {
    using namespace v_bias;
    using namespace v_blas;
    ⟨class FadbadVarODE 352⟩
  }
#endif
```

361 ⟨`fadbad_advar.cc` 361⟩ ≡

```
#include "fadbad_advar.h"
  namespace vnodelp {
    ⟨constructor (FadbadVarODE) 353⟩
    ⟨destructor (FadbadVarODE) 354⟩
    ⟨initialize Taylor coefficients (FadbadVarODE) 355⟩
    ⟨compute Taylor coefficients (FadbadVarODE) 356⟩
    ⟨sum terms (FadbadVarODE) 357⟩
    ⟨get term (FadbadVarODE) 358⟩
  }
```

22.4 Encapsulated FADBAD++ AD

Now we encapsulate all AD using FADBAD++ in

362 ⟨encapsulated FADBAD++ AD 362⟩ ≡

```
  class FADBAD_AD : public AD {

  public:
        FADBAD_AD(int n, Tfunction f, TFfunction tf)
    : AD(n, new FadbadODE(n, f), new FadbadVarODE(n, tf)),
        max_order(MaxLength − 2) { }
    FADBAD_AD(int n, Tfunction f, TFfunction tf, void *p)

    : AD(n, new FadbadODE(n, f, p), new FadbadVarODE(n, tf, p)),
        max_order(MaxLength − 2) { }
    virtual int getMaxOrder( ) const {
      return max_order;
    }
  private: const int max_order;
  };
```

This code is used in chunk 363.

and store the new class in

363 ⟨`fadbadad.h` 363⟩ ≡

```
#ifndef FADBADAD_H
#define FADBADAD_H
#include "fadbad_ad.h"
#include "fadbad_advar.h"
#include "allad.h"
  namespace vnodelp {
    ⟨ encapsulated FADBAD++ AD 362 ⟩
  }
#endif
```

Appendix A

Miscellaneous Functions

A.1 Vector output

We output a vector to the standard output by

366 ⟨print vector 366⟩ ≡
```
#include <iostream>
  using namespace std;
  template⟨class T⟩ void printVector(const T &v, const char *s = 0)
  {
    if (s) cout ≪ s ≪ "␣=␣" ≪ endl;
    for (unsigned int i = 0; i < v_blas::sizeV(v); i++) cout ≪ v[i] ≪ endl;
    cout ≪ endl;
  }
```
This code is used in chunk 137.

A.2 Check if an interval is finite

367 ⟨check finite 367⟩ ≡
```
  namespace v_bias {
    inline bool finite_interval(const interval &a)
    {
      return (isfinite(inf(a)) ∧ isfinite(sup(a)));
    }
  }
```
See also chunk 368.

This code is used in chunk 374.

We check if an interval vector contains finite intervals by

368 ⟨check finite 367⟩ +≡
```
  namespace v_blas {
    inline bool finite_interval(const iVector &a)
    {
      for (unsigned int i = 0; i < a.size( ); i++)
        if (¬v_bias::finite_interval(a[i])) return false;
      return true;
    }
  }
```

A.3 Message printing

We would like to print various messages.

369 ⟨ message printing 369 ⟩ ≡
```
#ifdef VNODE_DEBUG
#define printMessage(s)
  {
    cerr ≪ "\n␣***␣" ≪ __FILE__":" ≪ __LINE__ ≪ "␣␣␣" ≪ s ≪ endl;
  }
#else
#define printMessage(s)   (0)
#endif
#ifdef VNODE_DEBUG
#define exitOnError(s)
  {
    printMessage(s);
    exit(−1);
  }
#else
#define exitOnError(s)   (0)
#endif
```
This code is used in chunk 372.

370 ⟨ VNODE-LP message 370 ⟩ ≡
```
  void vnodeMessage(const char *s)
  {
    cerr ≪ endl ≪ "␣***␣VNODE-LP:␣" ≪ s ≪ endl;
  }
```
This code is used in chunk 373.

A.4 Check intersection

371 ⟨ check vector intersection 371 ⟩ ≡
```
  void checkIntersection(const iVector &a, const iVector &b)
  {
    v_bias::interval ai, bi;
    for (unsigned int i = 0; i < vnodelp::sizeV(a); i++) {
      ai = a[i];
      bi = b[i];
      if (vnodelp::disjoint(ai, bi))
        cout << "i␣=␣" << i << "␣␣␣␣" << endl << ai << endl << bi;
    }
  }
```

This code is used in chunk 373.

Files

372 ⟨debug.h 372⟩ ≡

```
#ifndef DEBUG_H
#define DEBUG_H
#include "basiclinalg.h"
  ⟨message printing 369⟩
  using namespace v_blas;
  void checkIntersection(const iVector &a, const iVector &b);
#endif
```

373 ⟨debug.cc 373⟩ ≡

```
#include <ostream>
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
  using namespace std;
#include "vnodeinterval.h"
#include "basiclinalg.h"
  using namespace std; namespace vnodelp {
    using namespace v_bias;
    using namespace v_blas;
    ⟨check vector intersection 371⟩
    ⟨VNODE-LP message 370⟩
  }
```

374 ⟨miscfuns.h 374⟩ ≡

```
#ifndef MISCFUNS_H
#define MISCFUNS_H
#include <cmath>
#include "basiclinalg.h"
  using namespace std; ⟨check finite 367⟩ namespace vnodelp {
        void vnodeMessage(const char *s);
      }
#endif
```

A.5 Timing

The *getTime* function returns the current user time. The *getTotalTime* subtracts the end time from the start time and returns the result.

375 ⟨vtiming.h 375⟩ ≡

```
#ifndef VTIMING_H
#define VTIMING_H
  double getTime( );
  double getTotalTime(double start_time, double end_time);
#endif
```

376 ⟨vtiming.cc 376⟩ ≡

```
#include <cassert>
#include <sys/times.h>
#include <unistd.h>
#include <ctime>
#include "vnodeinterval.h"
#include "vnoderound.h"
  using namespace std;
  static struct tms Tms;
  double getTime( )
  {
    times(&Tms);
    v_bias::round_nearest( );
    long int ClockTcks = sysconf(_SC_CLK_TCK);
    return (Tms.tms_utime)/(double(ClockTcks));
  }
  double getTotalTime(double start_time, double end_time)
  {
    assert(start_time ≤ end_time);
    v_bias::round_nearest( );
    return (end_time − start_time);
  }
```

Bibliography

[1] *BLAS — Basic Linear Algebra Subprograms*. `www.netlib.org/blas/`.

[2] *LAPACK — Linear Algebra PACKage*. `www.netlib.org/lapack/`.

[3] D. Achlioptas, *Setting 2 variables at a time yields a new lower bound for random 3-SAT*, Tech. Rep. MSR-TR-99-96, Microsoft Research, Microsoft Corp., One Microsoft Way, Redmond, WA 98052, December 1999.

[4] U. M. Ascher and L. R. Petzold, *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*, SIAM, Philadelphia, 1998.

[5] E. Auer, A. Kecskeméthy, M. Tändl, and H. Traczinski, *Interval algorithms in modelling of multibody systems*, in Numerical Software with Result Verification, vol. 2991 of Lecture Notes in Computer Lecture Notes in Computer Science, Springer-Verlag, 2004, pp. 132–159.

[6] C. Bendsten and O. Stauning, *FADBAD, a flexible C++ package for automatic differentiation using the forward and backward methods*, Tech. Rep. 1996-x5-94, Department of Mathematical Modelling, Technical University of Denmark, DK-2800, Lyngby, Denmark, August 1996.

[7] M. Berz, K. Makino, and J. Hoefkens, *Verified integration of dynamics in the solar system*, Nonlinear Analysis: Theory, Methods & Applications, 47 (2001).

[8] B. M. Brown, M. Langer, M. Marletta, C. Tretter, and M. Wagenhofer, *Eigenvalue bounds for the singular Sturm-Liouville problem with a complex potential*, J. Phys. A: Math. Gen., 36 (2003), pp. 3773–3787.

[9] G. F. Corliss and R. Rihm, *Validating an a priori enclosure using high-order Taylor series*, in Scientific Computing, Computer Arithmetic, and Validated Numerics, G. Alefeld and A. Frommer, eds., Akademie Verlag, Berlin, 1996, pp. 228–238.

[10] G. I. Hargreaves, *Interval analysis in MATLAB*, Tech. Rep. No. 416, Department of Mathematics, University of Manchester, UK, December 2002.

[11] W. Hayes, *Rigorous shadowing of numerical solutions of ordinary differential equations by containment*, PhD thesis, Department of Computer Science, University of Toronto, Toronto, Canada, 2001.

[12] W. Hayes and K. R. Jackson, *Rigorous shadowing of numerical solutions of ordinary differential equations by containment*, SIAM J. Num. Anal., to appear (2005).

[13] T. E. Hull and W. H. Enright, *A structure for programs that solve ordinary differential equations*, Tech. Rep. 66, Department of Computer Science, University of Toronto, May 1974.

[14] T. E. Hull, W. H. Enright, B. M. Fellen, and A. E. Sedgwick, *Comparing numerical methods for ordinary differential equations*, SIAM J. on Numerical Analysis, 9 (1972), pp. 603–637.

[15] M. Kieffer and E. Walter, *Nonlinear parameter and state estimation for cooperative systems in a bounded-error context*, in Numerical Software with Result Verification, vol. 2991 of Lecture Notes in Computer Lecture Notes in Computer Science, Springer-Verlag, 2004, pp. 107–123.

[16] O. Knüppel, *PROFIL/BIAS – a fast interval library*, Computing, 53 (1994), pp. 277–287.

[17] D. E. Knuth, *Literate Programming*, Center for the Study of Language and Information, Stanford, CA, USA, 1992.

[18] D. E. Knuth and S. Levy, *The CWEB System of Structured Documentation*, Addison-Wesley, Reading, Massachusetts, 1993.

[19] M. Lerch, G. Tischler, and J. Wolff von Gudenberg, FILIB++ — *interval library specification and reference manual*, Tech. Rep. 279, Universität Würzburg, Germany, 2001.

[20] R. J. Lohner, *Einschließung der Lösung gewöhnlicher Anfangs– und Randwertaufgaben und Anwendungen*, PhD thesis, Universität Karlsruhe, 1988.

[21] F. Mazzia and F. Iavernaro, *Test set for initial value problem solvers*, Tech. Rep. 40, Department of Mathematics, University of Bari, Italy, 2003. `http://pitagora.dm.uniba.it/~testset/`.

[22] H. Mukundan, K. H. Ko, T. Maekawa, T. Sakkalis, and N. M. Patrikalakis, *Tracing surface intersections with a validated ODE system solver*, in Proceedings of the Ninth EG/ACM Symposium on Solid Modeling and Applications, G. Elber and G. Taubin, eds., Eurographics Press, June 2004, June 2004.

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

[24] N. S. Nedialkov, *An interval method for initial value problems in linear ordinary differential equations*, SIAM Journal Numerical Analysis, (2004). Submitted.

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

[26] N. S. Nedialkov, K. R. Jackson, and G. F. Corliss, *Validated solutions of initial value problems for ordinary differential equations*, Applied Mathematics and Computation, 105 (1999), pp. 21–68.

[27] N. S. Nedialkov, K. R. Jackson, and J. D. Pryce, *An effective high-order interval method for validating existence and uniqueness of the solution of an IVP for an ODE*, Reliable Computing, 7 (2001), pp. 449–465.

[28] N. M. Patrikalakis, T. Maekawa, K. H. Ko, and H. Mukundan, *Surface to surface intersection*, in International CAD Conference and Exhibition, CAD'04, L. Piegl, ed., Thailand, May 2004.

[29] O. Stauning and C. Bendtsen, *FADBAD++ web page*, May 2003. FADBAD++ is available at `www.imm.dtu.dk/fadbad.html`.

[30] W. Tucker, *A rigorous ODE solver and Smale's 14th problem*, Found. Comput. Math., 2 (2002), pp. 53–117.

Index

List of Refinements

⟨ DETEST E1 37 ⟩ Used in chunk 40.
⟨ `E1.cc` 40 ⟩
⟨ FadbadODE destructor 341 ⟩ Used in chunk 350.
⟨ Krawczyk's iteration 162, 163 ⟩ Used in chunk 165.
⟨ Lorenz 19 ⟩ Used in chunks 20, 45, 48, 61, and 70.
⟨ VNODE accept solution 326 ⟩ Used in chunk 334.
⟨ VNODE-LP message 370 ⟩ Used in chunk 373.
⟨ a priori enclosure representation 181 ⟩ Used in chunk 184.
⟨ accept solution (HOE) 220 ⟩ Used in chunk 227.
⟨ accept solution (IHO) 276 ⟩ Used in chunk 293.
⟨ `ad_ode.h` 194 ⟩
⟨ `ad_var.h` 195 ⟩
⟨ `allad.h` 196 ⟩
⟨ `basic.cc` 27 ⟩
⟨ `basiclinalg.h` 137 ⟩
⟨ case 1 320 ⟩ Used in chunk 318.
⟨ case 2 321 ⟩ Used in chunk 318.
⟨ case 3 323 ⟩ Used in chunk 318.
⟨ changing the rounding mode (BIAS) 119 ⟩ Used in chunk 120.
⟨ changing the rounding mode (FILIB++) 118 ⟩ Used in chunk 120.
⟨ check finite 367, 368 ⟩ Used in chunk 374.
⟨ check if a number is even 281 ⟩ Used in chunk 293.
⟨ check if in the interior 140 ⟩ Used in chunk 151.
⟨ check if last step 318 ⟩ Used in chunk 296.
⟨ check if sorting is needed 287 ⟩ Used in chunk 284.
⟨ check if success 25 ⟩ Used in chunks 20, 35, 40, and 68.
⟨ check input correctness 298, 299, 300, 301, 302, 303, 304 ⟩ Used in chunk 296.
⟨ check vector inclusion 139, 141 ⟩ Used in chunk 151.
⟨ check vector intersection 371 ⟩ Used in chunk 373.
⟨ check t 316 ⟩ Used in chunk 315.
⟨ class FadbadODE 339 ⟩ Used in chunk 349.
⟨ class FadbadVarODE 352 ⟩ Used in chunk 360.
⟨ class HOE 204 ⟩ Used in chunk 226.
⟨ class IHO 235 ⟩ Used in chunk 292.
⟨ class MatrixInverse 156 ⟩ Used in chunk 172.
⟨ class VNODE 295 ⟩ Used in chunk 333.
⟨ close files 63 ⟩ Used in chunk 61.
⟨ compare function 288 ⟩ Used in chunk 291.
⟨ compute $(-1)^q q!p!/(p+q)!$ 282 ⟩ Used in chunk 293.
⟨ compute A^{-1} 158 ⟩ Used in chunk 173.
⟨ compute Q_{j+1} 273 ⟩ Used in chunk 265.
⟨ compute β 224 ⟩ Used in chunk 227.
⟨ compute $\boldsymbol{p}_j$ 206 ⟩ Used in chunk 217.
⟨ compute $\boldsymbol{u}_j$ and $\widetilde{\boldsymbol{y}}_j$ 207 ⟩ Used in chunk 217.
⟨ compute $h_{\min}$ 314 ⟩ Used in chunk 334.
⟨ compute IHO method coefficients 239 ⟩ Used in chunk 293.

⟨compute QR factorization 153⟩ Used in chunk 154.
⟨compute Taylor coefficients (FadbadODE) 343⟩ Used in chunk 350.
⟨compute Taylor coefficients (FadbadVarODE) 356⟩ Used in chunk 361.
⟨compute column norms of A 286⟩ Used in chunk 284.
⟨compute enclosure 324⟩ Used in chunk 296.
⟨compute initial box 161⟩ Used in chunk 160.
⟨compute initial stepsize 315⟩ Used in chunk 334.
⟨compute stepsize 210, 212, 214⟩ Used in chunk 217.
⟨compute tight enclosure 236⟩ Used in chunk 293.
⟨constants Lorenz 51⟩ Used in chunk 58.
⟨constructor (FadbadODE) 340⟩ Used in chunk 350.
⟨constructor (FadbadVarODE) 353⟩ Used in chunk 361.
⟨constructor (VNODE) 327⟩ Used in chunk 334.
⟨constructor IHO 274⟩ Used in chunk 293.
⟨constructor-destructor HOE 219⟩ Used in chunk 227.
⟨control class 200⟩ Used in chunk 201.
⟨`control.h` 201⟩
⟨corrector: compute $\boldsymbol{y}_{j+1}$ 247⟩ Used in chunk 236.
⟨create AD object 22⟩ Used in chunks 20, 45, 48, 61, and 68.
⟨create E1 39⟩ Used in chunk 40.
⟨create a solver 23⟩ Used in chunks 20, 35, 40, 45, 48, 58, 61, and 68.
⟨create file name 69⟩ Used in chunk 68.
⟨create problem object with parameters 54⟩ Used in chunk 58.
⟨create scalar AD object 34⟩ Used in chunk 35.
⟨create **Apriori** 182⟩ Used in chunk 185.
⟨create **Solution** 179⟩ Used in chunk 185.
⟨`debug.cc` 373⟩
⟨`debug.h` 372⟩
⟨decide 325⟩ Used in chunk 296.
⟨destructor (FadbadVarODE) 354⟩ Used in chunk 361.
⟨destructor (VNODE) 328⟩ Used in chunk 334.
⟨destructor IHO 275⟩ Used in chunk 293.
⟨determine direction 305, 306, 307⟩ Used in chunk 296.
⟨`detest_c3.cc` 71⟩
⟨do Krawczyk's iteration 165⟩ Used in chunk 160.
⟨encapsulated AD 191⟩ Used in chunk 196.
⟨encapsulated FADBAD++ AD 362⟩ Used in chunk 363.
⟨enclose each column 169⟩ Used in chunks 166 and 170.
⟨enclose the inverse of a matrix 166⟩ Used in chunk 173.
⟨enclose the inverse of an orthogonal matrix 170⟩ Used in chunk 173.
⟨enclose the solution to $Ax = b$ 160⟩ Used in chunk 173.
⟨`fadbad_ad.cc` 350⟩
⟨`fadbad_ad.h` 349⟩
⟨`fadbad_advar.cc` 361⟩
⟨`fadbad_advar.h` 360⟩
⟨`fadbadad.h` 363⟩

⟨sort columns 284⟩ Used in chunk 291.
⟨`sortcolumns.cc` 291⟩
⟨sum radii 164⟩ Used in chunk 165.
⟨sum terms (FadbadODE) 344⟩ Used in chunk 350.
⟨sum terms (FadbadVarODE) 357⟩ Used in chunk 361.
⟨template ODE function 18⟩
⟨tight enclosure representation 178⟩ Used in chunk 184.
⟨validate and select stepsize 317⟩ Used in chunk 296.
⟨validate existence and uniqueness 217⟩ Used in chunk 227.
⟨`vanderpol.cc` 74⟩
⟨vector and matrix types 122⟩ Used in chunk 127.
⟨vector operations 129⟩ Used in chunk 137.
⟨`vector_matrix.h` 127⟩
⟨`vnode.h` 335⟩
⟨`vnodeint.h` 333⟩
⟨`vnodeinterval.h` 115⟩
⟨`vnoderound.h` 120⟩
⟨`vtiming.cc` 376⟩
⟨`vtiming.h` 375⟩
⟨width 144⟩ Used in chunk 151.
⟨**MatrixInverse** constructor/destructor 171⟩ Used in chunk 173.