ARGONNE NATIONAL LABORATORY
Argonne, IL 60439 U.S.A.

ANL/MCS-TM-225

Users' Guide to ADIC 1.1*

Paul D. Hovland and Boyana Norris
hovland@mcs.anl.gov, norris@mcs.anl.gov

Mathematics and Computer Science Division

Technical Memorandum ANL/MCS-TM-225

July 2004

*This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38; by the National Aerospace Agency under Purchase Orders L25935D and L64948D; and by the National Science Foundation, through the Center for Research on Parallel Computation, under Cooperative Agreement No. CCR-9120008.

Argonne National Laboratory, a U.S. Dept. of Energy Office of Science laboratory, is operated by The University of Chicago under Contract W-31-109-ENG-38.

DISCLAIMER

Available electronically at http://www.osti.gov/bridge/
Available for a processing fee to U.S. Department of Energy and its contractors, in paper, from:
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831-0062
phone: (865) 576-8401
fax: (865) 576-5728
email: reports@adonis.osti.gov

Contents

Users' Guide to ADIC 1.1

by

Paul D. Hovland and Boyana Norris

Abstract

This guide describes the use of the Automatic Differentiation in C (ADIC) system. ADIC is a suite of tools and libraries that automates the process of generating derivatives for scientific programs. In the context of solving PDEs, optimizations, sensitivity analysis, and inverse problems, researchers often require the derivatives $\partial f/\partial x$ of a function f expressed as a program with respect to some input parameter(s) x. Automatic differentiation (AD) techniques augment the program with derivative computation by applying the chain rule of calculus to elementary operations in an automated fashion. ADIC uses sophisticated compiler techniques to augment the input C programs with derivative computation capability in an automatic fashion. It also provides a finer control of derivative code generation process via control scripts and pragmas. Another significant capability of ADIC is its component architecture, AIF, that allows ADIC's capability to be extended via plug-in modules.

For more information about ADIC, see **http://www.mcs.anl.gov/adic**.

Chapter 1

Introduction

Derivatives play an important role in computational science and engineering. Automatic differentiation (AD) is a technique for evaluating derivatives of a function written as a computer program by applying the chain rule of differential calculus at the elementary operation level. Since AD differentiates algorithms rather than formulas, it can deal with arbitrary programs representing these algorithms.

ADIC (Automatic Differentiation in C) is an AD tool to get accurate derivatives of programs written in ANSI-C. Given a set of C source files, ADIC produces a new set of C source files enhanced with derivative computation capabilities. The generated sources are made as portable as possible. In addition to ANSI-C, ADIC currently supports some C++ programs, although the coverage of the language is incomplete, notably with regard to templates, exceptions, and operator overloading. Readers are referred to [4] for a system-level overview of ADIC.

1.1 Terminology

We first define several terms that are used throughout this manual:

- *Independent variables* are program input variables with respect to which derivatives are desired.
- *Dependent variables* are program output variables whose derivatives are desired.
- A *derivative object* represents some derivative information, such as a vector of partial derivatives $(\partial z/\partial x_1, \ldots, \partial z/\partial x_n)$ of some variable z with respect to a vector x.
- Any program variable with which a derivative object is associated is called an *active* variable. Put another way, an *inactive* variable is a floating-point variable that does not have an associated derivative object, determined either through analysis or user annotations. ADIC currently considers all floating-point variables as active, unless explicitly specified otherwise by the user. We can extend this notion further for functions: an *inactive* function does not perform any derivative computation, even if floating-point computations are involved. The user can also specify inactive user-defined types. All elements of an inactive type are also inactive; for example, if a struct that contains floating-point fields has been declared inactive, no derivative objects would be associated with its components.

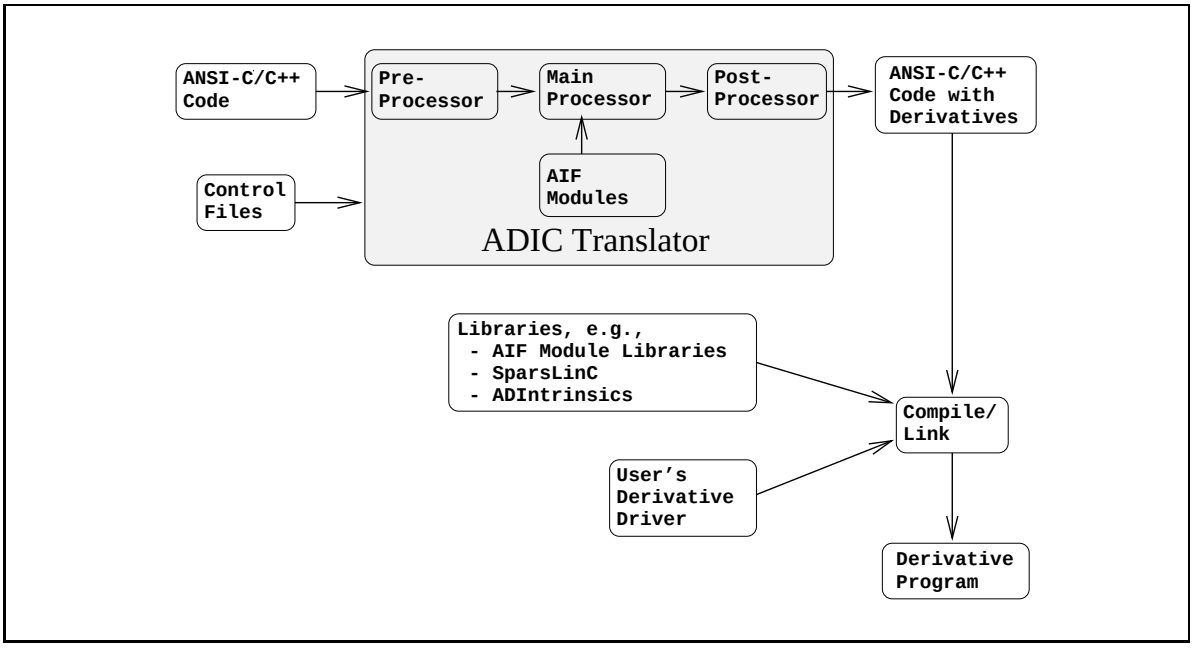

Figure 1.1: Generating derivative code with ADIC.

1.2 Overview of the ADIC System

The ADIC system comprises various programs, scripts, headers, and libraries. The processing stages of automatic differentiation using ADIC are shown in Figure 1.1. The main stages are as follows:

- A list of source files to be differentiated is collected. The list along with an optional control script is submitted to the ADIC system.
- For flexibility, the processing of source code is divided into several stages. The sources are first fed into the *preprocessor*, which deals with C preprocessor directives and macro expansions (like the C preprocessor `cpp`) but also embeds information required to recreate certain original directives and macros when generating the derivative code. The results are fed into the *main processor*, which generates the derivative code with the help of *AIF modules*. *AIF modules* are analogous to PC peripheral cards in the sense that they can be "plugged" into the ADIC "motherboard". Each module provides a certain defined functionality. Similarly to peripheral cards, AIF modules can be field installed by users. The details of AIF modules and their types are given in [4] and [3]. Here it is sufficient to note that AIF modules contain the core "rules" for performing automatic differentiation. The generated code is then fed into a series of *postprocessors* (in a pipelined fashion) that provide the ability to perform further textual transformations. One routinely used postprocessor is `purse`, a component of the ADIntrinsics subsystem to handle intrinsic function calls (e.g., `sqrt` and `cos`).
- The derivative code and a driver are then compiled and linked together to produce the final program. The ADIC system provides a set of headers and libraries required to compile and link the derivative code.

Chapter 2

Getting Started

In this chapter, we show the installation of the ADIC system and the required setup.

2.1 System Requirements

ADIC uses a standard C preprocessor to expand macros and handle preprocessor directives. The default preprocessor is **cpp** available on most Unix systems. The preprocessor of the GNU **gcc** compiler may be specified instead through a command-line option or a control script entry.

2.2 Installation

The ADIC system is distributed as a compressed archive file:

1. Note the machine and operating system you are using, and make sure you have the correct version of the software. The format of the ADIC distribution archive is "adic-x.xx-*machine*.tar.gz", where "x.xx" specifies the software version and *machine* specifies a particular machine and operating system combination, such as Linux for Intel x86 or Solaris on Sparcs. ADIC is supported on the following platforms: Solaris (SPARC), Linux (x86), HPUX, AIX (IBM RS6000), and IRIX. It is usually possible to compile ADIC-*generated* code on architectures not listed here. The source code for the libraries required for linking is included in the distribution.

2. Decide where to install the ADIC system. The archive file unpacks into a directory called `adic`. The following example installs ADIC under `install-dir`.

```
% cd install-dir
% gunzip -c adic-x.xx-solaris.tar.gz | tar xvf -
```

2.3 Directory Structure

In this section, we present an overview of the standard directory structure of the ADIC system, which is common for all platforms. The bin and lib directories contain subdirectories for each platform where architecture-dependent files are located. The standard platform directory names are `irix`, `linux`, `rs6000`, `hpux`, `solaris`, and `win32`.

`bin`: Various executables and scripts.

`bin/`*arch*: Architecture-dependent programs.

`include`: Header files for runtime library routines needed during the compilation of the derivative code.

`lib`: Precompiled runtime libraries and necessary files. Architecture-dependent files are in subdirectories.

`lib/`*arch*: Architecture-dependent library files.

`lib/src/`*module*: The source to the runtime libraries of the corresponding module.

`modules/`*module*: Module-specific data are stored under the corresponding *module* subdirectory; only modules that require such data have a subdirectory.

`doc, doc/`*module*: Documentation for ADIC and some *modules*.

`examples, examples/`*module*: Sample codes to test the system and modules.

In addition, the ADIC distribution currently contains the following AIF modules:

`Jacobian`: the standard Jacobian module.

`Hessian`: the standard Hessian module.

2.4 Setting Up the Environment

Before running `adiC`, you must set the `ADIC` and `ADIC_ARCH` environment variables and update the search path (examples below assume the use of the `csh` or `tcsh` shell):

1. Set the environment variable `ADIC` to the base directory where ADIC is installed. For example, if ADIC was unpacked in the /home/user directory, set the `ADIC` variable as follows:

   ```
   setenv ADIC /home/user/adic
   ```

2. Set the environment variable `ADIC_ARCH` to one of the following: `irix`, `linux`, `rs6000`, `hpux`, `solaris`, or `win32`, for example,

   ```
   setenv ADIC_ARCH solaris
   ```

   Alternatively, set the `ADIC_ARCH` variable to the type returned by the `$ADIC/bin/adicarch` command:

   ```
   setenv ADIC_ARCH ‘$ADIC/bin/adicarch‘
   ```

3. Add the ADIC executables directory to the command search path. The following example adds the ADIC executables directory as the first directory to be searched.

   ```
   setenv PATH $ADIC/bin/$ADIC_ARCH:$PATH
   ```

2.5 Building Runtime Libraries

Runtime libraries are distributed in compiled form for each supported platform (also referred to as a host machine). If the ADIC-generated derivative code is to be compiled for a non-host machine (hereafter referred to as the target machine), the libraries must be compiled for that machine. The source code for essential libraries is included in all distributions.

To compile all the runtime libraries:

1. copy the `lib/src` directory to the new platform.
2. type `make clean` to clean up the old object files and config files.
3. type `make` to build the library. GNU automake and autoconfig are used to generate the appropriate makefiles for the given platform.

Generally, in addition to the standard runtime libraries that come with the ADIC system, each AIF module also comes with its own set of runtime libraries. Currently, the standard ADIC libraries are

- **libadic.a:** This library is a container for a number of distinct libraries that may be used with the ADIC-generated code.
- **libADIntrinsics-C.a:** The source code is in the `lib/src/adintrinsics` directory.
- **libJacobian.a:** The source code is in the `lib/src/Jacobian` directory.
- **libHessian.a:** The source code is in the `lib/src/Hessian` directory.

As additional AIF modules are installed, their libraries must be built and installed for the desired target machine.

Chapter 3

Running ADIC: An Example

In this chapter, we show a simple example program that we would like to differentiate through ADIC. The example program consists of four source files: `func_main.c`, `func.h`, `func.c`, and `norm2.c`. We have deliberately tried to use various C features for illustration purposes. The source codes of this example come with the standard ADIC distribution and can be found in the `examples/Jacobian` subdirectory.

```
 1  #include <stdio.h>
 2  #include "func.h"
 3
 4  int main() {
 5      int    i, n;
 6      data_t data;
 7      double x[MAXLEN], y[MAXLEN], r;
 8
 9      /* read in values*/
10      scanf("%d", &n);
11      for (i = 0; i < n; i++) {
12          scanf("%lf %lf", &x[i], &y[i]);
13      }
14      data.len = n; data.x = x; data.y = y;
15
16      /*invoke the function*/
17      cos_angle(&data);
18
19      /*print the result*/
20      printf("%le\n", data.r);
21  }
```

Figure 3.1: File `func_main.c`.

The program computes the cosine of the angle between two vectors x and y. Figure 3.1 shows the main function that reads in the length and values of the two vectors from the user, packs the data into a `data_t` data structure, calls the `func` function that actually computes the value, and finally prints the result. The header file containing the definition of `data_t` is shown in Figure 3.2. The `func` function (shown in Figure 3.3) uses the formula $\frac{x \cdot y}{\|x\|_2 \cdot \|y\|_2}$ to calculate $\cos\theta$. It calls the `norm2` function (shown in Figure 3.4) to calculate the 2-norm of a vector.

Our goal is to generate derivative code that computes the Jacobian of the result with respect to x. The steps are as follows:

```
 1  #define MAXLEN  100
 2
 3  typedef struct {
 4     int    len;
 5     double *x, *y, r;
 6  } data_t;
 7
 8  void cos_angle(data_t*);
 9  double norm2(int, double*);
```

Figure 3.2: File `func.h`.

```
 1  #include <math.h>
 2  #include "func.h"
 3
 4  void cos_angle(data_t* pdata)
 5  {
 6      double *x = pdata->x, *y = pdata->y, dotp, norm_x, norm_y;
 7      int i;
 8
 9      for (dotp = 0.0, i = 0; i < pdata->len; i++, dotp += *x++ * *y++)
10          ;
11      pdata->r = dotp/(norm2(pdata->len,pdata->x)*norm2(pdata->len,pdata->y));
12      return;
13  }
```

Figure 3.3: File `func.c`.

```
 1  #include <math.h>
 2
 3  double norm2(int n, double *x)
 4  {
 5    double norm = 0.0;
 6    int    i;
 7
 8    for ( i=0; i<n; i++ ) {
 9        norm += x[i]*x[i];
10    }
11    norm = sqrt(norm);
12    return norm;
13  }
```

Figure 3.4: File `norm2.c`.

1. Properly set the environment variables and the search path for ADIC executables as specified in Section 2.4.

2. Generate the first-order derivative code for the functions by invoking ADIC with the following options (`-v` turns on the verbose mode, and `-d Jacobian` specifies the `Jacobian` module):

   ```
   % adiC -vd Jacobian func.c
   % adiC -vd Jacobian norm2.c
   ```

   or by combining the source files on the same command line (this is not recommended for large source files):

   ```
   % adiC -vd Jacobian func.c norm2.c
   ```

   This generates files called `func.ad.c` and `norm2.ad.c` shown in Figure 3.6 and 3.7. In addition, ADIC also generates a header, `ad_deriv.h` (shown in Figure 3.8), which is included in the generated codes. The generated files have been slightly reformatted for readability.

3. (*OPTIONAL*) Instead of, or in addition to, specifying source files at the command line, you can create a control script called `func.init` that lists the files to be differentiated (arbitrary script names can be used, but the init extension is convenient for distinguishing ADIC control script files). We want to differentiate the two functions `func` and `norm2`. The control script is shown in Figure 3.5. The source files do not have to be listed in the control script; they may be specified on the command line following the script name. A control script can be specified on the command line using the `-i` option, for example:

   ```
   % adiC -vd Jacobian -i func.init
   ```

```
[SOURCE_FILES]
    func.c
    norm2.c
```

Figure 3.5: File `func.init`.

4. Create a driver, based on the `main` function in `func_main.c`, that calls the derivative function. Figure 3.9 shows the driver, which sets up the independent variables, calls the derivative function, and extracts the derivative values.

5. Compile and link all the files, making sure the path to the appropriate include directory is set and the necessary libraries are linked. Also, define the `ad_GRAD_MAX` macro which designates the maximum number of independent variables. If this is left undefined, the maximum number of independent variables defaults to 5. An example of compiling and linking the differentiated files and the driver follows:

   ```
   % gcc -I$ADIC/include -Dad_GRAD_MAX=3 -c func.ad.c norm2.ad.c func_driver.c
   % gcc -o ad_func norm2.ad.o func.ad.o func_driver.o -L$ADIC/lib/$ADIC_ARCH
         -lADIntrinsics-C -lJacobian -lm
   ```

```
 1  /************************* DISCLAIMER ********************************/
 2  /*                                                                   */
 3  /*   This file was generated on 01/08/01 15:16:12 by the version of  */
 4  /*   ADIC compiled on  12/18/00 16:11:29                             */
 5  /*                                                                   */
 6  /*   ADIC was prepared as an account of work sponsored by an         */
 7  /*   agency of the United States Government and the University of    */
 8  /*   Chicago.  NEITHER THE AUTHOR(S), THE UNITED STATES GOVERNMENT   */
 9  /*   NOR ANY AGENCY THEREOF, NOR THE UNIVERSITY OF CHICAGO, INCLUDING */
10  /*   ANY OF THEIR EMPLOYEES OR OFFICERS, MAKES ANY WARRANTY, EXPRESS  */
11  /*   OR IMPLIED, OR ASSUMES ANY LEGAL LIABILITY OR RESPONSIBILITY FOR */
12  /*   THE ACCURACY, COMPLETENESS, OR USEFULNESS OF ANY INFORMATION OR  */
13  /*   PROCESS DISCLOSED, OR REPRESENTS THAT ITS USE WOULD NOT INFRINGE */
14  /*   PRIVATELY OWNED RIGHTS.                                         */
15  /*                                                                   */
16  /*********************************************************************/
17  #include "ad_deriv.h"
18  #include <math.h>
19  #include "adintrinsics.h"
20  typedef struct  {
21    int  len;
22    DERIV_TYPE  *x, *y, r;
23  } data_t;
24  void   ad_cos_angle(data_t  *);
25  void   ad_norm2(DERIV_TYPE  *ad_var_ret,int  ,DERIV_TYPE  *);
26  void   ad_cos_angle(data_t  *pdata) {
27    DERIV_TYPE  *x = pdata->x, *y = pdata->y, dotp, norm_x, norm_y;
28    int  i;
29    int  ad_var_0;
30    DERIV_TYPE  *ad_var_1, *ad_var_2, ad_var_3, ad_var_4;
31    double  ad_loc_0, ad_loc_1;
32    double  ad_adj_0, ad_adj_1, ad_adj_2, ad_adj_3;
33    ad_grad_axpy_0(&(dotp));
34    DERIV_val(dotp) = 0.0;
35    for (i = 0; i < pdata->len; ) {
36      ad_var_0 = i++;
37      ad_var_1 = x++;
38      ad_var_2 = y++;
39      ad_loc_0 = DERIV_val(*ad_var_1) * DERIV_val(*ad_var_2);
40      ad_loc_1 = DERIV_val(dotp) + ad_loc_0;
41      ad_grad_axpy_3(&(dotp), 1.000000000000000e+00, &(dotp), DERIV_val(*ad_var_2),
42                     &(*ad_var_1), DERIV_val(*ad_var_1), &(*ad_var_2));
43      DERIV_val(dotp) = ad_loc_1;
44    }
45    ad_norm2( &ad_var_3, pdata->len, pdata->x);
46    ad_norm2( &ad_var_4, pdata->len, pdata->y);
47    ad_loc_0 = DERIV_val(ad_var_3) * DERIV_val(ad_var_4);
48    ad_loc_1 = DERIV_val(dotp) / ad_loc_0;
49    ad_adj_0 =  -ad_loc_1 / ad_loc_0;
50    ad_adj_1 = 1.000000000000000e+00 / ad_loc_0;
51    ad_adj_2 = DERIV_val(ad_var_3) * ad_adj_0;
52    ad_adj_3 = DERIV_val(ad_var_4) * ad_adj_0;
53    ad_grad_axpy_3(&(pdata->r), ad_adj_1, &(dotp), ad_adj_3,
54                   &(ad_var_3), ad_adj_2, &(ad_var_4));
55    DERIV_val(pdata->r) = ad_loc_1;
56  }
57  void   ad_AD_Init(int  arg0) { ad_AD_GradInit(arg0); }
58  void   ad_AD_Final() { ad_AD_GradFinal(); }
```

Figure 3.6: File `func.ad.c`.

```
 1  /************************ DISCLAIMER ********************************/
 2  /*                                                                  */
 3  /*   This file was generated on 01/08/01 15:16:37 by the version of   */
 4  /*   ADIC compiled on  12/18/00 16:11:29                            */
 5  /*                                                                  */
 6  /*   ADIC was prepared as an account of work sponsored by an        */
 7  /*   agency of the United States Government and the University of    */
 8  /*   Chicago.  NEITHER THE AUTHOR(S), THE UNITED STATES GOVERNMENT   */
 9  /*   NOR ANY AGENCY THEREOF, NOR THE UNIVERSITY OF CHICAGO, INCLUDING */
10  /*   ANY OF THEIR EMPLOYEES OR OFFICERS, MAKES ANY WARRANTY, EXPRESS  */
11  /*   OR IMPLIED, OR ASSUMES ANY LEGAL LIABILITY OR RESPONSIBILITY FOR */
12  /*   THE ACCURACY, COMPLETENESS, OR USEFULNESS OF ANY INFORMATION OR  */
13  /*   PROCESS DISCLOSED, OR REPRESENTS THAT ITS USE WOULD NOT INFRINGE */
14  /*   PRIVATELY OWNED RIGHTS.                                        */
15  /*                                                                  */
16  /********************************************************************/
17  #include "ad_deriv.h"
18  #include <math.h>
19  #include "adintrinsics.h"
20  void ad_norm2(DERIV_TYPE  *ad_var_ret,int  n,DERIV_TYPE  *x) {
21    DERIV_TYPE  norm;
22    int  i;
23    int  ad_var_0;
24    double  ad_adji_0, ad_loc_0, ad_loc_1;
25
26    static int g_filenum = 0;
27    if (g_filenum == 0) {
28      adintr_ehsfid(&g_filenum, __FILE__, "ad_norm2");
29    }
30    ad_grad_axpy_0(&(norm));
31    DERIV_val(norm) = 0.0;
32    for (i = 0;     i < n;     )    {
33      ad_loc_0 = DERIV_val(x[i]) * DERIV_val(x[i]);
34      ad_loc_1 = DERIV_val(norm) + ad_loc_0;
35      ad_grad_axpy_3(&(norm), 1.000000000000000e+00, &(norm),
36                     DERIV_val(x[i]), &(x[i]), DERIV_val(x[i]), &(x[i]));
37      DERIV_val(norm) = ad_loc_1;
38      ad_var_0 = i++;
39    }
40    DERIV_val(norm) = sqrt( DERIV_val(norm)); /*sqrt*/
41    if (  DERIV_val(norm) > 0.0 ) {
42      ad_adji_0 = 1.0 / (2.0 * DERIV_val(norm));
43    }
44    else {
45      adintr_sqrt(1, g_filenum, __LINE__, & ad_adji_0);
46    }
47    ad_grad_axpy_1(&(norm), ad_adji_0, &(norm));
48    ad_grad_axpy_copy(&(*ad_var_ret), &(norm));
49    DERIV_val(*ad_var_ret) = DERIV_val(norm);
50    return;
51  }
```

Figure 3.7: File `norm2.ad.c`.

3.1 A Closer Look at the Preparation

Specifying source files. We do not process the driver `func_main.c` through ADIC since it does not contain any code that needs to be differentiated. We also do not separately process `func.h` since the header will be indirectly processed through `func.c`, which includes it: ADIC inlines the differentiated header in the generated code*.

Writing a control script. A control script is a text file optionally used to fine-tune the behavior of ADIC. It contains a set of bindings (key-value pairs) organized into sections. For example, the list of functions or variables that should be made inactive can be specified in the script. Control scripts can be nested. See Chapter 7 for further details and a description of all valid bindings.

Selecting a derivative AIF module. There are many different ways of computing and propagating derivatives through exploiting the chain rule associativity. These differentiation "rules" are embodied in AIF modules. ADIC makes use of one or more of these modules for generating the derivative code. The user must select a module when invoking ADIC. The current distribution includes the `Jacobian` and `Hessian` modules, which compute the first and second derivatives, respectively.

3.2 A Closer Look at the Generated Code

ADIC makes the following changes in the course of generating the code that compute the derivatives. See Chapter 8 for further details on compiling ADIC-generated derivative code.

Generated Files. For each source file name with suffix `.x` (e.g., `foo.c`), ADIC generates a corresponding derivative source file with the suffix `.ad.x` (e.g., `foo.ad.c`). ADIC also generates a header file, `ad_deriv.h` (or rather, `<prefix>deriv.h`, where `<prefix>` can be specified in the control script, see Chapter 7; the default is `ad_`), automatically included by all the generated source files. The header contains appropriate ADIC type declarations and prototypes. Different header files are generated depending on the derivative module chosen and also the options selected when ADIC is invoked. A command line option can disable the generation of the header. To see a summary of all command line options, run ADIC with no arguments.

Type change. ADIC changes the type of `double` or `float` variables into `DERIV_TYPE` defined in `ad_deriv.h` shown in Figure 3.8. In this case, `DERIV_TYPE` is defined as a structure containing a floating-point value and an array of floating-point values, corresponding to the Jacobian of the variable with respect to the independent variables. The actual definition of the type depends on how ADIC is invoked. The type change results in the change of any data structure containing an (active) floating-point variable. It is possible to disable certain types from being changed. This can be done either in a control script (see Chapter 7 or by changing the type name in the source file from float or double to InactiveFloat or InactiveDouble, respectively. This is useful when we know variables of those types are inactive and thus need not have any associated derivative objects.

Function name change. At times, derivative code must coexist with the original functions. This is especially the case in libraries containing both the original and differentiated versions. In order to reduce name conflicts

*There is an option that specifies that header files should not be inlined, in which case those header files must be processed through ADIC separately.

```
 1  /************************* DISCLAIMER ********************************/
 2  /*                                                                   */
 3  /*   This file was generated on 01/09/01 11:16:01 by the version of  */
 4  /*   ADIC compiled on  12/18/00 16:11:29                             */
 5  /*                                                                   */
 6  /*   ADIC was prepared as an account of work sponsored by an         */
 7  /*   agency of the United States Government and the University of    */
 8  /*   Chicago.  NEITHER THE AUTHOR(S), THE UNITED STATES GOVERNMENT   */
 9  /*   NOR ANY AGENCY THEREOF, NOR THE UNIVERSITY OF CHICAGO, INCLUDING */
10  /*   ANY OF THEIR EMPLOYEES OR OFFICERS, MAKES ANY WARRANTY, EXPRESS  */
11  /*   OR IMPLIED, OR ASSUMES ANY LEGAL LIABILITY OR RESPONSIBILITY FOR */
12  /*   THE ACCURACY, COMPLETENESS, OR USEFULNESS OF ANY INFORMATION OR  */
13  /*   PROCESS DISCLOSED, OR REPRESENTS THAT ITS USE WOULD NOT INFRINGE */
14  /*   PRIVATELY OWNED RIGHTS.                                         */
15  /*                                                                   */
16  /*********************************************************************/
17  #if !defined(AD_DERIV_H)
18  #define AD_DERIV_H
19
20  typedef double InactiveDouble;
21  typedef float InactiveFloat;
22
23  #if !defined(ad_GRAD_PTR)
24  #define ad_GRAD_PTR 0
25  #endif
26  #if !defined(ad_GRAD_MAX)
27  #define ad_GRAD_MAX 5
28  #endif
29  #define AD_INIT_MAP()
30  #define AD_CLEANUP_MAP()
31  #define AD_GET_DERIV_OBJ(x) ((void*)(&x.value+1))
32  #define AD_FREE_DERIV_OBJ(x)
33  typedef struct {
34  double value;
35  double  grad[ad_GRAD_MAX];
36  } DERIV_TYPE;
37
38  #define DERIV_val(a) ((a).value)
39  #define DERIV_grad(a) ((a).grad)
40  #define _FLOAT_INITIALIZER_(x) { x, 0.0 }
41
42  void ad_AD_Init(int);
43  void ad_AD_Final();
44  #include "ad_grad.h"
45  #define nullFunc(x) 0
46  #endif
```

Figure 3.8: File `ad_deriv.h`.

```
 1  #include "ad_deriv.h"
 2  #include <stdio.h>
 3
 4  #define MAXLEN 100
 5  typedef struct {
 6     int         len;
 7     DERIV_TYPE* x, *y, r;
 8  } data_t;
 9  void ad_cos_angle(data_t*);
10
11  int main() {
12      int    i, n;
13      double grad[ad_GRAD_MAX], t1, t2;
14      data_t data;
15      DERIV_TYPE x[MAXLEN], y[MAXLEN], r;
16
17      /*inialize*/
18      ad_AD_Init(ad_GRAD_MAX);
19
20      /* read in values*/
21      scanf("%d", &n);
22      for (i = 0; i < n; i++) {
23          scanf("%lf %lf", &t1, &t2);
24          ad_grad_axpy_0(DERIV_grad(x[i]));
25          DERIV_val(x[i]) = t1;
26          ad_grad_axpy_0(DERIV_grad(y[i]));
27          DERIV_val(y[i]) = t2;
28      }
29      data.len = n; data.x = x; data.y = y;
30
31      /*set independent variables*/
32      ad_AD_SetIndepArray(x, n);
33      ad_AD_SetIndepDone();
34
35      /*invoke the function*/
36      ad_cos_angle(&data);
37
38      /*extract the gradient vector*/
39      ad_AD_ExtractGrad(grad, data.r);
40
41      /*print the result and partials*/
42      printf("%le\n", DERIV_val(data.r));
43      for (i = 0; i < n; i++) {
44          printf ("%le\n", grad[i]);      /*partials*/
45      }
46
47      /*cleanup*/
48      ad_AD_Final();
49  }
```

Figure 3.9: File `func_driver.c`.

with the original source, ADIC prepends each function name in the source with a prefix. The default prefix is `ad_`, but a different prefix may be specified in a control script. See Chapter 10 for details.

Function type change. ADIC takes each function definition and augments it with derivative computations. As part of this process, if the original function returns a floating-point value, then the function is turned into a *procedure* (e.g., a function that doesn't return a value) that returns the result through the first argument. All calls to the function are suitably modified. The `norm2` function shows this process. This modification does not occur for overloaded operators in C++.

Intrinsic function calls. The `ad_norm2` function shown in Figure 3.7 contains some extra code in handling the `sqrt` intrinsic call. Functions such as `sqrt` are not continuously differentiable (e.g., `sqrt(x)` when `x` equals zero). To alert the user to such an occurrence, ADIC can check for it and print a warning message. In other cases, the user may want to skip these checks to improve the performance of derivative computations.

To support checking and reporting of potential exceptions, ADIC includes the header `adintrinsics.h` whenever `math.h` is included in the source. ADIC also provides a reasonable default value for the derivatives so that the execution can proceed. In fact, in most cases, the evaluation of an intrinsic at a point of nondifferentiability does not compromise the overall result.

Generated special functions. ADIC generates two special functions, `ad_AD_Init` and `ad_AD_Final`, that should be called from the driver. These functions are defined in the first generated source file (for the example, in `func.ad.c` file). The user can prevent ADIC from generating these functions by using a command line option. The `ad_AD_Init` function performs initializations necessary before any other differentiation steps. In the example above, the `ad_AD_Init` function makes a single call to `ad_AD_GradInit` which is defined in the `Jacobian` module library. The `ad_AD_Final` function performs any cleanup that may be necessary.

3.3 A Closer Look at the Driver

The driver sets up independent variables, calls the derivative functions, and extracts the derivative values. Independent variables can be nominated at runtime, rather than at translation time. Thus there is no need to rerun ADIC if the set of independent variables changes. This strategy facilitates the construction of differentiated libraries in which one cannot know in advance which of the inputs will be elected as independents.

The `func_driver.c` file shown in Figure 3.9 contains the calls to various ADIC functions. The independent variables are specified through a series of calls to `ad_AD_SetIndepArray` and `ad_AD_SetIndep`, and terminated by the call to `ad_AD_SetIndepDone`. The call

```
ad_AD_SetIndepArray(x, n);
```

specifies that `n` consecutive elements of the array `x` are independent variables. The call

```
ad_AD_ExtractGrad(grad, r);
```

extracts the Jacobian of the result `r` into the array `grad`. The call

```
ad_AD_ExtractVal(val, r);
```

extracts the result itself into the floating-point variable `val`.

Except for the variables `grad` and `val`, all the other floating-point variables are of type `DERIV_TYPE`. Manually writing drivers can be tedious and error prone. For example, we had to make sure that `MAXLEN` defined on

line 4 of Figure 3.9 was the same as defined in `func.h`. We can use ADIC to ease the task of writing this driver. First, we rewrite `func_main.c` to include the necessary calls as shown in Figure 3.10. Note that we declared `grad` array as `InactiveDouble` to make it inactive (`InactiveDouble` is defined internally by ADIC as an inactive type). The result of running it through ADIC is shown in Figure 3.11. This code can be then directly compiled and linked.

```
 1  #include <stdio.h>
 2  #include "func.h"
 3
 4  int main() {
 5      int    i, n;
 6      data_t data;
 7      InactiveDouble grad[MAXLEN], t1, t2, val;
 8      double x[MAXLEN], y[MAXLEN], r;
 9
10  #if defined(ADIC)
11      AD_Init(ad_GRAD_MAX);
12  #endif
13
14      /* read in values*/
15      scanf("%d", &n);
16      for (i = 0; i < n; i++) {
17          scanf("%lf %lf", &t1, &t2);
18          x[i]= t1;
19          y[i] = t2;
20      }
21      data.len = n; data.x = x; data.y = y;
22
23  #if defined(ADIC)
24      AD_SetIndepArray(x, n);
25      AD_SetIndepDone();
26  #endif
27
28      /*invoke the function*/
29      cos_angle(&data);
30
31  #if defined(ADIC)
32      AD_ExtractGrad(grad, data.r);
33      AD_ExtractVal(val, data.r);
34  #endif
35
36      /*print the result*/
37      printf("%le\n", val);
38  #if defined(ADIC)
39      for (i = 0; i < n; i++) {
40  printf ("%le\n", grad[i]);      /*partials*/
41      }
42      AD_Final();
43  #endif
44  }
```

Figure 3.10: File `func_main2.c`.

```
 1  /************************* DISCLAIMER ********************************/
 2  /*                                                                   */
 3  /*   This file was generated on 01/08/01 16:05:16 by the version of  */
 4  /*   ADIC compiled on  12/18/00 16:11:29                             */
 5  /*                                                                   */
 6  /*   ADIC was prepared as an account of work sponsored by an         */
 7  /*   agency of the United States Government and the University of    */
 8  /*   Chicago.  NEITHER THE AUTHOR(S), THE UNITED STATES GOVERNMENT   */
 9  /*   NOR ANY AGENCY THEREOF, NOR THE UNIVERSITY OF CHICAGO, INCLUDING */
10  /*   ANY OF THEIR EMPLOYEES OR OFFICERS, MAKES ANY WARRANTY, EXPRESS  */
11  /*   OR IMPLIED, OR ASSUMES ANY LEGAL LIABILITY OR RESPONSIBILITY FOR */
12  /*   THE ACCURACY, COMPLETENESS, OR USEFULNESS OF ANY INFORMATION OR  */
13  /*   PROCESS DISCLOSED, OR REPRESENTS THAT ITS USE WOULD NOT INFRINGE */
14  /*   PRIVATELY OWNED RIGHTS.                                         */
15  /*                                                                   */
16  /*********************************************************************/
17  #include "ad_deriv.h"
18  #include <stdio.h>
19  typedef struct  {
20    int  len;
21    DERIV_TYPE  *x, *y, r;
22  } data_t;
23  void   ad_cos_angle(data_t  *);
24  void   ad_norm2(DERIV_TYPE  *ad_var_ret,int  ,DERIV_TYPE  *);
25  int  main() {
26    int  i, n;
27    data_t  data;
28    InactiveDouble  grad[100], t1, t2, val;
29    DERIV_TYPE  x[100], y[100], r;
30    int  ad_var_0, ad_var_1;
31    ad_AD_Init(ad_GRAD_MAX);
32    scanf("%d",  &n);
33    for (i = 0;     i < n;     )    {
34      scanf("%lf %lf",  &t1,  &t2);
35      ad_grad_axpy_0(&(x[i]));
36      DERIV_val(x[i]) = t1;
37      ad_grad_axpy_0(&(y[i]));
38      DERIV_val(y[i]) = t2;
39      ad_var_0 = i++;
40    }
41    data.len = n;
42    data.x = x;
43    data.y = y;
44    ad_AD_SetIndepArray(x, n);
45    ad_AD_SetIndepDone();
46    ad_cos_angle( &data);
47    ad_AD_ExtractGrad(grad, data.r);
48    ad_AD_ExtractVal(val, data.r);
49    val = DERIV_val(data.r);
50    printf("%le\n", val);
51    for (i = 0;     i < n;     )    {
52      printf("%le\n", grad[i]);
53      ad_var_1 = i++;
54    }
55    ad_AD_Final();
56  }
57  void   ad_AD_Init(int  arg0) { ad_AD_GradInit(arg0); }
58  void   ad_AD_Final() { ad_AD_GradFinal(); }
```

Figure 3.11: File `func_main2.ad.c`.

Chapter 4

Using ADIC

4.1 Preparing the Source

In this chapter we describe the invocation semantics of ADIC.

Working with pre-ANSI C. ADIC expects ANSI-C source. If the user program uses the old K&R style function declarations, it should be first run through GNU **protoize** to convert to using ANSI-C style declarations and generate proper function prototypes.

```
% protoize file1 file2 ...
```

Working with C++. ADIC can also handle some C++ source. However, the coverage of the C++ language is not complete at this point. A command-line option must be specified when processing C++ source. For a summary of all command line options, run ADIC with no arguments.

Working with multiple source files. Multiple source files can be specified with ADIC. Since ADIC currently does not perform any interprocedural analysis, it makes a little difference whether the source files are specified all at once or one per invocation. It is recommended, however, that each nontrivial source file is processed separately.

Working with header files. The system headers (included using `#include` directives with the filename enclosed in angle brackets) mostly deal with the noncomputational system functions and therefore do not need to be differentiated. ADIC assumes that all functions and global variables declared inside system headers are inactive, which means that any user code referencing them is processed accordingly. The sole exception is `math.h`, which declares intrinsic numeric functions. The declarations in user headers (included using `#include` directives with the filename enclosed in quotes) are not made inactive by default (although the user can explicitly make them inactive through a control script). It is important that all system header `#include` directives use angle brackets to prevent incorrect code generation or inability to process the source file.

In addition, ADIC by default inlines user headers in the source that includes them. By using the `-u` command line option, user headers will not get inlined but instead the generated source will contain `#include` directives for the differentiated user headers. If this option is used, the header file must be processed separately by ADIC. See Chapter 9 for further details.

For the example in Chapter 3, if we use the command

```
% adiC -uv -d Jacobian -i func.init
```

and add the `func.h` entry to the `func.init`, we get the following `func.ad.c` (the body of the function is omitted since it is unchanged from the previous version):

```
#include "ad_deriv.h"
#include <math.h>
#include "adintrinsics.h"
#include "func.ad.h"

void ad_cos_angle(data_t *pdata) {
    ...
}
```

The differentiated header, `func.ad.h`, contains

```
typedef struct  {
  int        len;
  DERIV_TYPE *x, *y, r;
} data_t;

void ad_cos_angle(data_t*);
void ad_norm2(DERIV_TYPE*, int, DERIV_TYPE*);
```

Selecting different prefixes. In order to reduce name conflicts with the original source code, ADIC can prepend each type, function, and variable name in the with a prefix. Different prefixes may be specified for different name types. For example, we can specify one prefix for all the function names and another for all the type names. The prefixes are specified in the control script. See Chapter 10 for details.

Incorporating domain knowledge. The user may know that certain functions do not need to be augmented with derivatives (for example, debugging functions). When these function names are specified in the control script, ADIC will not augment them with derivatives. If a function is inactive, then all arguments are made inactive. ADIC ensures that any call to the function with active arguments are properly handled; that is, only the value part of an active variable is passed, rather than the entire `DERIV_TYPE`.

Checking. If a source contains a call to an intrinsic function such as `sin`, ensure that `math.h` is included using angle brackets. Just declaring the proper prototype of the intrinsic function by itself is not enough.

4.2 Invoking ADIC

Invoking ADIC to generate derivative code is quite simple. The basic format is:

```
% adiC -OPTIONS [file1 ...]
```

When no option or invalid options are specified, the usage information will be printed. The command-line options are as follows:

-d module_name The user can specify the derivative AIF module that determines the type of derivatives that will be generated. If this option is not specified, then ADIC will print out the available modules.

-v This turns on the verbose mode and details various phases of ADIC processing.

-a Instead of using `cpp` preprocessor, the user can instruct ADIC to use the GNU `gcc` preprocessor with this flag.

-t This specifies that the special initialization and finalization functions should not be created. This option generally is used when a number of files that will be linked together are being differentiated and only one of them needs to contain the special function definitions.

-C If the user has a C++ program, this option should be specified.

-i control_file This specifies a control script named `control_file`.

-I include_dir This specifies a directory to be searched to find `include` files. This directory is searched before the standard directories. Multiple **-I** options are allowed.

-k This directs ADIC to regenerate inactive function definitions completely.

-D This specifies a macro definition that will be passed down to the C preprocessor. For example, the user can use this to select the proper conditional directives. Multiple **-D** options are allowed. The macros may also be specified in the control file (under the `DEFINES` section).

-h This specifies that the special header file `ad_deriv.h` should not be created.

-g This directs ADIC to use guards around unmodified global declarations.

-s This specifies the silent operation. It will not print any messages, unless there is an error or a warning.

-u This specifies that user header files should not be expanded inline. In this case, the user should make sure the header files are processed through ADIC.

While processing, ADIC generates various working files, which will automatically be deleted unless errors are encountered.*

* After errors are fixed and ADIC runs successfully, the working files will be deleted.

Chapter 5

The Jacobian Module

The `Jacobian` module can be used to compute the Jacobian J of m dependent variables with respect to n independent variables. The cost of the computation is on the order of n times the function evaluation. By appropriately initializing the gradient objects of the independent variables (seed matrix M), we can actually compute the product $J * M$ at the cost of computing m columns of M. Hence, we can directly compute m directional derivatives.

The default `Jacobian` module defines one derivative object called `grad` associated with each active variable. `grad` is an array of float values. The array size is fixed at compile time and is equal to the `ad_GRAD_MAX` macro value. The user must define this macro at compile time. The derivative object of x is referenced by `DERIV_`*`grad`*`(`x`)`.

The following set of bindings is available (specified in the control script). See Chapter 7 for details of the control script format.

GRAD_INLINE = $\{$**no,yes**$\}$ The accumulation loop may be either inlined or performed through a function call.

GRAD_MAX = 0..n This value specifies the default maximum size of the gradient object. This value can be overridden during the compilation stage.

GRAD_CALL_PARAMS = $\{$**full,lite**$\}$ This binding is not used currently.

GRAD_MAX_ARITY = 0..n The runtime library that comes with the `Jacobian` module for handling partial derivative accumulation has an optimized version for each number of unique variables on the right-hand side, up to n. Beyond that, a general-purpose routine performs the accumulation. The user should not change this setting in general.

GRAD_NO_OPTIMIZE = $\{$**0,1**$\}$ Except for debugging purposes, this value should always be 0.

In order to compute the Jacobian, the following calls may be used:

ad_AD_SetIndep(var) This call increases the number of independent variables by one. It sets the variable `var` to be the ith independent variable. `var` may be either local or global. It also initializes the derivative object associated with this variable (e.g., sets the ith entry of the derivative object to 1.0 and the rest to 0.0). The order of calls to this function mirrors the order of Jacobian values in derivative objects.

ad_AD_SetIndepArray(var, n) This call is used to set an array of n active variables to be independent variables. Starting from the first element of the array, each succeeding element is initialized as the next independent

variable (e.g., the derivative objects are initialized correctly). The number of independent variables is thus increased by n.

ad_AD_SetIndepDone() This call signifies that all independent variables have been specified.

ad_AD_ResetIndep() This call can be called to set a new set of independent variables.

ad_AD_ExtractGrad(grad, var) This call extracts the whole Jacobian from `var` and copies to `grad`. `grad` is an array of inactive floating-point variables.

In the following example, we compute the first derivative of variable `r` with respect to `x`, `y`, and `z`.

```
#include "ad_deriv.h"

void main()
{
    InactiveDouble grad[ad_GRAD_MAX];
    DERIV_TYPE x, y, z, r;

    ad_AD_Init(ad_GRAD_MAX);

    set values of x, y, z;

    ad_AD_SetIndep(x);           /*variable x is the independent variable 1*/
    ad_AD_SetIndep(y);           /*variable y is the independent variable 2*/
    ad_AD_SetIndep(z);           /*variable z is the independent variable 3*/
    ad_AD_SetIndepDone();        /*done setting independent variables*/

    ad_function(&r, x, y, z);    /*invoke the derivative function*/

    ad_AD_ExtractGrad(grad, r);  /*extract the Jacobian into grad*/

    print(grad);                 /*grad[0] is dr/dx, grad[1] is dr/dy,
                                   grad[2] is dr/dz; print is a user-defined
                          routine */

    ad_AD_Final();
}
```

5.1 Computation of $J * M$

To support the initialization of the seed matrix, the following routines are provided:

ad_AD_ClearGrad(var) All elements of the derivative object of `var` variable are set to zero.

ad_AD_SetGrad(grad, var) This call is the complement of `ad_AD_ExtractGrad`: it initializes the Jacobian of `var` with `grad`. `grad` is an array of inactive floating-point variables.

To initialize the seed matrix M with m columns, the following steps can be used (we assume $n > m$, where n is the number of independent variables):

1. Call `ad_AD_SetIndep` for any set of m independent variables.
2. Use the above routines (i.e., ad_AD_ClearGrad, ad_AD_SetGrad) to reset the Jacobian of independent variables to the desired values. Each row of M corresponds to a particular Jacobian of an independent variable.

In the following example, we compute Jacobian*Vector ($J * v$, where J is a vector and v is a column vector) resulting in a scalar value.

```
#include "ad_deriv.h"

void main()
{
    InactiveDouble grad[ad_GRAD_MAX];
    DERIV_TYPE x, y, z, r;

    ad_AD_Init(ad_GRAD_MAX);

    /* ... set values of x, y, z; ... */

    ad_AD_SetIndep(x);           /* select any independent variable*/
    ad_AD_SetIndepDone();        /* done setting independent variables*/

    /* set seed matrix (a vector with all ones)*/
    grad[0] = 1.0;
    ad_AD_SetGrad(grad, x);      /* set row 1*/
    ad_AD_SetGrad(grad, y);      /* set row 2*/
    ad_AD_SetGrad(grad, z);      /* set row 3*/

    ad_function(&r, x, y, z);    /* invoke the derivative function*/

    /* extract the Jacobian (really just a value) into grad*/
    ad_AD_ExtractGrad(grad, r);

    /* ... print grad[0]; ...*/  /* print J*V*/

    ad_AD_Final();
}
```

Chapter 6

Intrinsic Functions

Two intrinsic functions, `max` and `min`, usually are defined as macros. They are not defined in `math.h`. In order for ADIC to treat them as intrinsic functions, the user must provide prototypes for them. The following code may be inserted in the code to be differentiated:

```
#undef max                    /*remove possible macro definitions*/
#undef min
double max(double,double);
double min(double,double);
```

Since `min/max` can be used for nonfloats, instead of overloading these functions, `fmin/fmax` may be used instead. Again, the user must provide the proper prototypes for them:

```
double fmax(double,double);
double fmin(double,double);
```

Important: The `#include` directive should be <math.h> rather than "math.h". If the code to be differentiated has been generated with `f2c` or a similar tool, the incorrect include directive may be generated and would need to be fixed manually before applying ADIC.

Chapter 7

Using Control Scripts

Control scripts are text files structured as a series of sections. Each section is headed by a section name enclosed in brackets, followed by a set of control lines called bindings. A binding represents a particular attribute that we wish to control and consists of a key and a value. A section is ended by a blank line, the beginning of another section, or the end of the file. Control scripts can include other control scripts.

ADIC defines a set of predefined sections and bindings. In addition, each module contributes its own set of bindings under a section name equivalent to the module name. Refer to each module's manual for the description of its bindings.

The standard ADIC distribution contains a control script named `adic.init`. This system script is always read first during ADIC processing.

In the addition to the global script adic.init, an architecture-specific control script for each platform is provided. The naming convention is `adic.init.`*arch*, where *arch* is the appropriate platform name (e.g., adic.init.solaris for the Solaris distribution). Most commonly, the architecture-specific control files are used for specifying preprocessor bindings and the location of the perl executable. Some system-specific macros can also be listed in the `[UNEXPANDED_MACROS]` section (see Chapter 9 for more details).

Any user-specified control scripts are processed next. The bindings specified in the user-specified control scripts override the same bindings in the adic.init script and the appropriate architecture-specific script.

The general format of script files is

```
[section1]
    key1 [= value]
    key2 [= value]

[section2]
    key1 [= value]
    key2 [= value]

...
```

On any line, comments begin right after the semicolon and are thus ignored. A good way to temporarily disable a particular binding is to put the semicolon in front of it.

The value portion of bindings should be enclosed in double quotes like the C strings if it contains special characters such as spaces and semicolons. Double quotes in the value string should be prefixed by the backslash character. Examples:

```
key = hello
key = "may be"
key2 = "value contains \"quotes\" and semicolons; "
```

The same section may appear multiple times. All the bindings from the same section will be collected together. If the same binding key are repeated, the later binding will override the earlier ones.

In the rest of this chapter, we describe each control script section. The valid range of numeric binding values is indicated by [begin..end]. Some binding values may only take on one of several predefined values. The result is unpredictable if the value is not one of the predefined values.

7.1 [GENERAL]

This control file section contains a grab-bag collection of bindings that affect the overall operation of ADIC.

include = string When this binding is encountered, the processing of the current script file is suspended and the script file specified by the binding value is processed. There may be multiple such includes.

prefix = string ADIC prepends each function name with a prefix. This prefix is also used in generating most internal variable names. The default prefix is **ad_** The user may override the default with this binding. If the user specifies the prefix binding without the value, then no prefix is prepended. In the case of C++, the member functions of a class do not get changed (this is unnecessary because the class name can be changed; see below).

var_prefix = string By default, ADIC does not change variable names. In that case the original function and the derivative-enhanced version cannot be used within the same scope, since ADIC changes the type of active variables, thus causing type conflicts. To avoid this, the user may define a var_prefix binding, which would be used to prepend all variable names in the differentiated code.

type_prefix = string Type names such as a `typedef` name, struct/union tag names, and C++ class name may be prepended with a prefix. For C++, in order to distinguish between the original class and the ADIC-processed class, the user should specify this binding. The default is no prefix, which in some cases may cause type conflicts (see explanation for var_prefix).

aif_version = 10..n ADIC implements the AIF interface used by plug-in modules. As AIF evolves, they are assigned different version numbers. ADIC notifies the module the latest AIF version that it can support. However, if this binding is present, it will override this internal value. This line is used for testing different versions of plug-in modules. Normally, the user would not need to modify this binding.

order = 0..n Each derivative module has a default derivative order. In many cases, each module may support only a single order. For example, the `Jacobian` module supports only first-order derivatives. The `Hessian` module supports first- and second-order derivatives defaulting to second order. The user may override the default value with this binding. Other derivative modules may generate different order derivatives.

include_dir = string The binding value specifies a list of directories to be searched for included files. The directories are separated by either colons or commas. These directories are searched before the standard directories.

7.2 [MODULES]

This control file section contains the specification of all the available plug-in modules. This section is listed in the system script file. The user normally does not need to deal with this section except when manually installing plug-ins. Each binding has the following format:

```
name = type,order,filepath
```

The **name** identifies a particular module and hence must be unique. In particular, this name is used to specify a particular module in the ADIC command line. Modules also may be embedded into ADIC. Make certain that the names of internal modules do not conflict with external module names. To see the list of both embedded and external module names, invoke ADIC without any argument. The **type** field specifies the type of modules. The derivative module has the value **deriv**; there may be different types of modules. The **order** specifies the default derivative order for the module. The user should not normally change this value. The **filepath** tells the file path of the module. The directory part of the path may be either relative to the ADIC directory or an absolute path in which case it must begin with a '/'.

Example:

```
[MODULES]
        Jacobian = deriv,1,bin/solaris/Jacobian
        Hessian = deriv,2,/home/me/Hessian/Hessian
```

The install process for a new plug-in usually sets the correct binding automatically in the system control script.

7.3 [INACTIVE_FUNCTIONS]

ADIC normally considers all functions to be active. The user can specify certain functions to be inactive (i.e., no derivative computations are to be performed nor any data structure be changed) by listing the function names in this section. This will also prevent the name change. Normally, when an inactive function definition is encountered, ADIC will not recreate the body (since it is already defined in the original source which presumably will be linked together), but rather only the prototype is generated at the point of the definition. However, this is not sufficient for static functions as the original static function will not be visible to the generated derivative source and hence the compiler will produce an error. As another option, the user can specify that the original definition be recreated.

ADIC normally considers functions declared within system headers to be inactive (this however can be overridden).

Example:

```
[INACTIVE_FUNCTIONS]
        creat
        open
        close
        write
        printf
```

The default system control script includes a number of these function names (such as `creat`). Although most of these functions are standard library functions, many programs do not contain the proper headers that declare them. Therefore, ADIC considers them to be regular functions and will change the function names. To prevent this, we have listed common library functions in the system script.

7.4 [NO_PREFIX_FUNCTIONS]

In some cases, the user may wish to prevent certain function names, whether active or not, from being prefixed, for example:

```
[NO_PREFIX_FUNCTIONS]
        main
```

The default system control script includes one function, namely `main`, which does not get prefixed (the function itself may be active).

7.5 [ACTIVE_FUNCTIONS]

It may be useful in some cases to notify ADIC to treat certain functions as if they were active. As an example, the user may have defined its own `atof` function that should be treated as a normal function by ADIC. The list in the example below overrides the `INACTIVE_FUNCTIONS` list and any functions declared in system headers.

Example:

```
[ACTIVE_FUNCTIONS]
        atof
```

7.6 [INACTIVE_VARIABLES]

ADIC normally considers all floating-point variables to be active. The user can specify, however, that certain global variables are to be made inactive by listing their names in this section. This will also prevent any name change.

Example:

```
[INACTIVE_VARIABLES]
        var1
        var2
```

7.7 [INACTIVE_TYPES]

A type is considered active if it is equivalent to a floating-point type (e.g., defined with a `typedef` from a floating-point type) or contains a subtype that is active (e.g., a structure containing a float type). The user can, however, explicitly specify certain type names as inactive by listing them under this section. Inactive types do not get modified. Variables of an inactive type are considered inactive. All the components of a compound inactive type (e.g., a struct) are also considered inactive.

Example:

```
[INACTIVE_TYPES]
        mydouble
        myStructType
```

ADIC predefines two inactive types, `InactiveDouble` and `InactiveFloat`, that are `typedef`ed to `double` and `float`, respectively. They may be used to declare inactive variables in user code.

7.8 [INTRINSIC_CONTROL]

The entries in this control file section controls the behavior of intrinsic function handling.

reporting = verbose,reportonce,counting,terse,performance When an floating-point exception occurs during derivative computation of an intrinsic function, this binding specifies how the exception should be reported. The `performance` mode turns off checking for many of the possible exception conditions and performs a default action (e.g., reportonce).

7.9 [INTRINSIC_FUNCTIONS]

The names of supported intrinsic functions are listed under this section. The default system script contains a standard set of intrinsic functions that it currently supports. Normally, nothing here should be changed, unless the user has added support for new intrinsics.

Example:

```
[INTRINSIC_FUNCTIONS]
        log
        sqrt
        cos
        sin
```

7.10 [DEFINES]

In order to process source files, all C macros must be properly defined. C macro definitions can be listed either under this section or on the command-line with the `-D` flag. The generated code does not contain any macros other than those that are explicitly specified (see the next section).

Example:

```
[DEFINES]
        DEBUG
        MAX_SIZE = 20
```

7.11 [UNEXPANDED_MACROS]

In order to prevent certain macros from being expanded by the preprocessor, their names and optional definitions may be added under this section. See Chapter 9 for further details and the motivation behind this option.

Example:

```
[UNEXPANDED_MACROS]
        putc = "int putc(char, FILE*);"
        getc = "int getc(FILE*);"
```

7.12 [SOURCE_FILES]

The set of source files to be processed can be listed either under this section or on the command line. The first source is considered the "master" file and will contain the definitions of special functions (e.g, `ad_AD_Init()`).

Example:

```
[SOURCE_FILES]
        source1.c
        source2.c
        header1.h
```

7.13 [STANDARD_INCLUDES]

Certain user header files have properties similar to those of system header files. Examples are common non-numeric library headers, such as `mpi.h`. The user can notify ADIC to treat certain header files as system header files by listing them under this section.

Example:

```
[STANDARD_INCLUDES]
        mpi.h
```

7.14 [NO_INLINE_INCLUDES]

If the inlining of user headers option is chosen (this is the default), then all user headers are inlined. To prevent certain headers from being inlined, the user can list them under this section.

Example:

```
[NO_INLINE_INCLUDES]
        portable.h
```

7.15 Derivative Modules

Each module can have a section that has the same name as the module name under which it defines its own set of bindings. If a module defines any bindings, its manual specifies the valid values of these bindings.

Chapter 8

Building ADIC-Generated Derivative Code

In order to integrate the derivative code into a larger system or to turn it into a standalone program, an appropriate driver must be written.

All floating-point variables are converted into the `DERIV_TYPE` unless they have been listed as inactive in a control script or have been declared as `InactiveDouble` or `InactiveFloat`. In general, the `DERIV_TYPE` will be a structure type. The "value" of an active variable x is referenced by `DERIV_val(`x`)`. Each derivative module defines one or more named derivative objects associated with each active variable. A particular derivative object of an active variable x is accessed by `DERIV_`*`name`*`(`x`)` where *name* specifies the name of the particular derivative object.

A typical process in writing the driver involves the following:

1. Declare all input floating-point variables to be referenced by the derivative function (either as parameters or as globals) as type `DERIV_TYPE`.
2. To initialize a variable x to value 2.0, use

   ```
   DERIV_val(x) = 2.0;
   ```

3. Call the initialization function generated by ADIC, typically called `ad_AD_Init()` (the actual name depends on the prefix string).
4. Specify independent variables, and initialize them. The exact procedure depends on the particular derivative module to be used.
5. Invoke the derivative function(s).
6. Extract the derivative values. The exact procedure also depends on the particular derivative module chosen.
7. Call the finalization function generated by ADIC, typically called `ad_AD_Final()` (the actual name depends on the prefix string).

In the following example, we compile and link two files, `func.ad.c` and `driver.c`. We generated the derivatives using the `Jacobian` module. We also set the maximum number of independent variables to 5.

```
% gcc -I$(ADIC)/include -Dad_GRAD_MAX=5 -c func.ad.c driver.c
% gcc -o program func.ad.o driver.o -L$(ADIC)/lib/$(ADIC_ARCH)
      -lJacobian -lADIntrinsics-C -lm
```

Chapter 9

Handling C Preprocessor Directives and Macros

The C preprocessing facility allows macros and directives to be defined. The portability and flexibility of C programs derive in part from this preprocessing facility. However, a translator typically sees the source only after it has been run through the preprocessor, converting it into legal ANSI-C translation units, which then can be correctly parsed. Hence, any preprocessor directives and macro names will be lost in the translated source. In most instances, no problems result. However, some of the portability and flexibility provided by the preprocessor facility are lost. The areas where this loss can occur are examined below:

- Preprocessor conditional directives are used to selectively include or leave out groups of lines within source files at compile time. However, this decision must be made at the time of translation rather than be deferred until the compile time of the translated source. The directives are often used for debugging or handling sections of the source that are machine specific.
- `#include` directives are used to include the contents of the standard or user header files. Since the implementation of the standard headers is system specific, their contents are not portable across machines in general.
- A macro can represent any text. A functionlike macro can also take arguments, performing argument substitutions during the preprocessing stage. Wherever the macro name occurs in the source, it gets expanded. The expansion of macros does not present any problem if the translated source is compiled on the same system. However, the expanded macros may not be portable across machines or even across compilers. For example, some of the macros defined in the standard headers are system-dependent. In fact, a name may be declared differently depending on the system. A typical example is `FILE`, declared in `stdio.h` standard header file. It is a macro name in SunOS but a `typedef` name in HPUX.

To deal with these potential problem areas, ADIC performs a number of steps to ensure more portability of the augmented source. ADIC has a preprocessing stage that "pre-treats" the source before it is run through the C preprocessor and sent on to the parsing stage.

- ADIC must be run separately for each desired choice of conditional directives. To ensure that the C preprocessor will select the appropriate text block, proper macro values can be passed to ADIC, which will in turn pass them on to the preprocessor. Macro values may be specified either via the control file or through a command-line option.

In the following example, the original source can either count up or count down depending on the value of the COUNT_MODE macro.

```
#if COUNT_MODE
   for (i = 0; i < n; i++) { //count up
#else
   for (i = n-1; i >= 0; i--) { //count down
#endif
       func(i, k);
   }
```

We must push forward the choice of value for COUNT_MODE to the translation stage by using the control file segment:

```
[DEFINES]
    COUNT_MODE = 1
```

or via the command-line option:

```
% adiC -v -d Jacobian -D COUNT_MODE=1 source.c
```

- To handle the #include directive problem, ADIC marks the locations of any included text and stores the names of the original header files. When ADIC is generating the augmented source, the entire contents of the standard headers are replaced with the original directives.

  In the case of user headers, in most cases, it is not necessary to reconstruct the original #include directives since it is assumed that the user headers are written in a portable fashion. However, the user could specify a command line option (-u), which will perform the reconstruction, with one difference. The reconstructed #include directive is changed to include the augmented header file, using the standard naming scheme. For example, names are changed as follows:

  ```
  #include "myinclude.h"      =>    #include "myinclude.ad.h"
  #include "commoncode.c"     =>    #include "commoncode.ad.c"
  ```

  With the -u option, the header files must be processed separately by ADIC. Without this option, the header files need not be processed by ADIC.

- In order to prevent expansion of certain macros, the ADIC preprocessing stage can either remove the macro definition if defined within a user include (when the -u option is used) or undefine it immediately after its definition in a standard header. Hence, the C preprocessor will not see the macro definition and thus will not expand the macro used in the source. To turn the macro name into a syntactically valid C construct, the user has to specify a suitable replacement definition (e.g., a variable declaration or a function prototype).

  The list of unexpanded macro names along with their replacement definitions can be specified in the control file under the UNEXPANDED_MACROS section. The replacement definition does not need to make semantic sense, the only requirement is that it can be parsed correctly. For example, suppose MY_MAX is defined in terms of DBL_MAX, which is a system-dependent maximum double-precision float value (defined in the standard header float.h):

  ```
  #define MY_MAX    DBL_MAX/2.0
  ```

To preserve this name wherever it is used instead of expanding it into the actual number (since DBL_MAX will get expanded also), the following binding can be added:

```
[UNEXPANDED_MACROS]
        MY_MAX    =   [static double MY_MAX;]
```

In essence we are moving `MY_MAX` defined in the macro name space into a static floating-point variable in the C name space. It is important that we turn it into a variable rather than a constant, since ADIC might optimize a constant away.

As another example, a functionlike macro can be handled as follows:

```
[UNEXPANDED_MACROS]
        getc    =   [extern int getc(FILE*);]
```

In this fashion we turn the `getc()` macro (defined in `stdio.h`) into a function prototype declaration. Actually, for ANSI-C source processing (but not for C++), no argument information need be provided. Hence, the following works just as well:

```
[UNEXPANDED_MACROS]
        getc    =   [extern int getc();]
```

To handle a macro that represents a type, we can do the following:

```
[UNEXPANDED_MACROS]
        FILE    =   [typedef struct _iobuf FILE;]
```

If the user does not have an idea about what a proper definition should be, the user can manually run the source through the C preprocessor (e.g., `cpp`) and check the expansion, then make the appropriate binding.

A potential problem can occur by undefining macros. In some older systems, the definition of a macro is used to determine whether a standard header has already been included or not. If that macro is the one undefined, then the header may be included again, causing multiple definition conflicts. For example, on BSD, the definition of `FILE` is checked to determine whether `stdio.h` has already been included. To prevent this from happening, the preprocessing stage will keep track of encountered standard headers and will ensure that it is not included twice.

Chapter 10

Controlling Naming

The original source declares names (identifiers) in various namespaces. Typical names may be function, variable, or type names. In the process of augmentation, the semantics of these names are changed. For example, `double` variable may be changed into `DERIV_TYPE` variable. In other cases, the semantics remain the same; for example, an `int` variable is unchanged. A conflict may occur if the original source and the augmented source are used (e.g., linked) together causing semantic or syntactic conflicts.

To handle these conflicts, names may be systematically changed by prepending them with a certain prefix. Depending on the type of names, we need to handle them differently:

- *Function names:* Each derivative enhanced function must have its name changed if the original function is also to be used in the same program. The default solution adopted by ADIC is to always prepend all occurrences of function names (both use and definition) with the prefix. The prefix is controlled by the `prefix` binding. There are two exceptions to prefixing function names. First, in the case of the standard library routines, the names should not be changed. Since we cannot easily know the names of all the standard library routines, we store the names of all library routines declared inside any standard header included in the source. Whenever such a name is encountered, the function is considered inactive and hence its name does not get modified. Second, the calls to standard math library routines are handled specially. The names of the standard math routines are listed in the control script file.

- *Typedefs:* Typedefs are effectively an alias of some other type, a shorthand for a type composition. If the base type contains a float type somewhere, then we may wish to change the name. The `type_prefix` binding may be used to attach a prefix to all type names.

- *structs, unions, and classes:* Struct and union tag names can also be attached with a prefix through the `type_prefix` binding.

- *C++ Methods:* Since class methods are declared with respect to a particular class, these need not and should not change, especially the overloaded operators.

- *Variable names:* Only global variables need to be changed, and only if both the original and the derivative functions are to be used in the same program. The prefix is controlled by `var_prefix` binding.

All identifiers declared inside standard headers should not be changed, since the derivative code will include the same standard headers. In the case of external numerical libraries such as the BLAS, either the source must be available to be run through ADIC or the differentiated version must be available. In future versions, ADIC distributions may include the derivative-enhanced versions of popular numerical libraries.

Chapter 11

Handling Intrinsic Calls

Invoking an intrinsic function through a function pointer may cause problems, since all intrinsic functions must be handled specially and replaced by a specific section of code. The problem arises from the lack of an address associated with this section of code.

The solution is to create a wrapper function that calls the intrinsic function. Whenever an address of an intrinsic function is taken, it should be changed into the address of the corresponding wrapper function.

Eventually, ADIC will automatically perform this step. For now, the user must do this step manually.

Chapter 12

Advanced Control

This section is for advanced users. Our current focus is on guarding global declarations.

Global declarations such as enums, and any typedefs/variables/structs declared as inactive can be guarded by ADIC-generated unique #ifdef directives. This also obviates the need for changing the enum values.

To generate guards, specify the flag `-g` when invoking adiC.

The guard will be added only to header files; it doesn't make sense to guard the generated source code. To determine whether a file is a header or not, ADIC uses the suffix name. The header filename must have the suffix .h or .H.

The guard macro is distinct for each header file; otherwise the user must include original headers for each differentiated header included even if the user may be interested in using only one of the original headers. A bigger problem is that different make rules must be added for user codes that include the original and the differentiated headers, and for ADIC-generated code that includes only the differentiated code. To solve this problem and the extra hassle of keeping track of which macros to define, ADIC uses the fact that most headers have guards themselves; these same guards are used for our purpose as well. The naming scheme of these guards is usually based on some rule (e.g., __basename_suffix__). The user can specify this type of rule in the control script using a printf-like format string.

```
[GENERAL]
    guard = __%b_%s__   ;(generates the above rule)
    guard = __%B_%S__   ;(generates the above rule with uppercasing
                        ;the letters)

%b -- base name  ( e.g., xx in xx.yy, and  x.y in x.y.z )
%s -- suffix  (  yy in xx.yy, and z in x.y.z )
%B -- uppercased base name
%S -- uppercased suffix name
```

Output filename generation rule: The filename generation rule uses the same format string as the guard format.

```
[GENERAL]
    filename_trans = %b.ad.%s   ;(the default rule)
    filename_trans = ad_%b.%s   ;(generates what the user wants)
```

Chapter 13

Troubleshooting

In this section, potential problems are discussed and possible solutions offered:

- If (supported) intrinsic function calls with the prefix (e.g., `ad_sin`) are generated, the user must make sure that `math.h` is included using angle brackets.
- If the collection of source files is too large, ADIC will not be able to handle them due to lack of memory. In this case, the user must invoke ADIC separately for each source.
- If multiple definitions are generated after processing each source separately, the user must make sure to use `-t` option for all sources except one.
- ADIC tries to ensure that any function referenced in the source and is declared within standard headers does not get prepended with the prefix. However, it is possible that some function gets prefixed (especially when only a partial or no prototype info is given). When this occurs, the user must manually put the name of the function in the `INACTIVE_FUNCTIONS` section. This problem usually occurs when many source files are processed together.
- ADIC expands/processes all macros during derivative generation, unless those macros are specified under the `UNEXPANDED_MACROS` section. If the source contains C preprocessor directives such `#if defined(XXX)` and these macros are defined through the compiler command-line arguments, the user must make sure to define the proper macro definition either through ADIC command-line or through the control scripts. If the macro definitions change, ADIC would need to be run again.
- When testing new modules, the user must make sure the corresponding control section associated with the new module is set up correctly.

Chapter 14

Known Problems

In this section, known problems and possible workarounds are discussed:

- In C++ mode, the parser does not recognize the external language specifier "C++". This problem will break some standard C++ headers such as `iostream.h`. Also, default arguments cannot be handled. No workaround.
- In ANSI C, goto labels have function scope. ADIC expects goto label names to be unique within each translation unit.
- Currently, C++ keywords such as `new` and `delete` are recognized as reserved tokens even when processing C files. Hence, make sure these names are not used.

Check the ADIC web site for a more current list of problems and workarounds.

Bibliography

[1] Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank. *Computational Differentiation: Techniques, Applications, and Tools*. SIAM, Philadelphia, 1996.

[2] Christian Bischof, Alan Carle, Peyvand Khademi, and Andrew Mauer. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science & Engineering*, 3(3):18–32, 1996.

[3] Christian Bischof and Lucas Roh. The automatic differentiation intermediate form (AIF), 1996. Unpublished Information.

[4] Christian Bischof, Lucas Roh, and Andrew Mauer. ADIC — An Extensible Automatic Differentiation Tool for ANSI-C. Preprint ANL/MCS-P626-1196, Mathematics and Computer Science Division, Argonne National Laboratory, 1996.

[5] S. Brown. OPRAD - a users guide to the OPtima Reverse Automatic Differentiation tool. Technical report, Numerical Optimization Centre, University of Hertfordsshire, 1995.

[6] Andreas Griewank. On automatic differentiation. In *Mathematical Programming: Recent Developments and Applications*, pages 83–108, Amsterdam, 1989. Kluwer Academic Publishers.

[7] Andreas Griewank, David Juedes, and Jean Utke. ADOL-C, a package for the automatic differentiation of algorithms written in C/C++. *ACM Transactions on Mathematical Software*, 22(2):131–167, 1996.

[8] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI – Portable Parallel Programming with the Message Passing Interface*. MIT Press, Cambridge, 1994.

[9] Andrew Mauer, Christian Bischof, and Alan Carle. The ADIntrinsics system for handling automatic differentiation of Fortran 77 intrinsics, 1996. Unpublished information, Argonne National Laboratory.

[10] Michael Monagan and Rene R. Rodoni. An implementation of the forward and reverse mode of automatic differentiation in Maple. In Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 353–362. SIAM, Philadelphia, 1996.

[11] Louis B. Rall. *Automatic Differentiation: Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1981.

[12] Nicole Rostaing, Stephane Dalmas, and Andre Galligo. Automatic differentiation in Odyssee. *Tellus*, 45a(5):558–568, October 1993.

[13] Dimitri Shiriaev and Andreas Griewank. ADOL-F: Automatic differentiation of Fortran codes. In Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 375–384, Philadelphia, 1996. SIAM.