An Introduction to Using Software Tools for Automatic Differentiation

by

Uwe Naumann and Andrea Walther†

naumann@mcs.anl.gov awalther@math.tu-dresden.de

Mathematics and Computer Science Division

Technical Memorandum No. 254, Revision A

July 2003

*This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-Eng-38.

† Institute for Scientific Computing, Technical Universität Dresden, Germany
Argonne National Laboratory, with facilities in the states of Illinois and Idaho, is owned by the United States Government and operated by The University of Chicago under the provisions of a contract with the Department of Energy.

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor The University of Chicago, nor any of their employees or officers, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of document authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof, Argonne National Laboratory, or The University of Chicago.

Available electronically at http://www.doe.gov/bridge

Available for a processing fee to U.S. Dept. 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
An Introduction to Using Software Tools for Automatic Differentiation

by

Uwe Naumann and Andrea Walther

Abstract

We give a gentle introduction to using various software tools for Automatic Differentiation (AD). Ready-to-use examples are discussed and links to further information are presented. Our target audience includes all those who are looking for a straight-forward way to get started using the available AD technology. The document is supposed to be dynamic in the sense that its content will be kept up-to-date as the AD software covered is evolving.

1 Intention and References

This document explains how to use the software tools for Automatic Differentiation (AD)

- ADIFOR 2.0 Revision D (http://www.mcs.anl.gov/adifor)
- ADIC 1.1 (http://www.mcs.anl.gov/adic)
- ADOL-C 1.8 (http://www.math.tu-dresden.de/wir/project/adolic)
- TAPENADE 1.0-alpha (http://www-sop.inria.fr/tropics)

to generate first order derivative code for a small example problem. It does not describe the AD algorithms used by the software tools covered. Various publications cover the theoretical foundations of the algorithms referenced in our discussion. For an introduction and more advanced issues we point the reader to [9]. Three workshops have been dedicated to AD, and the proceedings [5, 6, 7] contain numerous references to other AD-related papers that have appeared in scientific journals and proceedings of international conferences. A large variety of successful applications of AD software to real-world applications is discussed there as well.

2 Getting Started

For information on how to obtain, install, and use the AD tools discussed in this document, visit the websites listed above. The examples in this document are discussed in the context of the computer environment found on a laptop computer, specifically an Intel Pentium system running Mandrake Linux Version 8.1. ADIFOR, ADIC, and TAPENADE are installed in the directory /home/uwe/ADT00LS. The ADOL-C library and include files are assumed to be in the same directory as the source files.

ADIFOR, ADIC and TAPENADE are available on the Web and can be accessed online using the corresponding Web servers. Visit

- http://www.mcs.anl.gov/autodiff/adiforserver for ADIFOR,
- http://www.mcs.anl.gov/autodiff/adicserver for ADIC, and

All source files presented in this document can be downloaded from


The makefiles have to be adapted to the user’s environment.
3 Explosion Equation

We have selected a variant of the Bratu problem [1] to illustrate the use of source transformation AD tools. The code in Appendix A models the thermal explosion of solid fuels, which can be described by the system of differential equations

\[ x''(\tau) + s \cdot e^{-\frac{x(\tau)}{s}} = 0, \]

where \( \tau \in (-1, 1) \) and \( x(-1) = x(1) = 0 \). The problem has been discretized by using step size \( h \) as

\[ F_i = x_{i-1} - 2x_i + x_{i+1} + h^2 [f_{i-1} + 10f_i + f_{i+1}] / 12 \]

for \( i = 1, \ldots, 10000 \), with \( x_0 = x_{10001} = 0 \) and \( f_i = s \exp(x_i/(1+tx_i)) \). Of interest are the derivatives of the component functions \( F_i \) with respect to the current state \( x_i \) as well as the parameters \( s \) and \( t \).

We consider the following problems:

1. ADIFOR
   (a) Use the forward vector mode to compute the Jacobian of \( F \) with respect to \( x \) and the parameters \( s \) and \( t \). In our implementation \( F \) is represented by the variable \( f \), \( x \) is represented by \( x \), and \( s \) and \( t \) are combined into the parameter vector \( \text{prm} \). All three program variables are declared as arrays of double-precision floating-point numbers, namely
      - double precision \( x[7], \text{prm}[2], f[\text{dim}] \) in Fortran and
      - double \( x[7], \text{prm}[2], f[\text{dim}] \) in C.
      (See section 3.1.)
   (b) Use Curtis-Powell-Reed [8] seeding to compute the compressed Jacobian in forward vector mode (see section 3.1.1).
   (c) Use the SparseLinC library to compute the sparse Jacobian by sparse forward mode (see section 3.1.2).

2. ADOL-C
   (a) Use one of the Easy-to-Use driver to compute the Jacobian of \( F \) with respect to \( x \) and the parameters \( s \) and \( t \).
   (b) Compute the sparsity structure of the Jacobian.
   (c) Use the low level routines for forward and reverse mode to compute the Jacobian of \( F \) with respect to \( x \) and the parameters \( s \) and \( t \).
      (see section 3.3).

3. ADIC
   (a) Compute the Jacobian of \( F \) with respect to \( x \) in forward vector mode
      (see section 3.2).

4. TAPENADE
   (a) Compute the Jacobian of \( F \) with respect to both \( x \) and the parameters \( s \) and \( t \) in reverse mode
      (see section 3.4).
3.1 ADIFOR 2.0

ADIFOR generates a differentiated version of the subroutine `exp1(...) with the following header.

```fortran
subroutine g_exp1(g_p, dim, parmax, x, g_x, ldg_x, prm, g_prm,
+ ldg_prm, f, g_f, ldg_f)
  integer dim, parmax
  double precision x(dim), prm(parmax)
  double precision f(dim)

  integer g_pmax_
  parameter (g_pmax_ = 9)
  integer g_p, ldg_x, ldg_prm, ldg_f
  double precision g_x(ldg_x, dim), g_prm(ldg_prm, parmax),
+ g_f(ldg_f, dim)
```

The parameter `g_pmax_` is the maximal number of directional derivatives that can be computed. Its value is set via `AD_PMAX` in `explosion.adf`. The parameter is required because of the lack of dynamic memory allocation in Fortran 77. The derivative components of all scalar program variables are allocated as vectors of length `g_pmax_`. For example, to compute the Jacobian matrix of `f` with respect to `x`, the value of `g_pmax_` must be at least equal to 7, so that the derivative components of `x` can accommodate the identity in `df`.

The actual number of directional derivatives `g_p` must be less than or equal to `g_pmax_`. The parameter `g_p` determines the upper bound for the loops that compute the values of the derivative components.

To compute the Jacobian of `f` with respect to both `x` and `prm`, we set `g_pmax_` to be greater than or equal to the sum of the numbers of elements in both vectors, that is, `9 = 7 + 2 = dim + parmax`. Consequently, `g_x` contains the first `dim` columns of the seed matrix and `g_prm` its last `parmax` columns. The argument `g_f` is the transposed of the `(dim x ldg_f)`-Jacobian, where `ldg_f` must be initialized to 9. All this is done in the driver routine, which is shown in Appendix B.3.2.

The "dense:" section of the makefile in Appendix B.2 builds an executable that produces the following result:

```
-1.88  1.01  0.  0.  0.  0.  0.21 -0.48
 1.01 -1.87  1.01  0.  0.  0.  0.39 -1.78
 0.  1.01 -1.87  1.01  0.  0.  0.48 -2.69
 0.  0.  1.01 -1.87  1.01  0.  0.55 -3.49
 0.  0.  0.  1.01 -1.87  1.01  0.  0.48 -2.69
 0.  0.  0.  0.  1.01 -1.87  1.01  0.39 -1.78
 0.  0.  0.  0.  0.  1.01 -1.88  0.21 -0.48
```

The output has been formatted for better readability. It shows the full `7 x 9` Jacobian evaluated at the argument

```
x(1) = 1.72
x(2) = 3.45
x(3) = 4.16
x(4) = 4.87
x(5) = 4.16
x(6) = 3.45
x(7) = 1.72
```
\[
pr_m(1) = 1.3 \\
pr_m(2) = 0.245828
\]

### 3.1.1 Compressed Jacobian – Seeding

The full Jacobian computed in the preceding section is sparse, and its sparsity pattern can be visualized as follows.

```
  * *       * *       * *       * *       * *
  * * *     * * *     * * *     * * *     * * *
  * * *     * * *     * * *     * * *     * * *
  * * *     * * *     * * *     * * *     * * *
  * * *     * * *     * * *     * * *     * * *
  * * *     * * *     * * *     * * *     * * *
  * * *     * * *     * * *     * * *     * * *
```

Here, \(*\) stands for a nonzero entry, and blanks represent structural zero entries in the Jacobian. In other words, no dependence exists between the corresponding dependent and independent variables.

Curtis-Powell-Reid (CPR) [8] seeding is based on the idea that certain columns of the Jacobian can be merged to share storage. For example, column 1 and column 7 could share one column, thus resulting in a compressed version of the Jacobian. This implies that the sparsity pattern must be known in advance in order to exploit matrix compression techniques. Remember that the derivative code generated by ADIFOR always loops over the derivative components of all active variables.1 Many of them are equal to zero, leading to predictably trivial multiplications that one would like to avoid. Therefore, instead of computing the Jacobian as a Jacobian times identity matrix product, one could try to compute a compressed Jacobian using a seed matrix with fewer columns than the identity. Since the number of independent variables is often very large, the size of the seed matrix can become much smaller, leading to a decreased complexity of the forward vector mode Jacobian computation.

In CPR seeding, one considers the column incidence graph of the Jacobian to try to determine a minimal vertex coloring. Whenever two vertices share the same color, the corresponding columns can be stored in the same column of the compressed Jacobian. In our example, the column incidence graph has the following structure.

![Column Incidence Graph](image)

---

1An active variable \( x \) can be characterized as follows: At some point in the program \( (1) \) the value of \( x \) depends on the value of some independent variables and \( (2) \) there is some dependent variable whose value depends on \( x \). Variables that are not active are referred to as passive.
Unfortunately, since the vertex coloring problem is known to be NP-complete in general, the use of heuristics is essential. The coloring in the example graph has been found “by inspection”. Different colors are represented by different vertex shapes.

The number \( \nu \) of different colors used determines the number of columns in the CPR seed matrix. Its rows are Cartesian basis vectors in \( \mathbb{R}^\nu \). Whenever two vertices share the same color, the corresponding rows in the seed matrix contain the same Cartesian basis vector. This leads to the following seed matrix for our example.

\[
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1
\end{bmatrix}
\]

The “compressed” section of the makefile in Appendix B.2 builds the executable, which produces an output similar to the following.

\[
\begin{bmatrix}
-1.88 & 1.01 & 0. & 0.21 & -0.48 \\
1.01 & -1.87 & 1.01 & 0.39 & -1.78 \\
1.01 & 1.01 & -1.87 & 0.48 & -2.69 \\
-1.87 & 1.01 & 1.01 & 0.55 & -3.49 \\
1.01 & -1.87 & 1.01 & 0.48 & -2.69 \\
1.01 & 1.01 & -1.87 & 0.39 & -1.78 \\
-1.88 & 0. & 1.01 & 0.21 & -0.48 \\
\end{bmatrix}
\]

It shows the compressed \( 7 \times 5 \) Jacobian evaluated at the current argument. The reconstruction of the original Jacobian is a simple substitution process as described in [9, Chapter 7].

3.1.2 Sparse Forward Mode – SparsLinC

We compute the Jacobian of \( f \) with respect to \( x \). The sparsity pattern of the Jacobian need not be known a priori. Derivative components are sparse vectors of (index, value) pairs. The computational overhead of sparse vector arithmetic results from the index calculations. Structural sparsity of the extended Jacobian [9, Chapter 2] is exploited by avoiding trivial multiplications by zero. The decision on when to apply runtime sparsity methods depends on the problem structure.

The usage of SparsLinC with our example can be described by the following steps:

1. The file `explosion.cmp` can be used without change.
2. Add the entry `AD_FLAVOR=sparse` to `explosion.adf`.
3. Write driver program as shown in Appendix B.5.
4. Generate `g_explosion.f` by calling Adifor2.1 `AD_SCRIPT=explosion.adf`, and copy it from `output_files` to the current directory.
5. Compile `g_explosion.f` and the driver program.
6. Link with `ReqADIntrinsics-Linux86.o`, `libADIntrinsics-Linux86.a`, and `libSparsLinC-Linux86.a`.
7. Run the executable.
Let us have a closer look at the driver program. The differentiated subroutine

\[ g_{\text{expl}}(\text{dim}, \text{parmax}, x, g_x, \text{prm}, f, g_f) \]

is generated in \( g_{\text{explosion}}.f \), where \( g_x \) and \( f_x \) are integer arrays of dimension \( \text{dim} \) containing pointers to the corresponding sparse derivative objects. Seeding \( g_x \) is performed in the driver by

\[
\text{do 10 } i=1,\text{dim} \\
g_x(i)=0 \\
\text{CALL DSPSD}(g_x(i), i, 1.d0, 1) \\
10 \text{ continue}
\]

Notice that \( g_x(i) \) must be initialized properly before calling DSPSD. The call initializes the sparse derivative object pointed at by \( g_x(i) \) as \((1,1.d0)\). Consequently, after executing this loop \( g_x \) contains the sparse identity in \( F^{\text{dim}} \).

The sparse derivative components are extracted by

\[
\text{do 30 } i=1,\text{dim} \\
g_f(i)=0 \\
\text{CALL DSPXSQ}(\text{indvec}, \text{valvec}, \text{dim}, g_f(i), \text{outlen}, \text{info}) \\
30 \text{ continue}
\]

where \( \text{indvec} \) is the index vector and \( \text{valvec} \) the corresponding value vector making up the sparse representation of the derivative object pointed at by \( g_f(i) \). The parameter \( \text{outlen} \) is the number of nonzero entries in the \( i \)th row of the Jacobian. Consequently, the \( \text{outlen} \) first entries of \( \text{indvec} \) and \( \text{valvec} \) define the nonzero entries in the \( i \)th row of the Jacobian as index value pairs \((\text{indvec}(j), \text{valvec}(j))\).

The "sparse:" section of the makefile in Appendix B.2 builds the executable, which produces an output similar to the following.

\[
(1, -1.88) \ (2, \ 1.01) \\
(1, \ 1.01) \ (2, -1.87) \ (3, \ 1.01) \\
(2, \ 1.01) \ (3, -1.87) \ (4, \ 1.01) \\
(3, \ 1.01) \ (4, -1.87) \ (4, \ 1.01) \\
(4, \ 1.01) \ (5, -1.87) \ (6, \ 1.01) \\
(5, \ 1.01) \ (6, -1.87) \ (7, \ 1.01) \\
(6, \ 1.01) \ (7, -1.88)
\]

The sparse Jacobian is given by sparse row vectors whose nonzero entries are represented by (index, value) pairs. The reconstruction of the full Jacobian is straightforward. Refer to [3] for further information on how to use SparsLinC.

### 3.2 ADIC 1.1

All files involved in using ADIC with the C version of our example problem are shown in Appendix C. The Jacobian of \( F \) with respect to \( x \) is computed in forward vector mode. ADIC uses the derived data type \( \text{DERIV\_TYPE} \) to associate derivative components with active variables. The data type \( \text{InactiveDouble} \) is used to indicate that some floating-point variable is not active. The call of \( \text{ad\_AD\_Init(\text{dim})} \) causes the derivative code to use only the first \( \text{dim} \) elements of the derivative components. The latter are vectors of length \( \text{GRAD\_MAX} \) (see Appendix C). The vector \( x \) is declared to contain the \( \text{dim} \) independent variables by calling
ad_AD_SetIndepArray(x,dim);
ad_AD_SetIndepDone();

Access to the function value components of DERIV_TYPE variables is provided via the macro DERIV_val.
ADIC generates a differentiated version of the subroutine explosion and names it ad_exlosion. The argument list remains unchanged because all AD-related information is encapsulated in the new data type DERIV_TYPE. It changes, however, if the function returns a double. In this case, the differentiated function becomes a procedure, and the returned value becomes the first argument.
The derivative components of scalar variables of this type can be extracted by calling the routine ad_AD_ExtractGrad. The actual function value can be extracted with ad_AD_ExtractVal. In the example we call ad_AD_ExtractGrad(jac,P[i]) inside a loop over the dim elements of the vector of dependent variables P. The auxiliary variable jac is declared as a passive vector of size dim and contains the i-th row of the Jacobian.
A makefile is provided to build the executable explosion.ad to compute the first seven columns of the Jacobian shown in Section 3.1.

3.3 ADOL-C

The AD-tool ADOL-C is based on operator overloading. Using this technique, one can log for each operation during the program execution the operator and the variables that are involved. Hence, one obtains a new internal representation of the function evaluation. Based on the generated execution log, ADOL-C computes the desired derivatives.

In order to apply ADOL-C for derivative calculations one first has to modify the evaluation program to record the internal representation called tape. This modification starts with including header-file(s) that introduce the new data types and functions. Here, the easiest way is to simply include adolc.h. As second step, one has to define the part of the program for which one wants to compute the derivatives. From now on, this part is called active section. The statement trace_on(tag,keep); determines the beginning of the active section, the statement trace_off(file); the end of the active section. The parameter tag identifies the function to be differentiated. Hence, several function representations can be kept at the same time. Due to the shortness of the tutorial the optional parameters keep and file are not explained here but in the documentation [4]. As third step, one has to change the types of the independents to adoubles and mark them as independents using the overloaded operator <<=. The same must be done for the dependents using the overloaded operator >>=. Finally all variables that lie on the way from the independents to the dependents must be declared as adoubles. This step includes the generation of a new function explosion.ad which contains the same source code as before but in the interface the double-variables have to be changed to adouble-variables. The required modification of the original source code are illustrated by Appendix D.

3.3.1 Jacobian Calculation using the Easy-to-use Driver

After the tapes are generated during the execution of the active section the required derivative objects can be calculated. For that purpose ADOL-C provides a variety of easy-to-use drivers, e.g. gradient(...), jacobian(...), hessian(...), vec_jac(...) computing the product vector times Jacobian, jac_vec(...) computing the product Jacobian times vector, and so on. Moreover, ADOL-C supplies routines evaluating the Taylor coefficient vectors and their Jacobians with respect to the current state vector of solution curves defined by ordinary differential equations. Furthermore, there are drivers for derivative tensors and for the differentiation of implicit and inverse functions.

If one wants to compute the Jacobian of $F$ with respect to $x$ and the parameters $s$ and $t$, one has to declare a variable storing the derivative information

```c
double **J = myalloc(dim,dim+parmax);
```

where myalloc is provided by ADOL-C to allocate a two-dimensional array. Then, the statement
jacobian(tag, dim, dim+parmax, v, J);
causes the computation of the Jacobian of the function representation contained in the tape with
the number tag at the point v. For a consistency check, the second and third parameter determine
the number of dependents and independents, respectively. Hence, only the two statements given
above have to be inserted after trace_off() in order to compute the Jacobian of F.

3.3.2 Calculation of Sparsity Pattern
ADOL-C provides for the computation of sparsity patterns the driver
jaccpat(tag, dim, dim+parmax, v, rb, cb, Jsp, option);
If one sets rb and cb to NULL, jaccpat computes the sparsity structure of the complete Jacobian at the
point where the tape was generated and stores it in the unsigned int-array Jsp. The corresponding
statements that have to be added to the source code are shown in Appendix 7. In the case that a
certain block structure of the Jacobian is known prior the unsigned int-vectors rb and cb can be
applied to describe a compressed form of the independent and dependent variables.

3.3.3 Forward and Reverse mode using Low-level Routines
Arbitrarily high-order derivatives can be calculated using the low-level functions of ADOL-C for
the forward and reverse mode of AD. These routines are explained in detail in a short reference
available from the ADOL-C web-page. In this article, only the computation of first order derivatives
is sketched. If one wants to compute the full Jacobian it is preferable to use vector modes of AD. For that
purpose, ADOL-C provides the drivers forward(...) and reverse(...). Here, forward stands for first
order vector. The other acronyms of the low level routines have a corresponding meaning. The
statement
forward(tag, dim, dim+parmax, p, v, X, Fp, Y);
computes the derivative object Y = F'(v)X for X ∈ Rp×p. The statement
reverse(tag, dim, dim+parmax, q, U, Z);
computes the derivative object Z = UF'(v) for U ∈ Rp×q. In order to prepare this reverse sweep
one has to call an appropriate forward routines, the choice of which is described in detail by the
short reference mentioned above. The code segments required for computing the full Jacobian with
the low level routines are contained in Appendix 7.

3.4 TAPENADE
Our last example uses the alpha version of TAPENADE 1.0 to illustrate the use of reverse-mode
AD for computing the Jacobian of f with respect to x and y. All files needed to run the example
are described in Appendix 8. The makefile in Appendix E.2 can be used to generate the executable
equation. Running the latter results in the computation of the full Jacobian as in section 3.1 but
this time using the reverse mode of AD.

The following command-line parameters are used to call tapenade:

• "-head": The name of the head routine.
• "-vars": The names of the independent variables.
• "-c1": Switch indicating that reverse, or cotangent linear, mode is used.

It generates derivative code in exp1c1.f, which must be compiled together with the driver program
shown in Appendix E.2 and linked with the routines for storing and restoring the values of the tape
as described in [9, Chapter 2].

The current version of TAPENADE does not provide the reverse vector mode. Hence, the
Jacobian must be computed as a sequence of Jacobian transposed times vector products. This is
done in the driver program by successively initializing the derivative components $\mathbf{g}_i \mathbf{f}$ of $\mathbf{f}$ as the Cartesian basis vectors in $\mathbb{R}^{n_{\text{in}}}$. Thus, the Jacobian is accumulated row-by-row at a computational complexity which is proportional to the number of dependent variables. This becomes particularly interesting in the case of large single gradients. Refer to [9] for further details on reverse mode AD.

References


A Original Code

```fortran
subroutine bratu(dim,parmax,x,prm,F)
integer dim , parmax
C independent variables
double precision x(dim), prm(parmax)
C dependent variables
double precision F(dim)
C
integer i
double precision h

h = 2.0/(dim+1)
F(1) = -2*x(1)+h*h*prm(1)/12.0*(1+10*exp(x(1)/(1.0+prm(2)*x(1))))
F(2) = x(1)+h*h*prm(1)/12.0*exp(x(1)/(1.0+prm(2)*x(1))))

do 1 i=2,dim-1
   F(i-1) = F(i-1)+x(i)+h*h*prm(1)/12.0*exp(x(i)/(1.0+prm(2)*x(i))))
   F(i) = F(i)-2*x(i)+h*h*prm(1)/1.2*exp(x(i)/(1.0+prm(2)*x(i))))
   F(i+1) = x(i)+h*h*prm(1)/12.0*exp(x(i)/(1.0+prm(2)*x(i))))
1 continue

F(dim-1) = F(dim-1)+x(dim)+h*h*prm(1)/12.0*exp(x(dim)/(1.0
   +pram(2)*x(dim))))
F(dim) = F(dim)-2*x(dim)
F(dim) = F(dim)+h*h*prm(1)/12.0*(1+10*exp(x(dim)/(1.0
   +pram(2)*x(dim))))
end
```

B ADIFOR 2.0

The following files can be downloaded from

http://www-unix.mcs.anl.gov/~haumann/adtools.html

explosion.adf
explosion.cmp
explosion.driver.compressed.f
explosion.driver.f
explosion.driver.sparse.f
explosion.f
explosion.sparse.adf
makefile

B.1 explosion.cmp

The composition file lists the names of all files containing subroutines subject to differentiation. In our simple example there is just explosion.f.
B.2 makefile

The makefile can be used for computing the full, compressed, and sparse Jacobians using ADIFOR 2.0. It also ensures a proper cleanup of all files that are generated automatically during this process.

```
AD_LIB=/home/uwe/ADTOOLS/ADIFOR2/ADIFOR2.0D.lib
dense:
  Adifor2.1 AD_SCRIPT=explosion.adf
cp output_files/g_explosion.f .
g77 -g -c g_explosion.f explosion.driver.f
g77 -g -o explosion.ad.dense -L$(AD_LIB)/lib *.o 
$(AD_LIB)/lib/ReqADIntrinsics-Linux86.o 
-1ADIntrinsics-Linux86

compressed:
  Adifor2.1 AD_SCRIPT=explosion.adf
  cp output_files/g_explosion.f .
g77 -g -c g_explosion.f explosion.driver.compressed.f
g77 -g -o explosion.ad.compressed -L$(AD_LIB)/lib *.o 
$(AD_LIB)/lib/ReqADIntrinsics-Linux86.o 
-1ADIntrinsics-Linux86

sparse:
  Adifor2.1 AD_SCRIPT=explosion.sparse.adf
  cp output_files/g_explosion.f .
g77 -g -c g_explosion.f explosion.driver.sparse.f
g77 -g -o explosion.ad.sparse -L$(AD_LIB)/lib *.o 
$(AD_LIB)/lib/ReqADIntrinsics-Linux86.o 
-1SolarsLinC-Linux86

clean:
  rm -fr output_files
  rm -fr AD_cache
  rm *.o
  rm g_ *
  rm explosion.ad.*
  rm *
```

B.3 Jacobian

B.3.1 explosion.adf

The Jacobian of the output variable \( f \) with respect to the two input variables \( x \) and \( prm \) of the top-level routine \( \text{exp1} \) is computed. The parameter \( \text{AD\_P\_MAX} \) must be set to the total number of independent variables, that is, \( 9^6 \).

```
AD\_PROG = explosion.cmp
```

\(^1\)This is because the Jacobian is computed by forward vector mode with a seed matrix that is equal to the identity in \( \mathbb{R}^9 \).
B.3.2 explosion.driver.f

program main
   implicit none
C Example: Explosion Equation
C Driver for computing Jacobian
C
integer dim, parmax, n
parameter (dim=7, parmax=2, n=9)
integer i,j
C independent variables
double precision x(dim), prm(parmax)
C derivative components of independent variables
double precision g_x(n,dim)
double precision g_prm(n,parmax)
C dependent variables
double precision f(dim)
C derivative components of dependent variables
double precision g_f(n,dim)
C
Initialization of input variables
x(1) = 1.72
x(2) = 3.45
x(3) = 4.16
x(4) = 4.87
x(5) = 4.16
x(6) = 3.45
x(7) = 1.72
prm(1) = 1.3
prm(2) = 0.245828

C Seeding (identity)
do 20 i=1,n
   do 10 j=1,parmax
      if (i.eq.j+dim) then
         g_prm(i,j)=1.0
      else
         g_prm(i,j)=0.0
      endif
   10 continue
20 continue

do 40 i=1,n
   do 30 j=1,dim
      if (i.eq.j) then

\begin{verbatim}
g_x(i,j)=1.0
else
  g_x(i,j)=0.0
endif
30  continue
40  continue

C call differentiated subroutine
call gexpl(n,dim,parmax,x,g_x,n,prm,g_prm,n,f,g_f,n);

C print Jacobian
do 60 i=1,dim
do 50 j=1,n
  print*, "f'(i, i",", j")=",g_f(j,i)
50  continue
60  continue
end

B.4 Compressed Jacobian
B.4.1 explosion.adf
... is the same as in the dense case.

B.4.2 explosion.driver.compressed.f

program main
implicit none

C
C Example: Explosion Equation
C Driver for computing compressed Jacobian
C
integer dim, parmax, n
parameter (dim=7, parmax=2, n=5)
integer i,j
C independent variables
double precision x(dim), prm(parmax)
C derivative components of independent variables
double precision g_x(n,dim)
double precision g_prm(n,parmax)
C dependent variables
double precision f(dim)
C derivative components of dependent variables
double precision g_f(n,dim)

C Initialization of input variables
x(1) = 1.72
x(2) = 3.45
x(3) = 4.16
x(4) = 4.87
x(5) = 4.16
\end{verbatim}
\[ x(6) = 3.45 \\
\] \[ x(7) = 1.72 \\
\] \[ \text{prm}(1) = 1.3 \\
\] \[ \text{prm}(2) = 0.245828 \]

C Seeding (CPR)
do 20 i=1,n
do 10 j=1,prmax
\[ g_{\text{prm}}(i,j)=0.0 \]
10 continue
20 continue

\[ g_{\text{prm}}(n-1,1)=1.0 \]
\[ g_{\text{prm}}(n,2)=1.0 \]

do 40 i=1,n
do 30 j=1,dim
\[ g_{x}(i,j)=0.0 \]
30 continue
40 continue

\[ g_{x}(1,1)=1.0 \]
\[ g_{x}(1,4)=1.0 \]
\[ g_{x}(1,7)=1.0 \]
\[ g_{x}(2,2)=1.0 \]
\[ g_{x}(2,5)=1.0 \]
\[ g_{x}(3,3)=1.0 \]
\[ g_{x}(3,6)=1.0 \]

C call differentiated subroutine
call g_expl(n,dim,prmax,x,g_x,n,prm,g prm,n,f,g_f,n);

C print compressed Jacobian
do 60 i=1,dim
do 50 j=1,n
\[ \text{print}, "f"(\text{"\,"}, i, \"\,\", j, \"\,\")="\,g_f(j,i) \]
50 continue
60 continue

end

B.5 Sparse Jacobian

B.5.1 explosion.sparse.adf

To make ADIFOR generate derivative code that can use the sparse forward mode provided by
SparLinC, the parameter AD_FLAVOR must be set to sparse.

\[ \text{AD_PROG} = \text{explosion.cmp} \]
\[ \text{AD_FLAVOR} = \text{sparse} \]
\[ \text{AD_TOP} = \text{expl} \]
\[ \text{AD_PMAX} = 9 \]

14
AD_IVARS= x
AD_DVARS= f

B.5.2 explosion.driver.sparse.f

program main
implicit none

C
C Example: Explosion Equation
C Driver for computing sparse Jacobian using SparsLinC
C
integer dim, parmax, n
parameter (dim=7, parmax=2, n=9)
integer i,j
C independent variables
double precision x(dim)
C passive inputs
double precision prm(parmax)
C pointers to sparse derivative components of
C independent variables
integer g_x(dim)
C dependent variables
double precision f(dim)
C pointers to sparse derivative components of
C dependent variables
integer g_f(dim)

C (index, value) pairs
integer indexes(dim)
double precision values(dim)

C values used in extraction routine
integer outlen, info

C Initialization of input variables
x(1) = 1.72
x(2) = 3.45
x(3) = 4.16
x(4) = 4.87
x(5) = 4.16
x(6) = 3.45
x(7) = 1.72
prm(1) = 1.3
prm(2) = 0.245828

C Initialization of sparse data structures
call XSPINI
Seeding (sparse identity)
do 10 i=1,dim
    g_x(i)=0
    g_f(i)=0
call DSPSD(g_x(i),i,1.d0,1)
10    continue

Call differentiated subroutine

call g_expl(dim,parmax,x,g_x,prm,f,g_f)

Extract derivative components
do 30 i=1,dim
    call DSPXSQ(indexes,values,dim,g_f(i),outlen,info)
    if (info.eq.0) then
        do 20 j=1,outlen
            print*, "indexes(" i "", "j ")=" indexes(j)
            print*, "values(" i "", "j ")=" values(j)
        20        continue
    endif
30    continue
end

ADIC 1.1

The following files can be downloaded from

http://www-unix.mcs.anl.gov/haumann/ad_tools.html

explosion.c
explosion.driver.c
explosion.init
makefile

makefile

AD_INC = -I$(ADIC)/include -I.
AD_LIB = -L$(ADIC)/lib/$(ADIC_ARCH)

all:
  adic -d gradient -i explosion.init
  g++ -g -o explosion.ad $(AD_INC) $(AD_LIB) \
explosion.ad c explosion.driver.c \
-1ADIntrinsics-C -laif_grad -lm

clean:
  rm explosion.ad*
  rm ad_deriv.h
  rm *

The environment variable ADIC must be set to the directory in which ADIC has been installed.
C.2 explosion.init

[SOURCE_FILES]
   explosion.c

[gradient]
   GRAD_MAX=7

   The script file specifies the input files (only one in this case) and a variety of other parameters that can be looked up in [2]. The definition of GRAD_MAX, the length of the derivative components, is important. Its default value is 5, which would cause trouble in our case where dim=7 > 5. Refer to [2] for other ways to set the value of GRAD_MAX.

C.3 explosion.driver.c

#include "ad_deriv.h"
#include <stdio.h>
#include <iostream>
#include <stdlib.h>

extern void ad_explosion(int, DERIV_TYPE*, DERIV_TYPE*, DERIV_TYPE*);

void main()
{
   int i,j;
   int dim=7;
   int parmax=2;
   // independent variables
   DERIV_TYPE *x = new DERIV_TYPE[dim];
   DERIV_TYPE *prm = new DERIV_TYPE[parmax];
   // independent variables
   DERIV_TYPE *F = new DERIV_TYPE[dim];
   // independent variables
   InactiveDouble *jac = new InactiveDouble[dim];

   ad_AD_Init(dim);

   ad_AD_SetIndepArray(x,dim);
   ad_AD_SetIndepDone();

   DERIV_val(x[0]) = 1.72;
   DERIV_val(x[1]) = 3.45;
   DERIV_val(x[2]) = 4.16;
   DERIV_val(x[3]) = 4.87;
   DERIV_val(x[4]) = 4.16;
   DERIV_val(x[5]) = 3.45;
   DERIV_val(x[6]) = 1.72;
   DERIV_val(prm[0]) = 1.3;
   DERIV_val(prm[1]) = 0.245828;

   ad_explosion(dim,x,prm,F);

   ...
for (i=0;i<dim;i++) {
    ad_AD_ExtractGrad(jac,F[i]);
    for (j=0;j<dim;j++)
        cout << "f"["i" << i+1 << "," << j+1 << "]=" << jac[j] << endl;
}

ad_AD_Final();
}

D ADOL-C

D.1 Modifications of original source code

#include <adolc.h>
#include <SPARSE/sparse.h>
#include <stdio.h>
#include <iostream>
#include <stdlib.h>

extern void explosion_ad(int, adouble*, adouble*, adouble*);

void main()
{
    int i,j;
    int dim=7;
    int parmax=2;
    int tag = 1;

    // independent variables (passiv)
    double *v = new double[dim+parmax];
    // dependent variables (passiv)
    double *Fp = new double[dim];

    // independent variables (active)
    adouble *x = new adouble[dim];
    adouble *prm = new adouble[parmax];
    // dependent variables (active)
    adouble *F = new adouble[dim];

    v[0] = 1.72;
    v[1] = 3.45;
    v[2] = 4.16;
    v[3] = 4.87;
    v[4] = 4.16;
    v[5] = 3.45;
    v[6] = 1.72;
    v[7] = 1.3;
    v[8] = 0.245828;
trace_on(tag);
for(j=0; j<dim; j++)
  x[j] = v[j];
for(j=0; j<parmax; j++)
  prm[j] = v[j];
explosion_ad(dim,x,prm,F);
for(j=0; j<dim; j++)
  F[j] = Fp[j];
trace_off();

D.2 Computation of Sparsity Pattern

// Sparsity pattern declarations
int option[3];
unsigned int** Jsp = new unsigned int*[dim];

for(j=0; j<dim; j++)
  Jsp[j] = new unsigned int[parmax];
option[0] = 0; // automatic detection for AD mode
option[1] = 0; // save propagation of bit-pattern
option[2] = 0; // no output

... // Sparsity pattern computation after trace_off
jac_pat(tag,dim,dim+parmax,v,NULL,NULL,Jsp,option);

D.3 Jacobian Calculation using Low-level Routines

// Calculate Jacobian using forward mode driver
double** X = myalloc(dim+parmax,dim+parmax);
// Calculate Jacobian using reverse mode driver
double** U = myalloc(dim,dim);

... // Use low level routines to compute Jacobian
// forward:

for (j=0; j<dim+parmax; j++)
{
  for (i=0; i<dim+parmax; i++)
    X[j][i] = 0.0;
  X[j][j] = 1.0;


E  TAPENADE

The following files can be downloaded from

    http://www-unix.mcs.anl.gov/~haumann/ad_tools.html

explosion.f
explosion.driver.f
makefile

E.1  makefile

ADSTACK = $(HOME)/ADTOOLS/TAPENADE/tapenade1.0/stack/adStack.o

all:
tapenade -head expl -vars "x prm" -cl explosion.f
g77 -o explosion.ad explcl.f explosion.driver.f \ $(ADSTACK)

clean:
rm explosion.ad
rm explcl.f
rm -fr diffgen
rm *~

E.2  explosion.driver.f

    program main
    implicit none
Example: Explosion Equation

Driver for computing Jacobian using TAPENADE's Reverse Mode

integer dim, parmax, n
parameter (dim=7, parmax=2, n=9)
integer i,j
C independent variables
double precision x(dim), prm(parmax)
C derivative components of independent variables
double precision g_x(dim)
double precision g_prm(parmax)
C dependent variables
double precision f(dim)
C derivative components of dependent variables
double precision g_f(dim)
C the whole Jacobian matrix
double precision jac(dim,n)

Initialization of input variables
x(1) = 1.72
x(2) = 3.45
x(3) = 4.16
x(4) = 4.87
x(5) = 4.16
x(6) = 3.45
x(7) = 1.72

prm(1) = 1.3
prm(2) = 0.245828

Compute Jacobian as sequence of Jacobian transposed
C times vector products
do 40 i=1,dim
   do 10 j=1,dim
      g_f(j)=0.d0
      g_x(j)=0.d0
10 continue
g_f(i)=1.d0
g_prm(1)=0.d0
g_prm(2)=0.d0
call explcl(dim,parmax,x,g_x,prm,g_prm,f,g_f);
do 20 j=1,dim
   jac(i,j)=g_x(j)
20 continue
do 30 j=1,parmax
   jac(i,j+dim)=g_prm(j)
30 continue
40 continue

C print Jacobian
   do 60 i=1,dim
       do 50 j=1,n
           print*, "f'(" , i, ",", j, ")=", jac(i,j)
   50 continue
50 continue
60 continue

end