Getting Started with ADOL-C

ndrea Walther[1]

thematics University of Paderborn
Paderborn, Germany
ner@math.uni-paderborn.de

Abstract. The C++ package ADOL-C described in this paper facilitates the evaluation of first and higher derivatives of vector functions that are defined by computer programs written in C or C++. The numerical values of derivative vectors are obtained free of truncation errors at mostly a small multiple of the run time and a fix small multiple random access memory required by the given function evaluation program. Derivative matrices are obtained by columns, by rows or in sparse format. This tutorial describes the source code modification required for the application of ADOL-C, the most frequently used drivers to evaluate derivatives and some recent developments.

Keywords. ADOL-C, algorithmic differentiation of C/C++ programs

1 Introduction

ADOL-C is an operator overloading based AD-tool that allows to compute derivatives for functions given as C or C++ source code. ADOL-C can handle codes based on classes, templates and other advanced C++-features. The resulting derivative evaluation routines may be called from C, C++, Fortran, or any other language that can be linked with C.

ADOL-C facilitates the simultaneous evaluation of arbitrarily high directional derivatives and the gradients of these Taylor coefficients with respect to all independent variables. Hence, ADOL-C covers the computation of standard objects required for optimization purposes as gradients, Jacobians, Hessians, Jacobian $\times$ vector products, Hessian $\times$ vector products, etc. The exploitation of sparsity is possible via a coupling with the graph coloring library ColPack [1] developed by the authors of [2] and [3]. For solution curves defined by ordinary differential equations, special routines are provided that evaluate the Taylor coefficient vectors and their Jacobians with respect to the current state vector. For explicitly or implicitly defined functions derivative tensors are obtained with a complexity that grows only quadratically in their degree. The numerical values of derivative vectors are obtained free of truncation errors at a small multiple of random access memory required by the given function evaluation program. The derivative calculations involve a possibly substantial but always predictable amount of data. Most of this data is accessed strictly sequentially. Therefore, it can be automatically paged out to external files if necessary. Furthermore,

Dagstuhl Seminar Proceedings 09061
Combinatorial Scientific Computing
http://drops.dagstuhl.de/opus/volltexte/2009/2084

ADOL-C provides a so-called tapeless forward mode, where the derivatives are directly p[illegible]h the function values. In this case, no additionally sequ[illegible]s required.

Appli[illegible]OL-C can be found in many fields of science and techn[illegible].g., fish stock assessment by the software package CAS[illegible] simulation of electronic circuits by fREEDA [5] and the n[illegible] optimal control problems by MUSCOD-II [6]. Currently[illegible]developed and maintained at the University of Paderbor[illegible]hat is guided by Andrea Walther.

The key ingredient of automatic differentiation by overloading is the concept of an *active variable*. All variables that may be considered as differentiable quantities at some time during the program execution must be of an active type. Hence, all variables that lie on the way from the input variables, i.e., the independents, to the output variables, i.e., the dependents have to be redeclared to be of the active type. For this purpose, ADOL-C introduces the new data type adouble, whose real part is of the standard type double. In data flow terminology, the set of active variable names must contain all its successors in the dependency graph. Variables that do not depend on the independent variables but enter the calculation, for example, as parameters, may remain one of the *passive* types double, float, or int. There is no implicit type conversion from adouble to any of these passive types; thus, failure to declare variables as active when they depend on other active variables will result in a compile-time error message.

The derivative calculation is based on an internal function representation, which is created during a separate so-called taping phase that starts with a call to the routine trace_on provided by ADOL-C and is finalized by calling the ADOL-C routine trace_off. All calculations involving active variables that occur between the function calls trace_on(tag,...) and trace_off(...) are recorded on a sequential data set called *tape*. Pairs of these function calls can appear anywhere in a C++ program, but they must not overlap. The nonnegative integer argument tag identifies the particular tape, i.e., internal function representation. Once, the internal function representation is available, the drivers provided by ADOL-C can be applied to compute the desired derivatives.

In some situations it may be desirable to calculate the derivatives of the function at arbitrary arguments by using a tape of the function evaluation at another argument. Due to the avoidance of an additional taping process, this approach can significantly reduce the overall run time. Therefore, the routines provided by ADOL-C for the evaluation of derivatives can be used at arguments other than the point at which the tape was generated, provided that all comparisons involving adoubles yield the same result. The last condition implies that the control flow is unaltered by the change of the independent variable values. The return value of all ADOL-C drivers indicate this validity of the tape. If the user finds the return value of an ADOL-C routine to be negative the taping process simply has to be repeated by executing the active section again, since the tape records only the operations that are executed during one particular evaluation of the function.

2 Preparing a Code Segment for Differentiation

The mod[illegible] the algorithmic differentiation with ADOL-C form a fi[illegible]

1. Inclu[illegible]
   For b[illegible]ons the easiest way is to put
   #incl[illegible]
   at th[illegible]
   The [illegible] and the parallel computation of derivatives require additional header files.
2. Define the region that has to be differentiated
   That is, mark the *active section* with the two commands:

trace_on(tag, *keep*);	Start of
...	active section
trace_off(*file*);	and its end

   These two statements define the part of the program for which an internal representation is created
3. Declare all independent variables and dependent variables of type adouble and mark them in the active section:

xa $\ll=$ xp;	mark and initialize independents
...	calculations
ya $\gg=$ yp;	mark dependents

4. Declare all active variables, i.e., all variables on the way from the independent variables to the dependent variables of type adouble
5. Calculate derivative objects after trace_off(*file*)

Before a small example is discussed, some additional comments will be given. The optional integer argument keep of trace_on as it occurs in step 2 determines whether the numerical values of all active variables are recorded in a buffered temporary file before they will be overwritten. This option takes effect if keep $= 1$ and prepares the scene for an immediately following reverse mode differentiation as described in more detail in the sections 4 and 5 of the ADOL-C manual. By setting the optional integer argument file of trace_off to 1, the user may force a tape file to be written on disc even if it could be kept in main memory. If the argument file is omitted, it defaults to 0, so that the tape array is written onto an external file only if the length of any of the buffers exceeds BUFSIZE elements, where BUFSIZE is a user-defined size.

ADOL-C overloads the two rarely used binary shift operators $\ll=$ and $\gg=$ to identify independent variables and dependent variables, respectively, as described in step 3. For the independent variables the value of the right hand side is used as the initial value of the independent variable on the left hand side.

Choosing the set of variables that has to be declared of the augmented data type in step 4 is basically up to the user. A simple strategy that can be applied with minimal effort in many cases is the redeclaration of every floating point variable [illegible] use the provided augmented data type adouble. This can [illegible] MAKRO statement. Then, code changes are necessary [illegible]d usually very small part of the source files. However, [illegible]ing effort, this simple approach may result in an unnecess[illegible] both the function evaluation and the derivation calculatio[illegible]echnique to determine the active variables is the compiler-[illegible]or this purpose, only the independent variables are redec[illegible]nted data type adouble. During the compilation process o[illegible]piler will issue error messages concerning data type conversions that are not allowed. Based on these error messages the user redeclares additional variables, i.e., certain intermediates or dependents, to be of the augmented data type adouble. Then, the program compilation process is invoked again, possibly issuing different error messages. This loop of changing the type of variables and examining the compiler messages has to be carried out until the last error message has been resolved. Compared to the global change strategy described above, a reduced set of augmented variables is created resulting in a smaller internal function representation that is generated during the taping step.

3 Example of a Modified Code Segment

To illustrate the required source code modification, we consider the following lighthouse example from [7]. The lighthouse on the left emanates a light beam that hits the quay-wall at a point (y_1, y_2). Applying planar geometry yields that its coordinates are given by

$$y_1 = \frac{\nu * \tan(\omega * t)}{\gamma - \tan(\omega * t)}$$

and

$$y_2 = \frac{\gamma * \nu * \tan(\omega * t)}{\gamma - \tan(\omega * t)}.$$

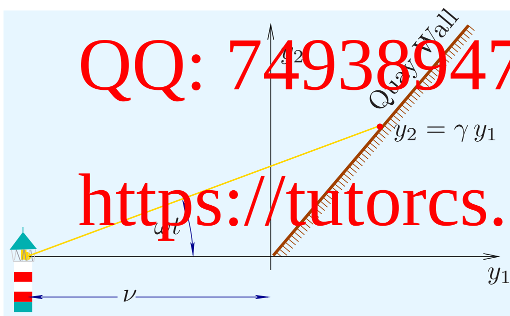

Fig. 1. Lighthouse Example

These two symbolic expressions might be evaluated by the simple program shown below in Fig. 2. Here, the distance $x_1 = \nu$, the slope $x_2 = \gamma$, the angular velocity $x_3 = \omega$, and the time $x_4 = t$ form the independent variables. Subsequently, six statements are evaluated using arithmetic operations and elementary functions. Finally, the last two intermediate values are assigned to the dependent variables y_1 and y_2.

For the computation of derivatives with ADOL-C, one has to perform the changes of the source code as described in the previous section. This yields the code segment given on the left hand side of Fig. 3, where modified lines are marked with /* ! */. Note that the function evaluation itself is completely unchanged. If this

```
...
int main()
                                        ;    /* inputs nu, gamma, omega, t */
                                             /* intermediates */
                                             /* outputs */

                                             /* some input values */


                                             /* function evaluation */
    v2 = tan(v1);
    v1 = x2-v2;
    v3 = x1*v2;
    v2 = v3/v1;
    v3 = v2*x2;

    y1 = v2; y2 = v3;                        /* output values */
}
```


Fig. 2. Source Code for Lighthouse Example (double Version)


```
/* ! */   #include "adolc.h"
          ...
          int main()
          {
/* ! */     adouble x1, x2, x3, x4;
/* ! */     adouble v1, v2, v3;
/* ! */     adouble y1, y2;

/* ! */     trace_on(1);

/* ! */     x1 <<= 3.7; x2 <<= 0.7;
/* ! */     x3 <<= 0.5; x4 <<= 0.5;

            v1 = x3*x4;
            v2 = tan(v1);
            v1 = x2-v2;
            v3 = x1*v2;
            v2 = v3/v1;
            v3 = v2*x2;

/* ! */     y1 >>= v2; y2 >>= v3;

/* ! */     trace_off();

          }
```

ADOL−C tape
trace_on, tag
<<=, x1, 3.7
<<=, x2, 0.7
<<=, x3, 0.5
<<=, x4, 0.5
*, x3, x4, v1
tan, v1, v2
−, x2, v2, v1
*, x1, v2, v3
/, v1, v3, v2
*, v2, x2, v3
>>=, v2, y1
>>=, v3, y2

Fig. 3. Source Code for Lighthouse Example (adouble Version) and Tape

adouble version of the program is executed, ADOL-C generates an internal function repr[illegible] the tape. The tape for the lighthouse example is sketch[illegible]de of Fig. 3. Once the internal representation is genera[illegible] ADOL-C can be used to compute the desired derivativ[illegible]

4 Eas[illegible]

For the [illegible], ADOL-C provides several easy-to-use drivers that compute the most frequently required derivative objects. Throughout, it is assumed that after the execution of an active section, the corresponding tape with the identifier tag contains a detailed record of the computational process by which the final values y of the dependent variables were obtained from the values x of the independent variables. This functional relation between the input variables x and the output variables y is denoted by

$$F : \mathbb{R}^n \mapsto \mathbb{R}^m, \qquad x \longrightarrow F(x) \equiv y.$$

The presented drivers are all C functions and therefore can be used within C and C++ programs. Some Fortran-callable companions can be found in the appropriate header files.

For the calculation of whole derivative vectors and matrices up to order 2 there are the following procedures:

```
int gradient(tag,n,x,g)
short int tag;              // tape identification
int n;                      // number of independents n and m = 1
double x[n];                // independent vector x
double g[n];                // resulting gradient ∇F(x)

int jacobian(tag,m,n,x,J)
short int tag;              // tape identification
int m;                      // number of dependent variables m
int n;                      // number of independent variables n
double x[n];                // independent vector x
double J[m][n];             // resulting Jacobian F'(x)

int hessian(tag,n,x,H)
short int tag;              // tape identification
int n;                      // number of independents n and m = 1
double x[n];                // independent vector x
double H[n][n];             // resulting Hessian matrix ∇²F(x)
```

The driver routine hessian computes only the lower half of $\nabla^2 f(x)$ so that all values H[i][j] with $j > i$ of H allocated as a square array remain untouched during the call of hessian. Hence only $i + 1$ doubles need to be allocated starting at the position H[i].

To use the full capability of automatic differentiation when the product of derivativ[...]ectors or directions are needed, ADOL-C offers the follo[...]

```
i...                                   // result z = F'(x)v
i...t,x,u,z)                           // result z = u^T F'(x)
i...                                   // result z = ∇^2 F(x)v
```

A detaile[...]rface of these drivers can be found in the ADOL-C docum[...] ADOL-C provides several drivers for special cases of [...]or solution curves defined by ordinary differential equations, special routines are provided that evaluate the Taylor coefficient vectors and their Jacobians with respect to the current state vector. For explicitly or implicitly defined functions derivative tensors are obtained with a complexity that grows only quadratically in their degree. In addition to the routines for derivative evaluation, ADOL-C provides functions for an appropriate memory allocation. Using these facilities, one may compute derivatives of the lighthouse example presented in the last section by the following code segment given in Fig. 4.


```
...                              /* as above */
trace_off();

double xp[4];                    /* passive inputs nu, gamma, omega, t */
xp[0] = 3.7; xp[1] = 0.7;        /* some input values */
xp[2] = 0.5; xp[3] = 0.5;

double** J;                      /* Calculate F' */
double *x1, *y1;                 /* Calculate y1 = F'(xp)*x1 */
double *x2, *y2;                 /* Calculate x2 = y2^T*F'(xp) */

J = myalloc(2,4);
x1 = myalloc(4); y1 = myalloc(2);
x2 = myalloc(4); y2 = myalloc(2);

jacobian(1, 2, 4, xp, J);

xp[0] = 2.0; xp[1] = 1.0;        /* change independents */

jac_vec(1, 2, 4, xp, y1, x1);
vev_jac(1, 2, 4, 0, xp, y2, x2);
...                              /* do something with the derivatives */
```

Fig. 4. Derivative Calculation with ADOL-C

Quite often, the Jacobians and Hessians that have to be computed are sparse matrices. Therefore, ADOL-C provides additionally drivers that allow the exploitation of sparsity. The exploitation of sparsity is frequently based on *graph coloring* [illegible] example in [2] and [3]. To compute the entries of sparse [illegible] Hessians, respectively, in coordinate format one may use

```
i[illegible]eat,x,&nnz,&rind,&cind,&values,&options)
i[illegible]at,x,&nnz,&rind,&cind,&values,&options)
```

Once mo[illegible]n of the calling structure can be found in the documen[illegible]

5 Tapeless Forward Differentiation

Up to version 1.9.0, the development of the ADOL-C software package was based on the decision to store all data necessary for derivative computations on tapes, since these tapes enable ADOL-C to offer a very broad functionality. However, really large-scale applications may require the tapes to be written out to corresponding files. In almost all cases this means a considerable drawback in terms of run time due to the excessive memory accesses. Nevertheless, there are some tasks with respect to derivative computation that do not require a tape.

Starting with version 1.10.0, ADOL-C now features a tapeless forward mode for computing first order derivatives in scalar mode, i.e., $\dot{y} = F'(x)\dot{x}$, and in vector mode, i.e., $\dot{Y} = F'(x)\dot{X}$. These tapeless variants coexist with the more universal tape based mode in the package. Because of the different implementation strategy, also the required modification of the source code are different as illustrated in Fig. 5 for the scalar mode. After defining the variables as adoubles only two things are left to do. First one needs to initialize the values of the independent variables for the function evaluation. This can be done by assigning the variables a double value. Then, the corresponding derivative value is set to zero. Alternatively, the ADOL-C offers a function named setValue for setting the value of a variable without changing the derivative part. To set the derivative components of the independent variables, ADOL-C provides two possibilities:

– Using the constructor

```
adouble x1(2,1), x2(4,0), y;
```

  This would create the three variables x_1, x_2 and y. Obviously, the latter remains uninitialized. The variable x_1 holds the value 2, x_2 the value 4 whereas the derivative values are initialized to $\dot{x}_1 = 1$ and $\dot{x}_2 = 0$, respectively.
– Setting point values directly

```
adouble x1=2, x2=4, y;
...
x1.setADValue(1);
x2.setADValue(0);
```

  The same example as above but now using setADValue-method for initializing the derivative values.

```
/* ! */     #ADOLC_TAPELESS
/* ! */     #include "adolc.h"
/* ! */     adouble;



/* ! */     x4;
/* ! */
/* ! */

            x3 = 0.5; x4 = 0.5;             /* initialization of x */

/* ! */       x1.setADValue(1); x2.setADValue(0);   /* initialization of ẋ */
/* ! */       x3.setADValue(0); x4.setADValue(0);

              v1 = x3*x4; v2 = tan(v1); v1 = x2-v2;   /* same as before */
              v3 = x1*v2; v2 = v3/v1; v3 = v2*x2;

              y1 = v2; y2 = v3;

/* ! */       cout ≪ y0.getADValue() ≪ " " ≪ y1.getADValue() ≪ endl;

            }
```

Fig. 5 Source Code for Lighthouse Example (Tapeless adouble Version)

The derivatives can be obtained at any time during the evaluation process by calling the getADValue method:

```
adouble y;
...
cout ≪ y.getADValue();
```

Similar to the tapeless scalar forward mode, the tapeless vector forward mode can be applied by defining ADOLC_TAPELESS and an additional preprocessor macro named NUMBER_DIRECTIONS. This macro takes the maximal number of directions to be used within the resulting vector mode. Just as ADOLC_TAPELESS the new macro must be defined before including the adolc.h header file since it is ignored otherwise. A more detailed description of the tapeless vector forward mode and its usage can be found in the documentation of ADOL-C.

6 Recent Developments

Advanced differentiation techniques had been integrated recently in the ADOL-C tool. This comprises for example the optimal checkpointing for time integrations when the number of time steps is known in advance. For this purpose, ADOL-C employs the routine revolve for a binomial checkpointing [8] to achieve a enormous reduction of the memory required for calculation the adjoint of the time-dep[illegible] more, ADOL-C allows now the exploitation of fixed poi[illegible]ng drivers for the reverse accumulation [9] for the mem[illegible]n of adjoints. Additionally, first drivers for the differenti[illegible]el programs are included in the current version of ADOL[illegible] of MPI-parallel programs with ADOL-C is the subject o[illegible] planed to integrate corresponding routines into ADOL-C[illegible]

References

1. ColPack. (http://www.cscapes.org/coloringpage/software.htm)
2. Gebremedhin, A., Manne, F., Pothen, A.: What color is your Jacobian? Graph coloring for computing derivatives. SIAM Rev. **47** (2005) 629–705
3. Gebremedhin, A., Tarafdar, A., Manne, F., Pothen, A.: New acyclic and star coloring algorithms with application to computing Hessians. SIAM J. Sci. Comput. **29** (2007) 1042–1072
4. Bull, B., Francis, R.I.C.C., Dunn, A., McKenzie, A., Gilbert, D.J., Smith, M.H.: CASAL (C++ algorithmic stock assessment laboratory) – User Manual. Technical Report 127, NIWA, Private Bag 14901, Kilbirnie, Wellington, New Zealand (2005)
5. Hart, F.P., Kriplani, N., Luniya, S.R., Christoffersen, C.E., Steer, M.B.: Streamlined Circuit Device Model Development with fREEDA® and ADOL-C. In Bücker, H.M., Corliss, G., Hovland, P., Naumann, U., Norris, B., eds.: Automatic Differentiation: Applications, Theory, and Implementations. Lecture Notes in Computational Science and Engineering. Springer (2005) 295–307
6. Bock, H.G., Leineweber, D.B., Schafer, A., Schloder, J.P.: An Efficient Multiple Shooting Based Reduced SQP Strategy for Large-Scale Dynamic Process Optimization – Part II: Software Aspects and Applications. Computers and Chemical Engineering **27** (2003) 167–174
7. Griewank, A., Walther, A.: Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation (Second Edition). SIAM, Philadelphia (2008)
8. Griewank, A., Walther, A.: Algorithm 799: Revolve: An implementation of checkpoint for the reverse or adjoint mode of computational differentiation. ACM Trans. Math. Softw. **26** (2000) 19–45
9. Christianson, B.: Reverse accumulation and attractive fixed points. Optimization Methods and Software **3** (1994) 311–326