Discussiones Mathematicae
Differential Inclusions, Control and Optimization 27 (2007) 119–134

EFFICIENT CALCULATION OF SENSITIVITIES FOR OPTIMIZATION PROBLEMS

Andreas Kowarz and Andrea Walther

Institute of Scientific Computing
TU Dresden, 01062 Dresden, Germany

Abstract

Sensitivity information is required by numerous applications such as, for example, optimization algorithms, parameter estimations or real time control. Sensitivities can be computed with working accuracy using the forward mode of automatic differentiation (AD).

ADOL-C is an AD-tool for programs written in C or C++. Originally, when applying ADOL-C, tapes for values, operations and locations are written during the function evaluation to generate an internal function representation. Subsequently, these tapes are evaluated to compute the derivatives, sparsity patterns etc., using the forward or reverse mode of AD. The generation of the tapes can be completely avoided by applying the recently implemented tapeless variant of the forward mode for scalar and vector calculations. The tapeless forward mode enables the joint computation of function and derivative values directly from main memory within one sweep. Compared to the original approach shorter runtimes are achieved due to the avoidance of tape handling and a more effective, joint optimization for function and derivative code.

Advantages and disadvantages of the tapeless forward mode provided by ADOL-C will be discussed. Furthermore, runtime comparisons for two implemented variants of the tapeless forward mode are presented. The results are based on two numerical examples that require the computation of sensitivity information.

Keywords: automatic differentiation, sensitivities, forward mode.

2000 Mathematics Subject Classification: 65D25, 65K05, 68Q25.

1. Introduction

Automatic Differentiation (AD) offers an efficient way of calculating derivative information with machine accuracy for a function given as source code in a supported programming language. The reverse mode [6] of AD yields the exact gradient of a given scalar-valued function at a cost of ω times the function evaluation, with $\omega \in [3, 4]$. This means, in particular, that in contrast to the application of Finite Differences the costs for evaluating the gradient are independent of its dimension. Despite the good complexity measure for the reverse mode many applications benefit from the forward mode of AD, e.g., if the minimization of a given optimization problem only requires certain gradient information. Furthermore, compression techniques may be used in practice, when handling sparse Jacobian matrices. Frequently, computing a compressed matrix makes the forward mode attractive and competitive. Other important applications are parameter estimation problems or the computation of complete Jacobians. Using the vector forward mode of AD one only needs one sweep to compute the n columns of the Jacobian in machine accuracy at a cost of ω times the function evaluation, with $\omega \in [1 + n, 1 + 1.5n]$ (see [6, Section 3.2]).

Two different implementation strategies have been frequently used for the Automatic Differentiation of source codes. One approach concentrates on the development of special compilers which analyze the given program, build optimized dependency trees and finally create appropriate derivative codes. This so-called "Source-to-Source" transformation that is applied, e.g., by TAF [4] and Tapenade [8], will not be discussed in this paper. Currently, the special AD-enabled compilers are mainly available for the FORTRAN programming language.

The other approach is characterized by the usage of the Operator Overloading abilities offered by most modern high level programming languages. The mechanism of overloading allows for extending or changing the meaning of operators and functions by replacing them by user defined source codes. Applying this facility within the AD context results in modified codes, able to jointly compute function and derivative information using the forward mode or to produce an internal function representation that serves for the required derivative calculations, especially the reverse mode differentiation. Operator Overloading is used by numerous tools, e.g., ADOL-C [5], CppAD [1] and FADBAD [2]. Due to the overloading on the level of operations and intrinsic functions, most branch and loop information as well as subroutine calls cannot be included into the internal function representation. This fact enforces a complete unrolling of the program that can result in a very large internal representation. If the required derivative information can be computed using the forward mode of AD, the generation of the internal representation can be avoided by computing function and derivative values jointly within one sweep. In this paper, we present a new work mode for the AD-tool ADOL-C, based on this strategy.

The structure of this paper is the following: In Section 2, we present the new mode of ADOL-C. This includes an analysis of the implemented derivative handling as well as a description of the interface extensions. At the end of the section, we present a short implementation example to demonstrate the demands on the user. Two numerical examples using the new work mode are presented in Section 3. We start with the Medical Akzo Nobel Problem as an academic example to demonstrate some results concerning the basic implementation strategies as well as the tapeless vector forward mode. We close the section with numerical results for a sensitivity computation of an optimal turnaround maneuver performed by an industrial robot as a more realistic application. Finally, conclusions and an outlook are given in Section 4.

2. Tapeless Computations using ADOL-C

Compared to the Source-to-Source approach, an implementation of the Automatic Differentiation based on Operator Overloading has to deal with certain additional difficulties. Since most control information is not available at the operation level an internal representation of the program, i.e., an operation trace, has to be used to enable derivative calculations based on the forward and reverse mode of AD. However, as mentioned in Section 1 applying the reverse mode of AD is not always the most efficient way of computing the required derivatives. In the case of Jacobians that are square matrices or a small number of required sensitivities the forward mode of AD is advantageous in terms of runtime and memory requirement. Under abandonment of the higher flexibility provided by the internal function representation one can compute function and derivative information jointly within one forward sweep. Necessary changes for a tapeless work mode in ADOL-C are discussed in the following subsection.

Implementation Details

Assuming that derivative information is propagated along with the function evaluation the implementation strategy is quite obvious. It must be ensured that the codes for computing the function and the derivative value are both executed within the operation's scope. The new ADOL-C work mode *Tapeless* uses the well established active class `adouble` with an implementation as described below. The basic layout of the class is the following:

```
class adouble {
   double val;
   double ad_val;
};
```

In this class all basic operations are overloaded to compute the normal operation as well as the corresponding derivative value, e.g.:

```
adouble adouble::operator + (const adouble &a) {
   adouble res;
   res.val = val + a.val;
   res.ad_val = ad_val + a.ad_val;
   return res;
};
```

This approach is well known from the AD theory [6] and is sometimes said to be the trivial solution. However, for the class of problems mentioned, it is a very promising way for an efficient implementation.

The type of the variable `ad_val` can be changed from `double` to a vector of double values for applying the vector version of the forward mode. The dimension of the vector is selected at compile time and therefore yields a fixed spatial complexity. Nevertheless, the propagation of all directional derivatives in the forward mode can be avoided at runtime by calling the special function `setNumDir(int)`, as described below in more detail.

Another important question concerns code maintainability and speed. The tape-based part of ADOL-C is provided as shared library. The main advantage of such an approach is the reduction of binary size for all programs using the library. Additionally, the provided code is better maintainable since bug fixes within the library are automatically a part of the programs that use this library. However, applying the shared library approach for the implementation of a tapeless mode would mean to separate the often small function code from the derivative code. A good optimization by the compiler is not very likely in this case. Therefore, we decided to implement the tapeless version as a completely inlined code. This avoids a lot of subroutine calls that correspond to jumps into and back from a library. Furthermore, the compiler gets the source code that contains the computations only. The downside is the code maintainability, which is reduced to a level comparable to the usage of static libraries.

To illustrate the demands on the user, we demonstrate the application of the new functionality of ADOL-C in the following subsection.

Implementation Examples

Assume that we want to evaluate for a given function $F : I\!R^n \to I\!R^m$ the Jacobian×vector product

$$\dot{y} = F'(x) * \dot{x} \qquad \dot{x} \in I\!R^n, \dot{y} \in I\!R^m, F'(x) \in I\!R^{m \times n} .$$

Applying the new tapeless scalar forward mode of ADOL-C results in a source code similar to the one shown in Figure 1. Most of the AD-related changes to the original source code are common to both, the tape-based

```
#define ADOLC_TAPELESS
#include <adolc.h>
...
adouble *xa=new adouble[n], *ya=new adouble[m];
for (int i=0; i<n; ++i) {
   xa[i]=<double value>;                // set x
   xa[i].setADValue(<double value>);   // set ẋ
}
...
< evaluate ya = F(xa) >
...
for (int i=0; i<m; ++i) {
   cout << ya[i].getValue();            // get y
   cout << ya[i].getADValue();          // get ẏ
}
```

Figure 1. Tapeless derivative computation using ADOL-C

and the tapeless forward mode in ADOL-C. The first important step is to define the preprocessor macro `ADOLC_TAPELESS`. This has to be done before including the header file `adolc.h`. Furthermore, all intermediate variables that depend on the input variables xa and that are required for the computation of $F(xa)$ must be declared of type `adouble`. As done in line 7 of Figure 1, the user has to provide the actual direction $\dot{x}$ by calling the function `setADValue(..)` for every independent variable. After evaluation of the function the derivative values are accessible by calling the function `getADValue()` for the dependent variables of interest. Hence, the Jacobian×vector product $F'(x)\dot{x}$ is computed together with the function evaluation.

Whenever one needs several directional derivatives, i.e., a Jacobian× matrix product of the form

$$\dot{Y} = F'(x) * \dot{X} \qquad \dot{X} \in I\!R^{n\times p}, \dot{Y} \in I\!R^{m\times p}, F'(x) \in I\!R^{m\times n},$$

one can either execute the scalar forward mode several times or benefit from the vector version of the forward mode of AD. The advantage of the latter approach is that the function values and common intermediate values as well as the local partial derivatives need to be computed only once. Then, the derivative values can be computed for all directions, simultaneously, using this information. This approach calls for a higher memory requirement but reduces the overall runtime considerably. Compared to the tapeless scalar forward mode in ADOL-C only a small number of changes have to be made for the vector version. First, the maximal number of directions to be propagated must be supplied by defining the preprocessor macro `NUMBER_DIRECTIONS`. This has to be done before including the ADOL-C header, for example in the following way:

```
#define ADOLC_TAPELESS
#define NUMBER_DIRECTIONS 10
#include <adolc.h>
```

Second, for the vector version of the forward mode the type of `ad_val` changes from a single `double` into a vector of `doubles`. The function for setting these values must be changed from

```
setADValue(double value)
```
to
```
setADValue(double *valueVector) or
setADValue(int index, double value).
```

Correspondingly, derivative values can be accessed using the functions

`double *getADValue()` or
`double getADValue(int index)`.

In addition, the function `setNumDir(int number)` can be used at runtime to set the actual number of computed directions to the value of `number` which has to be less than or equal to `NUMBER_DIRECTIONS`. Due to the size determination for `adouble` variables at compile time the storage size remains a multiple of `NUMBER_DIRECTIONS`, even for smaller values of `number`. Properties of the tapeless vector forward mode of ADOL-C are discussed as part of the numerical experiments in the following section.

3. Numerical Examples

To demonstrate the new tapeless forward mode contained in the current version 1.10.0 of ADOL-C, we use an academic example to demonstrate some basic results concerning the implementation strategy and the vector mode. Subsequently, we consider the sensitivity computation for an industrial robot as a more realistic application.

All results are based on the following two test systems:

Test system P3:

- Pentium-IIIE 700MHz (Coppermine)
- 16 KB L1-Data-Cache
- 256 KB L2-Cache
- 378 MB Main Memory
- GCC Version 3.2

Test system PM:

- Pentium-M 725 1,6 GHz (Dothan)
- 32 KB L1-Data-Cache
- 2 MB L2-Cache
- 1 GB Main Memory
- GCC Version 3.4.4

Medical Akzo Nobel Problem

The following problem was formulated by the Akzo Nobel research laboratories during their studies of the penetration of radio-labeled antibodies into a tumor infected tissue. It can be derived from the following two partial differential equations

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} - kuv\,,$$
$$\frac{\partial v}{\partial t} = -kuv\,.$$

Semi-discretization leads to a stiff ODE of the form

$$\frac{\mathrm{d}y}{\mathrm{d}t} = f(t,y), \quad y(0) = g \qquad y \in I\!R^{2N},\ 0 \le t \le 20,\ g \in I\!R^{2N}$$

with $g = (0, v_0, 0, v_0, \ldots, 0, v_0)^{\mathrm{T}}$ and the grid constant N as a user supplied parameter. In addition, one has $y_{-1}(t) = \phi(t)$ and $y_{2N+1} = y_{2N-1}$ with

$$\phi(t) = \begin{cases} 2 & \text{for } t \in (0,5]\,, \\ 0 & \text{for } t \in (5,20]\,. \end{cases}$$

The function f is given by

$$\left.\begin{aligned} f_{2j-1} &= \alpha_j \tfrac{y_{2j+1}-y_{2j-3}}{2\Delta\zeta} + \beta_j \tfrac{y_{2j-3}-2y_{2j-1}+y_{2j+1}}{(\Delta\zeta)^2} - ky_{2j-1}y_{2j}, \\ f_{2j} &= -ky_{2j-1}y_{2j}, \end{aligned}\right\} 1 \le j \le N$$

with the coefficients defined by

$$\alpha_j = \frac{2(j\Delta\zeta - 1)^3}{c^2}, \quad \beta_j = \frac{(j\Delta\zeta - 1)^4}{c^2} \quad \text{and} \quad \Delta\zeta = \frac{1}{N}.$$

Throughout the numerical tests the constants $k = 100$, $v_0 = 1$ and $c = 4$ have been used. The problem under consideration was contributed to the Test Set for Initial Value Problems by R. van der Hout from the Akzo Nobel Central Research. For a more detailed description see [9, II-4].

For our basic analysis we only used the right-hand side of the ODE, i.e., the function f. Since we were less interested in a good approximation of the solution of the problem itself but in information about the behavior of the tapeless code we used the vector mode for 10 directions $\dot{X} \in I\!R^{2N\times 10}$ to compute derivatives $F'(x)\dot{X}$ at a given argument x. However, to minimize the influence of operating system processes we averaged the time measurements for the combined function-derivative calculation over a loop of 50 iterations. Furthermore, we tested the code as both inlined and library version.

Table 1. Code sizes for the Medical Akzo Nobel Problem

double version:	5981 Byte
adouble version (tapeless):	12096 Byte
adouble version (library):	25227 Byte

As summarized in Table 1, the practical studies do not completely confirm our theoretical assumptions. The `double` version has the smallest size as expected. However, the size of the library version is significantly higher than the size of the inlined version. Obviously, the code for calling the derivative functions within the library has a larger size than the code of the function itself.

Table 2. Runtimes for the Medical Akzo Nobel Problem

System	T_d (double)	T_a (adouble)	T_a/T_d	Theory
P3	inlined version			
	0.1011 s	2.8809 s	28.50	11-16
	library version			
	0.1011 s	3.5067 s	34.69	11-16
PM	inlined version			
	0.0257 s	0.7072 s	27.52	11-16
	library version			
	0.0257 s	0.9139 s	35.56	11-16

Another important aspect concerns the runtime of the specific binaries. Table 2 illustrates the observed execution times. For all binaries we used the GNU Compiler Collection applying optimization level "O3" and appropriate CPU-based special optimization for the target system. As can be seen from the practical results, a smaller runtime could be achieved for the derivative computation when using the inlined version in comparison to the usage of the library-based version. This observation fulfills our theoretical assumption.
To proceed with theory–practice comparisons, we analyze the runtime ratio of the `double` version and the specific `adouble` version. For this ratio an upper bound, denoted by ω, is available due to the complexity theory of the Automatic Differentiation. One has

$$\omega \in [1 + p, 1 + 1.5p]$$

for the propagation of p directions within one sweep (see [6, Section 3.2]). In our example with $p = 10$ the theoretical upper bound of the runtime ratio is $\omega \in [11, 16]$ as stated in column 5 of Table 2. Comparing this theoretical measure with our practical results we observe a higher runtime ratio than expected. This result is hardly surprising since one of the basic assumptions for the theory (see [6, Section 2.5]) is a flat memory model that may not reflect the practical circumstances.

A more interesting observation can be received from the theory of the vector mode itself. For every elementary function

$$v_i = \varphi_i(v_j) \quad \text{with} \quad v_i, v_j \in I\!R,$$

evaluated to compute $F(x)$, i.e., for every operator or function that gets overloaded, we denote the elementary partials $\frac{\partial \varphi_i}{\partial v_j}(v_j)$ by c_{ij}. In the scalar mode, the derivative values have to be computed as

$$\dot{v}_i = c_{ij} * \dot{v}_j \quad \text{with} \quad \dot{v}_i, \dot{v}_j \in I\!R.$$

For the vector mode the derivative formula changes to

$$\dot{V}_i = c_{ij} * \dot{V}_j \quad \text{with} \quad \dot{V}_i, \dot{V}_j \in I\!R^p .$$

The basic idea of the vector mode is to reuse the elementary partials for computing the scalar vector product instead of recomputing them for all directions when using p scalar sweeps. Obviously, the runtime ratio `double` code to `adouble` code is the better the more impact the reusable fraction of the computation has. To demonstrate this aspect we use a modified version of the Medical Akzo Nobel Problem where we replace the formula for β_j by

$$\beta_j = \tan\left(\frac{(j\Delta\zeta - 1)^4}{c^2}\right) .$$

Then, the code structure is nearly the same but we increase the impact of the reusable fraction of the code. This means we add the computational expensive quotient $\frac{1}{\cos^2(\ldots)}$ to the set of elementary partials $\{c_{ij}\}$ that gets evaluated only once for the propagation of $p = 10$ directions.

The averaged runtimes for the modified Medical Akzo Nobel Problem are summarized in Table 3. A first interesting observation concerns the high computational costs for computing the trigonometrical functions. In the case

Table 3. Runtimes for the modified Medical Akzo Nobel Problem

System	T_d (double)	T_a (adouble)	T_a/T_d	Theory
P3	inlined version			
	0.3031 s	3.2965 s	10.88	11-16
	library version			
	0.3031 s	3.9230 s	12.94	11-16
PM	inlined version			
	0.0989 s	0.9139 s	9.24	11-16
	library version			
	0.0989 s	1.0791 s	10.91	11-16

of the `double` version the addition of the tangent leads to an increase in the runtime by factor 3 for test system P3 and by factor 4 for test system PM, respectively, whereas the runtime of the `adouble` version increases only by a small fraction. Accordingly, the runtime ratio is much better than in the original version and it now nicely complies to the theory. Results for the tapeless scalar mode are discussed in the following subsection in more detail.

Optimal Turn-around Maneuver of an Industrial Robot

The numerical example that serves here to illustrate the runtime effects of the tapeless scalar forward mode of ADOL-C is an industrial robot as depicted in Figure 2 that has to perform a fast turn-around maneuver. We denote by $q = (q_1, q_2, q_3)$ the angular coordinates of the robot's joints, q_1 referring to the angle between the base and the two-arm system. The robot is controlled via three control functions u_1 through u_3, denoting the respective angular momentum applied to the joints (from bottom to top) by electrical motors.

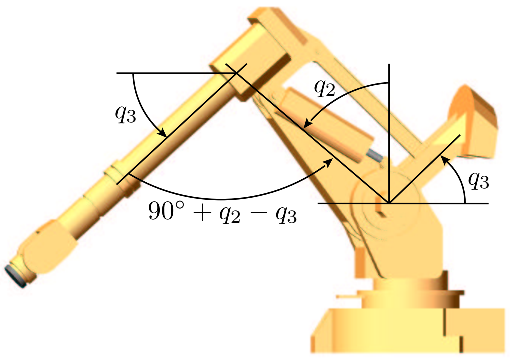

Figure 2. Industrial robot ABB IRB 6400

The control problem under consideration is to minimize the energy-related objective

$$J(q,u) = \int_0^{t_f} [u_1(t)^2 + u_2(t)^2 + u_3(t)^2]\, dt,$$

where the final time t_f is given. The robot's dynamics obey a system of three differential equations of second order:

$$M(q)\,\ddot{q} = v(q,\dot{q}) + w(q) + \tau_{\text{friction}}(\dot{q}) + \tau_{\text{reset}}(q) + u$$

where $M(q)$ is a 3×3 symmetric positive definite matrix containing moments of inertia, called a generalized mass matrix. The vector v is composed of centrifugal and Coriolis force entries, and w contains the gravitational influence. Finally, we allow for forces induced by dry friction and reset forces by means of τ_{friction} and τ_{reset}, respectively. The complete equations of motion can be found in [10]. The robot's task to perform a turn-around maneuver is expressed by means of initial and terminal conditions as well as control constraints [7]. To compute an approximation of the corresponding trajectory, we apply the standard Runge-Kutta method of order 4 for the integration resulting in about 800 lines of code.

Optimizing the target function J as done in the original problem is not the objective of our example. Here, we are interested in the sensitivities of the optimal trajectory with respect to perturbations of specific elements of the parameter vector p. Therefore, we need partial derivatives of the target function J with respect to the parameters in question which may be, e.g., the start position of the object the robot works on, the weight of the object and so on. Since only a small number of sensitivities are needed, the forward mode of the Automatic Differentiation is the instrument of choice for our example.

Table 4 summarizes the runtime results for one evaluation of J in the `double` case as well as J and $\partial J/\partial p_1$ in the `adouble` case. Throughout our experiments we used the values 100, 500, 1000, 5000 for the number of time steps l, thus enforcing different increments for the Runge-Kutta scheme. According to the number of time steps the memory requirements for the `adouble` version are roughly 10 KB, 50 KB, 100 KB and 500 KB, respectively, and half the size for the `double` version.

As in the vector case a runtime ratio ω based on a flat memory model is given by AD theory [6, Section 3.2]. One gets:

$$\omega \in [2, 2.5]\,.$$

Table 4. Runtimes for the robot example

System	# steps	T_d (double)	T_a (adouble)	T_a/T_d	Theory
P3	100	0.0046 s	0.0097 s	2.11	2-2.5
	500	0.0353 s	0.0728 s	2.06	
	1000	0.1006 s	0.2086 s	2.07	
	5000	4.1695 s	18.1271 s	4.35	
PM	100	0.0017 s	0.0030 s	1.76	2-2.5
	500	0.0102 s	0.0225 s	2.21	
	1000	0.0292 s	0.0610 s	2.09	
	5000	0.4064 s	1.5230 s	3.75	

As can be seen, the values of Table 4 fulfill the theoretical expectations for all test cases apart from the case $l = 5000$.

A reason for the higher runtime ratio in the test case $l = 5000$ on system P3 may be the small L2 cache of the processor that cannot hold all program data at the same time. As a result, expensive main memory accesses are necessary that burden the runtime of the `adouble` version. However, the same assumption cannot explain the result of the test case $l = 5000$ on system PM. The L2 cache of this system is large enough to hold all data for the specific test case. Obviously, a deeper insight into the real behavior of the code is necessary. Figure 3 summarizes some results we collected during our extended test runs.

We increased the number of test cases by varying the number of time steps l from 100 to 7000 with an increment of 100. The first four plots shown in Figure 3 depict the corresponding runtimes for all experiments as well as the resulting runtime ratio for the two test systems. Especially, the results for system PM clarify the runtime ratio of the case $l = 5000$. Investigating the runtimes, we can see a strong increase at $l = 2800$ for the `adouble` version and at $l = 5600$ for the `double` version. Between these two points the runtime ratio consequently jumps to a much higher level. The fourth test case depicted in Table 4 is clearly affected by this jump.

In the case of test system P3 a more complicated situation occurs since the small L2 cache of the processor has to be taken into account. To get more information about the practical code behavior, we use a modified Linux version that is able to capture hardware performance data through the PAPI system [3]. As expected, the increase in the L2 cache miss rate depicted in Figure 3 leads to a higher runtime that results in the higher ratio. Again, as

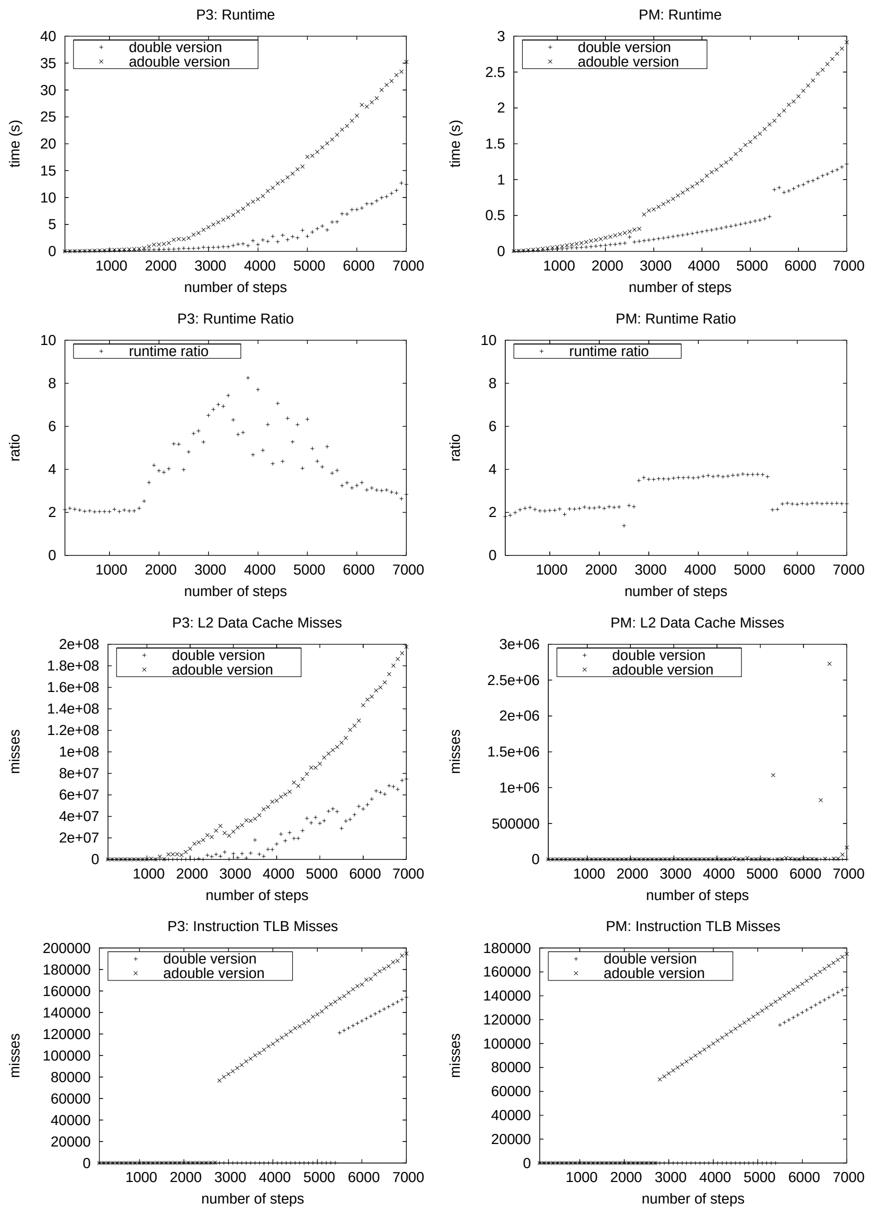

Figure 3. Detailed measurements for the two test systems

for the test system PM we observe a strange behavior between $l = 2800$ and $l = 5600$ that is not sufficiently described by L2 cache misses. The depicted results are based on the interaction of the cache misses and the jump in the runtime ratio already known from the system PM.

However, the results discussed so far do not explain the jumps in the runtime, especially for test system PM. The last two plots of Figure 3 summarize the Translation Lookaside Buffer (TLB) miss rate for the `double` and the `adouble` version of the code. The TLB contains pairs of virtual addresses, our program works with, and physical addresses, the caches work with. Using these fast buffers, the processor can avoid the expensive translation between the two kinds of addresses. Whenever a TLB miss occurs the mentioned translation is invoked by the processor. As a very likely result, operations are delayed thus causing an increase in the runtime. Unfortunately, the processors of our test systems do not offer facilities to capture all TLB misses but only Instruction-TLB misses. Control measurements on an AMD platform that also creates the runtime jumps in question show an analog behavior of the Data-TLB miss rates compared to Instruction-TLB miss rates. On our test systems we can observe a drastic increase, i.e., a jump, in the TLB miss rates for the two test systems at exactly the positions where the corresponding runtimes perform a jump. We can conclude that the jumps in the runtimes for $l = 2800$ and $l = 5600$ and hence also the jumps in the runtime ratio are caused by the high penalty for the TLB misses. Further investigations concerning the reasons for the jumps in the TLB miss rates are possible but require at least a code insight at assembler level what is far beyond the scope of this paper.

4. Conclusion and Outlook

We presented a new functionality for the AD-tool ADOL-C based on the tapeless evaluation of derivatives for function codes given in C/C++. This standalone forward mode should be applied to compute sensitivities for a small number of independents or for computing square sized Jacobians. Our practical results confirm good runtime ratios comparable to the assumptions of the AD theory.

Although first order derivatives suffice for many applications we want to provide support for more complicated derivative computations. Therefore, we plan to implement a tapeless forward version for computing higher order derivatives based on Taylor arithmetic. An equivalent mode already exists in ADOL-C for tape-based computations.

REFERENCES

[1] M.B. Bell, CppAD: A package for C++ Algorithmic Differentiation, COIN-OR foundation, Available at `http://www.coin-or.org/CppAD`

[2] C. Bendtsen and O. Stauning, *FADBAD, a flexible C++ package for automatic differentiation*, Department of Mathematical Modelling, Technical University of Denmark, 1996, Available at `http://www.imm.dtu.dk/fadbad.html`

[3] J. Dongarra, S. Moore, P. Mucci, K. Seymour, D. Terpstra, Q. Xia and H. You, Performance Application Programming Interface, Innovative Computing Laboratory, Department of Computer Science, University of Tennessee, Available at `http://icl.cs.utk.edu/papi`

[4] R. Giering and T. Kaminski, *Recipes for Adjoint Code Construction*, ACM Trans. Math. Software **24** (1998), 437–474. URL: `http://www.fastopt.com`

[5] A. Griewank, A. Kowarz and A. Walther, Documentation of ADOL-C, Available at `http://www.math.tu-dresden.de/~adol-c`, Updated Version of: A. Griewank, D. Juedes, J. Utke J, ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. ACM Trans. Math. Software **22** (1996), 131–167.

[6] A. Griewank A, Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, Number 19 in Frontiers in Appl. Math. SIAM, Philadelphia, 2000.

[7] R. Griesse and A. Walther, *Parametric sensitivities for optimal control problems using automatic differentiation*, Optimal Control Applications and Methods **24** (2003), 297–314.

[8] L. Hascoët and V. Pascual, Tapenade 2.1 user's guide. Tech. rep. 300, INRIA, 2004.

[9] F. Mazzia F and F. Iavernaro, Test Set for Initial Value Problem Solvers, Entire test set description, Department of Mathematics, University of Bari, August 2003, Available at `http://www.pitagora.dm.uniba.it/~testset`

[10] M. Knauer and C. Büskens, *Real-Time Trajectory Planning of the Industrial Robot* IRB6400, PAMM. **3** (2003), 515–516.

Received 21 March 2006