

# Performance of automatic differentiation tools in the dynamic simulation of multibody systems based on a semi-recursive penalty formulation

Alfonso Callejo<sup>a,\*</sup>, Sri Hari Krishna Narayanan<sup>b</sup>, Javier García de Jalón<sup>a</sup>, Boyana Norris<sup>b</sup>

<sup>a</sup>*Instituto Universitario de Investigación del Automóvil, Universidad Politécnica de Madrid, Madrid, Spain*

<sup>b</sup>*Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois, USA*

---

## Abstract

Within the multibody systems literature, few attempts have been made to use automatic differentiation for solving forward multibody dynamics and evaluating its computational efficiency. The most relevant implementations are found in the sensitivity analysis field, but they rarely address automatic differentiation issues in depth. This paper presents a thorough analysis of automatic differentiation tools in the time integration of multibody systems. To that end, a penalty formulation is implemented. First, open-chain generalized positions and velocities are computed recursively, while using Cartesian coordinates to define local geometry. Second, the equations of motion are implicitly integrated by using the trapezoidal rule and a Newton-Raphson iteration. Third, velocity and acceleration projections are carried out to enforce kinematic constraints. For the computation of Newton-Raphson's tangent matrix, instead of using numerical or analytical differentiation, automatic differentiation is implemented here. Specifically, the source-to-source transformation tool ADIC2 and the operator overloading tool ADOL-C are employed, in both dense and sparse modes. The theoretical approach is backed by the numerical analysis of a 1-DOF spatial four-bar mechanism, three different configurations of a 15-DOF multiple four-bar linkage, and a 16-DOF coach maneuver. Numerical and automatic differentiation are compared in terms of their computational efficiency and accuracy. Overall, we provide a global perspective of the efficiency of automatic differentiation in the field of multibody systems.

*Keywords:* Multibody dynamics, Semi-recursive penalty formulation, Automatic differentiation, Operator overloading, Source-to-source transformation, ADOL-C, ADIC2

---

## 1. Introduction

Multibody systems (MBS) are mechanical systems made up of rigid or flexible bodies interconnected by perfect or imperfect kinematic joints and subject to various forces. These systems are present in numerous areas of industry, including mechanisms, robots, vehicles, machinery, wind turbines, and aerospace engineering. After more than 35 years of simulation of multibody systems, the development of efficient multibody methods is still challenging. The kinematic constraints between the rigid bodies and the presence of closed loops in the mechanisms often make the integration of the differential-algebraic equations (DAEs) tricky, unstable, or slow. Yet, in some applications such as driving simulators, hardware-in-the-loop applications, on-board controllers, and optimization algorithms, the computation of multibody system dynamics in real-time is crucial. In order to improve the efficiency of multibody dynamics software, several strategies can be adopted, some of which are efficient formulations, efficient implementations, and parallel implementations. The first two are addressed here.

Among the great variety of contemporary MBS formulations [1], penalty schemes have proven to be a robust and efficient

approach for solving forward MBS dynamics using dependent coordinates [2, 3]. Basically, they avoid the direct enforcement of kinematic constraints by introducing penalty terms proportional to the nonfulfillment of constraints. When combined with implicit integrators and projections, they allow for long integration time-steps while keeping the simulation stable. One of the most interesting approaches in this direction was presented in [4] and is followed here in the preliminary stages. Natural (or fully Cartesian) coordinates<sup>1</sup> are used to define local geometry and constraint equations. This approach simplifies the modeling stage. Positions and velocities are then computed recursively, making the most of the system topology.

For the time integration of the equations of motion, the trapezoidal rule with velocity and acceleration projections is used. This scheme requires the solution of a nonlinear system of equations, which is generally solved with a Newton-Raphson algorithm. To that end, the Jacobian matrix of the open-chain forces with respect to the relative positions and velocities has to be computed. Because this step takes most of the computation time, it is worth exploring efficient and accurate ways of differentiating computer functions, while preserving the scalability of the implementation.

There are several ways of computing the derivative of a mathematical function with respect to its independent variables. For

---

\*Corresponding author. Phone: (+34)913365335

Email address: a.callejo@upm.es (Alfonso Callejo)

URL: <http://mat21.etsii.upm.es/mbs/mbs3d/> (Alfonso Callejo)

<sup>1</sup>Cartesian components of points and unit vectors [2].

example, one may apply differential calculus by hand and code the differentiated functions; this is usually called *analytical differentiation*. A similar but more automated technique is *symbolic differentiation*, which is based on symbolic mathematical programs that generate the derivative equations from the original function. The derivatives must be generated in the third-party software, exported, reimplemented, and compiled each time a change is introduced in the equation; and only purely analytic equations can be differentiated. This technique thus has considerable drawbacks from the scalability point of view.

Another way of computing derivatives is through the use of *numerical differentiation* (ND) techniques such as finite-differences. Let  $f$  be a scalar function that depends on variable  $x$ . The derivative can be numerically approximated as:

$$f'(x_0) \equiv \frac{df}{dx}(x_0) \approx \frac{f(x_0 + h) - f(x_0 - h)}{2h}, \quad (1)$$

which corresponds to a centered-difference formula. More accurate formulae can be obtained by evaluating the function in different points. The advantage of these methods is that they only require the original function. However, Eq. (1) demands a very small value of the perturbation  $h$ . When  $h$  is very small, two similar numbers are being subtracted in the numerator, and, because of the limited computer precision, the derivative is less accurate than the original function. These numerical errors are unavoidable. Moreover, in the case of vector functions, the computational cost increases quickly as the problem size grows.

*Automatic or algorithmic differentiation* (AD) allows differentiating a computer function (implemented in Fortran, C, C++, MATLAB, etc.) and automatically computing both first-order derivatives (e.g. gradients and Jacobian matrices) and higher-order derivatives (e.g. Hessian matrices). The development time is shorter than using analytical differentiation techniques, and AD generates machine-precision derivatives. In past investigations with the formulation presented here, the operator overloading tool ADOL-C [5] was used successfully [6]. However, a single AD tool was not enough for assessing the computational efficiency, since different types of AD tools working on different AD modes might have very different performances. Also, only academic examples were considered.

Few works in the MBS community have thoroughly addressed AD as a way of differentiating computer functions. In 1996, Bischof [7] used the source transformation tools ADIC and ADIFOR on a Fortran code to compute vehicle sensitivities, but general performance conclusions were not given. Three years later, Eberhard and Bischof [8] focused on the time integration of sensitivities using ADIFOR on a 5-DOF robot and concluded that AD was less efficient but simpler to implement than analytical derivatives. Later, in [9], Dürbaum, Klier and Hahn proved that the symbolic tool MACSYMA generated derivatives faster than did ADOL-C for two medium-size planar and spatial robots. In 2007, Ambrósio, Neto, and Leal [10] simulated a satellite antenna as a flexible multibody system and recommended AD over ND for accuracy reasons, even though with little implementation details. Recently, Hannemann et al. [11] applied the source transformation tool dcc and an operator overloading tool to dynamic models. In general, rough descrip-

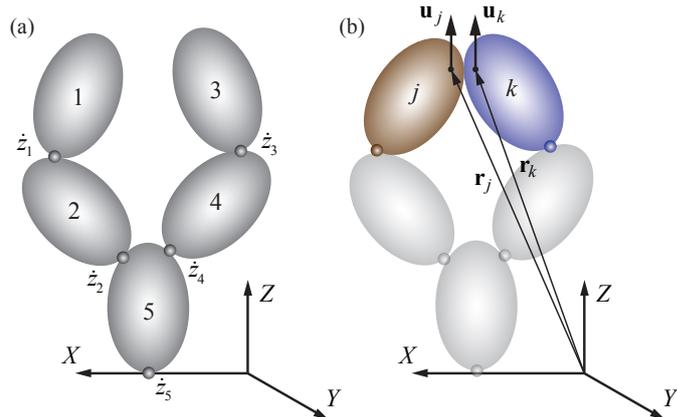


Figure 1: (a) Tree-configured MBS. (b) Closure-of-the-loop revolute joint.

tions of AD tools and their implementation are provided; the results are not compared with other AD tools; and academic rather than industrial numerical examples are considered. To the best of the authors' knowledge, the benefits of exploiting Jacobian sparsity in MBS formulations by using AD has not been shown before. In this work, both the source-to-source transformation tool ADIC2 [12] and the operator-overloading tool ADOL-C [5] are used on three numerical examples, namely a 1-DOF spatial four-bar mechanism, a 15-DOF multiple four-bar linkage and a 16-DOF coach model. These examples are used as medium to large benchmarks of ND and AD tools, with special focus on computational efficiency and exploiting sparse Jacobians.

## 2. Multibody formulation

In this section, a general-purpose MBS formulation is presented [4]. The formulation is explained in four steps: (1) the open-chain recursive differential equations are proposed; (2) the loops are closed by introducing position penalty terms; (3) the trapezoidal rule of integration is introduced; and (4) velocity and acceleration projections are carried out.

### 2.1. Open-chain equations

In order to apply recursion techniques, the system is considered as a tree-configured multibody system (see Fig. 1(a)). In the case of closed-chain systems, certain joints and rods<sup>2</sup> are temporarily removed and enforced later through constraint equations. Cartesian coordinates are used to define the velocity and acceleration of bodies:

$$\mathbf{Z}_i \equiv \begin{Bmatrix} \dot{\mathbf{s}}_i \\ \boldsymbol{\omega}_i \end{Bmatrix} \quad (2)$$

$$\dot{\mathbf{Z}}_i \equiv \begin{Bmatrix} \ddot{\mathbf{s}}_i \\ \dot{\boldsymbol{\omega}}_i \end{Bmatrix}, \quad (3)$$

<sup>2</sup>Slender bodies with two spherical joints and a negligible moment of inertia around the direction of the axis.

where  $\dot{\mathbf{s}}_i$  and  $\ddot{\mathbf{s}}_i$  are, respectively, the velocity and acceleration of the point attached to body  $i$  that instantaneously coincides with the origin of the inertial reference frame. In this way, all bodies share the same reference point, which brings interesting advantages [13]. The recursive expression of the Cartesian velocities and accelerations of body  $i$  in terms of those of body  $i - 1$  is

$$\mathbf{Z}_i = \mathbf{Z}_{i-1} + \mathbf{b}_i \dot{z}_i \quad (4)$$

$$\dot{\mathbf{Z}}_i = \dot{\mathbf{Z}}_{i-1} + \mathbf{b}_i \ddot{z}_i + \mathbf{d}_i. \quad (5)$$

Note the lack of transformation matrices in the previous equations. Scalar  $z_i$  is the relative coordinate of joint  $i$ . Vector  $\mathbf{b}_i$  represents the velocity of the point of body  $i$  that coincides with the origin of the global reference frame when  $\dot{z}_i = 1$  and  $\dot{z}_j = 0$ ,  $j \neq i$ ; and vector  $\mathbf{d}_i$  is the increase in acceleration from  $i - 1$  to  $i$  when  $\ddot{z}_i = 0$ . Both  $\mathbf{b}_i$  and  $\mathbf{d}_i$  depend on the type of joint between  $i$  and  $i - 1$ . Here, only revolute and prismatic joints (and combinations thereof) are considered. Vectors  $\mathbf{Z}^T \equiv \{\mathbf{Z}_1^T, \mathbf{Z}_2^T, \dots, \mathbf{Z}_n^T\}$  and  $\dot{\mathbf{Z}}^T \equiv \{\dot{\mathbf{Z}}_1^T, \dot{\mathbf{Z}}_2^T, \dots, \dot{\mathbf{Z}}_n^T\}$  group the system velocities and accelerations,  $n$  being the number of moving bodies. The virtual power of the inertia and external forces of the open-chain system can be expressed as

$$\sum_{i=1}^n \mathbf{Z}_i^{*T} (\overline{\mathbf{M}}_i \dot{\mathbf{Z}}_i - \overline{\mathbf{Q}}_i) = 0 \quad (6)$$

$$\overline{\mathbf{M}} \equiv \text{diag}(\overline{\mathbf{M}}_1, \overline{\mathbf{M}}_2, \dots, \overline{\mathbf{M}}_n) \quad (7)$$

$$\overline{\mathbf{Q}} \equiv \{\overline{\mathbf{Q}}_1^T, \overline{\mathbf{Q}}_2^T, \dots, \overline{\mathbf{Q}}_n^T\}^T \quad (8)$$

$$\overline{\mathbf{M}}_i = \begin{bmatrix} m_i \mathbf{I}_3 & -m_i \tilde{\mathbf{g}}_i \\ m_i \tilde{\mathbf{g}}_i & \mathbf{J}_i - m_i \tilde{\mathbf{g}}_i \tilde{\mathbf{g}}_i \end{bmatrix} \quad (9)$$

$$\overline{\mathbf{Q}}_i = \begin{Bmatrix} \mathbf{F}_i - m_i \tilde{\omega}_i \tilde{\omega}_i \mathbf{g}_i \\ \tilde{\mathbf{g}}_i \mathbf{F}_i - \tilde{\omega}_i \mathbf{J}_i \omega_i - m_i \tilde{\mathbf{g}}_i \tilde{\omega}_i \tilde{\omega}_i \mathbf{g}_i \end{Bmatrix}, \quad (10)$$

where  $\overline{\mathbf{M}}_i \in \mathbb{R}^{6 \times 6}$  and  $\overline{\mathbf{Q}}_i \in \mathbb{R}^{6 \times 1}$  are, respectively, the inertia matrix and the vector of external and velocity-dependent inertia forces acting on body  $i$ ;  $m_i$  is the mass;  $\mathbf{J}_i \in \mathbb{R}^{3 \times 3}$  is the inertia tensor;  $\mathbf{g}_i$  is the position of the center of gravity (COG);  $\omega_i$  is the angular velocity vector; and  $\mathbf{F}_i$  is the applied force vector. The upper bar indicates reference to the origin of the global reference frame. The upper tilde transforms the vector into the associated skew-symmetric matrix, such that, for generic  $3 \times 1$  vectors  $\alpha$  and  $\beta$ ,  $\alpha \times \beta = \tilde{\alpha} \beta$ .

A velocity transformation  $\mathbf{R} \in \mathbb{R}^{6n \times n}$  between Cartesian ( $\mathbf{Z}$ ) and relative ( $\dot{\mathbf{z}}$ ) velocities is now introduced. Bodies (and their corresponding input joints) are numbered from the leaves to the root of the spanning tree. The  $j^{\text{th}}$  column of matrix  $\mathbf{R}$  is the Cartesian velocities of all bodies that are upwards of joint  $j$  when a unit relative velocity is introduced in  $j$ , keeping the others null; because the origin of the global reference frame is the reference point for all bodies, these Cartesian velocities happen to be  $\mathbf{b}_i$  for all bodies, according to Eq. (4).

$$\mathbf{Z} = \mathbf{R} \dot{\mathbf{z}} = \mathbf{T} \text{diag}(\mathbf{b}_1, \mathbf{b}_2, \dots, \mathbf{b}_n) \dot{\mathbf{z}} \equiv \mathbf{TR}_d \dot{\mathbf{z}} \quad (11)$$

$$\dot{\mathbf{Z}} = \mathbf{TR}_d \ddot{\mathbf{z}} + \dot{\mathbf{TR}}_d \dot{\mathbf{z}}. \quad (12)$$

The connectivity of the mechanism has been defined through an upper triangular path matrix  $\mathbf{T} \in \mathbb{Z}^{6n \times 6n}$ . Submatrix  $T_{ij}$  is  $\mathbf{I}_6$

if body  $i$  is upwards of joint  $j$ , and  $\mathbf{0}_6$  otherwise. Introducing  $\mathbf{Z}^{*T} = \mathbf{R}_d^T \mathbf{T}^T \dot{\mathbf{z}}^*$  (from Eq. (11)) and Eq. (12) into Eq. (6), and knowing that the relative open-chain virtual velocities are independent, one can eliminate virtual velocities  $\dot{\mathbf{z}}^*$  and obtain a new set of differential equations

$$\underbrace{\mathbf{R}_d^T \mathbf{T}^T \overline{\mathbf{M}} \mathbf{T} \mathbf{R}_d}_{\mathbf{M}_d^\Sigma} \ddot{\mathbf{z}} = \underbrace{\mathbf{R}_d^T (\mathbf{T}^T \overline{\mathbf{Q}} - \mathbf{M}_d^\Sigma \dot{\mathbf{R}}_d \dot{\mathbf{z}})}_{\mathbf{Q}_d^\Sigma}, \quad (13)$$

where some terms have been grouped for the sake of clarity. These recursive equations constitute a set of  $n$  ODEs describing the motion of the open-chain system. In closed-loop systems, the constraint equations coming from the closure of the loops still need to be enforced.

## 2.2. Constraint enforcement and implicit integration

Closed-loop dynamic equations can be formulated by adding the constraint equations to the open-chain dynamic equations, which have just been obtained. The fulfillment of the position constraint equations is enforced by introducing a penalty term into Eq. (13). Then, velocity and acceleration constraints are imposed by carrying out velocity and acceleration projections.

First, let us add a penalty term to Eq. (13):

$$\mathbf{M}_d^\Sigma \ddot{\mathbf{z}} + \Phi_z^T \alpha \Phi = \mathbf{Q}_d^\Sigma, \quad (14)$$

where  $\alpha$  is the penalization coefficient,  $\Phi \in \mathbb{R}^{m \times 1}$  is the vector of  $m$  constraint equations, and  $\Phi_z \in \mathbb{R}^{m \times n}$  is the Jacobian matrix of the constraint equations with respect to relative positions. The penalty term has a physical meaning:  $\alpha \Phi$  is the value of the penalty forces ( $\omega$  for each constraint equation that is violated) and the columns of  $\Phi_z^T$  are the directions of the constraint forces in which penalties are applied. Figure 1(b) shows the way a closure-of-the-loop revolute joint can be formulated in terms of natural coordinates (for more details see [13]).

For the integration of Eq. (14), the implicit single-step trapezoidal rule with time-step  $h$  is used. Relative velocities and accelerations in time-step  $j + 1$  are written as follows.

$$\dot{\mathbf{z}}_{j+1} = \frac{2}{h} \mathbf{z}_{j+1} - \left( \frac{2}{h} \mathbf{z}_j + \dot{\mathbf{z}}_j \right) \quad (15)$$

$$\ddot{\mathbf{z}}_{j+1} = \frac{4}{h^2} \mathbf{z}_{j+1} - \underbrace{\left( \frac{4}{h^2} \mathbf{z}_j + \frac{4}{h} \dot{\mathbf{z}}_j + \ddot{\mathbf{z}}_j \right)}_{\hat{\mathbf{z}}_j}. \quad (16)$$

By introducing Eqs. (15) and (16) in Eq. (14), a nonlinear equation  $\mathbf{f}(\mathbf{z}_{j+1}) = \mathbf{0}$  is obtained:

$$\mathbf{M}_{d,j+1}^\Sigma \mathbf{z}_{j+1} + \frac{h^2}{4} \Phi_{z,j+1}^T \alpha \Phi_{j+1} - \frac{h^2}{4} \mathbf{Q}_{d,j+1}^\Sigma + \frac{h^2}{4} \mathbf{M}_{d,j+1}^\Sigma \hat{\mathbf{z}}_j = \mathbf{0}, \quad (17)$$

where  $\mathbf{M}_d^\Sigma = \mathbf{M}_d^\Sigma(\mathbf{z})$ ,  $\mathbf{Q}_d^\Sigma = \mathbf{Q}_d^\Sigma(\mathbf{z}, \dot{\mathbf{z}})$ ,  $\Phi = \Phi(\mathbf{z})$  and  $\Phi_z = \Phi_z(\mathbf{z})$ .

Equation (17) is a nonlinear system of equations, that has to be solved for unknown vector  $\mathbf{z}_{j+1}$ . To that end, it is customary to use the Newton-Raphson method, which has a quadratic convergence in the neighborhood of the solution. The use of this

iterative method implies the evaluation of a tangent matrix and a remainder, as indicated next. Let  $k + 1$  be the iteration.

$$\mathbf{z}_{j+1}^{k+1} = \mathbf{z}_{j+1}^k + \Delta \mathbf{z}_{j+1}^k \quad (18)$$

$$\left[ \frac{\partial \mathbf{f}(\mathbf{z})}{\partial \mathbf{z}} \right]_{j+1}^k \Delta \mathbf{z}_{j+1}^k = -[\mathbf{f}(\mathbf{z})]_{j+1}^k. \quad (19)$$

The solution of Eq. (18) implies the evaluation of the tangent matrix in Eq. (19). This tangent matrix can be approximated [4] with the following expression:

$$\left[ \frac{\partial \mathbf{f}(\mathbf{z})}{\partial \mathbf{z}} \right]_{j+1}^k \approx \left[ \mathbf{M}_d^\Sigma + \frac{h}{2} \mathbf{C} + \frac{h^2}{4} (\Phi_z^T \alpha \Phi_z + \mathbf{K}) \right]_{j+1}^k \quad (20)$$

$$\mathbf{K} \equiv -\frac{\partial \mathbf{Q}_d^\Sigma}{\partial \mathbf{z}} \quad (21)$$

$$\mathbf{C} \equiv -\frac{\partial \mathbf{Q}_d^\Sigma}{\partial \dot{\mathbf{z}}}, \quad (22)$$

where  $\mathbf{K} \in \mathbb{R}^{n \times n}$  and  $\mathbf{C} \in \mathbb{R}^{n \times n}$  have been introduced. If the state vector  $\mathbf{y} \equiv \{\mathbf{z}^T, \dot{\mathbf{z}}^T\}^T \in \mathbb{R}^{2n \times 1}$  is defined, both matrices can be grouped as:

$$\mathbf{J} \equiv -\frac{\partial \mathbf{Q}_d^\Sigma}{\partial \mathbf{y}}. \quad (23)$$

The Jacobian matrix  $\mathbf{J} \in \mathbb{R}^{n \times 2n}$  is the key computation of this algorithm. Some authors [4] formulate this matrix analytically, meaning that the derivatives have to be computed by hand for the most typical force types (springs, dampers, etc.). This is obviously not the most general-purpose approach and often leads to error-prone expressions. Numerical differentiation might be inefficient, and its error is difficult to control. On the other hand, AD in its various forms can be used as well to calculate these derivatives with minimal effort from the user and reasonable efficiency. The following sections investigate the different ways of computing this Jacobian matrix.

The previous equations impose the dynamics and the fulfillment of the position constraint equations, but the velocity and acceleration constraints have not been enforced yet. During the time integration process, Eqs. (15) and (16) yield a set of velocities  $\dot{\mathbf{z}}^*$  and accelerations  $\ddot{\mathbf{z}}^*$  that do not satisfy velocity and acceleration constraints. The reason is that both vectors have been obtained numerically from the integrator and not by differentiating the positions. This problem can be solved through velocity and acceleration projections [4]. Applying a projection method with penalty terms, one can obtain a set of velocities  $\dot{\mathbf{z}}$  that satisfy the constraints. Introducing a weight matrix  $\mathbf{P}$ , one can compute the projected velocities as:

$$\left( \mathbf{P} + \frac{h^2}{4} \Phi_z^T \alpha \Phi_z \right) \dot{\mathbf{z}} = \mathbf{P} \dot{\mathbf{z}}^* - \frac{h^2}{4} \Phi_z^T \alpha \Phi_t, \quad (24)$$

$$\mathbf{P} \equiv \mathbf{M}_d^\Sigma + \frac{h}{2} \mathbf{C} + \frac{h^2}{4} \mathbf{K}, \quad (25)$$

where the system matrix in the l.h.s. of Eq. (24) is the tangent matrix (20). In this way, the matrix factorization can be reused, and the projection is performed with a low computational cost.

Similarly, the expression of the projected accelerations is

$$\left( \mathbf{P} + \frac{h^2}{4} \Phi_z^T \alpha \Phi_z \right) \ddot{\mathbf{z}} = \mathbf{P} \ddot{\mathbf{z}}^* - \frac{h^2}{4} \Phi_z^T \alpha \dot{\Phi}_z \dot{\mathbf{z}} - \frac{h^2}{4} \Phi_z^T \alpha \Phi_t. \quad (26)$$

After solving the velocity and acceleration projections, all constraints (in position, velocity and acceleration) are fulfilled.

In standard Newton-Raphson problems, both the value and the factorization of the tangent matrix (20) can be reused over a number of iterations so that only a back substitution is needed to find  $\Delta \mathbf{z}_{j+1}^k$  in Eq. (19). In this case, however, the tangent matrix factorization is employed later to solve velocity (24) and acceleration (26) projections. These projections need an updated version of the tangent matrix (and its factorization) in order to achieve an accurate enforcement of constraints; hence, reusing the tangent matrix in the Newton-Raphson iteration is not compatible with velocity and acceleration projections. Since the computational burden of projections is greater, we chose to always refactorize Newton-Raphson's tangent matrix and use the last factorization for projections. In turn, we did reuse Jacobian matrices (21) and (22) for three Newton-Raphson iterations, as they are the heaviest tangent matrix components. This approach has proved to be an effective cost-accuracy tradeoff in real-life mechanical systems.

### 3. Automatic differentiation tools

Among the steps of the presented formulation, the computation of the Jacobian matrix in Eq. (23) is critical for performance. In this section, we introduce AD methodology and discuss how AD is used to produce the derivatives algorithmically.

#### 3.1. Algorithmic differentiation

AD [14, 15] is an approach to obtaining derivative computations based on source-code implementations of mathematical functions. AD combines rule-based differentiation of elementary operators (e.g. addition, subtraction) with derivative accumulation according to the chain rule of differential calculus. The derivatives produced by using AD are accurate to machine precision with respect to the original computation (but not necessarily the original mathematical function; and in the case of iterative algorithms, convergence rates may differ [16]) and can be used in many contexts, including numerical optimization, nonlinear partial differential equation solvers, or the solution of inverse problems using least squares. Many tools provide AD for different languages, including Fortran, MATLAB, C, and C++ (e.g. [14, 17, 18, 15]).

AD tools typically adopt one of two implementation approaches: operator overloading (in languages that support it) or source transformation. Operator overloading-based tools are easier to implement; but because they rely on runtime evaluation of partial derivatives, the ways in which the chain rule associativity can be exploited to attain better performing derivative code are limited. On the other hand, source transformation approaches enable static analysis of program source code, presenting opportunities for optimization over much larger scopes than a single statement, often resulting in significantly better

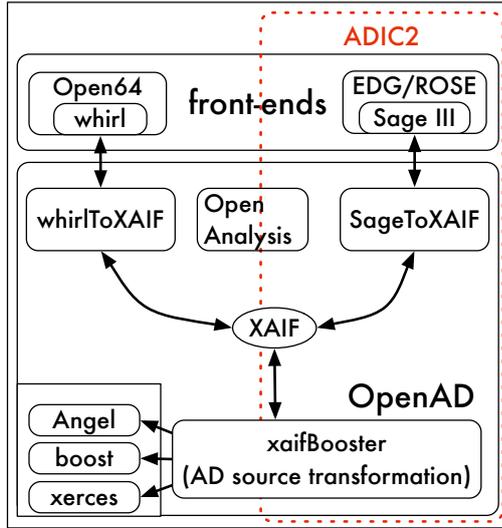


Figure 2: OpenAD component structure and source transformation workflow.

performance of the AD computation. Moreover, the resulting code can be tweaked and improved manually if necessary. However, source transformation-based AD has the same limitations as traditional compilers, which includes complexity of implementing parsing and analysis of general-purpose languages such as C++, as well as reliance on necessarily conservative static analysis (e.g., alias analysis), which may lead to the generation of suboptimal derivative code.

### 3.2. ADIC2

ADIC2 is a source-to-source transformation AD tool for C and C++ with support for the forward and reverse modes [12]. It is part of the OpenAD framework [19], illustrated in Fig. 2. The input code is input to the ROSE compiler framework [20, 21] which is parsed by EDG C/C++ parsers. Once converted into a ROSE abstract syntax tree (AST), the following processes occur to generate output derivative code:

1. Canonicalization: Several code constructs are simplified in order to make the later transformations feasible. For example, all function calls determined to affect the output are converted into subroutine calls.
2. Program analysis: The OpenAnalysis framework [22] is used to analyze the canonicalized code. It generates a call graph, a control flow graph, define-use and use-define chains, a scope hierarchy, and alias analysis results.
3. XAIF generation: The results generated by OpenAnalysis and any code statements that affect the output are converted into the XML Abstract Interface Form (XAIF), a language-independent to represent code.
4. Derivative propagation: xaifBooster [19] uses transformation algorithms to convert the input XAIF into derivative XAIF (AD-XAIF).
5. Conversion of AD-XAIF: The AD-XAIF is parsed and is converted into ROSE AST nodes.
6. Generation of derivative code: The Rose AST is converted into C/C++ using Rose's codegen facility. The output

```
void speelpenning(double *y,double *x,int n){
  int i;
  *y = 1.0;
  for(i=0; i<n; i++) {
    *y = (*y) * x[i];
  }
}
```

(a)

```
#include "ad_types.h"
#include "ad_grad_saxpy-n_dense.h"
void ad_speelpenning(DERIV_TYPE *y,DERIV_TYPE *x,int n){
  int ad_i;
  DERIV_val(*y) = 1.00000;
  ADIC_ZeroDeriv(*y);
  for (ad_i = 0, ad_i = 0; ad_i < n; ad_i = ad_i + 1) {
    DERIV_TYPE ad_prp_1;
    ADIC_Initialize(&ad_prp_1);
    DERIV_TYPE ad_prp_0;
    ADIC_Initialize(&ad_prp_0);
    double ad_lin_1;
    double ad_lin_0;
    ad_lin_0 = DERIV_val(x[ad_i]);
    ad_lin_1 = DERIV_val(*y);
    DERIV_val(*y) = DERIV_val(*y) * DERIV_val(x[ad_i]);
    ADIC_SetDeriv(*y,ad_prp_0);
    ADIC_SetDeriv(x[ad_i],ad_prp_1);
    ADIC_Sax_2(ad_lin_0,ad_prp_0,ad_lin_1,ad_prp_1,*y);
  }
}
```

(b)

```
typedef struct {
  double val;
  double grad[ADIC_GRADVEC_LENGTH];
} DERIV_TYPE;
```

(c)

Figure 3: Example of the use of ADIC2.

```
#include "adolc/adolc.h"
void speelpenning(double *yp,double *xp,int n){
  adouble *x = new adouble[n];
  adouble y = 1;
  trace_on(1);
  for(i=0; i<n; i++) {
    x[i] <=& xp[i];
    y *= x[i];
  }
  y >>= *yp;
  delete[] x;
  trace_off(1);
}
```

(a)

```
#include "adolc/adolc.h"
void speelpenning_driver(double* g,double *xp,int n){
  double* g = new double[n];
  gradient(1,n,xp,g);
}
```

(b)

Figure 4: Example of the use of ADOL-C.

code can be compiled with a runtime library provided by ADIC2 and executed to generate derivatives.

Figure 3 is an example of the use of ADIC2. Figure 3(a) is the classic example attributed to Speelpenning, and Fig. 3(b) is the corresponding output generated by ADIC2. The ADIC2 runtime library defines a structure called DERIV\_TYPE (shown in Fig. 3(c)) that contains a value field and an array field that holds derivative values. The size of the array field is the number of independent variables. Operations to manipulate the array

fields are defined within the library as well. At the end of the computation, the array fields of the dependent variable form the Jacobian matrix.

Because Jacobians can be sparse, using an array size that effectively computes a full Jacobian can be inefficient. Furthermore, for large Jacobians, not enough memory may be available to allocate an array for each `DERIV_TYPE` variable. Therefore, ADIC2 implements a framework to exploit Jacobian sparsity [23]. Specifically, given a function  $\mathbf{Q}_d^z : \mathbb{R}^{2n \times 1} \rightarrow \mathbb{R}^{n \times 1}$  whose Jacobian matrix  $\mathbf{J} \in \mathbb{R}^{n \times 2n}$  (see Eq. (23)) is sparse, ADIC2 employs the following framework to efficiently compute matrix  $\mathbf{J}$  using the following four steps:

1. Determine the *sparsity pattern* of matrix  $\mathbf{J}$ .
2. Using a *coloring* on an appropriate graph of  $\mathbf{J}$ , obtain an  $n \times p$  *seed matrix*  $\mathbf{S}$  with the smallest  $p$  that defines a partitioning of the columns of  $\mathbf{J}$  into  $p$  groups.
3. Compute the numerical values in the *compressed matrix*  $\mathbf{B} \equiv \mathbf{J}\mathbf{S}$ .
4. Recover the numerical values of the entries of  $\mathbf{J}$  from  $\mathbf{B}$ .

In step 1, the output derivative code is compiled with a runtime library called SparsLinC, which is used to detect the structure of the Jacobian. In step 2, the coloring package ColPack [24] is used. The number of colors  $p$  used to partition the Jacobian dictates the number of columns in the compressed matrix and consequently the new size of the array in the `DERIV_TYPE` structure. When computing the compressed matrix, having a smaller array can result in a performance improvement, provided that the overheads of the steps 1, 2, and 4 can be offset.

### 3.3. ADOL-C

ADOL-C is an operator overloading AD tool for C and C++ with support for the forward and reverse modes [25]. It generates gradients, Jacobians, Hessians, Jacobian  $\times$  vector products, Hessian  $\times$  vector products, and the like. Figure 4 is an example of the use of ADOL-C. Figure 4(a) is the Speelpenning example coded to obtain derivatives using ADOL-C, and Fig. 4(b) is the driver used to run the derivatives. ADOL-C defines a type called `adouble` to be used for *active variables* in the computation. Derivative calculation is based on a function representation created during the 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`.

Jacobians can be computed by ADOL-C in three different modes: forward, reverse and sparse. According to ADOL-C guidelines, the reverse mode should be used when the number of independent coordinates is twice the number of dependent coordinates or larger; otherwise the forward mode should be employed. In the present formulation, the size of the Jacobian lies in between. To compute a sparse Jacobian efficiently, ADOL-C follows a technique similar to that of ADIC2.

## 4. Results

Five different mechanical systems are simulated in order to assess the accuracy and efficiency of AD tools in the presented

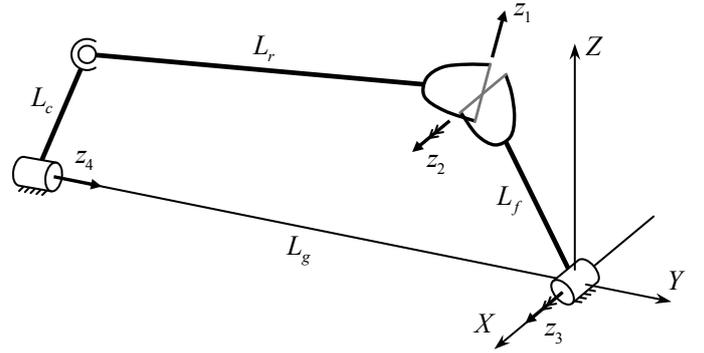


Figure 5: Schematic view of the spatial four-bar mechanism.

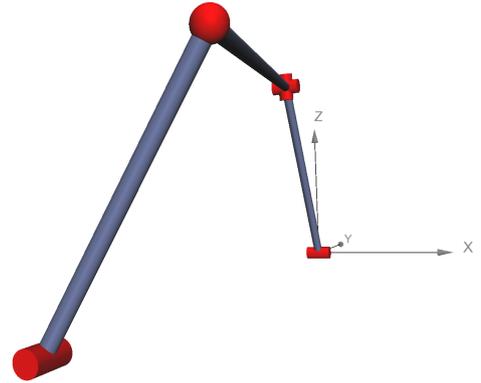


Figure 6: 3D view of the spatial four-bar mechanism.

Table 1: Jacobian matrices for the different models.

System	Indeps.	Deps.	Nonzeroes	Colors
Spatial four-bar	8	4	20	6
1 $\times$ 15 four-bar	90	45	1410	60
4 $\times$ 15 four-bar	270	135	4290	60
7 $\times$ 15 four-bar	900	450	7170	60
Coach	66	33	1078	62

formulation. The first one is a 1-DOF spatial four-bar mechanism. The second, third, and fourth examples are three different configurations of a 15-DOF multiple-four bar linkage. The fifth is a 16-DOF coach performing a lane-change maneuver. In all cases, gravity acts in the  $-Z$ -direction. All simulations have been run on two different platforms. Platform gcc-1 is a dual Intel<sup>®</sup> Xeon<sup>™</sup> (8 processors at 2.66 GHz) with 32 GB RAM running Ubuntu 12.04. Platform gcc-2 is an Intel<sup>®</sup> Core<sup>™</sup> i7 machine at 2.93 GHz with 6 GB RAM running Ubuntu 12.04.

The simulation time is 5 s in all experiments except for the coach case, where 2 s are simulated. Three different time-steps (1, 10, and 20 ms) have been tested, in order to capture the effect of the time-step length on the simulation accuracy and efficiency. To monitor the physical accuracy of the formulation, an energy balance is carried out. Kinetic and potential energies and the work of nonconservative forces are computed over time and summed. The variation in the total energy of the system (or *numerical drift*) is provided for each of the integrator

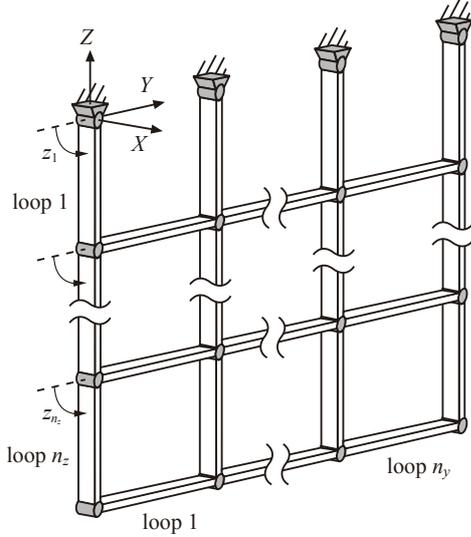


Figure 7: Schematic view of generic multiple four-bar linkage.

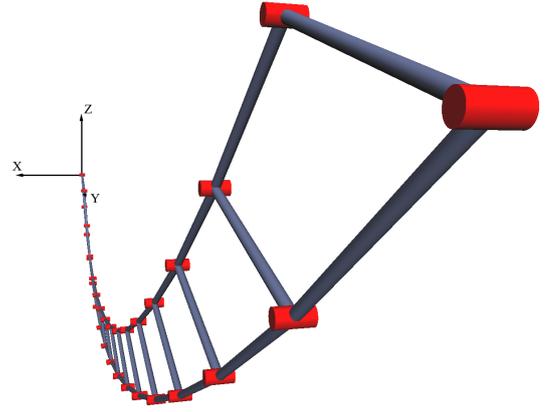


Figure 8: 3D view of the  $1 \times 15$  multiple four-bar linkage.

time-steps, showing how longer time-steps make the implicit integrator numerically competitive but physically less accurate.

For each system, derivatives are obtained first by employing centered-difference ND as in (1). Then, ADOL-C is run in the forward, reverse, and sparse modes, and ADIC2 is run in the forward and sparse modes. For each case, the time to complete the simulation, the numerical drift, and the number of Jacobians computed are noted.

#### 4.1. Spatial four-bar mechanism

The first model under study is the 1-DOF spatial four-bar mechanism shown in Figs. 5 and 6. The lengths of the crank, connecting rod, follower, and ground link are, respectively,  $L_c = 1.5$  m,  $L_r = 4$  m,  $L_f = 2$  m, and  $L_g = 5$  m. Each bar  $k$  has mass  $m_k = L_k$  and negligible inertia around the direction of the axis. The only initial condition is  $\dot{z}_4(t=0) = -0.1$  rad/s. Row 1 of Table 1 lists the number of independents, dependents, nonzeros, and colors used to partition the Jacobian. The size of the Jacobian in this model is quite small and is not very sparse, as can be seen in Fig. 10(a).

Computation times are shown in Table 2. Columns contain the elapsed times of the different differentiation methods (ND, AD with ADOL-C, and AD with ADIC2), and rows contain the different platforms used in the simulations. In this example, elapsed times are at the limit of what can be measured with standard timing functions. Nevertheless, a trend can already be observed: AD times are about the same as ND times, and sparse modes seem to be the fastest way of running AD. However, differences in AD modes and tools are not yet clearly quantifiable.

#### 4.2. Multiple four-bar linkage

The second sample model is a multiple four-bar linkage, made up of a series of concatenated four-bar mechanisms in the Y- and Z-directions. The number of quadrilaterals in both directions is denoted as  $n_y$  and  $n_z$ , respectively. Three different  $n_y \times n_z$  cases are considered:  $1 \times 15$ ,  $4 \times 15$ , and  $7 \times 15$ . See Fig.

7 for a generic case and Fig. 8 for the  $1 \times 15$  case. All joints in the system are parallel X-direction revolute joints, and all bodies are contained in the YZ-plane. Only the top joints are fixed; all bars are moving bodies. Bars have a uniformly distributed mass of 1 kg and a length of 1 m. The system is considered as a three-dimensional multibody system for the sake of generality. The only initial condition is  $\dot{z}_1(t=0) = \pi/3$  rad/s. Rows 2–4 of Table 1 list the number of independents, dependents, nonzeros, and colors used to partition the Jacobian for the  $1 \times 15$ ,  $4 \times 15$ , and  $7 \times 15$  cases, respectively. The Jacobians are sparse and the number of colors used to partition the Jacobian is considerably lower than the number of independent variables. This implies that for this model, ADIC2's sparse mode requires much less stack memory than ADIC2's dense mode.

The main objective of the multiple four-bar linkage experiments is to assess the effect of the problem size on the computational cost of Jacobian matrix (23). All three systems are 15-DOF systems, but they differ in the number of rigid bodies (45, 135, and 450, respectively). For a given number of DOFs, the higher the number of bodies (and thus joints), the higher the number of constraint equations.

Computation times are shown in Table 3. Several observations can be made for this models. First, both AD tools (ADIC2 and ADOL-C) have a very similar efficiency. Second, AD times are shorter than ND times in the  $4 \times 15$  and  $7 \times 15$  cases. Multiple four-bar systems with a high ratio of rigid bodies per DOF seem to favor AD performance. The reason is that the size of the compressed matrix is much smaller than the number of independent variables. This fact coincides with previous investigations using ADOL-C in similar contexts [6] and proves that ADIC2 follows the same trend. Third, because the grad field of DERIV\_TYPE is statically allocated, ADIC2-generated code exceeded the available stack space in the machine for the two infeasible cases. Fourth, ADOL-C's forward mode is faster than ADOL-C's reverse mode.

Table 2: Simulation results of the spatial four-bar mechanism.

$h$ (ms)	Platform	Elapsed Time (s)						Numerical Drift (%)	# Jacobians
		ND	ADOL-C			ADIC2			
			Forward	Reverse	Sparse	Forward	Sparse		
1	gcc-1	0.38	0.63	0.77	0.49	0.44	0.42	+0.413	5000
	gcc-2	0.36	0.62	0.91	0.54	0.32	0.30		
10	gcc-1	0.04	0.06	0.80	0.04	0.04	0.04	+0.345	500
	gcc-2	0.05	0.08	0.09	0.06	0.04	0.05		
20	gcc-1	0.02	0.30	0.30	0.04	0.02	0.02	-0.666	250
	gcc-2	0.03	0.05	0.05	0.03	0.03	0.03		

Table 3: Simulation results of the multiple four-bar linkages.

Case	$h$ (ms)	Platform	Elapsed Time (s)						Numerical Drift (%)	# Jacobians
			ND	ADOL-C			ADIC2			
				Forward	Reverse	Sparse	Forward	Sparse		
$1 \times 15$	1	gcc-1	44.80	82.13	49.51	51.17	69.03	51.85	-0.000	5000
		gcc-2	36.49	59.23	44.17	39.45	55.51	37.71		
	10	gcc-1	5.12	8.45	5.04	5.36	7.20	6.32	-0.015	500
		gcc-2	3.78	6.05	4.59	4.14	5.79	4.20		
	20	gcc-1	2.93	4.76	2.98	3.12	4.12	3.39	-0.057	287
		gcc-2	2.23	3.50	2.70	2.40	3.71	2.51		
$4 \times 15$	1	gcc-1	486.23	1,064.92	889.21	284.73	922.82	309.83	+0.000	5000
		gcc-2	453.45	673.61	538.85	205.57	665.96	214.49		
	10	gcc-1	58.16	106.19	92.77	34.27	97.43	39.09	-0.015	500
		gcc-2	50.38	72.86	58.19	25.26	71.64	28.27		
	20	gcc-1	34.58	60.78	53.97	20.33	55.65	23.82	-0.058	279
		gcc-2	29.85	41.33	33.28	15.13	41.16	17.34		
$7 \times 15$	1	gcc-1	1,870.84	3,688.65	3,041.90	1,147.13	3,468.08	1,103.67	+0.000	5000
		gcc-2	1,682.25	2,025.48	1,909.90	672.06	Infeasible	Infeasible		
	10	gcc-1	243.85	409.49	346.91	168.81	395.48	167.31	-0.015	500
		gcc-2	198.65	249.65	216.26	96.87	258.26	106.26		
	20	gcc-1	146.90	241.64	202.23	102.68	228.39	103.38	-0.058	275
		gcc-2	116.18	145.88	123.16	59.90	149.04	66.38		

Table 4: Simulation results of the coach dynamic maneuver.

$h$ (ms)	Platform	Elapsed Time (s)						Numerical Drift (%)	# Jacobians
		ND	ADOL-C			ADIC2			
			Forward	Reverse	Sparse	Forward	Sparse		
1	gcc-1	48.68	75.43	58.96	66.92	55.19	58.20	+3.742	5000
	gcc-2	36.01	57.93	55.14	50.46	54.44	52.88		
10	gcc-1	5.06	7.60	6.20	6.64	5.60	5.94	+3.731	506
	gcc-2	3.69	5.88	5.78	5.18	5.74	5.41		
20	gcc-1	2.70	4.04	3.19	3.51	3.01	3.26	+3.808	268
	gcc-2	2.03	3.13	2.99	2.74	2.98	2.91		

### 4.3. Coach dynamic maneuver

The coach under study is a Noge Touring 345 vehicle with frame from Mercedes-Benz. A general view of the coach model is shown in Fig. 9. A coordinate-measuring machine has been used on the unloaded real coach to obtain global dimensions and the position of key suspension points and joints. The coach has two axles: the front one has two wheels and the rear one four (assembled as two sets of dual wheels). The total mass of the loaded coach is 17,048 kg. The front suspension system is independent, while the rear one is a rigid axle. Suspension elasticity is provided through six air springs and two stabilizer bars, and damping through six regular dampers. The air spring force is considered linear w.r.t. the elongation of the spring, whereas the damping force is modeled as a piecewise bilinear function w.r.t. the relative velocity of the ends. On the other hand, stabilizer bars are modeled as an angular spring between both bar ends, neglecting bending stiffness. All vehicle parameters have been measured when possible, and estimated otherwise. The torsion stiffness of the bodywork, is considered by dividing the bodywork in two separate bodies linked by a revolute joint with a torsion spring acting along the X-direction.

The steering coordinate  $\delta$  corresponds to the rotation of the steering actuator, which acts on the steering rods through the steering mechanism. In the considered simulation, the coach performs a 2-second lane-change maneuver. To that end, coordinate  $\delta$  is kinematically guided by a predefined steering function  $\delta(t) = 0.013 \sin t$  [rad],  $t \in [0, 2]$ . For the tire forces, Pacejka's Magic Formula [26] is used to compute the contact point and the six contact forces (three linear forces and three torques). Also, a speed control is implemented so that the vehicle speed is always 50 km/h. The resulting model is a 16-DOF multibody system. Row 5 of Table 1 lists the number of independents, dependents, nonzeros and colors used to partition the Jacobian, which in this model is not sparse (see also Fig. 10(c)).

Computation times are shown in Table 4. In this case, AD times are slightly longer than ND times. Both for ADIC2 and ADOL-C, the benefits of the sparse mode are almost unnoticeable due to the dense nature of the Jacobian. Finally, ADOL-C's forward mode is faster than ADOL-C's reverse mode.

### 4.4. Discussion

When Jacobian matrices are considered naively to be dense, ADIC2 and ADOL-C codes are always slower than ND code. This result may be due to overheads in the implementation of ADIC2's and ADOL-C's runtime libraries. Furthermore, because of the nature of Newton's method, Jacobian accuracy is not essential for rapid numerical convergence. In fact, the inherent machine-precision accuracy of AD has almost no effect on the number of iterations required by Newton's algorithm to solve the nonlinear system of equations (17). Thus, in this formulation all methods converge in the same number of iterations.

The sparsity of the Jacobian has a vital effect on performance. Figure 10 shows the sparsity patterns of three models used here. The multiple four-bar linkage model is quite sparse and it has been exploited by both AD tools. While sparsity could certainly be exploited for the ND approach as well [27], the multibody



Figure 9: 3D view of the coach maneuver.

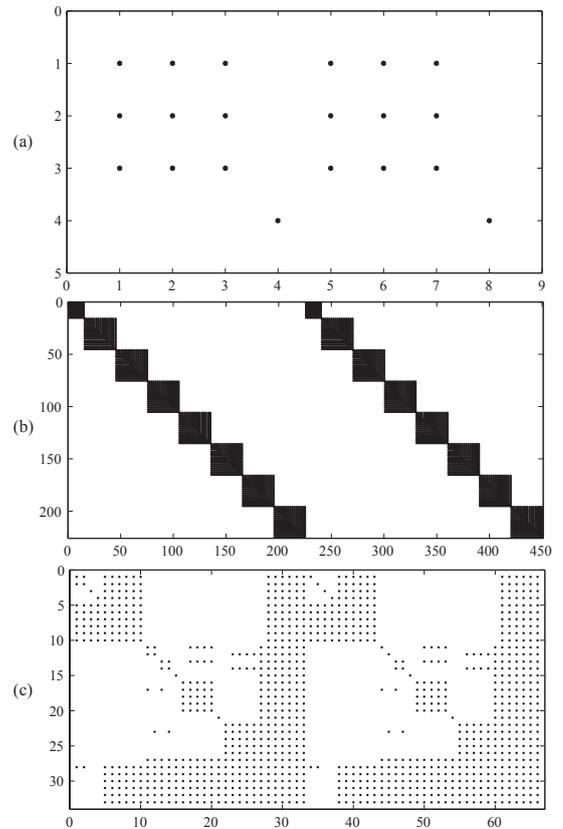


Figure 10: Sparsity pattern of (a) the spatial four-bar example, (b) the  $7 \times 15$  multiple four-bar linkage, and (c) the coach.

code used in this work is not easily amenable to such exploitation. The code was implemented without initially considering Jacobian sparsity, and modifying it requires nontrivial formulation changes. On the other hand, exploiting sparsity via ADIC2 and ADOL-C is as easy as changing drivers and using different runtime libraries during code compilation. When the Jacobian is sparse, ADIC2 and ADOL-C sparse Jacobians are faster than using the dense ND approach.

Considering the problem as a whole, the results suggest that AD cannot be accepted or rejected over ND by looking at just the computational efficiency of the code. Several aspects such as ease of implementation, development time, accuracy, Jacobian structure, and scalability should also be assessed.

## 5. Conclusions

A general-purpose multibody dynamics code, together with two academic examples and a realistic coach model, have been used to evaluate the performance of automatic differentiation tools in the context of multibody systems. Two different C/C++ tools for the automatic generation of derivatives have been used: the source transformation tool ADIC2 and the operator overloading tool ADOL-C. Both tools were relatively easy to use and provided accurate derivatives. The MBS simulation code is the most complex use case that ADIC2 has been applied to, and has required several enhancements to the tool.

When the Jacobians are considered to be dense, derivatives generated by AD are slower than using the finite-differences approach. However, support for sparse Jacobians can be provided by ADIC2 and ADOL-C in a direct way. Sparse AD derivatives are faster for certain systems than using the finite-differences approach without exploiting sparsity.

Whenever AD is faster than ND, probably no other differentiation method can generate more accurate and more efficient Jacobian matrices with such a short development time. Techniques like analytical (or manual) differentiation, which could provide machine-precision and efficient derivatives, would be nearly infeasible for complex formulations like the one considered here, and definitely not as scalable. Nevertheless, there is still room for the improvement of AD performance.

Overall, advantages and disadvantages of state-of-the-art tools for the automatic differentiation of C/C++ codes in the field of multibody dynamics have been presented rigorously.

## Acknowledgments

This work was supported by the U.S. Dept. of Energy Office of Science Applied Mathematics Program (DE-AC02-06CH11357), the Spanish Ministry of Science and Innovation (TRA2009-14513-C02-01) and the Government of Navarra.

## References

- [1] O. A. Bauchau, A. Laulusa, Review of contemporary approaches for constraint enforcement in multibody systems, *J. Comput. Nonlinear Dynam.* 3 (2007) 011005.
- [2] J. García de Jalón, E. Bayo, Kinematic and Dynamic Simulation of Multibody Systems. The Real-Time Challenge, Mechanical engineering series, Springer-Verlag, New York, 1994.
- [3] E. Bayo, R. Ledesma, Augmented lagrangian and mass-orthogonal projection methods for constrained multibody dynamics, *Nonlinear Dynamics* 9 (1996) 113–130.
- [4] J. Cuadrado, D. Dopico, M. Gonzalez, M. A. Naya, A combined penalty and recursive real-time formulation for multibody dynamics, *Journal of Mechanical Design* 126 (2004) 602–608.
- [5] A. Griewank, D. Juedes, J. Utke, ADOL-C, a package for the automatic differentiation of algorithms written in C/C++, *ACM Trans. Math. Software* 22 (1996) 131–167.
- [6] A. Callejo, J. García de Jalón, Automatic differentiation of forces in forward multibody dynamics, in: *ECCOMAS Multibody Dynamics 2011*, J.C. Samin, P. Fiset (eds.), Springer, 2011.
- [7] C. H. Bischof, On the automatic differentiation of computer programs and an application to multibody systems (1996) 41–48.
- [8] P. Eberhard, C. Bischof, Automatic differentiation of numerical integration algorithms, *Mathematics of Computation* 68 (1999) pp. 717–731.
- [9] A. Dürrbaum, W. Klier, H. Hahn, Comparison of automatic and symbolic differentiation in mathematical modeling and computer simulation of rigid-body systems, *Multibody System Dynamics* 7 (2002) 331–355.
- [10] J. A. C. Ambrósio, M. A. Neto, R. P. Leal, Optimization of a complex flexible multibody systems with composite materials, *Multibody System Dynamics* 18 (2007) 117–144.
- [11] R. Hannemann, W. Marquardt, U. Naumann, B. Gendler, Discrete first- and second-order adjoints and automatic differentiation for the sensitivity analysis of dynamic models, *Procedia Computer Science* 1 (2010) 297 – 305. ICCS 2010.
- [12] S. H. K. Narayanan, B. Norris, B. Winnicka, ADIC2: Development of a component source transformation system for differentiating C and C++, *Procedia Computer Science* 1 (2010) 1845–1853. ICCS 2010.
- [13] J. García de Jalón, A. Callejo, A. F. Hidalgo, Efficient solution of Maggi's equations, *Journal of computational and nonlinear dynamics* 7 (2012).
- [14] M. Berz, C. Bischof, G. Corliss, A. Griewank (Eds.), *Computational Differentiation: Techniques, Applications, and Tools*, SIAM, Philadelphia, PA, 1996.
- [15] A. Griewank, On automatic differentiation, in: M. Iri, K. Tanabe (Eds.), *Mathematical Programming: Recent Developments and Applications*, Kluwer Academic Publishers, Dordrecht, 1989, pp. 83–108.
- [16] A. Griewank, C. H. Bischof, G. F. Corliss, A. Carle, K. Williamson, Derivative convergence for iterative equation solvers, *Optimization Methods and Software* 2 (1993) 321–355.
- [17] J. Utke, OpenAD: Algorithm Implementation User Guide, Technical Memorandum ANL/MCS-TM-274, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, 2004.
- [18] C. H. Bischof, L. Roh, A. Mauer, ADIC — An extensible automatic differentiation tool for ANSI-C, *Software-Practice and Experience* 27 (1997) 1427–1456.
- [19] OpenAD, OpenAD Web Page, <http://www.mcs.anl.gov/OpenAD/>, 2013.
- [20] D. Quinlan, ROSE Web Page, <http://rosecompiler.org>, 2013.
- [21] M. Schordan, D. Quinlan, A source-to-source architecture for user-defined optimizations, in: *JMLC'03: Joint Modular Languages Conference*, volume 2789 of *Lecture Notes in Computer Science*, Springer Verlag, 2003, pp. 214–223.
- [22] M. M. Strout, J. Mellor-Crummey, P. Hovland, Representation-independent program analysis, *SIGSOFT Softw. Eng. Notes* 31 (2006) 67–74.
- [23] S. H. K. Narayanan, B. Norris, P. Hovland, D. C. Nguyen, A. H. Gebremedhin, Sparse jacobian computation using ADIC2 and ColPack, *Procedia Computer Science* 4 (2011) 2115 – 2123. Proceedings of the International Conference on Computational Science, ICCS 2011.
- [24] A. H. Gebremedhin, D. Nguyen, M. Patwary, A. Pothen, ColPack: Software for Graph Coloring and Related Problems in Scientific Computing, Technical Report, Purdue University, 2011.
- [25] A. Walther, A. Griewank, Getting started with ADOL-C, in: U. Naumann, O. Schenk (Eds.), *Combinatorial Scientific Computing*, Chapman-Hall CRC Computational Science, 2012, pp. 181–202.
- [26] H. Pacejka, Tire and Vehicle Dynamics, SAE-R, Society of Automotive Engineers, Incorporated, 2006.
- [27] A. R. Curtis, M. J. D. Powell, J. K. Reid, On the estimation of sparse Jacobian matrices, *J. Inst. Math. Appl.* 13 (1974) 117–119.

The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne"). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up, nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.