Contents lists available at ScienceDirect

Advances in Engineering Software

journal homepage: www.elsevier.com/locate/advengsoft

Performance of automatic differentiation tools in the dynamic simulation of multibody systems

Alfonso Callejo [a,*], Sri Hari Krishna Narayanan [b], Javier García de Jalón [a], Boyana Norris [b]

[a] *Instituto Universitario de Investigación del Automóvil, Universidad Politécnica de Madrid, Madrid, Spain*
[b] *Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, USA*

A R T I C L E I N F O

Article history:
Received 29 November 2013
Received in revised form 25 February 2014
Accepted 9 March 2014
Available online 12 April 2014

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

A B S T R A C T

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 with 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 system dynamics.

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 non-fulfillment 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 some of the stages. Natural (or fully Cartesian) coordinates[1] are used to define local geometry and constraint equations, which simplifies the modeling phase. Positions and velocities are then computed recursively, exploiting the system's tree-structured 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. Since this step takes most of the computation time, it is worth exploring

* Corresponding author. Tel.: +34 913365335.
E-mail address: a.callejo@upm.es (A. Callejo).

[1] Cartesian components of points and unit vectors [2].

http://dx.doi.org/10.1016/j.advengsoft.2014.03.002
0965-9978/© 2014 Elsevier Ltd. All rights reserved.

efficient and accurate ways of differentiating computer functions, while preserving the generality of the implementation.

There are several ways of computing the derivatives of a mathematical function with respect to its independent variables. For 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 and sometimes 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 generality point of view.

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

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

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. Moreover, in the case of vector functions, the computational cost increases quickly as the problem size grows. The ND error can be avoided through the complex-step derivative approximation [5], but this requires variables in the code to be changed to the complex type.

Automatic or *algorithmic differentiation* (AD) allows differentiating a computer function (implemented in Fortran, C, C++, MATLAB, etc.), automatically computing both first-order derivatives (e.g. gradients and Jacobian matrices) and higher-order derivatives (e.g. Hessian matrices). Its development time is shorter than using analytical differentiation techniques, and it generates machine-precision derivatives. In past investigations with the formulation presented here, the operator overloading tool ADOL-C [6] was used successfully [7]. However, a single AD tool was not enough for assessing the computational efficiency, since the different types of AD tools have very different performances. Also, only academic examples were considered.

Little work in the MBS community has thoroughly addressed AD as a way of differentiating computer functions. In 1996, Bischof [8] 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 [9] 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 [10], Dürrbaum et al. proved that the symbolic tool MACSYMA generated derivatives faster than did ADOL-C for two medium-size planar and spatial robots. In 2007, Ambrósio et al. [11] 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. [12] applied the source transformation tool `dcc` and an operator overloading tool to dynamic models. In general, rough descriptions 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. Also, 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 [13] and the operator-overloading tool ADOL-C [6] 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 sparse Jacobian exploitation.

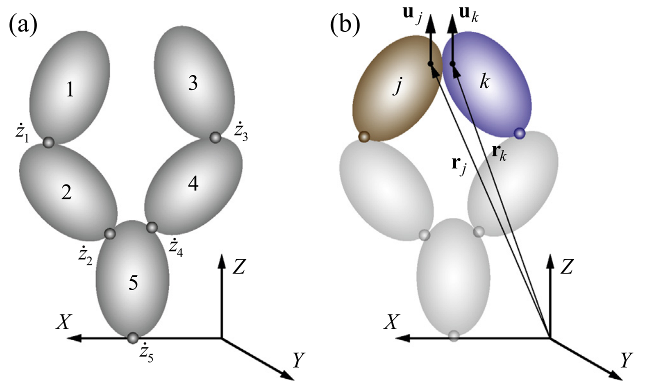

Fig. 1. (a) Tree-structured MBS. (b) Closure-of-the-loop revolute joint.

2. Multibody formulation

In this section, a general-purpose MBS formulation is presented [4]. The formulation is explained in four steps: first, the open-chain recursive differential equations are proposed; second, the loops are closed by introducing position penalty terms; third, the trapezoidal rule of integration is introduced; and fourth, 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-structured multibody system (see Fig. 1(a)). In the case of closed-chain systems, certain joints and rods[2] are temporarily removed and enforced later through constraint equations. Cartesian coordinates are used to define the velocity and acceleration of bodies:

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

$$\dot{\mathbf{Z}}_i \equiv \begin{Bmatrix} \ddot{\mathbf{s}}_i \\ \dot{\boldsymbol{\omega}}_i \end{Bmatrix} \tag{3}$$

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 has interesting advantages [14]. The recursive expression of the Cartesian velocities and accelerations of body i in terms of those of body $i-1$ is

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

$$\dot{\mathbf{Z}}_i = \dot{\mathbf{Z}}_{i-1} + \mathbf{b}_i \ddot{z}_i + \mathbf{d}_i \tag{5}$$

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

[2] Slender bodies with two spherical joints and a negligible moment of inertia around their axis.