PROJECT PART-FINANCED BY THE EUROPEAN UNION

North East South West INTERREG IIIC

MATEO

MATEO ANTASME

Integration of automatic differentiation tools within object-oriented codes: accuracy and timings

Deliverable 2.2
Marco Morandini
Dipartimento di Ingegneria Aerospaziale, Politecnico di Milano

Introduction

Automatic differentiation tools (ADTs) are gaining popularity because they dramatically reduce the theoretical and programming effort required to compute the partial derivatives of extremely complex nonlinear equations. With the aid of these tools it is possible to compute exact derivatives even when the resulting formulæ would be too complex to be derived by hand without making errors. It is also possible to take into account different code paths. The time required to obtain the derivatives is reduced, because it is no longer necessary to write the related formulæ and then the corresponding code, but only to write the code required to compute the function to be derived. The derivatives are also almost error-free because the manual steps are reduced, and they can be used to validate analytical derivatives. For all of these reasons, the use of automatic differentiation tools can help SMEs in reducing the development and validation burden when dealing with new, self-developed simulation codes, thus reducing the time to market of new, advanced products eventually.

Some of the available tools are able to perform a static analysis of the code, and to write the code required for the evaluation of the partial derivatives. The code generated by these tools should be very efficient, because the derivative code can be written without making use of virtual function calls, and can be well optimized by the compiler. Unfortunately, as shown in the first report, almost all the freely available tools that are able to analyse a given program are limited to some version of the FORTRAN language. Other tools perform the differentiation at run time, after having recorded all the floating point operations using a so-called "tape". The library CppAD [2], a COIN-OR project [3], was selected among these tools for the ANTASME automatic differentiation WP. The library has recently changed its licensing scheme to a double license, the Common Public License Version 1.0 (CPL) or the GNU General Public License Version 2 (GPL). Because the GPL is the same license of the multibody code MBDyn, it is possible to distribute a modified MBDyn version together with CppAD.

This document describes the development of an automatic differentiation enabled version of the multibody code MBDyn [1] using the freely available library CppAD [2]. Mbdyn is an implicit integration code; that is, at each time step, it has to solve a set of nonlinear equations, $\boldsymbol{f}(\boldsymbol{x})=\boldsymbol{0}$. The equations are solved by means of the iterative Newton-Raphson method, so requiring the availability of the Jacobian matrix $\boldsymbol{J}=\partial \boldsymbol{f}(\boldsymbol{x})/\partial \boldsymbol{x}$. The equations, as well as the Jacobian matrix, are computed assembling the contributions of different entities, such as nodes, bodies with inertia, deformable elements, constraints and so on. In this report, we will focus our attention on the use of ADTs for the rapid development of new, complex elements. For this reason, the scope of the work is the linearisation of the contribution of a single entity, and not of the whole set of nonlinear equations that are solved by the multibody code.

Implementation steps

In order to perform the automatic differentiation of an arbitrary function $\boldsymbol{f}(\boldsymbol{x})$ the library requires that

1. the values of the independent variables are stored in a vector containing `CppAD::AD<double>` elements, say `x_indep`, where the template `CppAD::AD<class T>` is defined by the library;
2. an analogous vector, say `y_dep`, is defined in order to store the result of the computations;
3. the function `CppAD::Independent(x_indep)` is called; after this step, all the computations involving the independent variables stored inside `x_indep` are taped, and the code should compute ***y*** = ***f***(***x***, ***possible additional parameters***);
4. a variable `f` of type `CppAD::ADFun<doublereal>` is created, stopping the taping;
5. at this point, one can define two vectors of doubles, say `x` and `J`, and store inside `J` the Jacobian matrix of $\boldsymbol{f}(\boldsymbol{x})$, `J = f.Jacobian(x)`; the elements $\partial f_i/\partial x_j$ of the Jacobian matrix are stored inside the vector `J` with the index j varying faster, i.e. by rows.

The code corresponding to steps 1-4 is reported below

```
//Define the independent and dependent vectors
std::vector<CppAD::AD<double> > x_indep(11), y_dep(11);
//Here we must fill x_indep with the current x values
//Identify the independent variables
CppAD::Independent(x_indep);
//Compute f(x_indep, additional parameters) and store the result
     //inside y_dep
AssVec(x_indep, y_dep, dCoef);
//Stop the taping and define the function f
```

```
CppAD::ADFun<double> f(x_indep, y_dep);
```

If the function ***f***(***x***) is computed following the steps outlined above then it is possible to compute the Jacobian matrix ***J*** for a given (possibly different) value of the independent vector, as per step 5 with this simple code:

```
//define the independent and Jacobian matrix vectors
std::vector<double> x(11), J(11 * 11);
//set the values of x (here copy from x_indep)
for (int i=0; i<11; i++) x[i] = CppAD::Value(x_indep[i]);
//compute the Jacobian matrix an store the result in J
f = f.Jacobian(x)
```

In theory, the above steps are the only ones required to perform the automatic differentiation of an arbitrarily complex function $\boldsymbol{y} = \boldsymbol{f}\,(\boldsymbol{x})$. In practice, however, it must be noted that all the computations involved in the evaluation of $\boldsymbol{f}\,(\boldsymbol{x})$, and all the intermediate results must be performed and stored using not the standard double type, but `CppAD::AD<double>` variables.

MBDyn performs most of its computations using two custom classes, called `Mat3x3` and `Vec3`, representing a 3x3 matrix and a 3 element vector, respectively. In order to allow the coexistence of standard and automatic-differentiated elements it was therefore necessary to templatize all the matrix and vector libraries in such a way that the same code could be used with the `double` or the `CppAD::AD<double>` data type, and that the code of all the already existing elements of MBDyn would be left untouched.

Another somewhat intrusive change to the code was required because MBDyn makes use of a whole set of internally defined scalar functions of scalar variables. Almost all these functions had to be modified as well in order to allow the use of the `CppAD::AD<double>` data type and the computations taping.

MBDyn Elements with ADT

Among the element library of MBDyn, three structural elements, namely the "revolute rotation", the "wheel2" and the "gimbal" joint, were modified in order to provide an ADT-computed Jacobian matrix. The first element is a rather simple joint that allows the relative rotation of two nodes along a given axis without imposing any constraint on their relative positions. The code implementing the exact linearization of this element is already available in MBDyn. The automatic differentiation version of the element has been developed only to assess the accuracy of the automatically computed Jacobian matrix and to compare the computational time required to perform a simple simulation. The other two elements are rather complex joints, and the evaluation of their function, ***f***(***x***), is by no means straightforward.

The "wheel2" element is an element that computes the forces exchanged between a rolling wheel and the ground. It can simulate impact conditions, such as during the very first phases of an aircraft landing simulation, braking and lateral sliding. The evaluation of the forces exchanged between the wheel and the ground is performed in a complex subroutine, with about 200 lines of code and lot of different code paths, functions of the wheel-ground relative position and velocity. For these reasons, the “wheel2” element, as currently available with MBDyn, does not provide any means to evaluate the Jacobian matrix of the forces, as the coding effort was estimated to be excessive, and thus implies an explicit integration of the related equations of motion.

The “gimbal” joint is a joint that allows the rotation between two nodes about two orthogonal axes. The angular velocity about the remaining axis is preserved regardless of the relative angle between the two nodes. It is a joint that is often used in the helicopter field in order to simulate the rotor of modern convertiplanes. The functional of the “gimbal” joint heavily depends on the compositions of finite rotation tensors $\boldsymbol{R} \in \mathrm{SO3}$ and on the underlying tangent algebra . Due to its complexity, only an approximate Jacobian matrix has been developed so far.

For all the joints, the code that evaluates the function $\boldsymbol{f}(\boldsymbol{x})$ was templatized is such a way that the same code can used to compute the function using the standard `double` or the `CppAD::AD<double>` data type. This allows to call the fast `double` version every time the code needs only to evaluate the function, and the `CppAD::AD<double>` version every time the Jacobian matrix is required. This step, together with the general modifications made to the MBDyn math library, is almost all that is needed to build an automatically differentiated element. The hand-coded, approximate Jacobian matrix code is compared to the automatic differentiation code in Tables 1 and 2 for the “revolute rotation” and “gimbal” joints, respectively.

```
//set up phase
Mat3x3 R1hTmp(pNode1->GetRRef()*R1h);
Mat3x3 R2hTmp(pNode2->GetRRef()*R2h);
Vec3 MTmp = M*dCoef;
Vec3 e3a(R1hTmp.GetVec(3));
Vec3 e1b(R2hTmp.GetVec(1));
Vec3 e2b(R2hTmp.GetVec(2));
MTmp = e2b*MTmp.dGet(1)-e1b*MTmp.dGet(2);
Mat3x3 MWedgee3aWedge(MTmp, e3a);
Mat3x3 e3aWedgeMWedge(e3a, MTmp);
WM.Sub(1, 1, MWedgee3aWedge);
WM.Add(1, 4, e3aWedgeMWedge);
WM.Add(4, 1, MWedgee3aWedge);
WM.Sub(4, 4, e3aWedgeMWedge);
Vec3 Tmp1(e2b.Cross(e3a));
Vec3 Tmp2(e3a.Cross(e1b));
for (int iCnt = 1; iCnt <= 3; iCnt++) {
   doublereal d = Tmp1.dGet(iCnt);
   WM.PutCoef(iCnt, 7, d);
   WM.PutCoef(3+iCnt, 7, -d);
   d = Tmp2.dGet(iCnt);
   WM.PutCoef(iCnt, 8, d);
   WM.PutCoef(3+iCnt, 8, -d);
}
for (int iCnt = 1; iCnt <= 3; iCnt++) {
   doublereal d = Tmp1.dGet(iCnt);
   WM.PutCoef(7, iCnt, d);
   WM.PutCoef(7, 3+iCnt, -d);
   d = Tmp2.dGet(iCnt);
   WM.PutCoef(8, iCnt, -d);
   WM.PutCoef(8, 3+iCnt, d);
}
```

```
typedef CppAD::AD<doublereal> T;
std::vector<T> x_indep(8, 0.), y_dep(8);
x_indep[6] = M[0];
x_indep[7] = M[1];
CppAD::Independent(x_indep);
AssVec(x_indep, y_dep, dCoef);
CppAD::ADFun<doublereal> f(x_indep, y_dep);
std::vector<doublereal> x(8);
for (int i=0; i<8; i++) {
   x[i] = CppAD::Value(x_indep[i]);
}
std::vector<doublereal> J(8*8);
J = f.Jacobian(xx);
for (integer row = 0; row < 8; row++) {
   for (integer col = 0; col < 8; col++) {
      WM.PutCoef(row+1, col+1, -J[row*8 + col]);
   }
}
```

Table 1: Exact hand-written (left) and automatic differentiation (right) Jacobian

matrix code for the "revolute rotation" joint.

Accuracy and timings

A first simulation was performed using the “revolute rotation” joint; it allows to verify that the Jacobian matrices produced by the hand-written code and by the ADT library are identical within machine precision.
The model is very simple: a node, without mass, is completely constrained with a “clamp” joint; a second node, with an initial angular velocity, is constrained to the first node by a “spherical” and a “revolute rotation” joint. Reaction forces and moments are null throughout the simulation. The number of nonlinear iterations, as well as the joint Jacobian matrices, are equal for the two models, as expected.
The simulation with the ADT version of the joint takes about 4.5 s of CPU time on an Athlon XP 2400+ with a CPU clock of 2 GHz, while it takes only about 3.7 s, on the same PC, with the original version of the joint. As expected, the hand-written code is computationally more efficient, because of the taping and of the run-time differentiation overhead that plagues the ADT library.

```
Mat3x3 Ra(pNode1->GetRRef()*R1h);
Mat3x3 RaT(Ra.Transpose());
double dCosTheta = cos(dTheta);
double dSinTheta = sin(dTheta);
double dCosPhi = cos(dPhi);
double dSinPhi = sin(dPhi);
WM.Add(1,6+1,Ra);
WM.Sub(3+1,6+1,Ra);
Mat3x3 MTmp(Ra*(M*dCoef));
WM.Sub(1,1,MTmp);
WM.Add(4,1,MTmp);
MTmp = RaT*dCoef;
WM.Add(6+1,1,MTmp);
WM.Sub(6+1,3+1, MTmp);
WM.IncCoef(6+1, 9+1, dSinTheta*dSinPhi);
WM.IncCoef(6+2, 9+1, 1. + dCosPhi);
WM.IncCoef(6+3, 9+1, dCosTheta*dSinPhi);
WM.IncCoef(6+1, 9+2, dCosTheta);
WM.DecCoef(6+3, 9+2, dSinTheta);
WM.IncCoef(9+1, 6+1, dSinTheta*dSinPhi);
WM.IncCoef(9+1, 6+2, 1. + dCosPhi);
WM.IncCoef(9+1, 6+3, dCosTheta*dSinPhi);
WM.IncCoef(9+1, 9+1, dSinPhi*(dCosTheta*M(1) -
dSinTheta*M(3)));
WM.IncCoef(9+1, 9+2, dCosPhi*(dSinTheta*M(1) +
dCosTheta*M(3)) - dSinPhi*M(2));
WM.IncCoef(9+2, 6+1, dCosTheta);
WM.DecCoef(9+2, 6+3, dSinTheta);
WM.DecCoef(9+2, 9+1, dSinTheta*M(1) +
dCosTheta*M(3));
```

```
typedef CppAD::AD<doublereal> T;
std::vector<T> x_indep(11, 0.), y_dep(11);
x_indep[6] = M[0];
x_indep[7] = M[1];
x_indep[8] = M[2];
x_indep[9] = dTheta;
x_indep[10] = dPhi;
CppAD::Independent(x_indep);
AssVec(x_indep, y_dep, dCoef);
CppAD::ADFun<doublereal> f(x_indep, y_dep);
std::vector<doublereal> xx(11), J(11 * 11);
for (int i=0; i<11; i++) xx[i] =
CppAD::Value(x_indep[i]);
J = f.Jacobian(xx);
for (int row = 0; row < 11; row++) {
  for (int col = 0; col < 11; col++) {
    WM.PutCoef(row+1, col+1, -J[row*11 + col]);
  }
}
```

Table 2: Approximated hand-written (left) and automatic differentiation (right) Jacobian matrix code for the "gimbal" joint.

The second simulation was performed using the “wheel2” element. This simple model has a wheel that, at the beginning of the simulation, has an initial horizontal velocity but a null angular velocity. The vertical position of the wheel and the vertical contact force change during the simulation (), even losing contact with the terrain three times; this happens because the wheel weight is not balanced in the initial configuration. Throughout the simulation, an increasing couple is applied to the wheel. The initial horizontal velocity quickly spins up the wheel; after that, the increasing couple reverses its angular velocity (Figure 1). Figure 3 shows that the number of iterations required for the convergence, at each time step, of a complex simulation with a "wheel2" joint is drastically reduced if we make use of the ADT-derived Jacobian matrix. The simulation of 2.5 seconds, with a fixed time step of 1.E-3 s, requires a total of 8084 and 10709 nonlinear solver iterations with and without the Jacobian matrix, respectively. Unfortunately, the total computation time is heavily affected by the taping and the run-time automatic differentiation, so that, even performing fewer iterations, the simulation with the ADT version of the joint takes about 7.7 s of CPU time on an Athlon XP 2400+ with a CPU clock of 2 GHz, while takes only about 6.1 s, on the same PC, with the original version of the joint. Note, however, that, despite the overhead due to the run-time construction of the Jacobian matrix, the required computational times are not dramatically different. Moreover, the reduced number of iterations can substantially reduce the total computational time of more complex models.

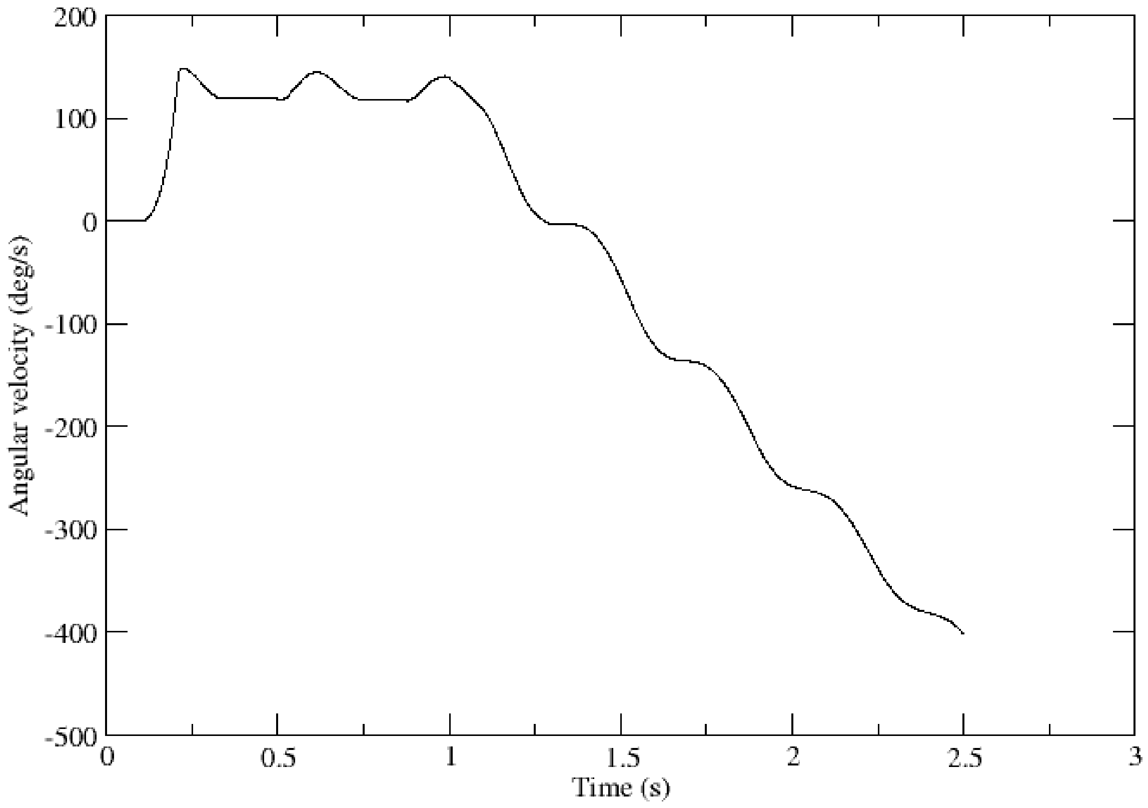

Figure 1: Wheel angular velocity.

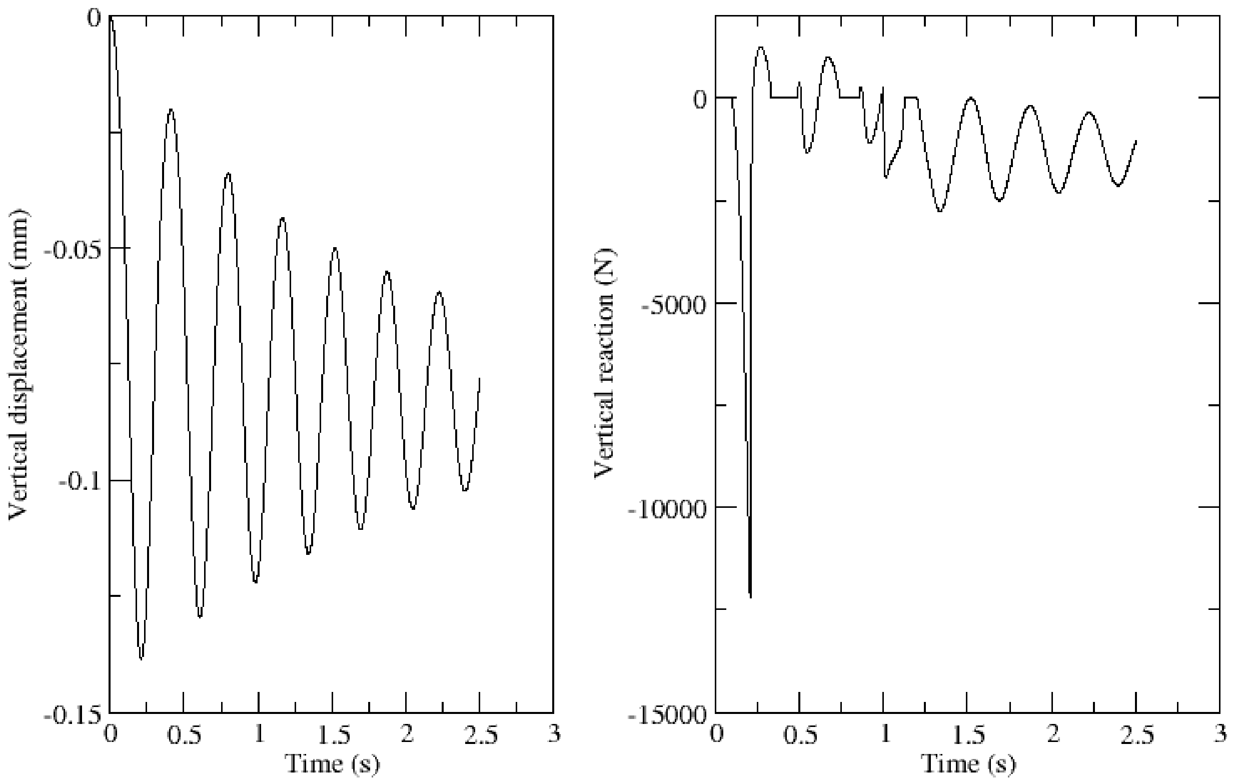

Figure 2: Wheel vertical position (left) and reaction force (right).

These findings are confirmed by a simulation performed using the "gimbal" joint. In this model (Figure 4), a body is kept at a constant angular velocity of 25 rad/s around a fixed axis, while a second body is connected to the first one by a gimbal and a spherical joint. The two body positions are also constrained to lie on a plane which contains the first body rotation axis; the relative positions of the two bodies are made to change during the simulation. The number of nonlinear iterations required, with a fixed time step of 1.E-3 s, is shown in Figure 5; the simulation with the hand-written Jacobian matrix takes 27799 iterations and about 20 s of CPU time, while the simulation with the ADT Jacobian matrix takes 18998 iterations and about 24 s. With this case it is possible to compare the behavior obtained with a larger time step of 1.E-2 s (the "wheel2" example does not converge using larger time steps). With the larger time step, the number of iterations required to reach convergence is much lower with the ADT joint than with the hand-written one (Figure 6). The total number of iterations is equal to 2881 and 6286, respectively, and this difference is so huge that the total computational time of the ADT joint, of about 3 s, is lower than that of the hand-written one, that takes about 3.6 s. Of course, the results are the same, because the code that computes the function $\boldsymbol{f}(\boldsymbol{x})$ is unchanged.

Conclusions

The development of the ADT version of the multibody code MBDyn was not as straightforward as anticipated at the beginning of the work. As expected, the elements with run-time ADT capabilities are slower than the corresponding elements with hand-coded Jacobian matrices. However, ADT techniques allow the rapid development of new, complex elements, without requiring the burden of the linearization of nonlinear equation contributions, still guaranteeing the correctness of the linearization even when the function evaluation can take different code paths. For these reasons, and because the run-time cost of the ADT library is not excessive, the rapid development of new, complex elements can benefit from these techniques. This is even more important with realistic, complex models, were a reduced number of iterations, even at the expense of a slow Jacobian matrix evaluation for one or two elements, can substantially reduce the total computational time.

Bibliography

1) http://www.mbdyn.org/
2) http://www.coin-or.org/CppAD/
3) http://www.coin-or.org/
4) B.A. Dubrovin, A.T. Fomenko, S.P. Novikov; Modern Geometry - Methods and Applications: Part I: The Geometry of Surfaces, Transformation Groups, and Fields, New York: Springer, 1985.

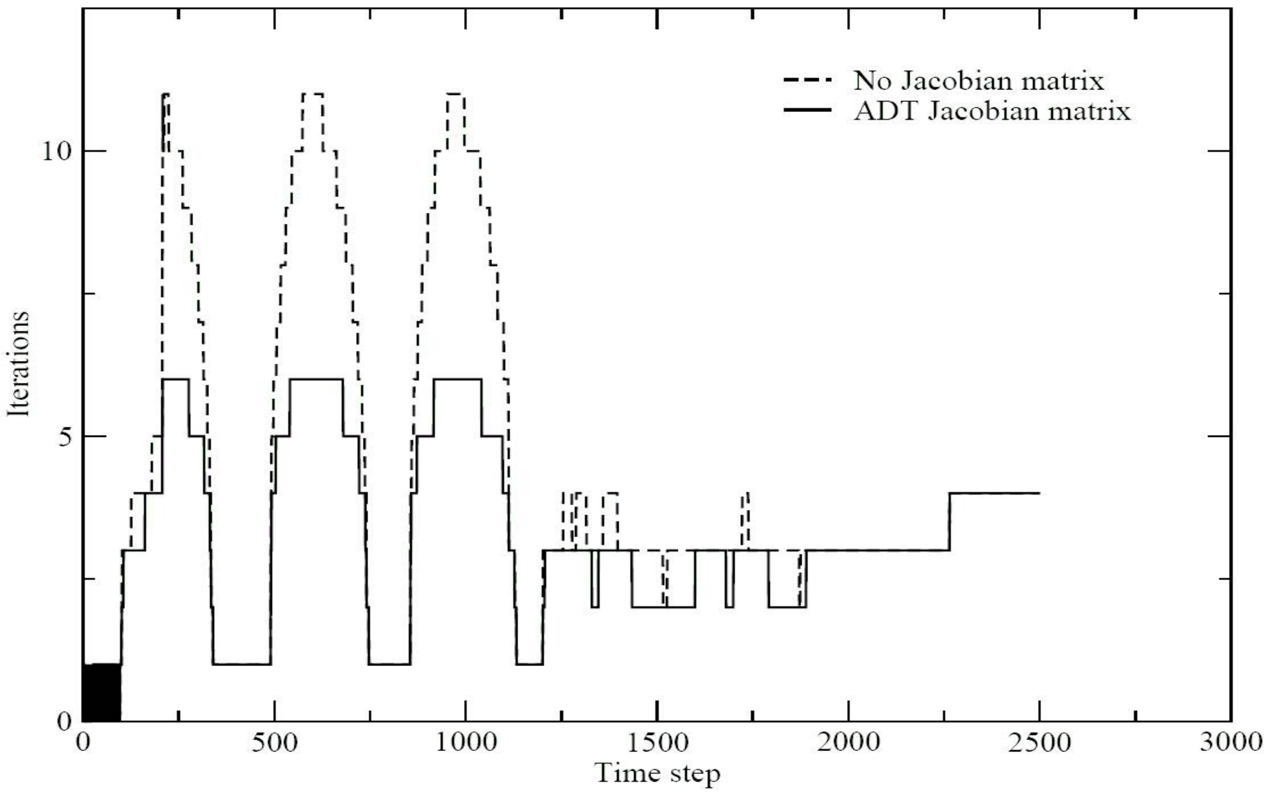

Figure 3: "Wheel2" joint iterations count required for convergence

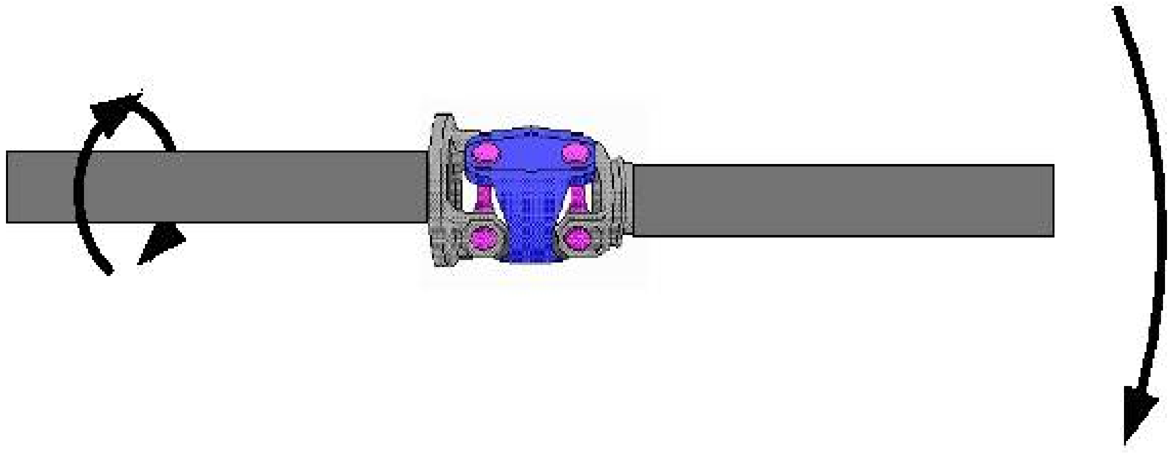
Figure 4: Sketch of the model using the gimbal joint.

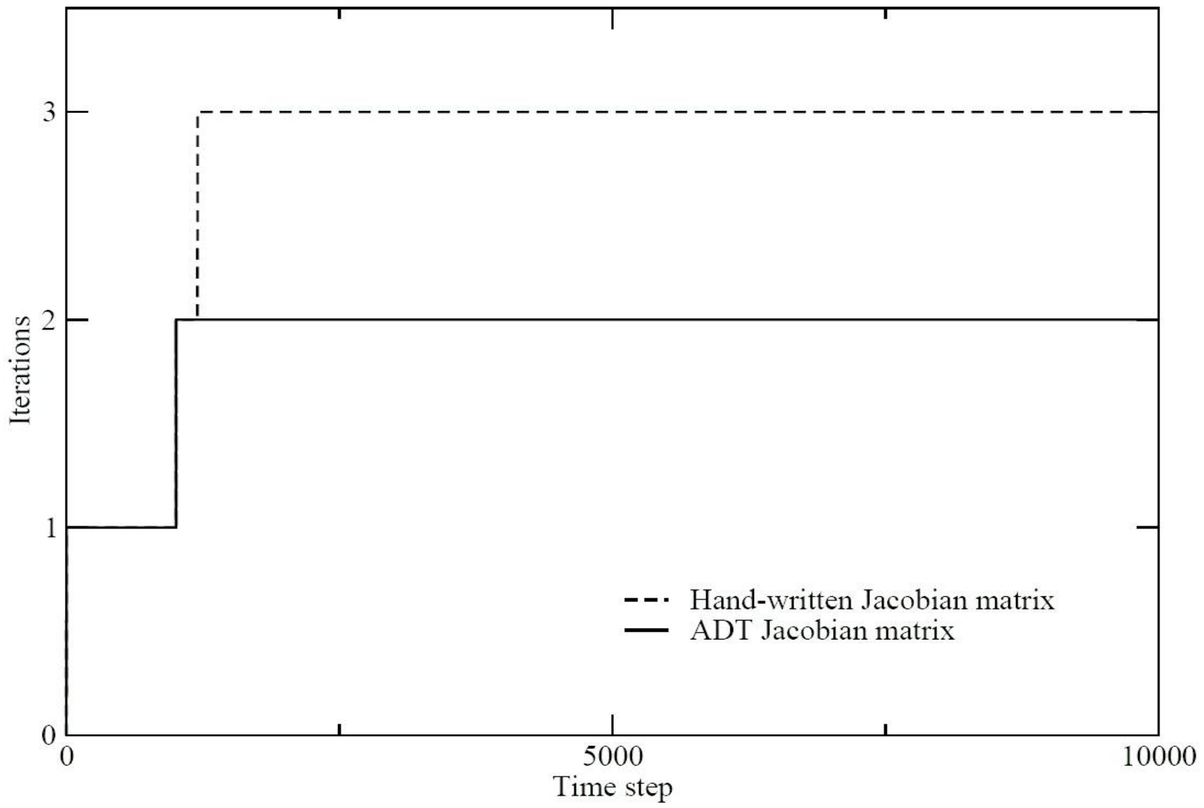

Figure 5: "Gimbal" joint iterations count required for convergence, dt = 1.E-3 s

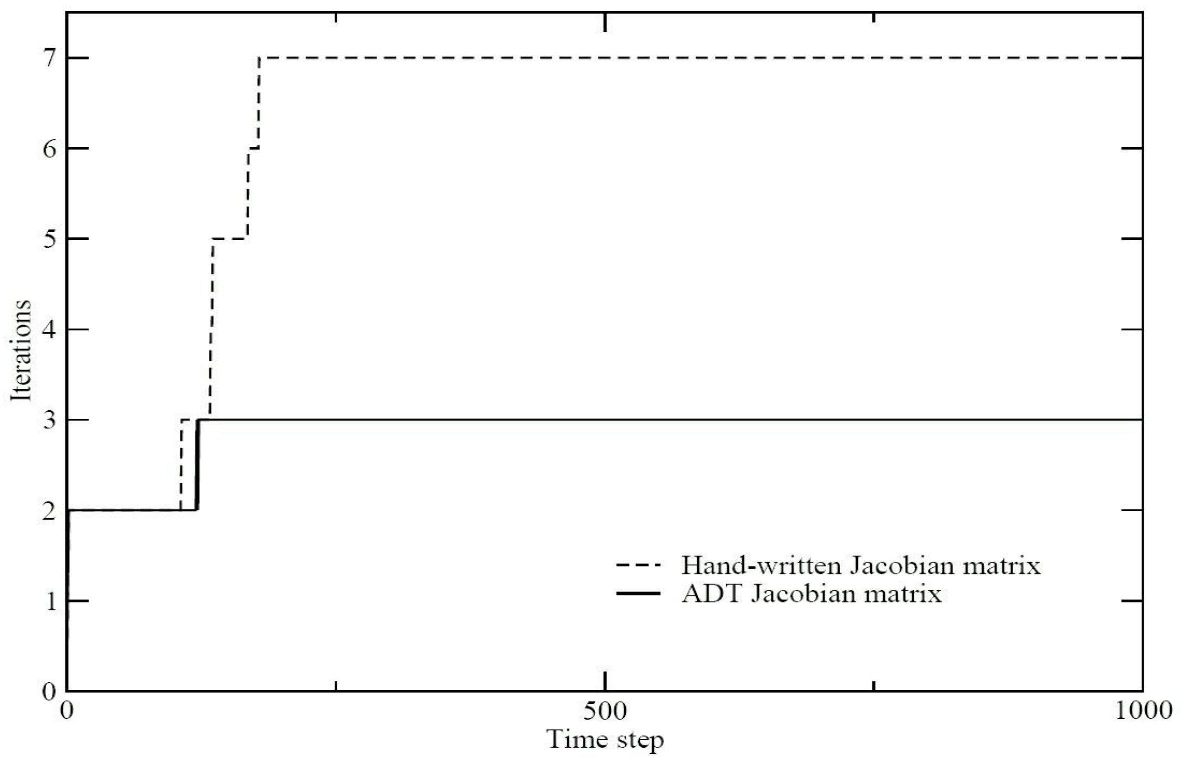

Figure 6: "Gimbal" joint iterations count required for convergence, dt = 1.E-2 s