Automatic differentiation techniques

Marco Morandini

Dipartimento di Ingegneria Aerospaziale – Politecnico di Milano

PROJECT PART-FINANCED BY THE EUROPEAN UNION
North East South West INTERREG IIIC
MATEO
POLITECNICO DI MILANO

WP2 - Automatic differentiation techniques

Implicit code:

- Set of nonlinear equations $\mathbf{f}(\mathbf{x}) = \mathbf{0}$
- Jacobian matrix $\mathbf{J} = \partial\mathbf{f} / \partial\mathbf{x}$ (cumbersome)

Rapid elements prototyping

- Code $\mathbf{f}(\mathbf{x}) = \mathbf{0}$
- Let the code compute $\mathbf{J}$

MBDyn:

- Multibody code

  http://www.aero.polimi.it/~mbdyn
- GPL
- C++

Automatic differentiation techniques

- Source code analysis

  (Fortran/C)

  - Requires: code instrumentation
  - Provides: compiled **J**
- Tape of operations

  (operator overloading, C++)

  - Requires: custom data types /templates (C++)
  - Provides: run-time **J**

Survey of available libraries

Tool	Language	License	Source code	Method
ADF95	Fortran77/Fortran95	Custom non.profit	Yes $	Code analysis?
ADIC	C/C++	?	No	
ADIFOR	Fortran77	Custom non-profit	Yes	Code analysis?
AdiMat	MATLAB	To be defined	No	
ADOL-C	C/C++	CPL	Yes	Tape
AUTO_DERIV	Fortran77/Fortran95	Custom non.profit	Yes $	Code analysis?
CppAD	C/C++	CPL/GPL	Yes	Tape
FAD	C/C++	Teaching, non profit	Yes	Tape?
FADBAD/TADIFF	C/C++	Custom non.profit	Yes	Tape
GRESS	Fortran77	?	Yes	?
OpenAD	Fortran77/Fortran95	?	No?(EDG front end)	Code analysis
TAMC	Fortran77	Non-profit	No	Code analysis?
TAPENADE	Fortran77/Fortran95	?	No	Code analysis?
TaylUR	Fortran95	Custom non.profit	Yes $	Code analysis?
TOMLAB/MAD	MATLAB	?	$$	Tape

Steps required

- Templatize MBDyn: double vs. CppAD<double>
- Differentation of rotation matrices $R \in SO(3)$
- Elements:
  - Revolute rotation (simple): relative rotation along a given axis allowed without position constraints
  - Wheel (no analytical Jacobian matrix): tyre model
  - Gimbal: rotation about two orthogonal axes allowed; constant angular velocity about the remaining axis

WP2 – Coding effort

- Once for all: enable the code (math library)

```
double Mat3x3::dDet(void) const
{
   double* p = (double*)pdMat;

   return p[M11]*(p[M22]*p[M33]-p[M23]*p[M32])
     +p[M12]*(p[M23]*p[M31]-p[M21]*p[M33])
     +p[M13]*(p[M21]*p[M32]-p[M22]*p[M31]);
}
```

```
template<class T>
T Mat3x3T<T>::dDet(void) const
{
   T* p = (T*)pdMat;

   return p[M11]*(p[M22]*p[M33]-p[M23]*p[M32])
     +p[M12]*(p[M23]*p[M31]-p[M21]*p[M33])
     +p[M13]*(p[M21]*p[M32]-p[M22]*p[M31]);
}

typedef Mat3x3T<double> Mat3x3
```

- Element level: **J** (requires **f**)

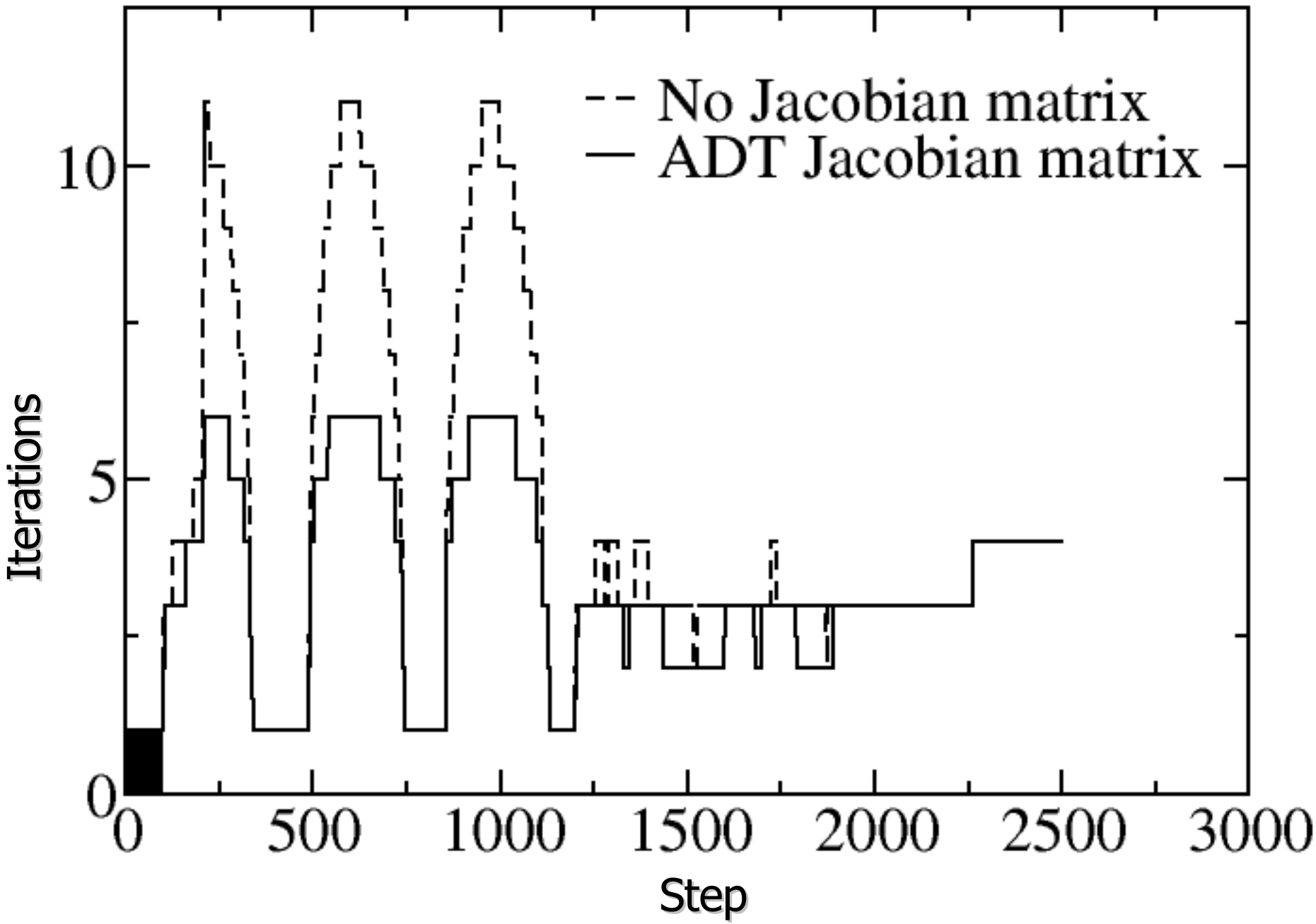
```
std::vector<T> y_dep(12);                          //declare dep variables
CppAD::Independent(x_indep);                       //declare indep variables
res_vec(p, x_indep, y_dep, pEl->GetLabel());       //compute f(x)
CppAD::ADFun<doublereal> f(x_indep, y_dep);

.....

J = f.Jacobian(xx);                                //Compute J(x)
```

Code comparison: revolute rotation joint

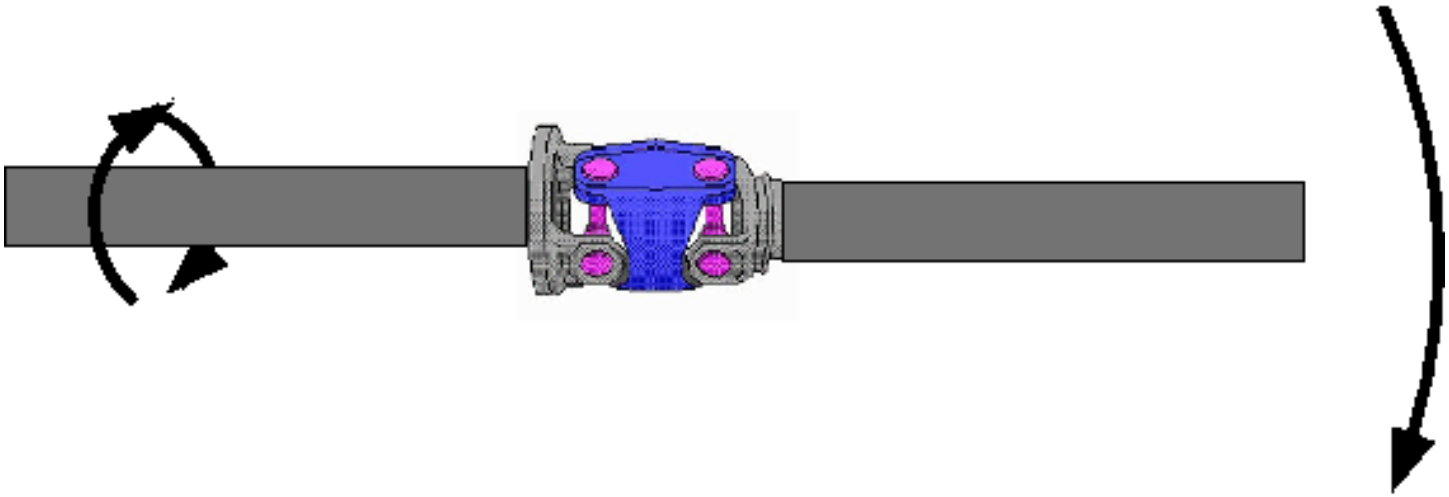
```
//set up phase
Mat3x3 R1hTmp(pNode1->GetRRef()*R1h);
Mat3x3 R2hTmp(pNode2->GetRRef()*R2h);
Vec3 MTmp = M*dCoef;
Vec3 e3a(R1hTmp.GetVec(3));
Vec3 e1b(R2hTmp.GetVec(1));
Vec3 e2b(R2hTmp.GetVec(2));
MTmp = e2b*MTmp.dGet(1)-e1b*MTmp.dGet(2);
Mat3x3 MWedgee3aWedge(MTmp, e3a);
Mat3x3 e3aWedgeMWedge(e3a, 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]);
   }
}
```

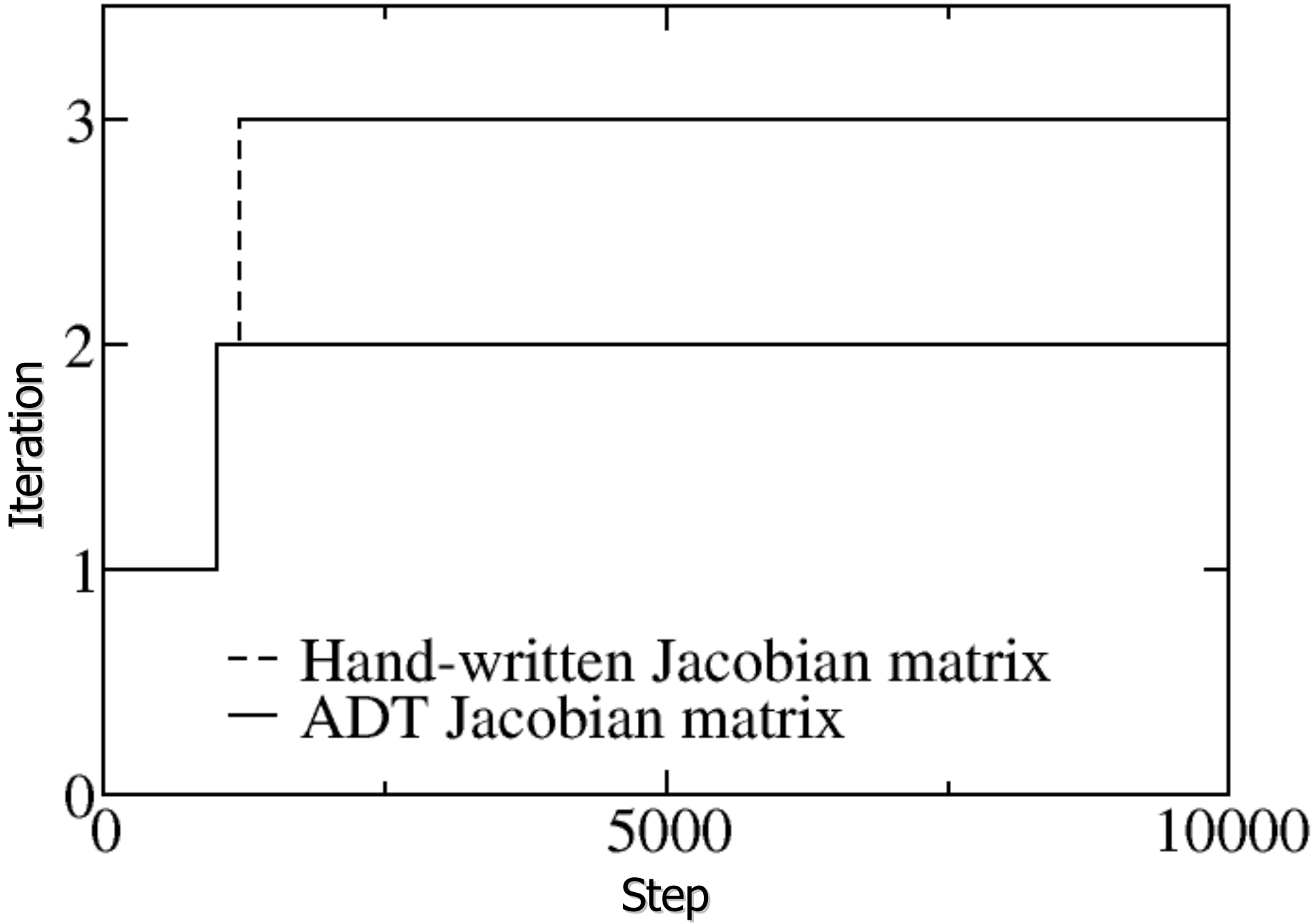
Wheel2: iterations



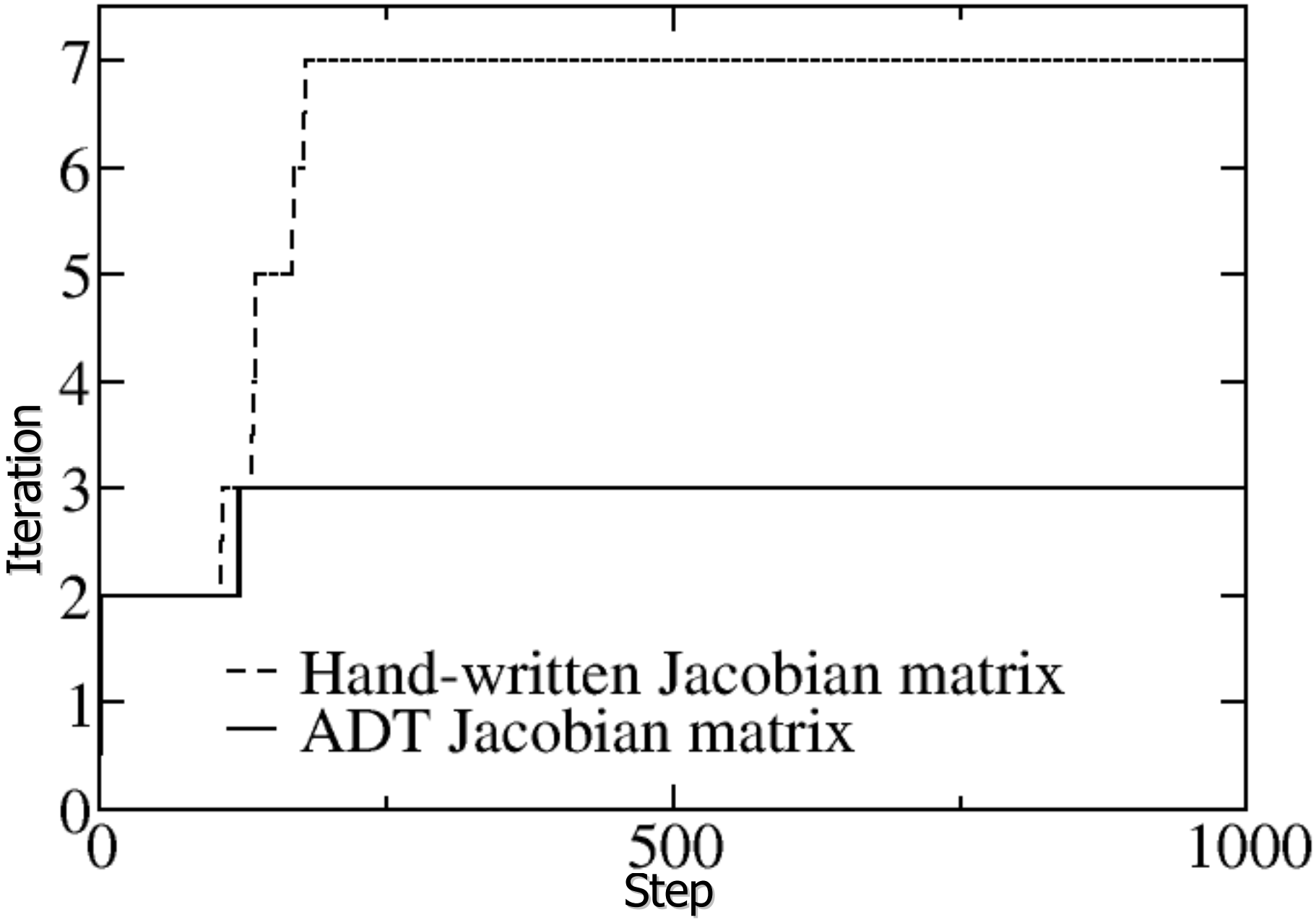
Gimbal model



Gimbal: iterations, dt = 1E-3 s



Gimbal: iterations, dt = 1E-2 s



Timings

Athlon XP 2400+

	Analytical $\boldsymbol{J}$		Autodiff $\boldsymbol{J}$	
Revolute rotation	=	3.7 *s*	=	4.5 *s*
Gimbal, dt = 1.E-3 s	27799 *its*	20 *s*	18998 *its*	24 s
Gimbal, dt = 1.E-2 s	6286 *its*	3.6 *s*	2881 *its*	3 *s*
Wheel	1079 *its*	6.1 *s*	8084 *its*	7.6 *s*

Conclusions

Automatic differentiation techniques:

- Validation of existing elements
- Rapid development of new elements
- Less iteration -> reduced (if not null) run-time cost
  - possible tun-time savings with complex models
- Autodiff-enabled MBDyn: soon downloadable

- Difficult step: enable the code

WP9 - Dissemination

http://www.aero.polimi.it/Antasme

Send us your

- Presentations
- Documents
- WPs