EASYDYN: A FRAMEWORK BASED ON FREE SYMBOLIC AND NUMERICAL TOOLS FOR TEACHING MULTIBODY SYSTEMS

O. Verlinden, G. Kouroussis, C. Conti

Department of Theoretical Mechanics, Dynamics and Vibrations
Faculté polytechnique de Mons, Bd Dolez 31, 7000 Mons, Belgium
e-mails: Oliver.Verlinden@fpms.ac.be Georges.Kouroussis@fpms.ac.be Calogero.Conti@fpms.ac.be web page: http://mecara.fpms.ac.be

Keywords: Multibody systems, education, object-oriented programming, open source, symbolic, numerical

Abstract. EasyDyn is an open source framework, available for free on the net, allowing to easily write a program which simulates the motion of a multibody system from only a description of the motion in terms of the chosen configuration parameters and the expression of the efforts exerted on each body. The configuration parameters necessarily correspond to generalized coordinates, whose number is equal to the number of degrees of freedom. EasyDyn obliges the user to understand some theoretical principles but automatic tools take charge of the tedious part of the job, principally the expression of velocities and accelerations by derivation of position, and the development of vector expressions.

EasyDyn provides on one hand a C++ library which offers the following components

- classes implementing the vector algebra: vectors, rotation and inertia tensors, homogeneous transformation matrices so that vector expressions can be written as is in the C++ code;
- routines to build 3D scenes that can be viewed by an external program called EasyAnim, also available for free;
- routines to numerically solve second-order differential equations;
- numerical routines to build the residual form of the equations of motion of a multibody system from kinematics and applied forces.

The C++ library is completed by CAGeM, a MuPAD script, generating a basic C++ EasyDyn application from inertia data and position expressions related to each body. After generation, the user just has to complete the C++ code with the expression of the efforts. EasyDyn was developed expressly for teaching with the concerns of readability and compactness instead of efficiency. Due to the elegant balance between numerical and symbolic tools, and the large use of existing packages, the development effort was minimal. It has also been tested successfully on reasonably complex systems.
1 INTRODUCTION

Anybody involved in the education of numerical methods like finite elements, computational fluid dynamics or, of course, multibody systems, is confronted with the organization of workshops which, on one hand, illustrate the developments and principles of the theoretical course and on the other hand let the students deal with tools and problems that they could encounter in industry. Clearly, commercial softwares are perfectly suitable only for the second task: they are developed for ease of use, efficiency and productivity and hide as far as possible the theoretical background. Rewriting a simulation program from scratch could illustrate all the theoretical and numerical aspects but, even limited to simple systems, would require a lot of time and also more programming than mechanical skills. Let us indeed recall that the laws of dynamics, although apparently quite simple when expressed in vector form, become awfully complex after the necessary transformation into the 3 equivalent scalar expressions obtained by projection onto a given coordinate system. This is namely due to the numerous transfers between local and global coordinate systems. The calculation of velocities and accelerations also leads to long mathematical expressions, although they result basically from a systematic derivation of the positions.

That’s the reason why the authors developed a framework called EasyDyn, which allows the student to perform simulations quite easily but only if he has a sufficient knowledge of the discipline. We kept in mind the following considerations during the development

- the tool must be open source [1], in the hope that it can be useful not only to our students but also to anybody interested in the subject;
- existing libraries or programs will be used for particular problems as far as they are freely available, at least for non commercial usage;
- it shouldn’t be limited to simplistic 2D systems;
- it should offer the possibility to be extended to mechanical systems involving particular components (controllers, hydraulic or electric actuators, ...);
- it should be portable and run under Windows and Unix;
- it should be composed of several independent elements with a specific task; more especially, the computational part won’t be mixed with any graphical result.

The adopted solution is not a simulation program but a C++ library, combined with a symbolic utility. The complete package, freely available from the internet [1], comprises a user guide, examples and a series of problems. The latter are especially useful for teachers. Under EasyDyn, the mechanical system is not defined in a data file but as a C++ program describing the kinematics and the applied forces. The equations of motion can then be built and integrated numerically with the help of particular routines of the library. To make the task easier and the code particularly readable, the object oriented features of the C++ language have been exploited to redefine a comprehensive vector algebra built from classes like vectors, rotation or inertia tensors and homogeneous transformation matrices, combined with the overloading of the usual operators. In such a way, vector expressions can be written as is in the C++ program, like a matrix expression under Matlab-alike programs. Last but not least, EasyDyn provides

a utility which generates symbolically a complete C++ code, where the kinematics (velocities and accelerations) has been derived symbolically from only the position information. This last module is actually a script running under MuPAD, developed at the university of Paderborn in Germany. Although available for free for non commercial usage, MuPAD offers features comparable to its commercial counterparts.

In this paper, we first describe the modules of the C++ library and then the symbolic script. Some illustrative and advanced examples will be presented, as well as some improvements with regard to computational efficiency.

2 THE C++ LIBRARY

The C++ library of EasyDyn library provides 4 modules with a specific task

- the vec module defines classes related to vector calculus (vectors, rotation tensors, inertia tensors, and homogeneous transformation matrices), and the related algebra so that vector expressions can be written in natural form in a C++ program; this module is completely independent;

- the sim module provides routines for the integration of second-order differential equations written in residual form; this module can be used on its own;

- the mbs module is a frontend to sim which automatically builds the differential equations of motion of a multibody system from the kinematics and the applied forces; the mbs module also features routines for the application of usual forces; this module relies on the vec and sim modules.

- the visu module provides routines to build 3D scenes, composed of moving objects, that can be viewed by external programs; the visu largely uses the vec module.

2.1 The vec module

The need for vector algebra is evident in multibody systems. Forces, position vectors, velocities, accelerations are all vectors. In practice, vector calculus needs the projection in an arbitrarily chosen frame (coordinate system). A frame $i$ is composed of 3 unit vectors $x_i$, $y_i$, and $z_i$ perpendicular to each other and positively oriented. Let us denote $\{a\}_i$ the 3x1 vector gathering the 3 components of vector $a$ in frame $i$

$$\begin{bmatrix} a_{x_i} \\ a_{y_i} \\ a_{z_i} \end{bmatrix}$$

The transfer between different coordinate systems $i$ and $j$ is easily written in the following matrix form

$$\{a\}_i = \begin{bmatrix} a_{x_i} \\ a_{y_i} \\ a_{z_i} \end{bmatrix} \leftrightarrow a = a_{x_i} \cdot x_i + a_{y_i} \cdot y_i + a_{z_i} \cdot z_i$$

(1)

The transfer between different coordinate systems $i$ and $j$ is easily written in the following matrix form

$$\{a\}_i = \begin{bmatrix} R_{i,j} \end{bmatrix} : \{a\}_j$$

with

$$R_{i,j} = \begin{bmatrix} \{x_j\}_i & \{y_j\}_i & \{z_j\}_i \end{bmatrix}$$

(2)

where $R_{i,j}$ is the rotation tensor describing the orientation of frame $j$ with respect to frame $i$.

Frames are also used to localize points in space. We will denote $r_{P,i}$ the coordinate vector of point P with respect to frame $i$, that’s to say the vector running from frame $i$ to point P (figure 1). Generally, a global reference exists and is referenced as frame 0. The coordinate vector of a
Figure 1: Points, frames and coordinate vectors

point \( \mathbf{P} \) with respect to the global reference frame, that’s to say \( \mathbf{r}_{\mathbf{P}/0} \) can be denoted for the sake of simplicity by \( \mathbf{e}_{\mathbf{P}} \).

In the same way, we need to be able to localize the situation of a body with respect to a reference frame. A frame \( j \) is then attached to the body and its situation with respect to the reference frame \( i \) will be expressed from the 4x4 homogeneous transformation matrix \( \mathbf{T}_{i,j} \) defined as

\[
\mathbf{T}_{i,j} = \begin{pmatrix} \mathbf{R}_{i,j} & \{ \mathbf{r}_{j/i} \}_i \\ 0 & 0 & 0 & 1 \end{pmatrix}
\] (3)

The homogeneous transformation matrices also have the advantage to enjoy the following two properties

\[
\left( \begin{pmatrix} \{ \mathbf{r}_{\mathbf{P}/i} \}_i \\ 1 \end{pmatrix} \right) = \mathbf{T}_{i,j} \cdot \left( \begin{pmatrix} \{ \mathbf{r}_{\mathbf{P}/j} \}_j \\ 1 \end{pmatrix} \right) \quad \mathbf{T}_{i,k} = \mathbf{T}_{i,j} \cdot \mathbf{T}_{j,k}
\] (4)

In particular, the last relationship allows to develop the motion along a kinematic chain as the product of successive elementary matrices.

Taking these considerations into account, the \texttt{vec} module offers the following classes

- vectors (class \texttt{vec}), consisting of 3 public variables, \( x \), \( y \) and \( z \) representing the coordinates of the vector in an orthonormal dextrosum coordinate system;
- rotation tensors (class \texttt{trot}), consisting of the 9 coefficients of the matrix (public variables \( r_{11}, r_{12}, ..., r_{33} \));
- inertia tensors (class \texttt{tiner}), equivalent to the 3x3 symmetric matrix built from the 3 moments of inertia \( I_{xx}, I_{yy}, I_{zz} \) and the 3 inertia products \( I_{xy}, I_{xz}, I_{yz} \);
- homogeneous transformation matrices (class \texttt{mth}) built from a vector and a rotation tensor.

The classical operators are overloaded and allow namely vector addition, scalar and vector products, multiplication by a rotation tensor or a homogeneous transformation matrix. Several methods also exist to assign the objects. In particular, a transformation matrix can be initialized from the multiplication of elementary matrices corresponding to displacements or rotations about a given axis. The following piece of code is given for the purpose of illustration
The vector algebra defined in the vec module assumes that the vectors are projected in the right coordinate system. It is the responsibility of the user to verify this condition.

2.2 The sim module

The sim module consists of routines to integrate second-order differential equations of the form

$$f(q, \dot{q}, \ddot{q}, t) = 0$$

expressed in terms of time $t$, the state variables $q$ and their first and second time derivatives $\dot{q}$ and $\ddot{q}$.

The only integration method available so far is based on the implicit Newmark formulas, classically used in finite elements and characterized by an accuracy order of 2. The integration process implements an adaptive time step and can deal with stiff equations. More details about the implementation can be found in [8], as well as the example of a hydraulic system showing that the tool is not limited to multibody systems.

Figure 2: Using the sim module of EasyDyn
The utilization of the module, illustrated in figure 2, generally consists in a call to the routine NewmarkIntegration which will perform the integration of the differential equations as far as the residuals $f$ are described in the routine ComputeResidual. At regular intervals, the integration process calls the routine SaveData provided by the user. Most of the time, it will come down to a call to the SaveStateVariables but the user is free to save some more data or to perform any other action expected to be done at regular intervals (a digital controller for example). The results are saved in text form and can be plotted by a tool like gnuplot.

Besides the integration, mbs provides a routine to determine the state of static equilibrium and a routine which determines the poles of the system after linearization of the differential equations about the current configuration.

The numerical methods implemented in mbs are largely based on matrices. All matrix operations rely on the Gnu Scientific Library (GSL), except for the eigenvalue problem which uses a routine from LAPACK.

2.3 The mbs module

The mbs module is a frontend to sim which automatically builds the residuals of the motion equations of a multibody system as far as the user formulates, for each body, the kinematics (position, velocities and accelerations) of the center of gravity and the resultant wrench of the applied efforts. The kinematics is expressed in terms of the chosen configuration parameters $q$ and their first and second time derivatives $\dot{q}$ and $\ddot{q}$. This implies that only generalized coordinates can be used, whose number corresponds exactly to the number of degrees of freedom. The module mbs cannot deal with constraint equations.

The differential equations governing the motion are constructed by application of the d’Alembert’s principle [5], which has the advantage to eliminate automatically all the joint forces. If the system comprises $n_B$ bodies and $n_{cp}$ configuration parameters (degrees of freedom), the $n_{cp}$ equations can be built from

$$
\sum_{i=1}^{n_B} \left[ d^{ij} \cdot (\mathbf{R}_i - m_i \mathbf{a}_i) + \theta^{ij} \cdot (\mathbf{M}_i - \Phi_G \mathbf{J}_i - \mathbf{\omega}_i \times \Phi_G \mathbf{\omega}_i) \right] = 0 \quad j = 1, n_{cp}
$$

Figure 3: Reference frame of a body
with

- $m_i$ and $\Phi_{G_i}$ the mass and the central inertia tensor of body $i$;
- $R_i$ and $M_{G_i}$ the resultant force and moment, at the center of gravity $G_i$, of all applied efforts exerted on body $i$;
- $a_i$ the acceleration of the center of gravity of body $i$;
- $d^{ij}$ the partial contributions of $\dot{q}_j$ in the velocity $v_i$ of the center of gravity of body $i$

$$v_i = \sum_{j=1}^{n_{cp}} d^{ij} \cdot \dot{q}_j$$  \hspace{1cm} (7)

- $\theta^{ij}$ the partial contributions of $\dot{q}_j$ in the rotational velocity $\omega_i$ of body $i$ such that

$$\omega_i = \sum_{j=1}^{n_{cp}} \theta^{ij} \cdot \dot{q}_j$$  \hspace{1cm} (8)

The $n_{cp}$ resulting equations of motion have the well known following form

$$M(q) \cdot \ddot{q} + h(q, \dot{q}, t) = 0$$  \hspace{1cm} (9)

with

- $M$ the mass matrix of dimension $n_{cp} \times n_{cp}$, defined by

$$M_{jk} = \sum_{i=1}^{n_{B}} \left[ m_i d^{ij} \cdot d^{ik} + \theta^{ij} \cdot (\Phi_{G_i} \cdot \theta^{ik}) \right]$$  \hspace{1cm} (10)

- $h$ a general term gathering the centrifugal and Coriolis terms and the contribution of the applied efforts.

Practically, the user provides three principal routines

- **SetInertiaData** expected to initialize the inertia properties of all the bodies of the system;
- **ComputeMotion** which describes, in terms of the configuration parameters and their time derivatives, the kinematics of each body, that’s to say, the position matrix $T_{0,i}$, the translation and rotation velocities $v_i$ and $\omega_i$ and the corresponding accelerations $a_i$ and $\dot{\omega}_i$.
- **AddAppliedEfforts** which builds the resultant force $R_i$ and the resultant moment $M_{G_i}$ of the applied efforts exerted on each body.

Once these routines are available, the residuals of the equations of motion can be built by a call to **ComputeResidualMbs**. By calling the latter from **ComputeResidual**, the integration routines of the **sim** module can be called directly. The separation of the two routines was made on purpose. The equations of the multibody part are built by **ComputeResidualMbs**.
and supplementary differential equations can be defined in ComputeResidual, to describe the electrical or hydraulic parts of the complete system.

Let us remark that the user doesn’t have to provide the expression of the partial contributions. It is indeed clear that the partial contribution $d_{i,j} (\theta_{i,j})$ is equal to the variation of the translation (rotation) velocity of body $i$ for a unit variation of $q_j$. Let us note that in the development version of EasyDyn, the user can provide a direct expression for the partial contributions which allows to considerably accelerate the computational process, especially with treelike systems.

The mbs module also provides routines to apply efforts related to classical force elements: spring, damper, gravity, contact point plane with friction and also a tire element. The latter is built on the model of the university of Arizona \cite{3,4} and is limited so far to a flat ground.

2.4 The visu module

The visu module allows to define a graphical scene composed of simple objects like boxes, frustums, lines, triangles, …. Each shape is attached to an homogeneous transformation matrix, generally the one giving the situation of a body, allowing to build successive configurations of the scene that can be saved to a file and animated by an independent viewer called EasyAnim also available from the web site of EasyDyn.

Figures 7 and 9 are examples of scenes built with the help of the visu module.

2.5 Example

Let us consider the double pendulum illustrated in figure 4. It is composed of 2 bodies, each one with its own frame. Two revolute joints constraint the motion of the bodies, on O between the ground and body 1 and on A between bodies 1 and 2. It is easy to figure out that the system has 2 degrees of freedom so that the configuration of the system can be univoquely defined from angles $q_1$ and $q_2$ indicated on the figure.

![Figure 4: Double pendulum](image)

Writing the routine SetInertiaData is just the matter of assigning the right variables

```cpp
void SetInertiaData()
{
    // Inertia data for body 0
```
The complete kinematics can be described in about ten explicit lines ($q[0]$ and $q[1]$ refer to $q_1$ and $q_2$)

```c
void ComputeMotion()
{
    body[0].T0G=Trotz(q[0])*Tdisp(0,-0.5*l1,0);
    body[0].omega.put(0,0,qd[0]);
    body[0].omegad.put(0,0,qdd[0]);
    vec OG=body[0].T0G.R*vcoord(0,-0.5*l1,0);
    body[0].vG=(body[0].omega^OG);
    body[0].aG=(body[0].omegad^OG)+(body[0].omega^(body[0].omega^OG));

    body[1].T0G=Trotz(q[0])*Tdisp(0,-l1,0)*Trotz(q[1])*Tdisp(0,-0.5*l2,0);
    body[1].omega=body[0].omega+vcoord(0,0,qd[1]);
    body[1].omegad=body[0].omegad+vcoord(0,0,qdd[1]);
    vec AG2=body[1].T0G.R*vcoord(0,-0.5*l2,0);
    body[1].vG=2*body[0].vG+(body[1].omega^AG2);
    body[1].aG=2*body[0].aG+(body[1].omegad^AG2) + (body[1].omega^((body[1].omega^AG2)));
}
```

This small piece of code illustrates some interesting utilizations of the `vec` module:

- the position matrix is built from the multiplication of predefined forms corresponding to rotations or displacements ($Trotz$ and $Tdisp$);

- the homogeneous transformation matrix ($T0G$) comprises a rotation tensor ($R$) which can be used to build a vector expressed in the global frame ($OG$ or $AG2$) from its local coordinates.

In this case where only the gravity is considered, the efforts would be easily described in the following way

```c
void AddAppliedEfforts()
{
    // Contribution of external applied forces
    vec gravity(0,-9.81,0);
    AddGravityForces(gravity);
}
```

The main part of an `EasyDyn` application is always based on the same model: specification of the number of degrees of freedom and bodies, initialization, call to the integration routine and closing. In the case of the double pendulum, we have

```c
int main()
{
    // Initialisation and memory allocation
    nbrdof=2; nbrbody=2; application="dp2";
    InitEasyDynmbs();
    // Initial configuration
    ```
The symbolic part of EasyDyn

Even with the help of the vector classes, the kinematics remains problematic for an unexperienced user. It can be dramatically simplified if we figure out that all the kinematics can be derived from only the homogeneous transformation matrices.

The translation velocity $v_i$ of body $i$ can indeed be derived directly from the homogeneous transformation matrix

$$\{v_i\}_0 = \frac{d}{dt}\{e_i\}_0 = \sum_{j=1}^{n_{\text{ep}}} \frac{\partial\{e_i\}_0}{\partial q_j} \cdot \dot{q}_j = \sum_{j=1}^{n_{\text{ep}}} \{d_{i,j}\}_0 \cdot \dot{q}_j \quad (11)$$

In the same way the rotation vector is related to the time derivative of the rotation tensor by

$$\{\dot{\omega}_i\}_0 = \left( \begin{array}{ccc} 0 & -\omega_{yi} & \omega_{xi} \\ \omega_{zi} & 0 & -\omega_{xi} \\ -\omega_{yi} & \omega_{xi} & 0 \end{array} \right) = \dot{R}_{0,i} \cdot R_{0,i}^T \quad (12)$$

One further derivation then naturally leads to the accelerations

$$\{a_i\}_0 = \frac{d}{dt}\{v_i\}_0 \quad \{\ddot{\omega}_i\} = \frac{d}{dt}\{\dot{\omega}_i\}_0 \quad (13)$$

---

**Figure 5**: Data flow when combining CAGE and the C++ library
A supplementary tool, called CAGeM (Computer Aided Generation of Motion) has been developed to help the users of EasyDyn. Practically, CAGeM is a MuPAD script which builds the core of a C++ application using mbs, to simulate the behaviour of a multibody system. The complete data flow when using CAGeM is illustrated in figure [5].

To use CAGeM, the user provides a MuPAD code with the following information:

• the number of bodies and the number of configuration parameters;
• the inertia data of each body;
• the expression of the homogeneous transformation matrices of each body, expressed in terms of the chosen configuration parameters;
• the initial conditions;
• the gravity vector;
• some option flags.

The script CAGeM uses the symbolic derivation features of MuPAD to build the expressions of velocities and accelerations from the position matrices.

The following code illustrates the user file related to the example of the double pendulum:

```plaintext
// Title of the application
title:="Simulation of a double pendulum":
nbrdof:= 2:
nbrbody:= 2:
// Gravity vector
gravity[1]:=0:
gravity[2]:=-9.81:
gravity[3]:=0:
// Eventual constants
l0:=1.2: l1:=1.1:
// Inertia characteristics
mass[0]:=1.1: mass[1]:=0.9:
Ixx[0]:=1: Ixx[1]:=1:
Iyy[0]:=1: Iyy[1]:=1:
Izz[0]:=10^2/12*mass[0]:
Izz[1]:=11^2/12*mass[1]:
// Definition of the position matrices
TG0 := Trotz(q[0]) * Tdisp(0,-l0/2,0):
TG[1] := Trotz(q[0]) * Tdisp(0,-l0,0) * Trotz(q[1]) * Tdisp(0,-l1/2,0):
// Initial conditions
qi[1]:=1:
// Simulation parameters
FinalTime:=5:
StepSave:=0.01:
StepMax:=0.005:
```

In this case, where the gravity is the only applied effort, no other task but compiling the resulting code is necessary. In some cases, the user will have to complete the procedure AddAppliedEfforts.

The CAGeM utility offers different options, activated by flags in the code describing the system. Let us mention the most significant ones.
PLOT: if this option is activated, a generic script file is generated to be used by gnuplot; the history of the configuration parameters and their first and second time derivatives is displayed on the screen and exported in postscript files;

ANIM: if activated, the generated C++ code comprises the statements needed to generate a basic animation during the integration process (a box is attached to each body);

LATEX_FR: allows the creation of a LaTeX report in French about the built application: inertia properties, expression of homogeneous transformation matrices, velocities, accelerations, some listings and the curves generated by the gnuplot script

LATEX_EN: same as LATEX_FR but the report is in English;

4 Illustrative examples

4.1 Slider-crank mechanism

The fact of working with generalized coordinates does not prevent to simulate closed-loop mechanisms, as shown by the slider-crank mechanism illustrated in figure 6. The parameters $\alpha$ and $x$ can indeed be expressed univocally in terms of the angle $q_0$ by

$$\alpha = \arcsin\left(\frac{l_1 \sin(q_0)}{l_2}\right) \quad x = l_1 \cos(q_0) + l_2 \cos(\alpha)$$

The relationships can be introduced symbolically in the MuPAD code describing the system as

```plaintext
11 := 1: 12 := 2:
alpha := arcsin(11*sin(q[0])/12):
x := 11*cos(q[0]) + 12*cos(alpha):
T0G[0] := Trotz(q[0]) * Tdisp(11/2,0,0):
T0G[1] := Tdisp(x,0,0) * Trotz(-alpha)
   * Tdisp(-12/2,0,0):
T0G[2] := Tdisp(x,0,0):
```

and the intermediary variables $\alpha$ and $x$ are automatically replaced by their expression and derived in a consistent way.

Such an approach leads however to a very long and inefficient code as, in the symbolic process, each occurrence of $\alpha$ or $x$ is replaced by its complete expression. That’s the reason why

\[\text{this option is particularly appreciated by the students}\]
the concept of *dependent variables* has been introduced in EasyDyn. The dependent variables, denoted \( p \), are intermediary variables, expressed in terms of the configuration parameters but they are calculated once for all at the beginning of ComputeMotion. The resulting code is shorter and also more efficient. For the slider crank mechanism, \( \alpha \) and \( x \) become \( p_0 \) and \( p_1 \). The MuPAD code describing the kinematics of the slider crank is given below.

\[
\begin{align*}
T_{G[0]} &= \text{Trotz}(q[0]) \times \text{Tdisp}(lc/2,0,0): \\
T_{G[1]} &= \text{Tdisp}(p[1],0,0) \times \text{Trotz}(-p[0]) \times \text{Tdisp}(-lr/2,0,0): \\
T_{G[2]} &= \text{Tdisp}(p[1],0,0): \\
&// \text{Dependent parameters} \\
p[0] &= \arcsin(lc \times \text{sin}(q[0])/lr): \\
p[1] &= lc \times \text{cos}(q[0]) + lr \times \text{cos}(p[0]):
\end{align*}
\]

Let us note that the dependent variables are defined after the position matrices. Otherwise, they would be replaced by their expression in the symbolic treatment of the position matrices.

For more complex systems, the dependent variables could eventually be determined in terms of the chosen configuration parameters from the symbolic or numeric resolution of constraint equations. This approach has been largely applied by Hiller et al. [5] on complex mechanical systems. At the time of writing this paper, such a feature has not been envisaged yet in EasyDyn.

![Figure 7: Image of the slider-crank mechanism](image)

### 4.2 Spatial robot

The robot illustrated in figure 8 is well documented as it was used as a benchmark for multi-body systems softwares [2]. It was studied with EasyDyn for the purpose of validation. We will present here only the principle of the model, the complete results being given in [8].

The robot owns 5 degrees of freedom, depicted in figure 8 and 3 bodies. In the script to be submitted to CAGeM, it makes no difference that the system is spatial, as shown in the following piece of code describing the kinematics

\[
\begin{align*}
T_{G[0]} &= \text{Trotz}(q[1]) \times \text{Tdisp}(0,0,q[0]): \\
T_{G[1]} &= \text{Trotz}(q[1]) \times \text{Tdisp}(0,0,q[0]) \times \text{Trotz}(q[3]) \times \text{Tdisp}(0,q[2],0): \\
T_{G[2]} &= \text{Trotz}(q[1]) \times \text{Tdisp}(0,0,q[0]) \times \text{Trotz}(q[3]) \times \text{Tdisp}(0,q[2]+L,0) \\
&\times \text{Trotz}(q[4]) \times \text{Tdisp}(0,C,0): \\
\end{align*}
\]

The force law at each actuator is defined in the benchmark for a point-to-point motion of the end point. These efforts must be programmed in the routine `AddAppliedEfforts` directly
in the C++ file. Once again, thanks to the available vector algebra, the spatial nature of the system doesn’t make this work dramatically more complex. Each effort (force or moment) is indeed defined as the product of the amplitude, given in the benchmark, and a unit vector in the right direction, obtained from the position matrices of the bodies. We get for example for the time interval $[0:0.5]$

```c
if (t<0.5)
{
    body[0].R += 6348 * body[0].T0G.R.uz();
    body[0].MG += (673*t-508) * body[0].T0G.R.uz();
    body[1].R += (36*t+986) * body[1].T0G.R.uy();
    body[0].R -= (36*t+986) * body[1].T0G.R.uy();
    body[2].MG+= 63.5 * body[1].T0G.R.ux();
    body[1].MG -= 63.5 * body[1].T0G.R.ux();
}
```

It is of interest to note that the user remains responsible of the coherence of the efforts applied on each body and in particular

- it is necessary to apply the action \textit{and the reaction};
- if a force isn’t applied directly on the center of gravity, the user must write himself the moment contribution.

The use of EasyDyn gives the opportunity to recall these fundamental principles, as they classically lead to student mistakes.

The 5 degrees of freedom robot is now a classical student exercise and requires less than 2 hours of work.

The robot reveals another shortcoming of the approach. It is easy to guess that the symbolic process doesn’t exploit the \textit{recursivity} along the kinematic chain. When the last body is treated, the complete product of the transformation matrices is derived without exploiting the fact that a part of the work has already been done. To avoid this waste of resources, EasyDyn was endowed with the ability to define the motion of a body as a relative motion with respect to another one. The code describing the motion of the robot becomes

```c
T0G[0] := Trotz(q[1]) * Tdisp(0,0,q[0]);
TrefG[1] := Troty(q[3]) * Tdisp(0,0,q[2],0);
```
BodyRef[1] := 0:
TrefG[2] := Tdisp(0,L,0) * Trotx(q[4]) * Tdisp(0,C,0):
BodyRef[2] := 1:

The relative motion (position, velocity and acceleration) is determined symbolically under MuPAD while the global motion is determined in the C++ code, by a call to the routine ComposeMotion, which composes the motion of the reference body and the relative motion. In the case of the robot, the resulting code is about 4 times faster than the original version.

5 More advanced examples

5.1 Dynamic analysis of a motorbike

For the purpose of validation again, EasyDyn was recently used to perform the stability analysis of a motorbike as described in [6]. Only the straight line case was considered. The model involves 15 degrees of freedom and 9 bodies and includes the refinements recognized as
necessary for a realistic representation of the motorbike dynamics. Some of these are: an elastic twist freedom of the front frame, a lateral stiffness of the tyres to account for the lag mechanism in the development of tyre forces, the aerodynamic effects and the roll motion of the rider’s upper body.

Figure 10 shows the root locus obtained by EasyDyn for a speed range from 10 to 90 m/s. The result is completely coherent with the work of Sharp [6]. For the purpose of illustration, figures 11 and 12 show the motion associated with the typical wobble and twist modes.

![Figure 11: Typical wobble mode](image1)

![Figure 12: Typical twist mode](image2)

5.2 Simulation of a walking robot

The walking robot AMRU5[10] illustrated in figure 13 is the most comprehensive example treated so far with the help of EasyDyn. It consists of 49 bodies (1 central body and 8 bodies per leg) and involves 24 configuration parameters (6 for the central body and 3 per leg) and 18 dependent parameters (3 per leg). All recent improvements of EasyDyn are used: relative motion, dependent parameters and the direct determination of the partial velocities from MuPAD. Moreover, the model includes the dynamics of the DC motors and their digital controllers and a contact with friction between the legs and the ground. The model is actually driven by the position signals sent to the leg controllers by the central unit. The purpose of the model is to help in tuning new types of controllers.

It is of interest to have a look at the kinematics of a leg, consisting of the pantograph mechanism illustrated in figure 14. The pantograph itself is attached to the central body through a motorized vertical hinge. The geometry of the pantograph mechanism is solved in terms of parameters $q_1$ and $q_2$ (actually driven by the motors), and begins by the determination of the 3 dependent variables $\alpha$, $\beta$ and $\gamma$, from the following relationships
The expression of the kinematics can be expressed naturally from \( q_1, q_2, \alpha, \beta, \) and \( \gamma \). We have for example for body 2

\[
T_{0G}[2] = T_{\text{disp}}(0, 0, q_1) * T_{\text{rot}}(\alpha - \gamma) * T_{\text{disp}}(AD, 0, 0) * T_{\text{rot}}(\gamma + \delta) * T_{\text{disp}}(DF/2, 0, 0)
\]

\( AC = \sqrt{q_1^2 + q_2^2} \) \hspace{1cm} (15)

\( \alpha = \arctan\left(\frac{q_1}{q_2}\right) \) \hspace{1cm} (16)

\( \delta = \arccos\left(\frac{AC^2 + BC^2 - AB^2}{2 \cdot AC \cdot BC}\right) \) \hspace{1cm} (17)

\( \gamma = \arccos\left(\frac{AB^2 + AC^2 - BC^2}{2 \cdot AB \cdot AC}\right) \) \hspace{1cm} (18)
For the purpose of illustration, figure 15 shows the evolution of the contact effort on the rearest leg during a straight walk in the Y direction. Of course, the contact effort is null when the leg is up. When the central body moves forward, it can be observed that the load is progressively transferred from the rear to the front legs. The irregularities in the curve show the discontinuous nature of walking.

![Figure 15: Contact efforts for leg 1](image)

6 Conclusion

We have presented a framework, called EasyDyn, that allows to easily build a C++ program performing the simulation of a dedicated multibody system. The system is described essentially from its kinematics and the expression of applied forces. Some utilities, like vector algebra classes and external symbolic tools, are provided to make the task of the user as easy as possible. EasyDyn is now used for 3 years in the workshops related to the course about computer-aided analysis of mechanical systems, in parallel with a commercial software. It offers some automatic tools which are concentrated on the mathematical and programming part. In this way, EasyDyn allows the students to think principally about the mechanical system but requires some knowledge of the laws of mechanics.

The proposed framework is not a replacement of available commercial simulation tools. In particular, the code was written for a maximum compacity, readability and scalability, often to the detriment of efficiency. The authors encourage anybody involved in education of multibody systems to have a look at the web site, at least for the suggested problems which are clearly defined in French and English, and come along with typical results. The authors are convinced that EasyDyn also constitutes a good foundation for the development of “proof of concept” applications.

REFERENCES


