Faculté Polytechnique de Mons
Service de Mécanique Rationnelle,
Dynamique et Vibrations

EasyDyn 1.2.4

C++ library for the easy simulation
of dynamic problems

Olivier Verlinden, Georges Kouroussis

July 24, 2008

Boulevard Dolez 31
7000 MONS
BELGIQUE
Tél. : 32.65.37.41.84
Fax. : 32.65.37.41.83
Email: Olivier.Verlinden@fpms.ac.be

Chapter 1

Getting and installing the library

1.1 Getting the library

The library can be downloaded in source form from the software part of the web site of the *Department of Theoretical Mechanics, Dynamics and Vibrations* of the *Faculté Polytechnique de Mons*, a university in Belgium

`http://mecara.fpms.ac.be`

either in tarball or zip format.

Once you have chosen a parent directory, decompact the archive with

```
unzip EasyDyn-x.y.z
```

or

```
tar xvzf EasyDyn-x.y.z.tgz
```

1.2 Getting a compiler

To compile `EasyDyn`, you will need a C++ compiler. I recommend `gcc` the compiler of the free software foundation, available for free from

`http://www.gnu.org/software/gcc`

Pay attention that you need a version of `gcc` above 3.0, with the new implementation of the I/O streams.

If you have not yet understood that you should drop Windows, it will probably be easier to download `mingw`, a complete and easy to install implementation of `gcc`, available from

`http://www.mingw.org`

Several integrated development environments (IDE) exist for `mingw`. I personnaly use `dev-cpp` available from

`http://www.bloodshed.net/dev/devcpp.html`
`http://sourceforge.net/projects/dev-cpp`

Have a look at the file `installFR.win` (unfortunately only in French so far) which describes the complete installation process under Windows.

However, as it is written in pure C++, it is very likely that `EasyDyn` can be compiled with any modern C++ compiler.

1.3 Getting the GNU scientific library

To compile `EasyDyn` you will need to compile and install the *GNU scientific Library* which can be obtained from `http://www.gnu.org/software/gsl`. From version 1.2.4 of `EasyDyn`, `LAPACK` is no longer necessary but you will need at least version 1.10 of the `GSL`.

Compiling and installing the `GSL` is trivial under Unix in general. Under Windows, you can do the job “in the unix way” under the `MSYS` environment available from the web site of `Mingw`. Otherwise, I provide a `Mingw` precompiled version of the `GSL` on the web site of `EasyDyn`. Other precompiled versions can generally be found quite rapidly by some search on the net.

1.4 Compiling and installing the library

To compile the `EasyDyn` library, a `makefile` is provided. It should work for any version of `gcc`. It has been tested under Linux and under Windows with `mingw`. Compiling the library is just a matter of entering

```
make
```

from the root directory of `EasyDyn`.

Please note that under `mingw`, `make` has been renamed `mingw32-make` to avoid confusion with any other implementation.

At the end you should have the library `libEasyDyn.a`. You may remove all the object files (`*.o`).

To install the library for general use, you just have to copy the subdirectory `include/EasyDyn` into the general `include` directory of `gcc`, like for example

- `c:`
  `mingw`
  `include` for `mingw`;
- `/usr/local/include` under Unix;

and `libEasyDyn.a` into the general `lib` directory of `gcc`, like for example

- `c:`
  `mingw`
  `lib` for `mingw`;
- `/usr/local/lib` under Unix;

Scripts are provided which perform this work automatically under Unix and Windows (for `mingw`).

Under Unix, the installation to `/usr/local` is performed by running

```
binlinux/EDinstall
```

from the root directory of `EasyDyn` (root privileges needed !). This operation also copies the script `EDbuild` (cf. later) to `/usr/local/bin`.

Under Windows, you can perform the installation by running

```
binwin\EDinstall C:\mingw
```

from the root directory of `EasyDyn` (administrator privileges needed !). Enventually, adapt `C:\mingw` to the actual directory of `mingw`. This operation also copies the script `EDbuild.bat` (cf. later) to the `bin` directory of `mingw`.

If you want to use another compiler, the library is just the collection of the objects obtained by compilation of all the `.cpp` files in the root directory of `EasyDyn`.

1.5 Testing the library and testing your applications

If you want to compile the examples, a `makefile` is provided for each of them. Either you have a file called `makefile` and you just type

```
make
```

or you have a file like `oscil1.mak` and you type

```
make -f oscil1.mak
```

If you want to compile a new application `applic.cpp` from a shell window, you will have to issue

- to compile the application

  ```
  c++ -c applic.cpp
  ```

  where `<EasyDynHome>` represents the root directory of `EasyDyn`; if successfull, the operation will result in the creation of a file called `applic.o`
- to link the application

```
c++ applic.o -lEasyDyn -lgsl -lgslcblas -o applic
```

which will result into the creation of an executable called `applic` (`applic.exe` under Windows).

You can eventually perform both steps together by

```
c++ applic.cpp -lEasyDyn -lgsl -lgslcblas -o applic
```

or, by using the provided `EDbuild` utility

```
EDbuild applic
```

Chapter 2

The `EasyDyn` vector library

2.1 Introduction

The `vec` part of the library defines 4 classes

- `vec` : defining a 3D vector;
- `trot` : defining a second-order 3D rotation tensor;
- `tiner` : defining a second-order 3D inertia tensor;
- `mth` : defining an homogenous transformation matrix.

and implements various operators between these classes. As seen later, the operators allow the user to program vector expressions as he would do by hand.

In the next sections

- variables a, b, and d represent real values;
- variables v, v_1, v_2 relate to vectors;
- variables R, R_1, R_2 relate to rotation tensors;
- variables Φ and Φ_G relate to inertia tensors;
- variables T, T_1, T_2 correspond to homogenous transformation matrices.

2.2 Some recalls about vector calculus

Vector calculus is a very powerful tool to develop the laws of mechanics. It is however important to be very clear about notations and conventions.

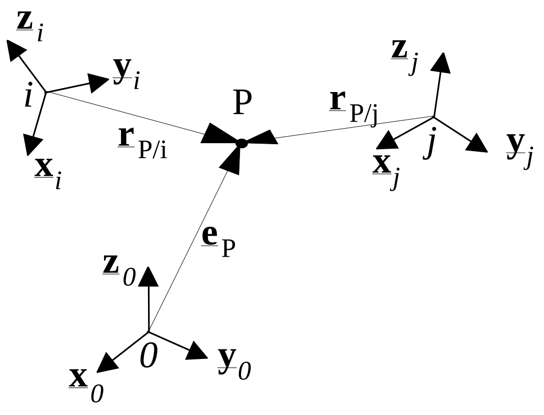

$\underline{\mathbf{z}}_i$
$\underline{\mathbf{y}}_i$
i
$\underline{\mathbf{x}}_i$
$\underline{\mathbf{r}}_{P/i}$
P
$\underline{\mathbf{r}}_{P/j}$
$\underline{\mathbf{z}}_j$
j
$\underline{\mathbf{x}}_j$
$\underline{\mathbf{y}}_j$
$\underline{\mathbf{e}}_P$
$\underline{\mathbf{z}}_0$
0
$\underline{\mathbf{x}}_0$
$\underline{\mathbf{y}}_0$

Figure 2.1: Points, vectors and frames

2.2.1 Points, vectors and frames

Let us denote vectors by underlined boldface characters, such as $\underline{\mathbf{a}}$, which represents a physical vector, characterized by a spatial direction and an amplitude.

For calculations, the vectors will have to be expressed with respect to a frame or coordinate system. We will assume here that each frame is cartesian (orthonormal) and composed of 3 dextrorsum axes X,Y and Z. The unit vectors corresponding to the axes of a frame i will be denoted by $\underline{\mathbf{x}}_i$, $\underline{\mathbf{y}}_i$ and $\underline{\mathbf{z}}_i$. The components a_{x_i}, a_{y_i} and a_{z_i} of a vector $\underline{\mathbf{a}}$ with respect to a frame i are such that vector $\underline{\mathbf{a}}$ can be reconstructed from its components by

$$\underline{\mathbf{a}} = a_{x_i} \cdot \underline{\mathbf{x}}_i + a_{y_i} \cdot \underline{\mathbf{y}}_i + a_{z_i} \cdot \underline{\mathbf{z}}_i \tag{2.1}$$

The set of components of a vector $\underline{\mathbf{a}}$, with respect to a frame i will be denoted by $\{\underline{\mathbf{a}}\}_i$, which represents a 3x1 matrix such as

$$\{\underline{\mathbf{a}}\}_i = \begin{pmatrix} a_{x_i} \\ a_{y_i} \\ a_{z_i} \end{pmatrix} \tag{2.2}$$

The position of any point P can be described by its position with respect to a chosen frame. We will denote $\underline{\mathbf{r}}_{P/i}$ the coordinate vector of point P with respect to frame i, that's to say the vector running from frame i to point P. The coordinates of point P in the coordinate system of frame i correspond to the 3 components of vector $\underline{\mathbf{r}}_{P/i}$, with respect to frame i, gathered in the column matrix $\{\underline{\mathbf{r}}_{P/i}\}_i$.

Generally, a global reference exists and is referenced as frame 0. The coordinate vector of a point P with respect to the global reference frame, that's to say $\underline{\mathbf{r}}_{P/0}$ can be denoted for the sake of simplicity by $\underline{\mathbf{e}}_P$.

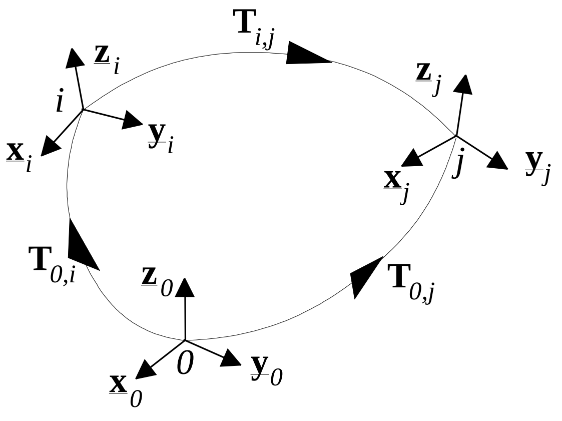
2.2.2 Homogeneous transformation matrices

The homogenous transformation matrix is a formalism which allows to represent the situation of a frame with respect to another one. According to this formalism, the relative situation of frame j with respect to frame i is expressed from the homogeneous transformation matrix $\mathbf{T}_{i,j}$, of dimension 4x4, which can be partitioned in the following way:

$$\mathbf{T}_{i,j} = \begin{pmatrix} \mathbf{R}_{i,j} & \{\underline{\mathbf{r}}_{j/i}\}_i \\ 0\ 0\ 0 & 1 \end{pmatrix} \tag{2.3}$$

where

- $\underline{\mathbf{r}}_{j/i}$ is the coordinate vector of frame j with respect to frame i;
- $\mathbf{R}_{i,j}$ is the rotation tensor describing the orientation of frame j with respect to frame i.



$\mathbf{T}_{i,j}$
$\underline{\mathbf{z}}_i$
i
$\underline{\mathbf{y}}_i$
$\underline{\mathbf{x}}_i$
$\underline{\mathbf{z}}_j$
j
$\underline{\mathbf{y}}_j$
$\underline{\mathbf{x}}_j$
$\mathbf{T}_{0,i}$
$\underline{\mathbf{z}}_0$
$\mathbf{T}_{0,j}$
0
$\underline{\mathbf{y}}_0$
$\underline{\mathbf{x}}_0$

Figure 2.2: Homogeneous transformation matrices

Let us recall that the columns of $\mathbf{R}_{i,j}$ correspond to the unit vectors $\underline{\mathbf{x}}_j$, $\underline{\mathbf{y}}_j$ and $\underline{\mathbf{z}}_j$, expressed in the axes of frame i

$$\mathbf{R}_{i,j} = \left(\{\underline{\mathbf{x}}_j\}_i \ \{\underline{\mathbf{y}}_j\}_i \ \{\underline{\mathbf{z}}_j\}_i \right) \tag{2.4}$$

and must of course hold the following constraints:

$$|\underline{\mathbf{x}}_j| = 1 \qquad |\underline{\mathbf{x}}_j| = 1 \qquad |\underline{\mathbf{x}}_j| = 1 \tag{2.5}$$

$$\underline{\mathbf{z}}_j = \underline{\mathbf{x}}_j \times \underline{\mathbf{y}}_j \qquad \underline{\mathbf{y}}_j = \underline{\mathbf{z}}_j \times \underline{\mathbf{x}}_j \qquad \underline{\mathbf{x}}_j = \underline{\mathbf{y}}_j \times \underline{\mathbf{z}}_j \tag{2.6}$$

The rotation tensor can be used on its own to determine the coordinates of a vector in a frame when they are given in an another one:

$$\{\underline{\mathbf{a}}\}_i = \mathbf{R}_{i,j} \cdot \{\underline{\mathbf{a}}\}_j \tag{2.7}$$

The homogeneous transformation matrices are not only used to describe the relative situation of two frames but also permit to easily express the coordinates of a point with respect to a frame from the ones with respect to another frame. The following relationship can indeed be easily demonstrated

$$\begin{pmatrix} \{\underline{\mathbf{r}}_{P/i}\}_i \\ 1 \end{pmatrix} = \mathbf{T}_{i,j} \cdot \begin{pmatrix} \{\underline{\mathbf{r}}_{P/j}\}_j \\ 1 \end{pmatrix} \tag{2.8}$$

which is equivalent to

$$\{\underline{\mathbf{r}}_{P/i}\}_i = \{\underline{\mathbf{r}}_{j/i}\}_i + \mathbf{R}_{i,j} \cdot \{\underline{\mathbf{r}}_{P/j}\}_j \tag{2.9}$$

For the sake of simplicity, we will introduce the $\circ$ operator such that

$$\{\underline{\mathbf{r}}_{P/i}\}_i = \mathbf{T}_{i,j} \circ \{\underline{\mathbf{r}}_{P/j}\}_j = \{\underline{\mathbf{r}}_{j/i}\}_i + \mathbf{R}_{i,j} \cdot \{\underline{\mathbf{r}}_{P/j}\}_j \tag{2.10}$$

To conclude this section, let us mention that the homogeneous transformation matrices have the advantage to enjoy the following property

$$\mathbf{T}_{i,k} = \mathbf{T}_{i,j} \cdot \mathbf{T}_{j,k} \tag{2.11}$$

This property is often used to obtain a complex transformation matrix from the successive multiplication of simpler matrices.

2.3 The 3D vector class `vec`

2.3.1 Variables of the class

The `vec` class consists of 3 public variables of type double: `x`, `y` and `z`, representing the components of the vector in a orthonormal, dextrosum coordinate system.

2.3.2 Assignment methods

By default, all components are set to zero by the constructor. After declaration

```
vec v;
```

vector `v` is equal to zero (`v.x`=`v.y`=`v.z`=0).

It is however allowed to assign the vector a given value in the declaration

```
vec v(1,2,3);
```

There are different ways to change the components of a vector

- by changing directly the coordinates (the variables are public)

  ```
  v.x=1; v.y=2; v.z=3;
  ```

- by using `put()`

  ```
  v.put(1,2,3);
  ```

To set all components to zero, the `zero()` method can be used

```
v.zero();
```

A vector can also be normalized with the method `unite()`

```
v.unite();
```

As a result, vector $\underline{\mathbf{v}}$ becomes a unitary vector with the same direction ($\underline{\mathbf{v}} = \underline{\mathbf{v}}/|\underline{\mathbf{v}}|$).

2.3.3 Utility methods

The method `length()` gives the length of a vector

```
double L=v.length();
```

2.3.4 Calculus operators

The classical operators for addition, substraction, dot product and cross product of vectors have been overloaded. Multiplication or division by a double have been overloaded as well. The use of operators is natural and self-explanatory. Table 2.1 illustrates the operators and their physical meaning. The left column gives the mathematical operation and the right one the way it is written in C++.

$\underline{\mathbf{v}} = \underline{\mathbf{v}}_1 + \underline{\mathbf{v}}_2$	`v=v1+v2;`
$\underline{\mathbf{v}} = \underline{\mathbf{v}} + \underline{\mathbf{v}}_1$	`v+=v1;`
$\underline{\mathbf{v}} = -\underline{\mathbf{v}}_1$	`v=-v1;`
$\underline{\mathbf{v}} = \underline{\mathbf{v}}_1 - \underline{\mathbf{v}}_2$	`v=v1-v2;`
$\underline{\mathbf{v}} = \underline{\mathbf{v}} - \underline{\mathbf{v}}_1$	`v-=v1;`
$d = \underline{\mathbf{v}}_1 \cdot \underline{\mathbf{v}}_2$	`d=v1*v2;`
$\underline{\mathbf{v}} = \underline{\mathbf{v}}_1 \times \underline{\mathbf{v}}_2$	`v=v1^v2;`
$\underline{\mathbf{v}} = \underline{\mathbf{v}}_1 \cdot d$	`v=v1*d;`
$\underline{\mathbf{v}} = d \cdot \underline{\mathbf{v}}_1$	`v=d*v1;`
$\underline{\mathbf{v}} = \underline{\mathbf{v}} \cdot d$	`v*=d;`
$\underline{\mathbf{v}} = \underline{\mathbf{v}}_1/d$	`v=v1/d;`
$\underline{\mathbf{v}} = \underline{\mathbf{v}}/d$	`v/=d;`

Table 2.1: Vector operators

All operators between vectors assume that their coordinates are expressed **in the same coordinate system**. If several coordinate systems are involved, the user must manage himself the necessary transformations. The operation is made easy with the other classes of the library: the rotation tensor and the homogenous transformation matrix.

Important remark

As the operator * was already used for the dot product, the operator ^ has been chosen, by reference to the classical scale (Λ) also used to represent the cross-product. Unfortunately, it doesn't have the priority of a classical multiplication operator as it is evaluated after + and -. Consequently the following statement

```
v1+v2^v3;
```

will be evaluated as $(\underline{\mathbf{v}}_1 + \underline{\mathbf{v}}_2) \times \underline{\mathbf{v}}_3$.

The statement to evaluate $\underline{\mathbf{v}}_1 + \underline{\mathbf{v}}_2 \times \underline{\mathbf{v}}_3$ in the way it is generally understood is then

```
v1+(v2^v3);
```

To make it short, brackets are welcome to avoid misunderstanding between you and the compiler in long statements with several operators.

2.3.5 Input/output

The input and output operators have been overloaded and can be used naturally with vectors. Here is an example

```
vec v(1,2,3);
cout << ``Vector v: `` << v << ``\n'';
vec v2;
cin >> v2;
```

On output, the three coordinates of the vector are sent to the stream, separated by some space but with no end-of-line. On input, three real values are expected.

Here again, the priority of operator ^ has surprising consequences as it is evaluated after << and >>. For example, the following code won't be understood by the compiler

```
cout << v1^v2 << ``\n'';
```

and must be written

```
cout << (v1^v2) << ``\n'';
```

2.4 The rotation tensor class `trot`

2.4.1 Variables of the class

The `trot` class consists of 9 public variables of type double: `r11`, `r12`, `r13`, `r21`, `r22`, `r23`, `r31`, `r32` and `r33`, representing the components of the rotation tensor in an orthonormal and positively oriented coordinate system.

The terms of the rotational tensor must verify the following constraints, expressing that coordinate systems after transformation must remain orthonormal and positively oriented

$$r_{11}^2 + r_{21}^2 + r_{31}^2 = 1 \tag{2.12}$$
$$r_{12}^2 + r_{22}^2 + r_{32}^2 = 1 \tag{2.13}$$
$$r_{13}^2 + r_{23}^2 + r_{33}^2 = 1 \tag{2.14}$$
$$r_{11}r_{12} + r_{21}r_{22} + r_{31}r_{32} = 0 \tag{2.15}$$
$$r_{11}r_{13} + r_{21}r_{23} + r_{31}r_{33} = 0 \tag{2.16}$$
$$r_{12}r_{13} + r_{22}r_{23} + r_{32}r_{33} = 0 \tag{2.17}$$

It is the responsability of the programmer to hold these constraints if the components of a rotation tensor are assigned directly. The library doesn't automatically verify them.

2.4.2 Assignment methods

By default, the rotation tensor corresponds to an identity matrix: all diagonal terms are equal to 1 while other ones are null. After declaration

```
trot R;
```

tensor `R` is such that `R.r11`=`R.r22`=`R.r33`=1 and all other terms are equal to zero.

The terms of a tensor are public and can be changed directly. For example, a rotation about the Z axis could be written

```
R.r11=cos(theta); R.r12=-sin(theta); R.r13=0;
R.r21=sin(theta); R.r22=cos(theta);  R.r23=0;
R.r31=0;          R.r32=0;           R.r33=1;
```

A rotation tensor can be defined back as identity with the `unite()` method

```
R.unite();
```

2.4.3 Utility methods

The methods `rotx(th)`, `roty(th)` and `rotz(th)` allow to directly define a rotation tensor as a rotation of angle `th` about the X, Y or Z axis.

```
trot R,R1,R2;
R.rotx(0.1); R1.roty(0.2); R2.rotz(0.3);
```

The equivalent functions `Rrotx`, `Rroty` and `Rrotz` exist and return a rotation tensor. The advantage is that a rotation tensor can be defined direcly as a combination of several rotations.

```
trot R;
R=Rrotx(0.1)*Rroty(0.2)*Rrotz(0.3);
```

The inverse rotation tensor is given by the method `inv()`

```
trot R,Rinv;
Rinv=R.inv();
```

2.4.4 Calculus operators

As already seen in previous examples, the rotation tensors can be multiplied by each other. A rotation tensor can also multiply a vector, implementing a change of coordinate system. The use of operators is natural and self-explanatory. Table 2.2 illustrates the operators and their physical meaning. The left column gives the mathematical operation and the right one the way it is written in C++.

$\underline{\mathbf{R}} = \underline{\mathbf{R}}_1 \cdot \underline{\mathbf{R}}_2$	`R=R1*R2;`
$\underline{\mathbf{R}} = \underline{\mathbf{R}} \cdot \underline{\mathbf{R}}_1$	`R*=R1;`
$\underline{\mathbf{v}} = \underline{\mathbf{R}} \cdot \underline{\mathbf{v}}_1$	`v=R*v1;`

Table 2.2: Rotation tensor operators

2.4.5 Unit vectors related to the rotation tensor

Each column of the rotation tensor corresponds to a unit vector. They are given respectively by the methods `ux()`, `uy()` and `uz()`, returning a data type `vec`. If the tensor is applied to a coordinate system, each vector gives the unit vectors of the new coordinate system.

```
trot R=Rroty(0.4);
vec v=R.ux();
```

The user can also set directly each of the unit vectors. The corresponding methods are `setux(vec v)`, `setuy(vec v)` and `setuz(vec v)`.

```
trot R;
vec v1(1,2,0),v2(-2,1,0);
v1.unite();
v2.unite();
R.setux(v1);
R.setuy(v2);
```

It is the responsability of the user to insure that the rotation matrix remains orthonormal.

2.4.6 Input/output

The input and output operators have been overloaded and can be used naturally with rotation tensors. Here is an example

```
trot R=Rrotz(1);
cout << ``Tensor R: \n`` << R;
trot R2;
cin >> R2;
```

On output, the 9 coordinates of the rotation tensor are sent to the stream, line by line (3 terms by line, separated by some space). A newline is output at the end of each line. On input, 9 real values are expected.

2.5 The inertia tensor class `tiner`

2.5.1 Variables of the class

The `tiner` class consists of 6 public variables of type double: `Ixx`, `Iyy`, `Izz`, `Ixy`, `Ixz` and `Iyz` representing the components of the inertia tensor of a body in an orthonormal and positively oriented coordinate system. In a given coordinate system, the inertia tensor can be seen as the following 3x3 matrix

$$\begin{pmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{xy} & I_{yy} & -I_{yz} \\ -I_{xz} & -I_{yz} & I_{zz} \end{pmatrix} \tag{2.18}$$

The terms on the diagonal are called moments of inertia while the others are called products of inertia.

The matrix representing the inertia tensor is constant when expressed in a coordinate system linked to the body. Otherwise, it depends on the orientation of the body.

2.5.2 Assignment methods

By default, all terms of the inertia tensor are null. At declaration

```
tiner Phi;
```

all variables (`Ixx`, `Iyy`, `Izz`, `Ixy`, `Ixz`, `Iyz`) related to tensor `Phi` are set to zero.

An inertia tensor can be assigned directly at declaration or with the `put` method

```
tiner Phi(1.1,1.2,1.3,0.1,0.2,0.3);
tiner Phi2;
Phi2.put(1.1,1.2,1.3,0.1,0.2,0.3);
```

the order of the parameters being Ixx, Iyy, Izz, Ixy, Ixz and Iyz.

Only the first 3 terms can be specified

```
tiner Phi(1.1,1.2,1.3);
tiner Phi2;
Phi2.put(1.1,1.2,1.3);
```

and the inertia products are then set to zero.

The terms of an inertia tensor are public and can also be assigned directly

```
Phi.Ixx=1.1; Phi.Iyy=1.2; Phi.Izz=1.3;
Phi.Ixy=0.1; Phi.Ixz=0.2; Phi.Iyz=0.3;
```

2.5.3 Calculus operators

The destiny of a tensor is to be applied to a vector to generate another vector. For example, the momentum of a body about its center of gravity G is calculated by

$$\underline{\mathbf{L}}_G = \mathbf{\Phi}_G \cdot \underline{\omega} \tag{2.19}$$

with $\mathbf{\Phi}_G$ the inertia tensor with respect to the center of gravity.

Only one operator is then defined: the multiplication by a vector (cf. table 2.3), corresponding to the following matrix product

$$\begin{pmatrix} v_{2x} \\ v_{2y} \\ v_{2z} \end{pmatrix} = \begin{pmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{xy} & I_{yy} & -I_{yz} \\ -I_{xz} & -I_{yz} & I_{zz} \end{pmatrix} \cdot \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} \tag{2.20}$$

It is of interest to note that the expression is meaningful **only if the components of the inertia tensor and the vector are expressed in the same coordinate system**.

$\underline{\mathbf{v}}_2 = \underline{\mathbf{\Phi}} \cdot \underline{\mathbf{v}}$	v2=Phi*v;

Table 2.3: Inertia tensor operators

2.5.4 Input/output

The input and output operators have been overloaded and can be used naturally with inertia tensors. Here is an example

```
tiner Phi(1.1,1.2,1.3);
cout << ‘‘Tensor Phi: ‘‘ << Phi << ‘‘\n’’;
tiner Phi2;
cin >> Phi2;
```

On output, the 6 parameters of the rotation tensor are sent to the stream, in the order `Ixx`, `Iyy`, `Izz`, `Ixy`, `Ixz` and `Iyz`, separated by some space. No newline is sent to the stream. On input, 6 real values are expected, the order of the parameters being the same as in output.

2.6 The homogeneous transformation matrix class `mth`

2.6.1 Variables of the class

The `mth` class consists of 2 public variables: a rotation tensor `R` and a vector `e`.

2.6.2 Assignment methods

At declaration, the vector and the rotation tensor get the default initial value of their class, that's to say the null vector and the identity tensor.

The terms of `R` and `e` are public and can be changed directly

```
mth T;
T.R.r11=cos(theta);
T.e.x=2;
```

An homogenous transformation matrix can be reinitialized with the `unite()` method

```
T.unite();
```

2.6.3 Utility methods

The methods `rotx(th)`, `roty(th)` and `rotz(th)` of the rotation tensor allow to directly define a transformation matrix corresponding to a rotation of angle `th` about the X, Y or Z axis.

```
mth T,T1,T2;
T.R.rotx(0.1); T1.R.roty(0.2); T2.R.rotz(0.3);
```

The equivalent functions `Trotx(th)`, `Troty(th)` and `Trotz(th)` exist and return an homogeneous transformation matrix

```
mth T;
T=Trotx(0.1)*Troty(0.2)*Trotz(0.3);
```

A transformation matrix corresponding to a displacement can be defined from a vector

```
vec v(1,2,3);
mth T;
T.e=v;
```

The functions `Tdisp(vec v)` and `Tdisp(double x, double y, double z)` return an `mth` and can also be used directly to define a matrix as a succession of elementary motions. For example a frame at the end of a pendulum can be defined by

```
vec v(0,-0.5,0);
mth T;
T=Trotz(th)*Tdisp(v); // or equivalently T=Trotz(th)*Tdisp(0,-0.5,0);
```

The inverse transformation matrix is given by the method `inv()`

```
mth T,Tinv;
Tinv=T.inv();
```

2.6.4 Calculus operators

The operators are similar to the ones of the rotation tensor and are self-explanatory. Table 2.4 illustrates the operators and their physical meaning. The left column gives the mathematical operation and the right one the way it is written in C++.

$\underline{\mathbf{T}} = \underline{\mathbf{T}}_1 \cdot \underline{\mathbf{T}}_2$	`T=T1*T2;`
$\underline{\mathbf{T}} = \underline{\mathbf{T}} \cdot \underline{\mathbf{T}}_1$	`T*=T1;`
$\underline{\mathbf{v}} = \underline{\mathbf{T}} \circ \underline{\mathbf{v}}_1$	`v=T*v1;`

Table 2.4: Homogeneous transformation matrices operators

2.6.5 Unit vectors related to the rotation tensor

The unit vectors can of course be retrieved directly from the ones of the rotation tensor

```
mth T=Troty(0.4);
vec v=T.R.ux();
```

2.6.6 Input/output

The input and output operators have been overloaded and can be used naturally with transformation matrices. Here is an example

```
mth T=Trotz(1);
cout << ‘‘Homogeneous transformation matrix T: \n‘‘ << T;
mth T2;
cin >> T2;
```

On output, the 12 coordinates of the rotation tensor are sent to the stream, line by line (4 terms by line, separated by some space). A newline is output at the end of each line. On input, 12 real values are expected.

2.7 Decomposition routines

2.7.1 Rotation decomposition

In order to decompose a rotation as 3 successive rotations about given axes, the library yields the following procedures

```
void getrotxyz(trot R1, trot R2, double & th1, double & th2, double & th3,
                            vec &axe1, vec &axe2, vec &axe3);
void getrotyzx(trot R1, trot R2, double & th1, double & th2, double & th3,
                            vec &axe1, vec &axe2, vec &axe3);
void getrotzxy(trot R1, trot R2, double & th1, double & th2, double & th3,
                            vec &axe1, vec &axe2, vec &axe3);
void getrotyxz(trot R1, trot R2, double & th1, double & th2, double & th3,
                            vec &axe1, vec &axe2, vec &axe3);
void getrotxzy(trot R1, trot R2, double & th1, double & th2, double & th3,
                            vec &axe1, vec &axe2, vec &axe3);
void getrotzyx(trot R1, trot R2, double & th1, double & th2, double & th3,
                            vec &axe1, vec &axe2, vec &axe3);
```

For example, `getrotxyz` determines the 3 successive rotations about local axes X, Y and Z to go from orientation `R1` to orientation `R2`. On return, variables `th1`, `th2` and `th3` are the rotation angles while `axe1`, `axe2` and `axe3` are unit vectors corresponding to the successive axes of rotation. Said in other words, after a call to `getrotxyz`, `R2` is such that

```
R2=R1*Rrotx(th1)*Rroty(th2)*Rrotz(th3);
```

Moreover, the rotation axes in this example are such that `axe1=R1.ux` and `axe3=R2.uz`.

The other procedures are equivalent but correspond to different orders in the axes of rotation.

2.7.2 Vector decomposition

The routine `decomposevec` allows to rebuild a vector as a linear combination of 3 given base axes and is defined as

```
int decomposevec(vec a,vec axe1, vec axe2, vec axe3,
                 double &a1,double &a2,double &a3);
```

On return, the components `a1`, `a2` and `a3` are such that

```
a=a1*axe1+a2*axe2+a3*axe3;
```

The value returned by the procedure should be equal to zero. Otherwise, the base vectors are not independent and the decomposition is not possible.

2.8 Example

The source file `testvec.cpp` under `examples`
`testvec` will allow to easily test all the procedures defined in the vector library. It also illustrates how to use the library.

If you dispose of gcc, a makefile is provided and you will get the excutable by typing

```
make
```

The comments given by the program are self-explanatory.

Chapter 3

The EasyDyn visualization library

3.1 Introduction

The `visu` part of the library allows to easily create `.vol` and `.van` files used by `EasyAnim` to visualize and animate a scene composed of moving objects. The `visu` module largely uses the vector part of the library.

`EasyDyn/visu` relies on two principal classes

- `shape` : a shape attached to a moving object;
- `scene` : a set of shapes.

To use this module, you will of course have to include the corresponding header in your source file with

```
#include <EasyDyn/visu.h>
```

3.2 The shapes

3.2.1 Structure

The abstract class `shape` is defined in the following way

```
class shape
  {
  protected:
  int nbrmth,*nbrnodes, nbredges, nbrsurf;
  int *edgecolor, *surfcolor;
  int *edgenode1, *edgenode2;
  int *nbrnodesurf, **numnodesurf;
  int security;
  public:
  mth **mthref;
  vec **nodecoord;
  shape(){};
  shape(int n_mth,mth **mt,int *n_pt_per_mth,vec **pt,int n_edges,
        int *enode1, int *enode2, int *ecolor,
        int n_surf,int *n_nodesurf,int **numnodesurf,
        int *sclor,int protection=1);
  shape(mth **mt_per_pt,int n_pt,vec **pt,int n_edges,
        int *enode1,int *enode2, int *eclr,
        int n_surf,int *n_nodesurf,int **num_nodesurf,
        int *sclr,int protection=1);
  shape(mth **mt_per_pt,int n_pt,vec **pt,int protection=1);
  shape(mth **mt_per_pt, istream &stream);
  shape(int n_mth,mth **mt,int *n_pt_per_mth,vec **pt,int protection=1);
  shape(int n_mth, mth **mt, istream &stream);
  ~shape();
  shape *nextshape;
  int GetNbrNodes();
  int GetNbrEdges();
  int GetNbrSurf();
  virtual void WriteNodes(ostream &stream, int initnode);
  virtual void WriteNodes(mth *mthvisu,ostream &stream, int initnode,
                          int norot=0);
  void WriteEdges(ostream &stream, int initnode, int initedge);
  void WriteSurf(ostream &stream, int initnode, int initsurf);
  virtual void WriteCoord(ostream &stream);
  virtual void WriteCoord(mth *mthvisu, ostream &stream, int norot=0);
  virtual char* GetShapeType() { return "SHAPE"; };
  virtual void WriteShape(ostream &stream);
  };
```

Basically, a shape consists of a set of nodes and a set of edges and surfaces defined between nodes.

The nodes are actually grouped according to the coordinate system they are attached to. The variable `nbrmth` gives the number of involved coordinate systems. The situtation of the latter is described by homogeneous transformation matrices, pointed to by the array of matrix pointers `mthref`. The evolution of these matrices is managed by the user, allowing to move the shapes and to create the animation. Each coordinate system i (starting from 0) owns a number of nodes equal to `nbrnodes[i]` the local coordinates being stored in the vector `nodecoord[i]`. The total number of nodes can be accessed by the method `GetNbrNodes()`. The arrays `nbrnodes` and `nodecoord` must be allocated by the user.

The number of edges is given by variable `nbredges` which can be accessed through the method `GetNbrEdges()`. Each edge connects two nodes, whose index is stored in integer arrays `edgenode1` and `edgenode2`, which must be allocated by the user. The color of each edge is stored in integer array `edgecolor` which must be allocated by the user. So far, only color numbers from 0 to 15 can be defined, corresponding for historical reasons to the ones defined in TurboPascal. The color codes are recalled in table 3.1. Note that the numbering of the nodes starts from the first node of the first coordinate system (number 0) up to the last node of the last coordinate system.

Code	Color	Code	Color
0	Black	8	Dark gray
1	Blue	9	Light blue
2	Green	10	Light green
3	Cyan	11	Light cyan
4	Red	12	Light red
5	Magenta	13	Light magenta
6	Brown	14	Yellow
7	Light gray	15	White

Table 3.1: Color codes in `EasyDyn/visu`

The number of surfaces is given by variable `nbrsurf` which can be accessed through the method `GetNbrSurf()`. Each surface is assumed to be polygonal and connects any number of nodes, specified in integer array `nbrnodesurf` (which must be allocated by the user). The indices of the nodes are stored in the array of integer arrays `numnodesurf`, which must be allocated by the user. The color of the surfaces is specified in the integer array `surfcolor`, according to the codes defined in table 3.1. The array `surfcolor` is also allocated by the user.

The functions `WriteNodes()`, `WriteEdges()`, `WriteSurf()` and `WriteCoord()` are used only by the scenes and are of little interest for the user. In the same way, the pointer `nextshape` is used only for the scene where the shapes are stored as a pointed list.

The function `WriteShape()` writes the information related to a shape to the screen or to a file. It is not strictly necessary and can be left empty but can be used for example to save the structure of a scene.

Several constructors exist for `shape`. The first one is the most general. If flag `protection` is not null (default), all arrays are copied locally. Otherwise, the data are accessed only through their address and the user must assure the consistence of the data.

The other constructors are dedicated to particular cases where for example only one node is attached to each coordinate system. Have a look at the code if necessary.

3.2.2 Predefined shapes

To make the use of the library easier, several predefined shapes are provided, corresponding to classes derived from base class `shape`. In a general way, the user will declare a pointer to base class `shape`, and allocate it as a derived class. Constructors are provided to correctly initialize the predefined shapes. There exist 7 predefined shapes: the line, the triangle, the box, the spline (class grspline), the frustum, the tyre and a last one (habfile) where the structure of the shape is defined in an external text file.

Remark: for each predefined shape, a general constructor is defined to read the information related to the shape from a stream (the most often, it will be a file). This information is assumed to be the one saved previously by the method `WriteShape()`. The constructor has the same form for each predefined shape and gives for example, for the line

```
line(mth *mt, istream &stream);
```

with `stream` the input stream from which the information is read. The pointer `mt` has the same meaning as in the other constructor form (see later) and relates to the coordinate system in which the coordinates of the nodes of the shape are expressed.

The line

The class `line` defines a straight line between two points. Besides the stream version, the following constructor is provided

```
line(mth *mt, vec pt1, vec pt2, int eclr);
```

with

- `mt` the pointer to the homogeneous transformation matrix describing the position of the local coordinate system of the shape;
- `pt1` and `pt2` the coordinate vectors of the end points of the line with respect to the local coordinate system;
- `eclr` the color code of the line.

Here is an example

```
shape *s1;
vec e1(0,0,0), e2(1,0,0);
mth T=Trotz(0.5);
s1=new line(&T,e1,e2,3);
```

The triangle

The class `triangle` defines a triangle by its three vertices. Besides the stream version, the following constructor is provided

```
triangle(mth *mt, vec c1, vec c2, vec c3, int eclr, int sclr);
```

with

- `mt` the pointer to the homogeneous transformation matrix describing the position of the local coordinate system of the shape;

- `c1`, `c2` and `c3` the coordinate vectors of the vertices with respect to the local coordinate system;
- `eclr` the color code of the edges;
- `sclr` the color code of the surface of the triangle.

Here is an example

```
shape *s1;
vec e1(0,0,0), e2(1,0,0), e3(0,1,0);
mth T=Troty(0.5);
s1=new triangle(&T,e1,e2,e3,3,4);
```

The class `triangle2` is a variant where the nodes of the triangle can be attached to different coordinate systems, as shown by the constructor

```
triangle(mth *mt1, vec c1, mth *mt2, vec c2, mth *mt3, vec c3,
         int eclr, int sclr);
```

The box

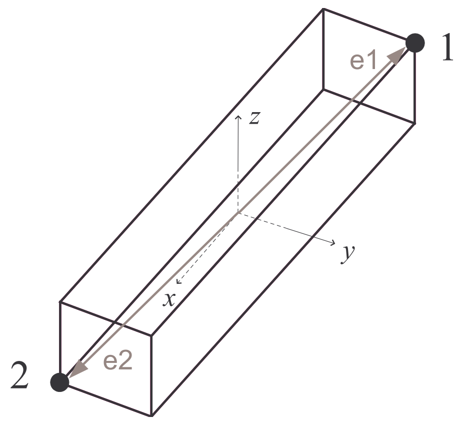


Figure 3.1: Parameters of the box shape

The class `box` defines a rectangular parallelepiped from two extreme vertices (cf. figure 3.1). The edges of the box are parallel to the X, Y and Z axes of the local coordinate system. Besides the stream version, the following constructor is provided

```
  box(mth *mt, vec c1, vec c2, int eclr, int sclr);
```

with

- `mt` the pointer to the homogeneous transformation matrix describing the position of the local coordinate system of the shape;

- `c1` and `c2` the coordinate vectors of the two extreme vertices with respect to the local coordinate system;
- `eclr` the color code of the edges;
- `sclr` the color code of the surfaces of the box.

Here is an example

```
shape *s1;
vec e1(-1,-1,-1), e2(1,1,1);
mth T=Tdisp(0.5,0,0);
s1=new box(&T,e1,e2,3,4);
```

The spline (class grspline)

The class `grspline` defines a spline from some points and the tangent vector of the curve at these points. Besides the stream version, the following constructor is provided

```
  grspline(mth *mt, vec *p, vec *t, int nt, int ns, int eclr);
```

with

- `mt` the pointer to the homogeneous transformation matrix describing the position of the local coordinate system of the shape;
- `p` the coordinate vectors of the successive points describing the spline, with respect to the local coordinate system;
- `t` the tangent vectors at the successive points with respect to the local coordinate system; pay attention that the amplitude of the tangent vector is relevant and modifies the layout of the curve;
- `nt` the number of sectors of the spline (equal to the number of points minus 1);
- `ns` the number of straight segments used to represent a sector of the spline;
- `eclr` the color code of the edges.

Here is an example

```
shape *s1;
vec e[2], t[2];
// We should get approximately a quarter of a circle
e[0].put(1,0,0); e[1].put(0,1,0);
t[0].put(0,1.567,0); t[1].put(-1.567,0,0);
mth T=Tdisp(0.5,0,0);
s1=new grspline(&T,e,t,1,10,3);
```

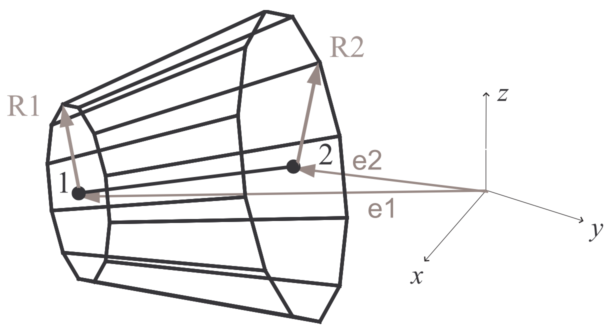


Figure 3.2: Parameters of the frustum shape

The frustum

The class `frustum` defines a frustum from the two extreme points of its axis and the corresponding radiuses (cf. figure 3.2). Besides the stream version, the following constructor is provided

```
frustum(mth *mt, vec e1, vec e2, double r1, double r2, int nbs,
        int eclr, int sclr);
```

with

- `mt` the pointer to the homogeneous transformation matrix describing the position of the local coordinate system of the shape;
- `e1` and `e2` the coordinate vectors of the two extreme axis points with respect to the local coordinate system;
- `r1` the radius at the first point;
- `r2` the radius at the second point;
- `nbs` the number of sectors used to represent the frustum;
- `eclr` the color code of the edges;
- `sclr` the color code of the surfaces of the frustum.

Here is an example

```
shape *s1;
vec e1(-1,0,0), e2(1,0,0);
mth T=Tdisp(0.5,0,0);
s1=new frustum(&T,e1,e2,1.5,1.2,24,3,4);
```

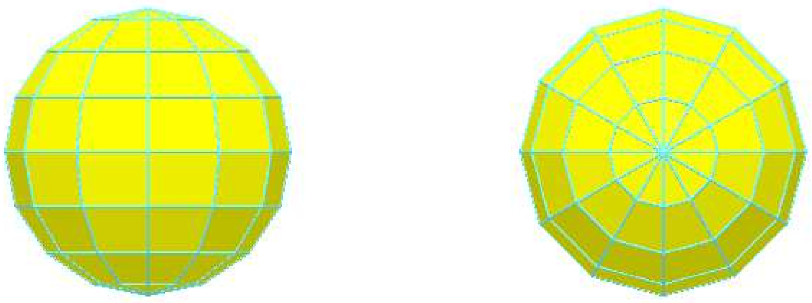


Figure 3.3: Image of a sphere composed of 7 slices and 12 sectors

The sphere

The class `sphere` defines a sphere from the center point and the radius. Besides the stream version, the following constructor is provided

```
sphere(mth *mt, vec e, double r, int nsl, int nbs,
        int eclr, int sclr);
```

with

- `mt` the pointer to the homogeneous transformation matrix describing the position of the local coordinate system of the shape;
- `e` the coordinate vector of the center with respect to the local coordinate system;
- `r` the radius of the sphere;
- `nsl` the number of slices used to represent the sphere (see figure 3.3);
- `nbs` the number of sectors used to represent the sphere (see figure 3.3);
- `eclr` the color code of the edges;
- `sclr` the color code of the surfaces.

Here is an example

```
shape *s1;
vec e(1,0,0);
mth T=Tdisp(0.5,0,0);
s1=new sphere(&T,e,1.5,7,12,3,4);
```

The tyre

The class `tyre` defines a tyre from the two extreme points of its axis and the external and internal radiuses (cf. figure 3.4). Besides the stream version, the following constructor is provided

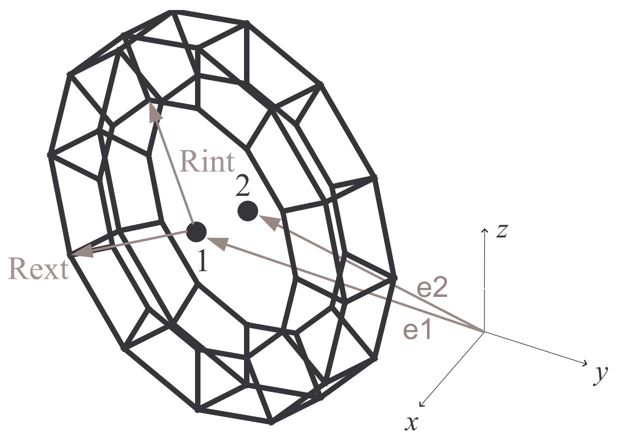


Figure 3.4: Parameters of the tyre shape

```
tyre(mth *mt, vec e1, vec e2, double ri, double re, int nbs,
         int eclr, int sclr);
```

with

- `mt` the pointer to the homogeneous transformation matrix describing the position of the local coordinate system of the shape;
- `e1` and `e2` the coordinate vectors of the two extreme axis points with respect to the local coordinate system;
- `ri` the internal radius;
- `re` the external radius;
- `nbs` the number of sectors used to represent the tyre;
- `eclr` the color code of the edges;
- `sclr` the color code of the surfaces of the frustum.

Here is an example

```
shape *s1;
vec e1(0,-0.1,0), e2(0,0.1,0);
mth T=Tdisp(1.3,0.8,0);
s1=new tyre(&T,e1,e2,0.18,0.25,12,3,4);
```

The gear

The class `gear` defines a gear from the two extreme points of its axis, its module, the number of teeth, and the hole radius. Besides the stream version, the following constructor is provided

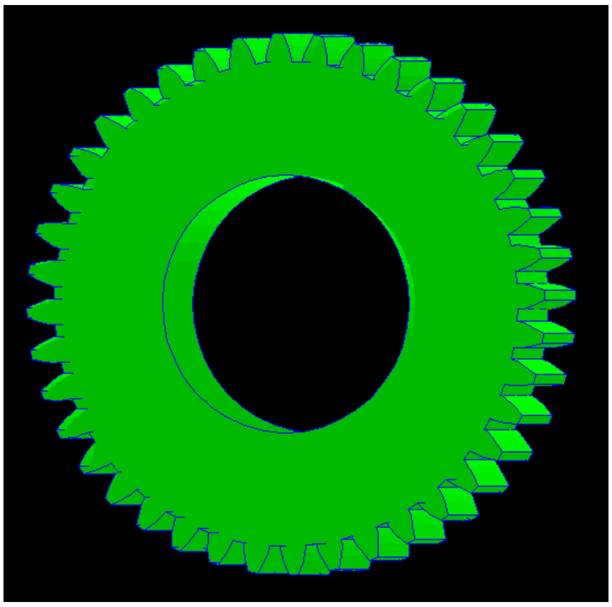


Figure 3.5: Example of a gear

```
gear(mth *mt, vec e1, vec e2, double module, int nt, double alpha, double R,
         int eclr, int sclr);
```

with

- `mt` the pointer to the homogeneous transformation matrix describing the position of the local coordinate system of the shape;
- `e1` and `e2` the coordinate vectors of the two extreme axis points with respect to the local coordinate system (definition similar to the one for frustum and tyre);
- `module` the module of the gear (for recall, the primitive diameter is equal to the module times the number of teeth);
- `alpha` an angle offset that can be used to adjust the gearing between two gears;
- `R` the hole radius (if the hole radius is higher than the primitive radius, the gear is assumed to be a crown);
- `eclr` the color code of the edges;
- `sclr` the color code of the surfaces of the gear.

Here is an example of two connected gears

```
int nd1=40, nd2=30;
double module=0.0254, R1=0.5*nd1*module, R2=0.5*nd2*module;
mth T1,T2=Tdisp(R1+R2,0,0);
vec e1(0,0,0.1), e2(0,0,0.1);
```

```
shape *s1, *s2;
s1=new gear(&T1,e1,e2,module,nd1,0,0.5*R1,1,2);
s2=new gear(&T2,e1,e2,module,nd2,3.1416/nd2,0.5*R2,1,4);
```

It is of interest to note that a gear should always own at least 25 teeth to avoid interference problems at the base of the teeth.

The spring

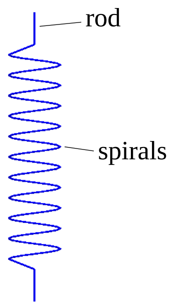


Figure 3.6: Example of a spring

The class `spring` defines a coil spring from the two extreme points, the rod length, the number of spirals and the radius of the spring. Besides the stream version, the following constructor is provided

```
spring(mth *mt1, vec e1, mth* mt2, vec e2, double R, double lrod,
       int nbsp, int nbs, int eclr);
```

with

- `mt1` the pointer to the homogeneous transformation matrix describing the position of the first local coordinate system of the shape;
- `e1` the coordinate vector of the first extreme point with respect to the first local coordinate system;
- `mt2` the pointer to the homogeneous transformation matrix describing the position of the second local coordinate system of the shape;
- `e2` the coordinate vector of the second extreme point with respect to the second local coordinate system;
- `R` the radius of the coilspring;
- `lrod` the length of the rod (cf. figure 3.6);
- `nbsp` the number of spirals;
- `nbs` the number of lines per round used to represent the spirals;

- `eclr` the color code of the edges of the spring.

Here is the code corresponding to figure 3.6

```
mth T1=Tdisp(0,5,0),T2=Tdisp(0,5,-2.5);
vec e1(0,0,3), e2(0,0,1);
shape *s1;
s1=new spring(&T1,e1,&T2,e2,0.4,0.5,10,12,1);
```

The spring has the particularity that its form changes with the relative position of the extremities so that the local coordinates of the nodes must be continuously recalculated. Consequently, the methods `WriteNodes` and `WriteCoord` are overriden. It is a good example if you want to develop a shape with such a feature.

habfile

The class `habfile` defines a shape from a structure defined in a text file. The constructor is declared in the following way

```
habfile(mth *mt, char *filename);
```

with `filename` the name of the file in which the structure of the shape is defined.

The structure of the file is the following

N_n N_e c_e	number of nodes, number of edges and edge color
x_1 y_1 z_1	local coordinates of node 1
x_2 y_2 z_2	local coordinates of node 2
$\vdots$	
x_{N_n} y_{N_n} z_{N_n}	local coordinates of last node
i_1 j_1	indices of end nodes of edge 1
i_2 j_2	indices of end nodes of edge 2
$\vdots$	
i_{N_e} j_{N_e}	indices of end nodes of last edge
N_s c_s	Number of polygonal surfaces and surface color
N_1 i_1 j_1 $\ldots$	number of vertices and node indices of polygon 1
N_2 i_2 j_2 $\ldots$	number of vertices and node indices of polygon 2
$\vdots$	
N_{N_s} i_{N_s} j_{N_s} $\ldots$	number of vertices and node indices of last polygon

Here is an example

```
shape *s1;
mth T=Trotz(0.9);
s1=new habfile(&T,"cube.hab");
```

var_surface

The class `var_surface` defines a shape corresponding to a polygonal surface, whose nodes may move relatively to each other. The constructor is declared in the following way

```
var_surface(mth **mt_per_pt,int n_pt,vec **pt,int sclr,int protection);
```

with `mt_per_pt` an array of matrix pointers (one for each point) giving the situation of the local coordinate systems of the nodes, `n_pt` the number of nodes, `pt` an array of vector pointers giving the position of each node with respect to its local coordinate system, `sclr` the color of the surface.

All mentioned arrays must have been allocated by the user.

The flag `protection` has the same meaning as for the base class `shape`.

var_path

The class `var_path` defines a shape corresponding to a polyline, whose nodes may move relatively to each other. The constructor is declared in the following way

```
var_path(mth **mt_per_pt,int n_pt,vec **pt,int eclr,int protection);
```

with `mt_per_pt` an array of matrix pointers (one for each point) giving the situation of the local coordinate systems of the nodes, `n_pt` the number of nodes, `pt` an array of vector pointers giving the position of each node with respect to its local coordinate system, `eclr` the color of the line.

All mentioned arrays must have been allocated by the user.

The flag `protection` has the same meaning as for the base class `shape`.

3.2.3 Defining your own shapes

The user is free to redefine other classes derived from the base class `shape`. A look at the code of the provided shapes will be more explanatory. However, note that you will have to define at least

- a constructor and, ideally, a destructor;
- the method `WriteShapes()` which can anyway be empty.

3.3 The scenes

3.3.1 Building the scene

A scene is built by successively adding shapes, with the method `AddShape`. Here is an example

```
shape *s1,*s2;
vec e1(-1,0,0), e2(1,0,0);
mth T1=Tdisp(5,0,0);
s1=new frustum(&T1,e1,e2,1.5,1.2,24,1,2);
```

```
vec e3(-1,-1,-1), e4(1,1,1);
mth T2;
s2=new box(&T2,e3,e4,3,4);
scene thescene;
thescene.AddShape(s1);
thescene.AddShape(s2);
```

3.3.2 Saving the scene

Once the scene has been built by adding shapes, it can be saved to a file with extension `.vol` by means of the method `CreateVolFile()`

```
thescene.CreateVolFile("myscene.vol");
```

In this file, the structure of the scene is described in terms of nodes, edges and surfaces (cf. later).

With only the `vol` file, the scene can be visualized with `EasyAnim`.

3.3.3 Creating the animation

An animation is created by saving successive configurations in a file with extension `.van`. This file comprises only the coordinates of the nodes, the structure in terms of edges and surfaces remaining the one described in the associated `.vol` file.

If we go on with the same example, here is how we can create an animation with rotating shapes

```
ofstream VanFile("myscene.van");
int i;
for (i=0;i<100;i++)
     {
     T1=Tdisp(5,0,0)*Trotz(i*0.02*6.2832);
     T2=Troty(i*0.03*6.2832);
     thescene.WriteCoord(VanFile);
     }
VanFile.close();
```

3.3.4 Changing the observer

By default, the scene is observed from the global coordinate system. If a moving observer is wanted, like for example the driver in a car, it can be specified with the method `SetVisuFrame`

```
mth Tref=Tdisp(10,10,10);
thescene.SetVisuFrame(&Tref);
```

When the animation is created, the homogeneous transformation matrix related to the observer evolves as defined by the user.

Sometimes, it is desirable to follow the scene from a moving frame but without rotating with it. The method `SetVisuFrame` will then be called like this

```
thescene.SetVisuFrame(&Tref,1);
```

Only the displacement part of matrix `Tref` will then be taken into account. The direction of view will remain parallel to the global axis.

3.4 Structure of the files

3.4.1 The `.vol` file

The structure of the file is the following

Comment line	the first line is just a comment
N_n	the number of nodes
$1\ x_1\ y_1\ z_1$	name (taken as the index the most often) and coordinates of node 1
$2\ x_2\ y_2\ z_2$	name and coordinates of node 2
$\vdots$	
$N_n\ x_{N_n}\ y_{N_n}\ z_{N_n}$	name and coordinates of last node
N_e	Number of edges
$1\ i_1\ j_1\ c_1$	name, indices of end nodes and color of edge 1
$2\ i_2\ j_2\ c_2$	name, indices of end nodes and color of edge 2
$\vdots$	
$N_e\ i_{N_e}\ j_{N_e}\ c_{N_e}$	name, indices of end nodes and color of last node
N_s	Number of polygonal surfaces
$1\ N_1\ i_1\ j_1\ \ldots c_1$	name, number of vertices, node indices and surface color of polygon 1
$2\ N_2\ i_2\ j_2\ \ldots c_2$	name, number of vertices, node indices and surface color of polygon 2
$\vdots$	
$N_s\ N_{N_s}\ i_{N_s}\ j_{N_s}\ \ldots c_{N_s}$	name, number of vertices, node indices and surface color of last polygon

3.4.2 The `.van` file

The structure of the file is simple. If the scene comprises N_n nodes, the file is a succession of blocks of N_n lines giving the X,Y and Z coordinates of each node, each block corresponding to a saved configuration. The structure of the block is

$x_1\ y_1\ z_1$	coordinates of node 1
$x_2\ y_2\ z_2$	coordinates of node 2
$\vdots$	
$x_{N_n}\ y_{N_n}\ z_{N_n}$	coordinates of last node

During animation by `EasyAnim`, the configurations are read successively and displayed on the screen.

Chapter 4

The EasyDyn simulation library

4.1 Introduction

The `sim` part of the library allows to easily simulate any system described by a set of second order differential equations of the form

$$\underline{\mathbf{f}}(\underline{\mathbf{q}}, \underline{\dot{\mathbf{q}}}, \underline{\ddot{\mathbf{q}}}, t) = 0 \tag{4.1}$$

with

- t the time;
- $\underline{\mathbf{q}}$, $\underline{\dot{\mathbf{q}}}$ and $\underline{\ddot{\mathbf{q}}}$ the vectors gathering the parameters of the system and their first and second time derivatives;
- $\underline{\mathbf{f}}$ the vector gathering the residuals of the differential equations governing the behaviour of the physical system.

Let's recall that the problem can be solved only if the number of parameters is the same as the number of differential equations.

The library was basically created to simulate the motion of mechanical systems but can be used for any physical system whose behaviour is described in terms of second-order differential equations.

First-order differential equations of the form

$$\underline{\mathbf{f}}(\underline{\mathbf{y}}, \underline{\dot{\mathbf{y}}}, t) = 0 \tag{4.2}$$

can be solved as well by considering $\underline{\mathbf{y}} = \underline{\dot{\mathbf{q}}}$ and $\underline{\dot{\mathbf{y}}} = \underline{\ddot{\mathbf{q}}}$, $\underline{\mathbf{q}}$ becoming meaningless.

4.2 Structure and routines of the library

4.2.1 Global variables

The structure of the library is illustrated in figure 4.1. It involves the following global variables

- `int nbrdof` the number of degrees of freedom of the system (or number of parameters or number of differential equations);
- `double *f` the array gathering the residuals of the considered differential equations
- `double *q, *qd, *qdd` the arrays gathering the parameters and their first and second time derivatives;
- `int nbrinput` the number of inputs to the system (necessary only if you want to export the input-output linearized system);
- `double *u` the array gathering the inputs of the system (inputs represent any entry exerting an action on the system: force, voltage, pressure,... that could be used to control it); the inputs are expected to play a role in the differential equations);
- `double t` the time;
- `char *application` the name of the application used namely to create the result file;
- `int DEBUG` an option which, if different from zero, activates the printing of various information during the simulation (equal to zero by default).

To be sure to declare the global variables only once, the main program will have to define `EASYDYNSIMMAIN` before including the header file.

4.2.2 Routines that the user must provide

The user manages the `main` routine and must provide the following functions

- `void ComputeResidual()` which computes the residuals `f` in terms of `q`, `qd`, `qdd` and time `t`;
- `WriteDataHeader(ostream &OutFile)` which sends to the `OutFile` output stream the **names**, separated by some space, of the data the user wants to save; this procedure is called only once at the beginning of an integration; eventually, it is sufficient to call the provided routine `WriteStateVariablesHeader` which sends the names of the data saved by the routine `SaveStateVariables`; pay attention that a final linebreak must be sent, which is not done by the latter routine;
- `SaveData(ostream &OutFile)` which sends to the `OutFile` output stream the **values**, separated by some space, of the data the user wants to save; this procedure is called automatically at regular intervals by some integration routines or can be called by the main program; eventually, it is sufficient to call the provided routine `SaveStateVariables` which sends the time followed, for each variable i, by the value of `q[i]`, `qd[i]` and `qdd[i]` (`t q[0] qd[0] qdd[0] q[1] ...`); pay attention that a final linebreak must be sent, which is not done by the latter routine.

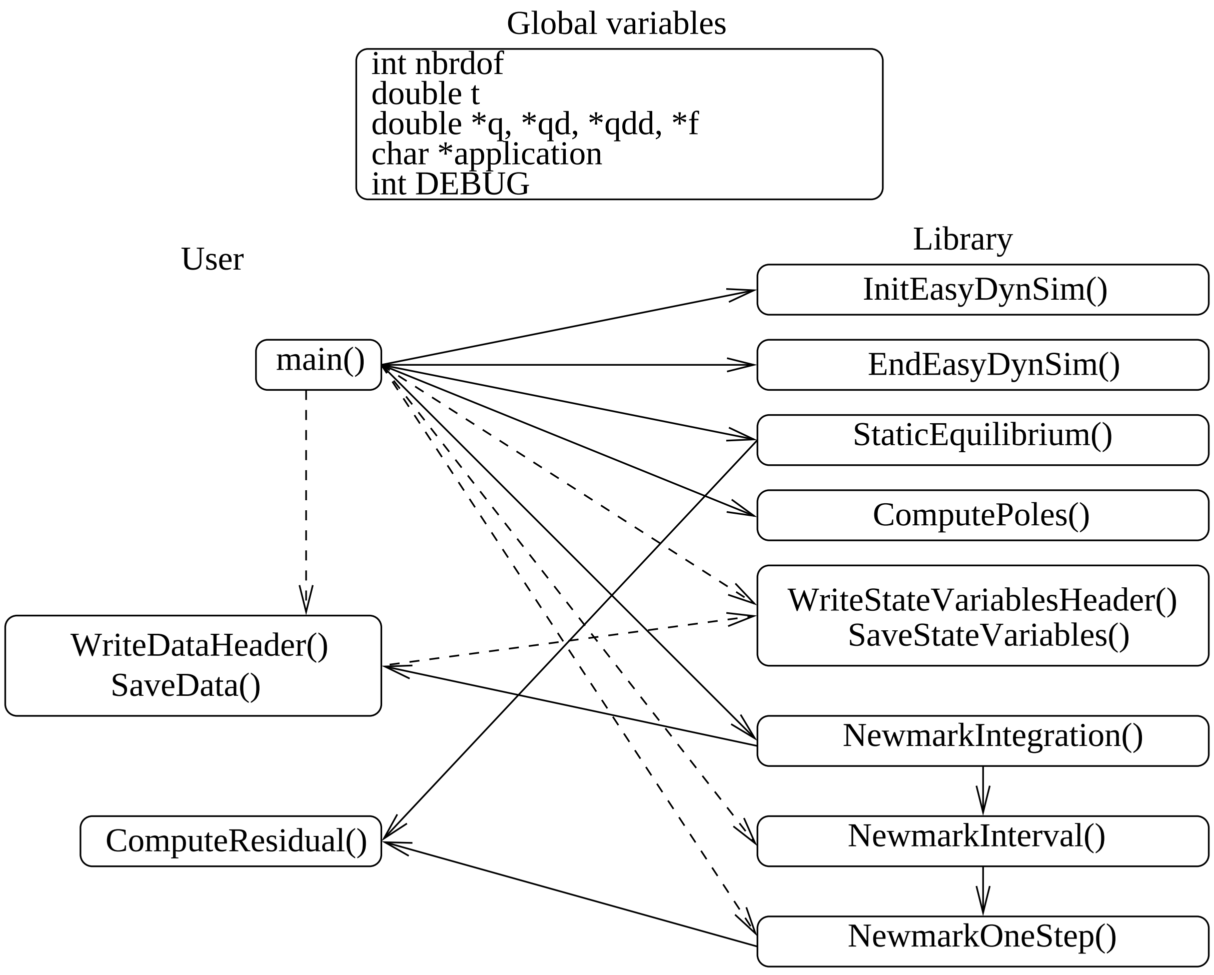


Figure 4.1: Structure of `EasyDyn sim`

4.2.3 Initializing the process

Before calling any other procedure, the user must set the variable `nbrdof` and call the routine `InitEasyDynsim()`, which allocates memory to all necessary arrays and sets all terms of `q`, `qd` and `qdd` to zero. If you intend to use inputs, the variable `nbrinput` must also be given and the initialization routine allocates memory for the array `u`. If not specified, `nbrinput` is equal to zero and inputs can be ignored.

When the initialization is done, the integration routines can be called. Similarly at the end of the program, the routine `EndEasyDynsim` should be called to clean the memory.

4.2.4 Integration routines

If the user wants the finest control on the integration, he will use the basic integration routine

```
int NewmarkOneStep(double h, double &errqd, int *doflocked=0);
```

which performs an integration step with a time step `h` according to the Newmark formulas. If the value returned by the function is different from zero, either the process was unable to converge (1), or there was a numerical trouble (2). On return, `errqd` gives an estimation of the rate of error on positions for the time step. The array `doflocked` allows to eventually lock selected degrees of freedom. If `doflocked` is equal to zero it is ignored. Otherwise, if `doflocked[i]` is different from zero, the ith degree of freedom is not modified by the integration procedure and the ith residual is ignored. The evolution of the corresponding degree of freedom can be specified in `ComputeResidual`. If not, it will follow a uniformly accelerated evolution according to the initial conditions. Due to the ability of C++ to use default arguments, `doflocked` will be automatically set to zero if it is not given in the call.

At a higher level, we find the procedure

```
void NewmarkInterval(double tfinal, double &h, double hmax,
                     double *doflocked=0)
```

which integrates the equations up to time `tfinal`, with an initial time step `h` and a maximum allowed time step `hmax`. To integrate over the interval, `NewmarkInterval` makes successive calls to `NewmarkOneStep` while automatically adapting the time step to keep the error below a chosen tolerance (1E-6).

The most often, the user will directly call the most general routine

```
void NewmarkIntegration(double tfinal, double hsave, double hmax,
                        int *doflocked=0)
```

which performs the integration up to time `tfinal` by regular time intervals equal to `hsave` and with the maximum allowed time step `hmax`. The routine `NewmarkIntegration` automatically opens a file called *application*`.res`, writes the header with the function `WriteDataHeader` and saves the data after each interval by a call to `SaveData`. Practically, `NewmarkInterval` is called for each integration interval.

4.2.5 Static equilibrium

For some particular cases, it is necessary to reach the static equilibrium before launching the integration. This can be done by calling the following routine

```
void StaticEquilibrium(int *doflocked)
```

which search for the parameters $\mathbf{q}_i$ verifying

$$\underline{\mathbf{f}}(\underline{\mathbf{q}}, \underline{\dot{\mathbf{q}}}^0, \underline{\ddot{\mathbf{q}}}^0, t^0) = 0 \tag{4.3}$$

at time t_0 for a given set of initial velocities and accelerations. The values of $\underline{\dot{\mathbf{q}}}^0$ and $\underline{\ddot{\mathbf{q}}}^0$ are the ones in memory when calling `StaticEquilibrium`.

The most often, the accelerations will be equal to zero, which leads to a stationnary behaviour. If velocities are equal to zero as well, the yielded configuration is an actual static equlibrium position.

The vector `doflocked` specifies whether some particular degrees of freedom should be locked during the process. Pratically, `q[i]` will be locked if `doflocked` is different from zero.

4.2.6 Linearization and eigen values

It is sometimes interesting to evaluate the behaviour of a system by considering only small perturbations with respect to a given reference configuration.

The reference configuration will be characterized by $\underline{\ddot{\mathbf{q}}}^0$, $\underline{\dot{\mathbf{q}}}^0$, $\underline{\mathbf{q}}^0$ which are expected to verify the equations of motion

$$\underline{\mathbf{f}}(\underline{\mathbf{q}}^0, \underline{\dot{\mathbf{q}}}^0, \underline{\ddot{\mathbf{q}}}^0, t^0) = 0 \qquad (4.4)$$

The most often, the configuration corresponds to a static equilibrium configuration or a stationnay motion (for example, a vehicle at constant speed in straight line).

The equations of motion can then be linearized in the following way

$$\underline{\mathbf{f}}(\underline{\mathbf{q}}^0 + \Delta\underline{\mathbf{q}}, \underline{\dot{\mathbf{q}}}^0 + \Delta\underline{\dot{\mathbf{q}}}, \underline{\ddot{\mathbf{q}}}^0 + \Delta\underline{\ddot{\mathbf{q}}}, t^0 + t) \simeq \mathbf{M} \cdot \Delta\underline{\ddot{\mathbf{q}}} + \mathbf{CT} \cdot \Delta\underline{\dot{\mathbf{q}}} + \mathbf{KT} \cdot \Delta\underline{\mathbf{q}} = 0 \qquad (4.5)$$

with **M**, **CT** and **KT** respectively the mass matrix, the tangent damping matrix and the tangent stiffness matrix, defined by

$$\mathbf{M}_{ij} = \left.\frac{\partial \mathbf{f}_j}{\partial \ddot{\mathbf{q}}_j}\right|_0 \qquad \mathbf{CT}_{ij} = \left.\frac{\partial \mathbf{f}_j}{\partial \dot{\mathbf{q}}_j}\right|_0 \qquad \mathbf{KT}_{ij} = \left.\frac{\partial \mathbf{f}_j}{\partial \mathbf{q}_j}\right|_0 \qquad (4.6)$$

The eigenvalues (or roots or poles) λ_i and the corresponding eigenvectors $\underline{\mathbf{\Psi}}^i$ result from the solution of the following eigenvalue problem

$$\left(\lambda_i^2\mathbf{M} + \lambda_i\mathbf{CT} + +\mathbf{KT}\right) \cdot \underline{\mathbf{\Psi}}^i = 0 \qquad (4.7)$$

As damping and stiffness tangent matrices are not necessarily symmetric, the eigenvalues and eigen vectors can be complex. However, as all matrices are real, complex roots are always complex conjugates.

The eigenvalue analysis allows to directly assess the stability of the system. For recall, the system will be unstable as soon as the real part of any pole is positive. If the root has an imaginary part, the associated motion is oscillatory, the imaginary part corresponding to the damped pulsation.

The determination of the poles and eigenvectors about the current configuration can be performed by means of the routine `ComputePoles`, declared in the following way

```
void ComputePoles(int *doflocked, double freqmin=0, double freqmax=1E30)
```

When calling `ComputePoles`, the equations of motion are first linearized. The eigen values and eigen modes are then determined and saved. The complete list of poles is saved in the file *application*`.lst`, with the following format (cf example `oscil5.cpp`)

```
 POLE        ALPHA          OMEGA          FREQ(Hz)    DAMP_RATIO
   2      -0.628319        6.25169        0.994987             0.1
```

The number of the pole does not necessarily begins from 1. This is due to the fact that in the case of complex conjugates poles, only the last one is saved. `ALPHA`, `OMEGA` represent the real and imaginary part of the pole and are related to the damped frequency (`FREQ`) and damping ratio (`DAMP_RATIO`) by

$$f = \frac{\omega}{2\pi} \qquad \xi = -\frac{\alpha}{\sqrt{\alpha^2 + \omega^2}} \tag{4.8}$$

The eigen modes are saved in the file *application*`.mod` but only the ones whose frequency is between `freqmin` and `freqmax`. With the default values of `freqmin` and `freqmax`, all modes should be saved. If different values of `freqmin` and `freqmax` are specified, another file called *application*`.ls2` is created, with only the roots whose frequency falls in the specified range.

The file *application*`.mod` consists of successive data sets corresponding to the saved modes. Each dat set holds the following information

NumberOfPole RealPart ImagPart Freq DampingRatio
Ψ_1,real Ψ_1,imag
Ψ_2,real Ψ_2,imag
:
:
Ψ_N,real Ψ_N,imag

where N represents the number of degrees of freedom of the system.

Practically, the linearization is performed by finite differences, the eigenvalue problem being solved by a call to the GSL routine for unsymmetric real eigenvalue problems (available from version 1.10). The latter replaces `dggev` of `LAPACK`, which is then no longer necessary to build `EasyDyn` applications.

4.2.7 Linearized input-ouput system

In a similar war, the linearized system of the form

$$\mathbf{M}\Delta\underline{\ddot{\mathbf{q}}} + \mathbf{C}\mathbf{T}\Delta\underline{\dot{\mathbf{q}}} + \mathbf{K}\mathbf{T}\Delta\underline{\mathbf{q}} = \mathbf{F}\Delta\underline{\mathbf{u}} \tag{4.9}$$

can be saved, for example, to set-up a controller of the system with the help of an appropriate tool.

When the reference configuration is specified in terms of positions, velocities, accelerations and inputs, the only thing to do is to call the routine

```
void SaveLinearizedSystem(int *doflocked=0)
```

where `doflocked`, if specified, allows to freeze some parameters.
The routine linarizes the system by finite differences and saves the mass, damping, stiffness and influence matrices, in ascii form, respectively in the files *application*`.mm`, *application*`.cc`, *application*`.kk` and *application*`.ff`.

Generally, inputs are defined in terms of time or according to a control strategy inside the routine `ComputeResidual()`. When calling `SaveLinearizedSystem()`, the inputs should remain unspecified. Otherwise, the numerical derivation will have no effect. Look at the example of the DC motor (files `dcmotor.cpp` and `dcmotorPID.cpp`) for illustration.

4.3 A simple example: the harmonic oscillator

4.3.1 Equations to solve

The motion of a body of mass m, attached to the ground by a spring of stiffness k is governed by the following equation

$$m\ddot{q} + kq = 0 \tag{4.10}$$

often rewritten as

$$\ddot{q} + \omega_0^2 q = 0 \tag{4.11}$$

with $\omega_0^2 = \frac{k}{m}$.

The solution of this differential equation is given by

$$q = A\cos(\omega_0 t + \phi) \tag{4.12}$$

the amplitude A and the phase ϕ depending on the initial conditions.

In particular, if we choose $\omega_0 = 2\pi$, that's to say a frequency of 1 Hz, and the initial conditions $q = 1$ and $\dot{q} = 0$, the solution will be

$$q = \cos(2\pi t) \tag{4.13}$$

We will see how we can retrieve this solution with the help of `EasyDyn`.

4.3.2 The simplest way

The code below (`oscil1.cpp`) shows the simplest code to implement our problem. It is based on a call to `NewmarkIntegration`

```
// Simulation of a mass-spring system (eigenfrequency=1 Hz)

#define EASYDYNSIMMAIN // to declare the global variables
#include <EasyDyn/sim.h>

double w0=2*3.14159265;

//---------------------------------------------------------------------
```

```
void ComputeResidual()
{
f[0]=qdd[0]+w0*w0*q[0];
}

//---------------------------------------------------------------------

void WriteDataHeader(ostream &OutFile)
{
WriteStateVariablesHeader(OutFile);
OutFile << endl;
}

//---------------------------------------------------------------------

void SaveData(ostream &OutFile)
{
SaveStateVariables(OutFile);
OutFile << endl;
}

//---------------------------------------------------------------------

int main()
{
  // Initialisation and memory allocation
  nbrdof=1; application="oscil1";
  InitEasyDynsim();
  // Initial configuration
  q[0]=1;
  // Let's go !
  NewmarkIntegration(5,0.01,0.005);
  EndEasyDynsim();
}
//---------------------------------------------------------------------
```

After compiling and running the code, the file `oscil1.res` contains the evolution of q, $\dot{q}$ and $\ddot{q}$ which can be easily plotted by GNUPLOT or a MATAB-alike program. The result is shown in figure 4.2

4.3.3 Alternative implementations

If you want to manage the step by step procedure of the integration, you can call the lower level routines `NewmarkInterval` or `NewmarkOneStep`.

If you call `NewmarkInterval`, you will only have to determine the initial accelerations

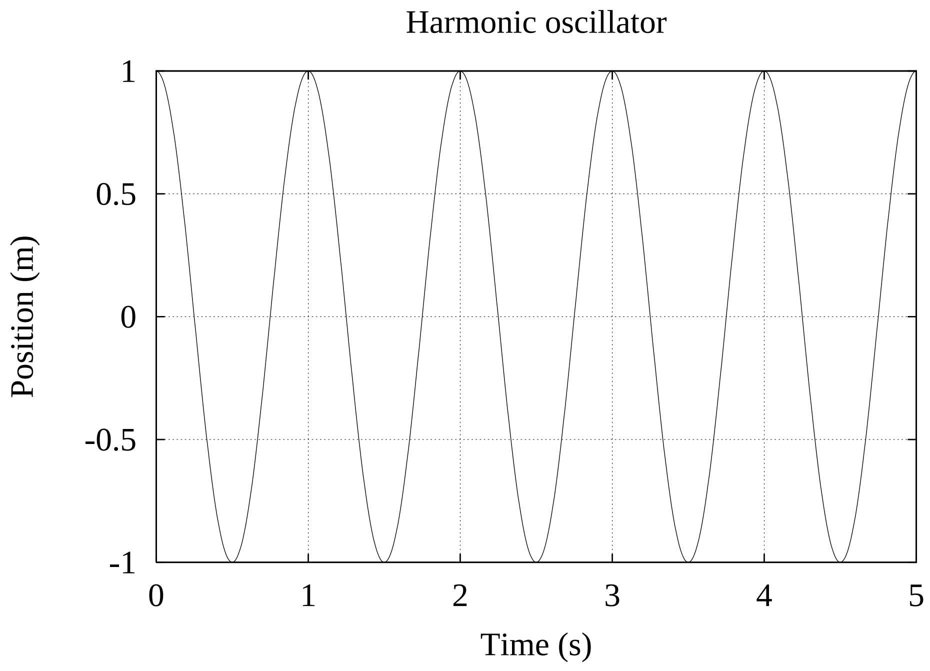


Figure 4.2: Results for the harmonic oscillator

and to manage the creation of the result file bu the integration should proceed fluently. The main routine of the program will look like this (example `oscil2.cpp`)

```
int main()
{
  // Initialisation and memory allocation
  nbrdof=1; application="oscil2";
  InitFpmsSim();
  // Initial configuration
  q[0]=1;
  // Opening of the result file
  ofstream ResFile("oscil2.res");
  // Determining initial accelerations
  t=0;
  double errqd;
  NewmarkOneStep(0,errqd);
  // Saving initial state
  SaveData(ResFile);
  // Let's go !
  double interval=0.01, tfinal=5, hmax=interval/2, h=hmax;
  while (t<tfinal)
      {
      NewmarkInterval(t+interval,h,hmax);
      cout << "t=" << t << endl;
      SaveData(ResFile);
      }
  ResFile.close();
  EndFpmsSim();
}
```

Note: the initial acceleration is determined by an initial call to `NewmarkOneStep` with a null time step.

If you call `NewmarkOneStep`, you will have moreover to look at the stability of the numerical integration. The main routine of the program will look like this (example `oscil3.cpp`)

```
int main()
{
  // Initialisation and memory allocation
  nbrdof=1; application="oscil3";
  InitFpmsSim();
  // Initial configuration
  q[0]=1;
  // Opening of the result file
  ofstream ResFile("oscil3.res");
  // Determining initial accelerations
  t=0;
  double errqd;
  NewmarkOneStep(0,errqd);
  // Saving initial state
  SaveData(ResFile);
  // Let's go !
  double h=0.05, tfinal=5;
  int code;
  while (t<tfinal)
      {
      code=NewmarkOneStep(h,errqd);
      cout << "t=" << t << endl;
      if (code==1) { cout << "No convergence\n"; exit(1); }
      if (code==2) { cout << "Numerical trouble\n"; exit(2); }
      SaveData(ResFile);
      }
  ResFile.close();
  EndFpmsSim();
```

A direct use of `NewmarkOneStep` can lead to inaccurate results if the time step is not appropriate. Figure 4.3 shows the result obtained if there are only 20 time steps per period.

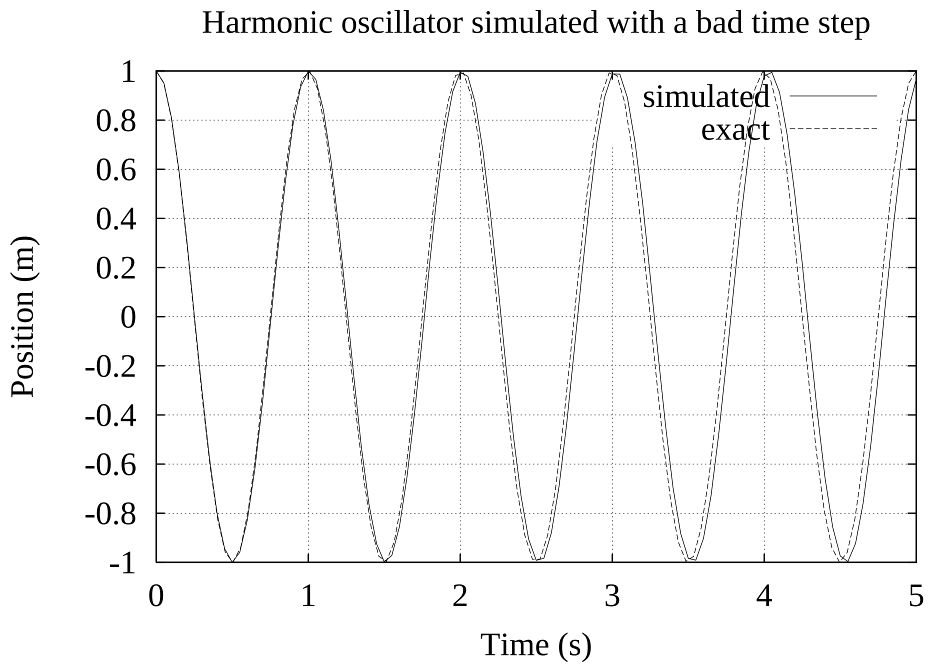


Figure 4.3: Results with a bad time step

4.3.4 Searching for the equilibrium position

You can determine the equilibrium position with a call to `StaticEquilibrium`. Here below the main routine doing the job for the harmonic oscillator. The obtained position should be equal to zero (example `oscil4.cpp`).

```
int main()
{
  // Initialisation and memory allocation
  nbrdof=1; application="oscil3";
  InitFpmsSim();
  // Initial configuration
  q[0]=1;
  t=0;
  StaticEquilibrium();
  cout << "Position reached for q=" << q[0] << endl;
  EndFpmsSim();
}
```

4.3.5 Performing the linear analysis (eigen values)

You can determine the eigenvalues with a call to `ComputePoles`. Here below the main routine doing the job for the harmonic oscillator. The operation is performed about the equilibrium position (example `oscil5.cpp`).

```
int main()
{
  // Initialisation and memory allocation
  nbrdof=1; application="oscil5";
  InitEasyDynsim();
  // Initial configuration
  q[0]=0;
  // Let's go !
  ComputePoles();
  EndEasyDynsim();
}
```

4.4 Another example: the hydraulic jack

The system represented in figure 4.4, consists of an hydraulic jack pushing a mass m, attached to the ground by a spring of stiffness k. The jack comprises two chambers numbered 1 and 2. It is a good test for EasyDyn as the hydraulic equations are well-known to be particularly stiff.

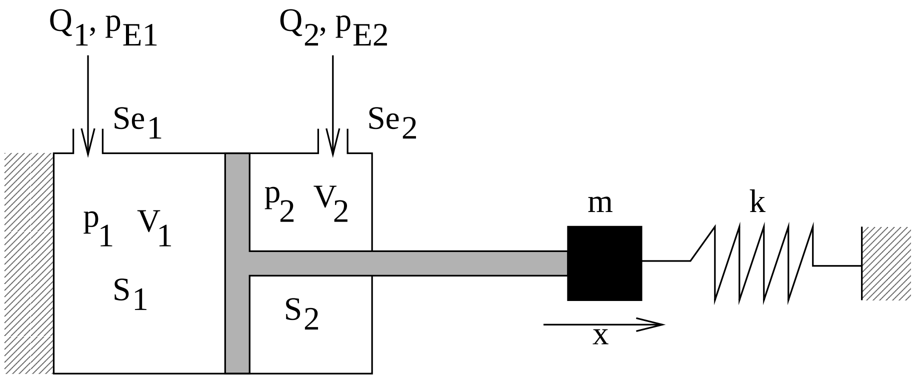


Figure 4.4: Hydraulic system

The volume flow Q_i entering in each chamber i (i=1,2) verifies on one hand the discharge law through the throttling section S_{ei}

$$Q_i = C_d S_{ei} \sqrt{\frac{2(p_{Ei} - p_i)}{\rho}} * \operatorname{sign}(p_{Ei} - p_i) \tag{4.14}$$

with C_d the orifice discharge coefficient, ρ the fluid density, p_i the fluid pressure inside the chamber, and p_{Ei} the circuit pressure at the entrance of the chamber.

On the other hand, the flow is driven by the variation of volume of the chamber and by the variation of the fluid pressure

$$Q_i = \dot{V}_i + \frac{V_i}{K}\dot{p}_i \tag{4.15}$$

with V_i the volume of the chamber and K the compressibility coefficient of the fluid.

If we consider that the position x=0 corresponds to the rest length of the spring, the dynamic equilibrium of mass m is given by

$$m\ddot{x} + kx - p_1 S_1 + p_2 S_2 = 0 \tag{4.16}$$

If V_{0i} is the volume of the chamber for the position x=0, the volume can be expressed in terms of x

$$V_1 = V_{01} + S_1 x \qquad V_2 = V_{02} - S_2 x \tag{4.17}$$

with S_1 and S_2 the cross sections of chambers 1 and 2.

The other equations of motion are obtained by matching the two expressions of the flow

$$\begin{aligned} S_1\dot{x} + \tfrac{V_{01}+S_1 x}{K}\dot{p}_1 \\ -C_d S_{e1}\sqrt{\tfrac{2(p_{E1}-p_1)}{\rho}} * \operatorname{sign}(p_{E1}-p_1) = 0 \end{aligned} \tag{4.18}$$

$$\begin{aligned} -S_2\dot{x} + \tfrac{V_{02}-S_2 x}{K}\dot{p}_2 \\ -C_d S_{e2}\sqrt{\tfrac{2(p_{E2}-p_2)}{\rho}} * \operatorname{sign}(p_{E2}-p_2) = 0 \end{aligned} \tag{4.19}$$

Although two equations are of first-order form, they can be naturally treated by specifying the 3 state variables as

$$q_1 = x \quad \dot{q}_2 = p_1 \quad \dot{q}_3 = p_2 \tag{4.20}$$

Here is the code implementing the simulation of the hydraulic system (`hydrjack.cpp` in the `examples\testsim` directory).

```
// Simulation of a hydraulic jack pushing a mass-spring system

#define EASYDYNSIMMAIN // to declare the global variables
#include <EasyDyn/sim.h>

double m=10, k=1E5, Kf=2E8, rho=860, pE1=1E6, pE2=1E6, S1=1E-3, S2=5E-4,
  V01=1E-4, V02=3E-4, Cd=0.611, Se1=1E-5, Se2=1E-5;

//----------------------------------------------------------------------

void ComputeResidual()
{
double p1,p2,p1d,p2d;
p1=1E5*qd[1]; p1d=1E5*qdd[1]; // In qd, the pressure is in bar
p2=1E5*qd[2]; p2d=1E5*qdd[2];
pE1=1E6+9E6*(100*t);
if (t>0.01) pE1=1E7;
f[0]=m*qdd[0]+k*q[0]-p1*S1+p2*S2;
f[1]=S1*qd[0]+(V01+S1*q[0])*p1d/Kf
    -Se1*Cd*sqrt(fabs(pE1-p1)/rho)*(pE1-p1)/(1+fabs(pE1-p1));
f[2]=-S2*qd[0]+(V02-S2*q[0])*p2d/Kf
    -Se2*Cd*sqrt(fabs(pE2-p2)/rho)*(pE2-p2)/(1+fabs(pE2-p2));
}

//--------------------------------------------------------------------

void WriteDataHeader(ostream &OutFile)
{
WriteStateVariablesHeader(OutFile);
OutFile << endl;
}

//--------------------------------------------------------------------

void SaveData(ostream &OutFile)
{
SaveStateVariables(OutFile);
OutFile << endl;
}

//--------------------------------------------------------------------

int main()
{
  // Initialization and memory allocation
  nbrdof=3; application="hydrjack";
  InitEasyDynsim();
  // Initial configuration
  qd[1]=pE2/1E5;
  qd[2]=pE2/1E5;
  // Let's go !
  DEBUG=1;
  NewmarkIntegration(1,0.001,0.0005);
  EndEasyDynsim();
}

//--------------------------------------------------------------------
```

The system has been simulated with the data in table 4.1 with the following initial conditions

$$x_0 = \dot{x}_0 = 0 \qquad p_{1_0} = p_{2_0} = 10 \text{ bar}$$

for the pressure p_{E1} increasing linearly from 10 to 100 bar, in the interval [0:0.01] seconds. The initial accelerations ($\ddot{x}$, $\dot{p}_1$ and $\dot{p}_2$) are determined from the equations of motion.

Table 4.1: Simulation data for the hydraulic jack

m=10 kg	$k = 10^5$ N/m	$K = 2 \cdot 10^8$ Pa
$\rho = 860$ kg/m^3	$p_{E2} = 10$ bar	S_1=0.001 m^2
S_2=0.0005 m^2	$V_{01} = 10^{-4}$ m^3	$V_{01} = 3 \cdot 10^{-4}$ m^3
$C_d = 0.611$	S_{e1}=10^{-5} m^2	S_{e2}=10^{-5} m^2

It is of interest to note that the pressure has been expressed in bar so as to avoid mixing state variables of completely different orders of magnitude.

Some results are presented in figures 4.5 to 4.8. The mass position (figure 4.5) is driven initially by the flow through the orifices and stops when the equilibrium is reached between the spring reaction and the force exerted by the jack. At the end of the simulation, the pressure in the chambers is the one of the corresponding circuit (figure 4.6). The stiff nature of the equations is illustrated on the evolution of the mass velocity (figure 4.7) where the oscillations due to the fluid compressibility can be clearly identified. Figure 4.8 shows how the integration procedure adapts the time step during the simulation. This evolution must be correlated with figure 4.7. In the beginning of the simulation, a small time step is necessary due to heavy oscillations of the system. Later, the behaviour is quasi-stationary and a larger time step is allowed. It can be seen that a maximum time step of 0.0005 seconds has been specified by the user.

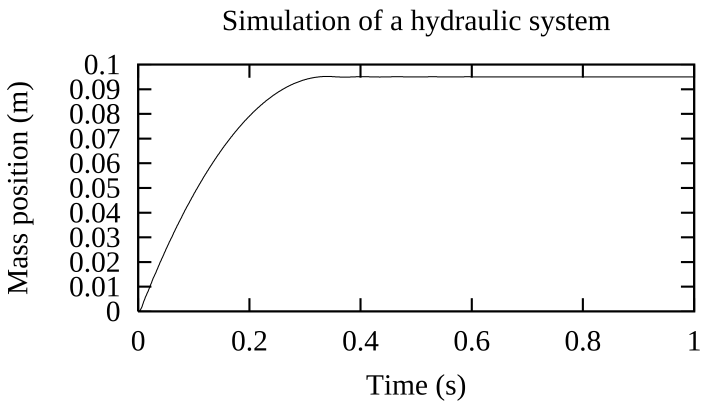


Figure 4.5: Mass position

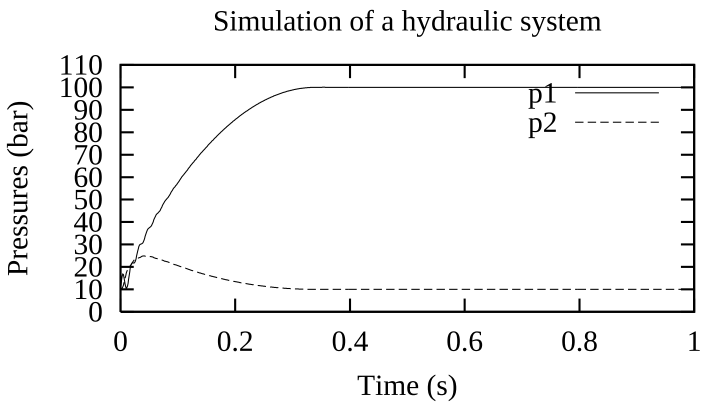


Figure 4.6: Pressure in the chambers

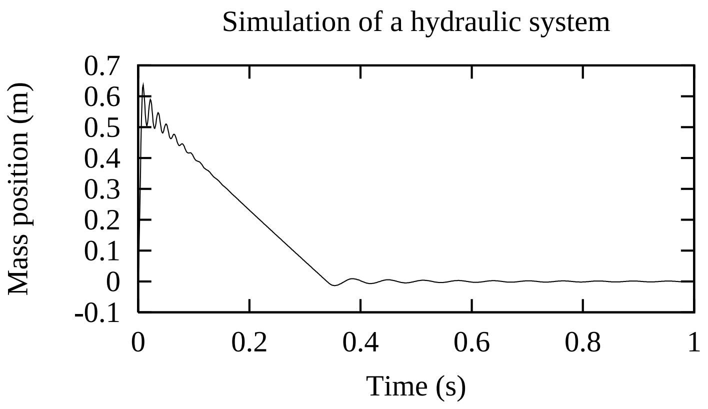


Figure 4.7: Mass velocity

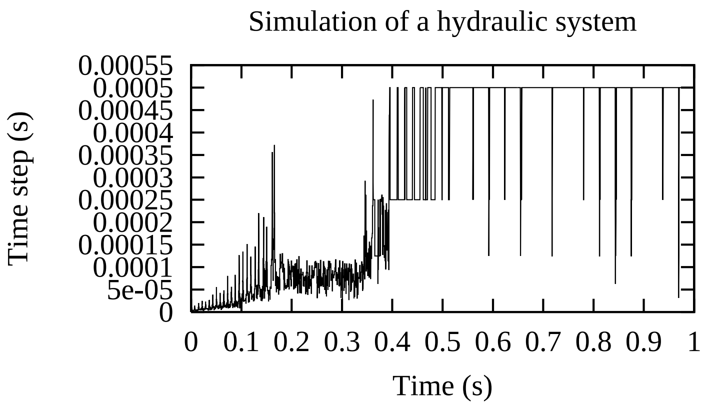


Figure 4.8: Evolution of time step

Chapter 5

The EasyDyn multibody library

5.1 Introduction

The `mbs` part of the library is a frontend to the `sim` library for the simulation of multibody systems. It provides the routine `ComputeResidual` which automatically builds the equations of motion if the user can provide

- the kinematics of the system, that's to say the expression of position, velocity and acceleration of each body in terms of the chosen configuration parameters and their first and second time derivatives;
- the expression of external forces exerted on each body in terms of time, configuration parameters and their time derivatives.

5.2 Structure of the library

5.2.1 Global variables

All the global variables of the `sim` library automatically exist in `mbs`. Morevover, we have

- `int nbrbody`: the number of bodies involved in the considered system;
- `structbody *body`: an array gathering the properties of all bodies in the system.

The structure `structbody` is declared in the following way

```
struct structbody
{
double mass;
tiner PhiG;
mth TOG,TrefG;
vec vG, aG, omega, omegad,
    vGrel, aGrel, omegarel, omegadrel,
    R, MG;
};
```

with

- `mass` the mass of the body;
- `PhiG` the inertia tensor of the body with respect to the center of gravity, expressed in the coordinate system of the body;
- `TOG` the homogeneous tranformation matrix giving the situation of the coordinate system of the body with respect to the global frame; the coordinate system of the body must be located at the center of gravity;
- `vG`, `aG` the velocity and acceleration vectors of the center of gravity, the coordinates being expressed in the global coordinate system;
- `omega`, `omegad` the rotational velocity and acceleration vectors of the body, the coordinates being expressed in the global coordinate system;
- `TrefG` the homogeneous tranformation matrix giving the situation of the coordinate system of the body with respect to the one of the body used as a reference to express the relative motion;
- `vGrel`, `aGrel` the relative velocity and acceleration vectors of the center of gravity with respect to the coordinate system of the body used as a reference to express the relative motion; the coordinates are expressed in the coordinate system of the reference body;
- `omegarel`, `omegadrel` the rotational velocity and acceleration vectors of the body with respect to the coordinate system of the body used as a reference to express the relative motion; the coordinates are expressed in the coordinate system of the reference body;
- `R,MG` the resultant force and the resultant moment with respect to the center of gravity, of inertia and applied efforts exerted on the body (the joint efforts don't need to be considered in the proposed approach); the coordinates are expressed in the global coordinate system.

All data related to the relative motion (`TrefG`, `vGrel`, `aGrel`, `omegarel` and `omegadrel`) are used only if the motion of the body is defined from the relative motion with respect to another body.

5.2.2 Procedures that the user must provide

Like in `sim`, the user manages the `main` routine and must provide the following functions

- `SetInertiaData`; this procedure is called only once at the beginning and initializes the inertia data of each body, that's to say `mass` and `PhiG`; the software verifies the following physical conditions
  - the mass must be positive;

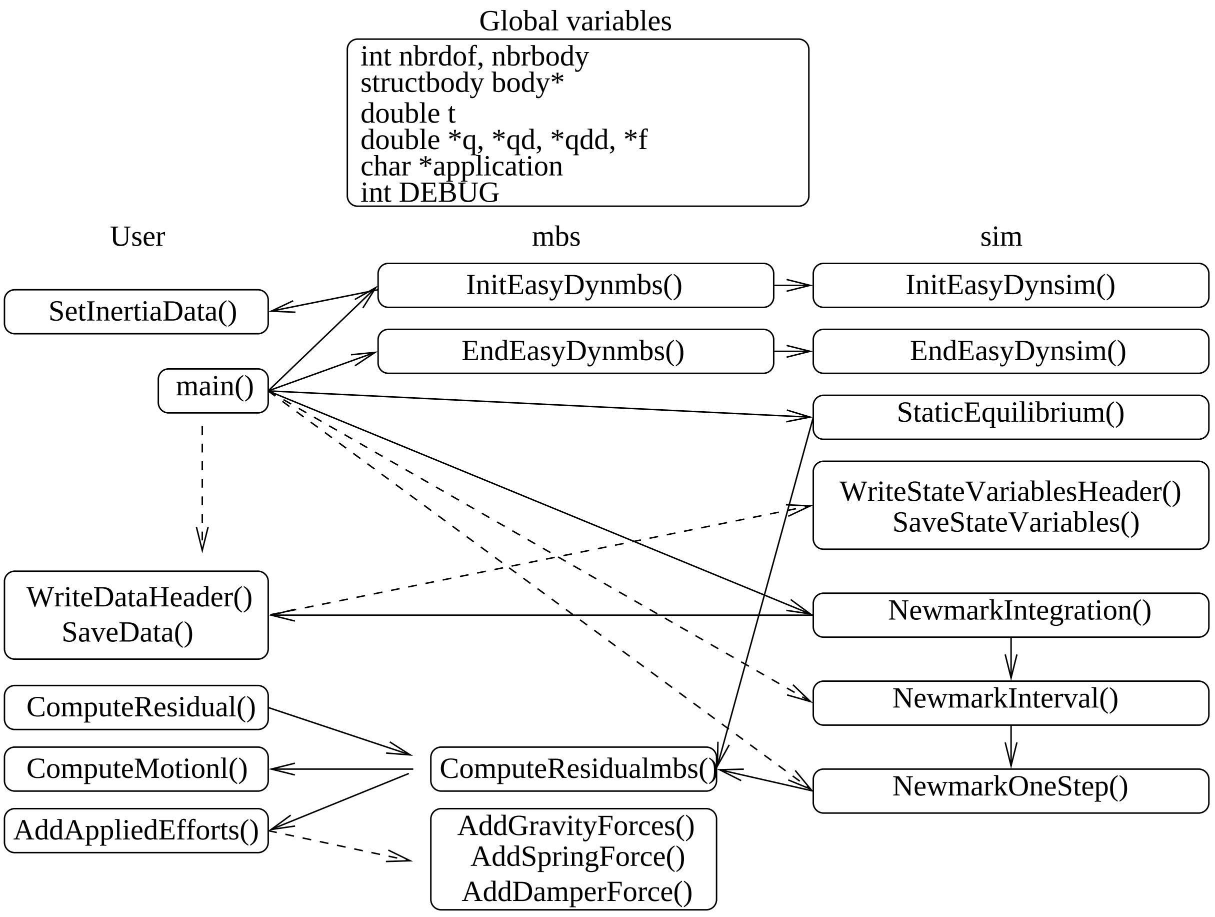


Figure 5.1: Structure of `EasyDyn mbs`

– any diagonal term of the inertia tensor mustn't be larger than the sum of the two other ones.

- `ComputeMotion()` building for each body the position matrix `TOG` and the vectors `vG`, `aG`, `omega` and `omegad` from the array of the configuration parameters `q` and their time derivatives `qd` and `qdd`. All entities can be described directly or from the relative motion with respect to another body. In this case, the user builds the relative position matrix `TrefG` and the vectors defining the relative motion `vGrel`, `aGrel`, `omegarel` and `omegadrel` from the array of the configuration parameters `q` and their time derivatives `qd` and `qdd`. The global motion is finally built from the motion of the reference body and the relative motion, by calling the procedure `ComposeMotion()` (cf. later).
- `AddAppliedEfforts` **adding** to the vectors `R` and `MG` the contribution of the efforts applied on each body (the inertia reactions being calculated automatically by the internal routine `ComputeInertiaEfforts`); the job is made easier by some routines implementing classical element forces like gravity, springs or dampers (cf. further).
- `ComputeResidual()` which the most often will simply call `ComputeResidualmbs()`,

which builds the equations of motion from the efforts and the kinematics. The procedures have been splitted in case the user would like to write supplementary differential equations relative to non kinematic parameters, like for example the model of a hydraulic actuator or a continuous controller.

5.2.3 Initializing the process

Before calling the simulation routines, the user must set the variables `nbrdof` and `nbrbody` and call the routine `InitEasyDynmbs()`, which itself calls `InitEasyDynsim`, allocates memory for the `body` array and calls `SetInertiaData`.

Similarly at the end of the program, the routine `EndEasyDynmbs` should be called to clean the memory.

5.2.4 Simulation routines

As `mbs` is a frontend to `sim` which simply implements in a particular way the routine `ComputeResidual`, the simulation routines are used in the same way as in `sim`.

The routine `ComputeResidualmbs` automatically computes the residuals of the equations of motion of a multibody system whose kinematics is described in `ComputeMotion` and subjected to efforts described in `AddAppliedEfforts`.

5.2.5 Motion composition

The routine `ComposeMotion` is declared

```
void ComposeMotion(int ibody, int ibodyref)
```

and builds the global motion of body `ibody` from the one of the reference body `ibodyref` and the relative motion with respect to this reference body.

The procedure assumes that the global motion (`TOG`, `vG`, `aG`, `omega` and `omegad`) of the reference body and the relative motion (`TrefG`, `vGrel`, `aGrel`, `omegarel` and `omegadrel`) of the considered body have been previously defined in terms of the configuration parameters and their first and second time derivatives.

5.2.6 Provided routines for applied efforts

Some routines are provided to help the user in writing the routine `AddAppliedEfforts`.

Gravity

The routine `AddGravityForces` adds the gravity contribution of all the bodies of the system; it is declared as

```
void AddGravityForces(vec grav);
```

with `grav` the gravity vector, expressed in the global coordinate system.

Springs and dampers

The routine `AddSpringForce` implements the efforts exerted by a spring attached to two bodies; it is declared as

```
void AddSpringForce(double K, double L0,
                    int ibodyA, vec rA, int ibodyB, vec rB);
```

with `K` and `L0` the stiffness and the rest length of the spring, `ibodyA` and `ibodyB` the numbers of the bodies connected by the spring (i and j on figure 5.2), and `rA` and rB the vectors giving the coordinates of the attachment points **in the coordinate system of the corresponding body**.

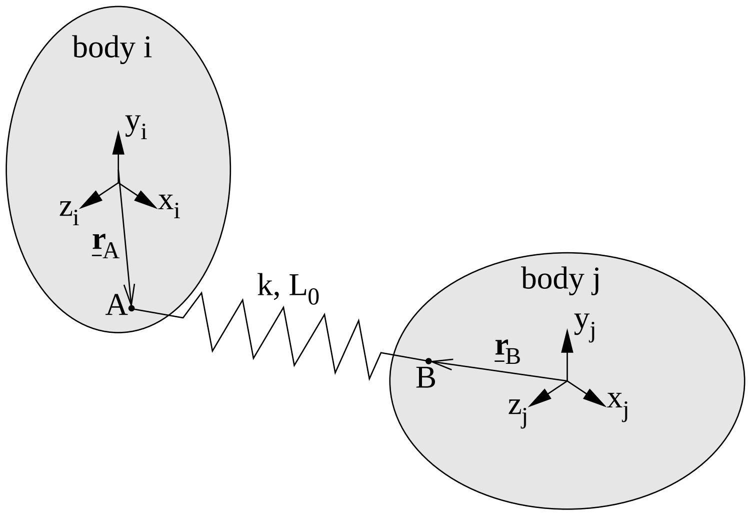


Figure 5.2: Spring between two bodies

The routine `AddDamperForce` implements the efforts exerted by a damper attached to two bodies; it is declared as

```
void AddDamperForce(double C,
                    int ibodyA, vec rA, int ibodyB, vec rB);
```

with `C` the damping coefficient of the damper; `ibodyA`, `ibodyB`, `rA` and `rB` have the same meaning as for the spring.

Contact

The user can specify a unilateral contact between a plane attached to a body and a point attached to another body. The element implements the contact stiffness (eventually nonlinear) as well as friction. The corresponding routine is specified as follows

```
void AddContactPlanePointForce(double Kn, double pk,
                               double Cdamp, double pd,
                               double ffrict, double vglim,
                               int ibodyplane, vec rplane, vec nplane,
                               int ibodypoint, vec rpoint)
```

with

- `Kn` the stiffness coefficient relative to the normal elastic contact force (cf. later for computation of contact force);
- `pk` the exponent of the penetration in the evaluation of the normal elastic contact force
- `Cdamp` the damping coefficient relative to normal contact force;
- `pd` the exponent of the penetration in the evaluation of the normal damping contact force
- `ffrict` the friction coefficient;
- `vglim` the limit slip velocity;
- `ibodyplane` the number of the body to which the plane is attached (body i in figure 5.3);
- `rplane` the vector coordinate of a point of the plane ($\underline{\mathbf{r}}_P$ in figure 5.3) with respect to the center of gravity of body `iplane`; this vector is expressed in the coordinate system attached to body `iplane`;
- `nplane` a vector normal to the plane, oriented outwards relatively to the material of body `iplane`; it is expressed in the coordinate system of the body `iplane`; this vector doesn't need to be unitary;
- `ibodypoint` the number of the body to which the contact point is attached (body j in figure 5.3);
- `rpoint` the vector coordinate of the contact point ($\underline{\mathbf{r}}_A$ in figure 5.3) with respect to the center of gravity of body `ipoint`; this vector is expressed in the coordinate system attached to body `ipoint`.

Practically, if pen represents the penetration of the contact point in the plane, the normal contact force is calculated as follows (in the direction of the normal vector)

$$F_n = K_n pen^{p_k} + C_{damp} * pen^{p_d} \frac{d\ pen}{dt} \tag{5.1}$$

A null value of p_d can be used for a damping coefficient independent of penetration but is not recommended for stability reasons.

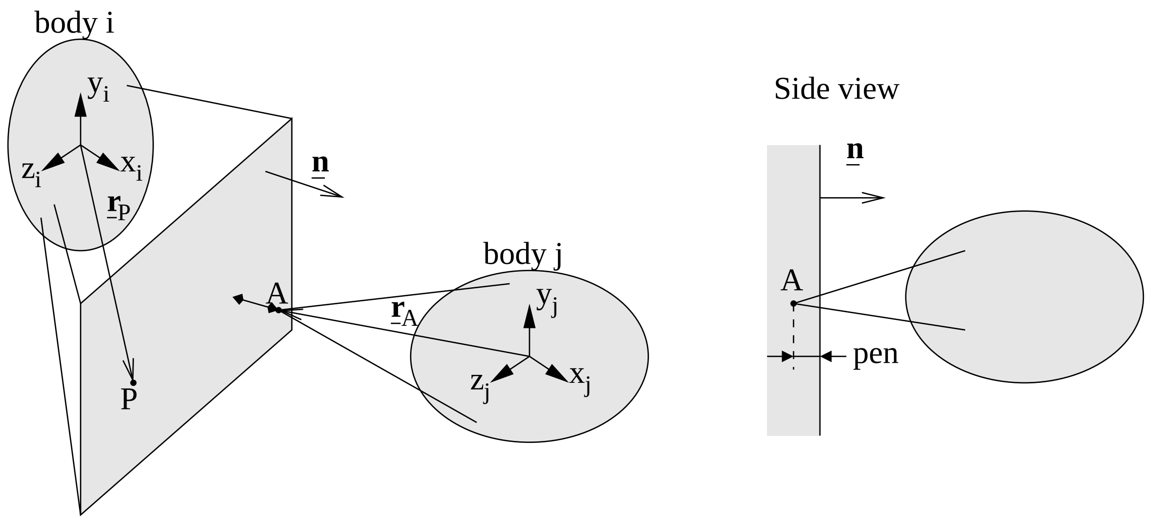


Figure 5.3: Contact between a plane and a point

On the other hand, if $\underline{\mathbf{v}}_g$ is the slip velocity of the contact point with respect to the plane, the tangential friction force is calculated as follows

$$\begin{aligned}\underline{\mathbf{F}}_t &= -fF_n * \frac{\underline{\mathbf{v}}_g}{v_{glim}} \text{ if } ||\underline{\mathbf{v}}_g|| < v_{glim} \\ &= -fF_n * \frac{\underline{\mathbf{v}}_g}{||\underline{\mathbf{v}}_g||} \text{ if } ||\underline{\mathbf{v}}_g|| \geq v_{glim} \qquad (5.2)\end{aligned}$$

Strictly speaking, adhesion never arises but the slip velocity is limited to v_{glim} as far as the tangential force is below the adhesion limit.

Tyre

The routine AddTyreEfforts allows the user to deal with vehicles. So far the ground is limited to the XY plane. The use of tyres needs a basic understanding of the classical tyre models. The one which is used in `EasyDyn` is the so-called model of the University of Arizona, developed by Gim Gwanghun, completely described in his Ph.D. thesis (Vehicle Dynamic Simulation with a Comprehensive Model for Pneumatic Tyres, Gim, Gwanghun, University of Arizona, 1988). It is an analytical model.

The routine is declared by

```
void AddTyreEfforts(int ibody, vec axe, structtyre tyre)
```

with

- `ibody` the number of the body representing the wheel (the user must give the wheel all necessary degrees of freedom, namely the axial rotation);
- `axe` a vector giving the direction of the rotation axis, expressed in the coordinate system of the body; this vector doesn't need to be unitary;

- `tyre` a variable of type `structtyre` gathering all physical data of the tyre.

The structure `structtyre` is declared in the following way, which is self explanatory

```
struct structtyre
{
double r1; // outer radius
double r2; // equivalent torus radius
double Kz; // vertical stiffness
double Cz; // vertical damping
double Fznom; // nominal vertical force on tyre
double Clongnom; // nominal longitudinal stiffness
double nlong; // Clong=Clongnom*(Fz/Fznom)^nlong
double Clatnom; // nominal cornering stiffness
double nlat; // Clat=Clatnom*(Fz/Fznom)^nlat
double Ccambernom; // nominal camber stiffness
double ncamber; // Ccamber=Ccambernom*(Fz/Fznom)^ncamber
double fClbs, fClbd; // Coulomb friction coefficients (static and dynamic)
};
```

As an indication, the longitudinal and cornering stiffnesses are worth, for a passenger car radial tyre, about 8 times the nominal vertical force. For the same type of tyre, the camber stiffness is about 0.6 times the nominal vertical force.

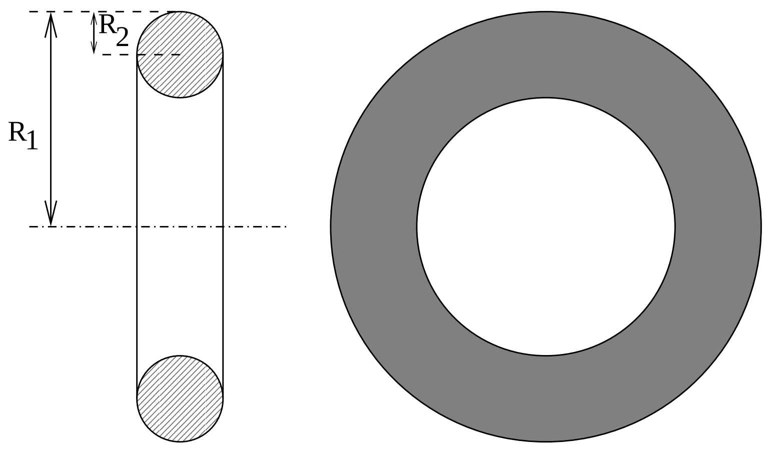


Figure 5.4: Tyre geometry

In figure 5.4, the parameters `r1` and `r2` are illustrated. It can be seen that the tyre is considered to be a torus. The research of the contact point is based on this assumption.

5.3 Examples

5.3.1 Mass-spring system

The mass-spring system presented in the previous chapter can also be simulated in a multibody way.

The corresponding code is given below (file `spdp.cpp` in the `examples\testmbs` directory).

```
// Simulation of a mass-spring-damper system

#define EASYDYNMBSMAIN // to declare the global variables
#include <EasyDyn/mbs.h>

// Definition of global application variables
vec ut; // direction of motion
// The result should be the same whatever ut

//--------------------------------------------------------------------

void WriteDataHeader(ostream &OutFile)

{
WriteStateVariablesHeader(OutFile);
OutFile << endl;
}

//--------------------------------------------------------------------

void SaveData(ostream &OutFile)

{
SaveStateVariables(OutFile);
OutFile << endl;
}

//--------------------------------------------------------------------

void SetInertiaData()

{

// Inertia for body 0 (=ground in this example)
body[0].mass=1;
body[0].PhiG.put(1,1,1); // no impact on the result anyway
// Inertia for body 1
body[1].mass=1;
body[1].PhiG.put(1,1,1); // no impact on the result anyway

}

//---------------------------------------------------------------------

void ComputeMotion()

{

// Kinematics for body 0
// =ground

// Kinematics for body 1
body[1].T0G=Tdisp(q[0]*ut);
body[1].vG=qd[0]*ut;
body[1].aG=qdd[0]*ut;

}

//---------------------------------------------------------------------

void AddAppliedEfforts()

{
// Contribution of external applied forces
vec gravity=39.478417*ut;
AddGravityForces(gravity);
AddSpringForce(39.478417,2,1,vcoord(0,0,0),0,2*ut);
// Uncomment the following if you want to add a damper
//AddDamperForce(1,0,2*ut,1,vcoord(0,0,0));
}

//---------------------------------------------------------------------

void ComputeResidual()

{
ComputeResidualmbs();
}

//---------------------------------------------------------------------
```

```
int main()

{
  // Initialization and memory allocation
  nbrdof=1;
  nbrbody=2;
  application="spdp";
  InitEasyDynmbs();
  // Initial configuration
  // Everything =0, so nothing to do :-)
  // Let's go !
  ut.put(1.1,2.2,3.3); ut.unite();
  NewmarkIntegration(5,0.01,0.005);
  // the clean way to finish
  EndEasyDynmbs();
}
//--------------------------------------------------------------------
```

It can be seen in the code that the motion can be defined along any unit vector `ut`, for the sake of testing the library. Moreover, a damper can be easily added.

5.3.2 Double pendulum

Let us consider the double pendulum illustrated in figure 5.5. 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.

The homogeneous transformation matrices giving the situation of the two bodies can be easily established as

$$\mathbf{T}_{0,1} = \begin{pmatrix} c_1 & -s_1 & 0 & \frac{l_1}{2} s_1 \\ s_1 & c_1 & 0 & -\frac{l_1}{2}\, c_1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.3}$$

$$\mathbf{T}_{0,2} = \begin{pmatrix} c_{12} & -s_{12} & 0 & l_1 s_1 + \frac{l_2}{2} s_{12} \\ s_{12} & c_{12} & 0 & -l_1 c_1 - \frac{l_2}{2}\, c_{12} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.4}$$

with l_1 and l_2 the lengths of the arms, $c_1 = \cos(q_1)$, $s_1 = \sin(q_1)$, $c_{12} = \cos(q_1 + q_2)$ and $s_{12} = \sin(q_1 + q_2)$.

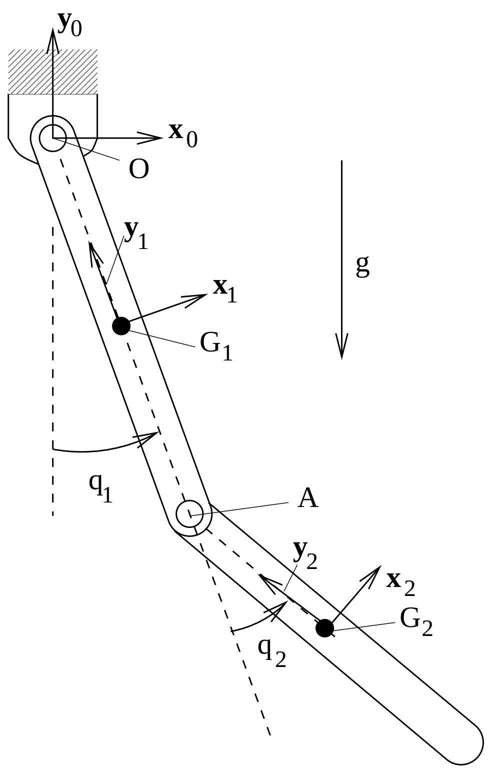


Figure 5.5: Double pendulum

They also correspond to a succession of rotations and translations and can be written in this form, from the utility routines provided in the `vec` module (cf. listing).

The translation and rotation velocities of each body are given in vector form by

$$\underline{\omega}_1 = \dot{q}_1 \vec{z}_0 \quad (5.5)$$

$$\underline{\mathbf{v}}_{G_1} = \underline{\dot{\omega}}_1 \times \underline{\mathbf{OG}}_1 \quad (5.6)$$

$$\underline{\omega}_2 = (\dot{q}_1 + \dot{q}_2)\vec{z}_0 \quad (5.7)$$

$$\underline{\mathbf{v}}_{G_2} = \underline{\dot{\omega}}_1 \times \underline{\mathbf{OA}} + \underline{\omega}_2 \times \underline{\mathbf{AG}}_2 \quad (5.8)$$

The accelerations are written

$$\underline{\dot{\omega}}_1 = \ddot{q}_1 \vec{z}_0 \quad (5.9)$$

$$\underline{\mathbf{a}}_{G_1} = \underline{\dot{\omega}}_1 \times \underline{\mathbf{OG}}_1 + \underline{\omega}_1 \times (\underline{\omega}_1 \times \underline{\mathbf{OG}}_1) \quad (5.10)$$

$$\underline{\dot{\omega}}_2 = (\ddot{q}_1 + \ddot{q}_2)\vec{z}_0 \quad (5.11)$$

$$\begin{aligned}\underline{\mathbf{a}}_{G_2} &= \underline{\dot{\omega}}_1 \times \underline{\mathbf{OA}} + \underline{\omega}_1 \times (\underline{\omega}_1 \times \underline{\mathbf{OA}}) \\ &\quad + \underline{\dot{\omega}}_2 \times \underline{\mathbf{AG_2}} + \underline{\omega}_2 \times (\underline{\omega}_2 \times \underline{\mathbf{AG_2}}) \quad (5.12)\end{aligned}$$

The gravity is the only force acting on the system.

Here is the complete code to simulate this double pendulum wih the `mbs` module (file `dp2.cpp` in the `examples\testmbs` directory)

```
// Simulation of a double pendulum with relative coordinates

#define EASYDYNMBSMAIN // to declare the global variables
#include <EasyDyn/mbs.h>

// Definition of global application variables
double l1=1.2, l2=1.1;

//----------------------------------------------------------------------

void WriteDataHeader(ostream &OutFile)

{
WriteStateVariablesHeader(OutFile);
OutFile << endl;
}

//----------------------------------------------------------------------

void SaveData(ostream &OutFile)

{
SaveStateVariables(OutFile);
OutFile << endl;
}

//----------------------------------------------------------------------

void SetInertiaData()

{

// Inertia for body 0
body[0].mass=1.1;
body[0].PhiG.put(1,1,body[0].mass*l1*l1/12);
// Inertia for body 1
body[1].mass=0.9;
body[1].PhiG.put(1,1,body[1].mass*l2*l2/12);

}

//----------------------------------------------------------------------

void ComputeMotion()
```

```
{

// Kinematics for body 0
body[0].TOG=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 OG1=body[0].TOG.e;
body[0].vG=(body[0].omega^OG1);
body[0].aG=(body[0].omegad^OG1)+(body[0].omega^(body[0].omega^OG1));
// Kinematics for body 1
body[1].TOG=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].TOG.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));
}

//----------------------------------------------------------------------

void AddAppliedEfforts()

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

//----------------------------------------------------------------------

int main()

{
  // Initialisation and memory allocation
  nbrdof=2;
  nbrbody=2;
  application="dp2";
  InitEasyDynmbs();
  // Initial configuration
  q[1]=1;
  // Let's go !
  NewmarkIntegration(5,0.01,0.005);
  // The clean way to finish !
  EndEasyDynmbs();
}
//-----------------------------------------------------------------
```

5.3.3 Double pendulum with a relative motion

In the case of the double pendulum presented in the previous section, the user can also choose to define the motion of the second body with respect to the first one. The function `ComputeMotion` will then be written in the following way.

```
void ComputeMotion()
{
// Kinematics for body 0
body[0].TOG=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 OG1=body[0].TOG.e;
body[0].vG=(body[0].omega^OG1);
body[0].aG=(body[0].omegad^OG1)+(body[0].omega^(body[0].omega^OG1));
// Kinematics for body 1
body[1].TrefG=Tdisp(0,-0.5*l1,0)*Trotz(q[1])*Tdisp(0,-0.5*l2,0);
body[1].omegarel=vcoord(0,0,qd[1]);
body[1].omegadrel=vcoord(0,0,qdd[1]);
vec AG2=body[1].TrefG.R*vcoord(0,-0.5*l2,0);
body[1].vGrel=(body[1].omegarel^AG2);
body[1].aGrel=(body[1].omegadrel^AG2)
          +(body[1].omegarel^(body[1].omegarel^AG2));
ComposeMotion(1,0);
}
```

It is important to notice that the relative motion is completely written in the axes of the intermediary reference frame.

5.4 Module `mbs` and animation

The motion of the multibody system resulting from a time integration can be animated with `EasyAnim`. For that purpose you will have to

1. globally declare a variable of type `scene` and an output stream for saving the images;
2. create graphical shapes attached to the bodies (practically to the homogeneous transformation matrix describing the motion of the body)
3. add the shapes to the scene;
4. save the structure of the scene (specified in a file with extension `.vol`);

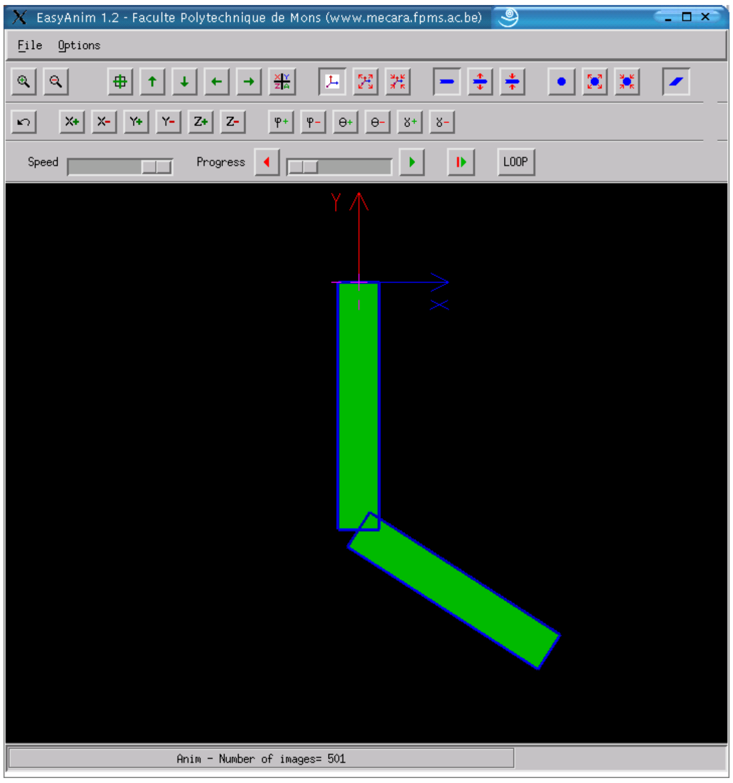

EasyAnim 1.2 - Faculte Polytechnique de Mons (www.mecara.fpms.ac.be)
File Options
Speed
Progress
LOOP
Y
X
Anim - Number of images= 501

Figure 5.6: Visualization of the double pendulum

5. save the successive node positions in a file with extension .van

It is much easier to show how this is implemented on the example of the double pendulum (cf. example dp2visu.cpp).

1. globally declare a variable of type `scene` and an output stream for saving the images; the beginning of the file will look like this

```
#define EASYDYNMBSMAIN // to declare the global variables
#include <EasyDyn/mbs.h>
#include <EasyDyn/visu.h> // so that the scene can be declared

#include <fstream>
using namespace std;

scene thescene;
ofstream VanFile; // this file will be used to save the node positions
```

2. create graphical shapes attached to the bodies (practically to the homogeneous transformation matrix describing the motion of the body)

```
  shape *s1,*s2;
```

```
    vec e1(-0.1,-0.6,-0.1), e2(0.1,0.6,0.1);
    s1=new box(&(body[0].TOG),e1,e2,1,2);
    vec e3(-0.1,-0.55,-0.1),e4(0.1,0.55,0.1);
    s2=new box(&(body[1].TOG),e3,e4,1,2);
```

3. add the shapes to the scene;

   ```
   thescene.AddShape(s1);
   thescene.AddShape(s2);
   ```

4. save the structure of the scene (specified in a file with extension `.vol`);

   ```
   ComputeMotion(); // to get relevant position matrices
   thescene.CreateVolFile("dp2visu.vol");
   ```

5. save the successive node positions in a file with extension `.van`; this requires three operations

   - open the file before launching the integration

     ```
     VanFile.open("dp2visu.van");
     ```

   - save regularly the node positions, the easiest way being to add a line to the routine `SaveData` which is automatically called by the integrator

     ```
     void SaveData(ostream &OutFile)
       {
       SaveStateVariables(OutFile);
       OutFile << endl;
       thescene.WriteCoord(VanFile);
       }
     ```

   - do not forget to close the file before closing the application

     ```
       EndEasyDynmbs();
       VanFile.close();
     ```

The expected result under `EasyAnim` is illustrated in figure 5.6.

5.5 Provided routine for animation of eigen modes

If a linear analysis of the multibody system has been performed by a call to `ComputePoles`, the files for animation can be created by a simple call to the routine

```
void CreateVmoFile(scene &sc)
```

The routine relies on the existence of the file *application*`.mod` as well as the one of a `scene` variable (`sc`) linking graphical objects to the bodies.

The routine creates the files *application*`.vol` and *application*`.vmo` which allow the visualization with `EasyAnim` (from version 1.2).

Chapter 6

The `CAGeM` module

6.1 Introduction

Even with the help of the vector library, the risk of mistake remains high when developing the expressions of velocities and accelerations. Moreover the kinematics remains problematic for an unexperienced user.

A supplementary tool, called `CAGeM` (*Computer Aided Generation of Motion*), has been developed to help the users of `EasyDyn`. This module is a simple `MuPAD` script which automatically builds a C++ application using the `mbs` part of `EasyDyn`. For the kinematic part, `CAGeM` symbolically derives the expression of the homogeneous transformation matrices and builds the expressions of the translation and rotation velocities and accelerations.

`CAGeM` is based on the fact that the homogeneous transformation matrix

$$\mathbf{T}_{0,i} = \begin{pmatrix} \mathbf{R}_{0,i} & \{\underline{\mathbf{e}}_i\}_0 \\ 0\ 0\ 0 & 1 \end{pmatrix} \tag{6.1}$$

yields, after time derivation, the rotation and the translation velocities as

$$\{\underline{\mathbf{v}}_i\}_0 = \frac{d}{dt}\{\underline{\mathbf{e}}_i\}_0 \tag{6.2}$$

$$\{\tilde{\underline{\omega}}_i\}_0 = \dot{\mathbf{R}}_{0,i} \cdot \mathbf{R}_{0,i}^T \tag{6.3}$$

where $\{\tilde{\underline{\omega}}_i\}_0$ is the skew symmetric matrix representing the vector product, such as:

$$\{\underline{\mathbf{a}}\} = \begin{pmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{pmatrix} \tag{6.4}$$

$$\tag{6.5}$$

One further derivation naturally leads to the accelerations

$$\{\underline{\mathbf{a}}_i\}_0 = \frac{d}{dt}\{\underline{\mathbf{v}}_i\}_0 \tag{6.6}$$

$$\{\underline{\dot{\omega}}_i\}_0 = \frac{d}{dt}\{\underline{\omega}_i\}_0 \tag{6.7}$$

These relations can be easily obtained by using symbolic software. This method gives an alternative to vector calculus to build the kinematics.

6.2 Installation

The `MuPAD` software is the result of several years of research at the *University of Paderborn* (Germany). Since 1997 the development of interfaces and special applications of `MuPAD` is done in a tight cooperation with *SciFace Software*. It can be downloaded free of charge from the net at `http://www.mupad.de` and the installation is very easy (on Windows and on Linux).

`MuPAD` is a general purpose computer algebra system for symbolic and numerical computations. The librairy is open source so that users can examine the code, implement their own routines and data types easily and can also dynamically link C/C++ compiled modules for raw speed and flexibility. It also offers features comparable to its commercial counterparts like `Mathematica` or `MathCad`.

For the installation of `CAGeM` on Windows, the user copies the file `cagem.mu` anywhere on the disc and give the corresponding directory on `MuPAD` session (menu `view >> options >> kernel` in `User-defined startup file` prompt).

On Linux, the installation is slightly different. The user must create a directory .mupad in his home directory, e.g. `mkdir ~/.mupad` and a new file userinit.mu, e.g. touch `~/.mupad/userinit.mu`. This file must be edited to specify the `MuPAD` commands to execute each time `MuPAD` starts. In this case, the only command is `read("XXX/cagem.mu")` where `XXX` is the complete path of the file `cagem.mu`.

6.3 Structure of the user's data file

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

- the number of bodies, the number of configuration parameters and, optionally, the number of dependent parameters;
- the inertia data of each body;
- the expression of the homogeneous transformation matrices (`TOG[i]` or `TrefG[i]` according as the motion is defined with respect to the ground or relatively to another body) of each body, expressed in terms of the chosen configuration parameters;

- the initial conditions;
- the gravity vector.

When the relative motion is defined with the variable `TrefG[i]`, the user must indicate the number of the reference body with the parameter `BodyRef[i]`. In this way, the C++ code will contain all the information to build the absolute motion (relative kinematics and procedure `ComposeMotion()`).

According to the definition of the class `tiner` of the `EasyDyn` vector library, the components of the inertia tensor are represented only by the 6 variables `Ixx[i]`, `Iyy[i]`, `Izz[i]`, `Ixy[i]`, `Ixz[i]` and `Iyz[i]`, i representing the number of body.

In the same way, the equivalent functions `Trotx(theta)`, `Troty(theta)`, `Trotz(theta)` and `Tdisp(x,y,z)` exist on `CAGeM` and return an homogeneous transformation matrix. A supplementary function `Trotn(nx,ny,nz,theta)` is included and defines a rotation about an axis whose direction cosines are equal to n_x, n_y and n_z

$$\begin{pmatrix} C+\mathtt{nx}^2\,(1-C) & \mathtt{nx}\,\mathtt{ny}\,(1-C)-\mathtt{nz}\,S & \mathtt{nx}\,\mathtt{nz}\,(1-C)-\mathtt{ny}\,S & 0 \\ \mathtt{nz}\,S+\mathtt{nx}\,\mathtt{ny}\,(1-C) & C+\mathtt{ny}^2\,(1-C) & \mathtt{ny}\,\mathtt{nz}\,(1-C)-\mathtt{nx}\,S & 0 \\ \mathtt{nx}\,\mathtt{nz}\,(1-C)-\mathtt{ny}\,S & \mathtt{nx}\,S+\mathtt{nx}\,\mathtt{nz}\,(1-C) & C+\mathtt{nz}^2\,(1-C) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{6.8}$$

where $C = \cos(\mathtt{theta})$, $S = \sin(\mathtt{theta})$.

Optionally various flags can be activated by the user to select different options defined in the table 6.3.

6.4 Calling CAGeM

To start, call CAGeM with the simple command `cagem()`. A new window appears to request the file (path and name) to analyze. In function of the options chosen by the user on the data file the program gives on screen the CPU–time needed of each operation.

6.5 Examples

6.5.1 Double pendulum

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

```
// Title.
title:="Simulation of a double pendulum":

// Definition of nbrdof  : number of degrees of freedom,
//               nbrbody : number of bodies,
//               nbrdep  : number of dependent parameter(s) (optional).
```

Flags	Functionality	Default
SIMPLIFY	Simplification of the various kinematics terms. This option, if equal to zero, is used where the simplification needs a long CPU–time.	1
FORCES	There exist applied forces other then gravity, expected to be defined by the user in the file `*.AppEff.cpp`, which will be included in the main code	0
ANIM	Some lines of code are added so as to create animation files during integration (a basic shape is attached to each body)	0
STATIC	The equilibrium position is determined before the integration	0
PLOT	A gnuplot script file is generated to plot the evolution of the configuration parameters and their first and second time derivatives	0
LATEX_FR	Creation of LaTeX 2_ε report in French	0
LATEX_EN	Creation of LaTeX 2_ε report in English	0

Table 6.1: Different flags in CAGeM

```
nbrdof:= 2:
nbrbody:= 2:
nbrdep:= 0:

// Gravity vector.
gravity[1]:=0:
gravity[2]:=-9.81:
gravity[3]:=0:

// Some constant for geometry.
l0:=1:
l1:=2:

// Inertia characteristics.
mass[0]:=1:
mass[1]:=2:
Ixx[0]:=1:
Ixx[1]:=1:
Iyy[0]:=1:
Iyy[1]:=1:
Izz[0]:=l0^2/12*mass[0]:
Izz[1]:=l1^2/12*mass[1]:

// The kinematic part only comes down to.
TOG[0] := Trotz(q[0]-PI/2) * Tdisp(l0/2,0,0):
TOG[1] := Trotz(q[0]-PI/2) * Tdisp(l0,0,0) * Trotz(q[1]) * Tdisp(l1/2,0,0):
// or TrefG[1] := Tdisp(l0/2,0,0) * Trotz(q[1]) * Tdisp(l1/2,0,0):
//    BodyRef[1] := 0:
```

```
// Initial conditions.
qi[0]:=3.14/2:

// Simulation condition.
FinalTime:=5:
StepSave:=0.01:
StepMax:=0.005:

// Optional flags.
SIMPLIFY:=1:
ANIM:=1:
PLOT:=1:
LATEX_EN:=1:
```

For the sake of illustration, the code generated to compute the accelerations of the two bodies looks like this

```
body[0].aG.x = (1.0/2.0)*qdd[0]*cos(q[0]) - (1.0/2.0)*(qd[0]*qd[0])*sin(q[0]) ;
body[0].aG.y = (1.0/2.0)*qdd[0]*sin(q[0]) + (1.0/2.0)*(qd[0]*qd[0])*cos(q[0]) ;
body[0].aG.z = 0.0 ;
body[1].aG.x = qdd[0]*cos(q[0]) - (qd[0]*qd[0])*sin(q[0]) + qdd[0]*cos(q[0] + q[1])
             + qdd[1]*cos(q[0] + q[1]) - (qd[0]*qd[0])*sin(q[0] + q[1])
             - (qd[1]*qd[1])*sin(q[0] + q[1]) - 2.0*qd[0]*qd[1]*sin(q[0] + q[1]) ;
body[1].aG.y = qdd[0]*sin(q[0]) + (qd[0]*qd[0])*cos(q[0]) + qdd[0]*sin(q[0] + q[1])
             + qdd[1]*sin(q[0] + q[1]) + (qd[0]*qd[0])*cos(q[0] + q[1])
             + (qd[1]*qd[1])*cos(q[0] + q[1]) + 2.0*qd[0]*qd[1]*cos(q[0] + q[1]) ;
body[1].aG.z = 0.0 ;
```

6.5.2 Slider-crank mechanism

The symbolic capabilities of MuPAD allow the user to define intermediary parameters. For example, with the slider–crank mechanism illustrated in figure 6.1, the parameters α and x can be expressed univoquely in terms of angle q_0 by

$$\alpha = \arcsin\left(\frac{l_1 \sin(q_0)}{l2}\right) \tag{6.9}$$

and

$$x = l_1 \cos(q_0) + l_1 \cos(\alpha) \tag{6.10}$$

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

```
// Title.
title:="Simulation of a slider-crank mechanism":

// Definition of nbrdof  : number of degrees of freedom,
//               nbrbody : number of bodies,
//               nbrdep  : number of dependent parameter(s) (optional).
```

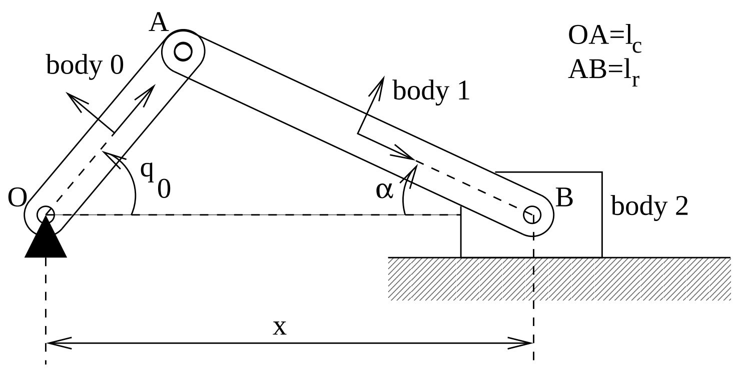


Figure 6.1: Slider–crank mechanism

```
nbrdof:= 1:
nbrbody:= 3:
nbrdep:= 0:

// Gravity vector.gravity[1]:=0:
gravity[2]:=-9.81:
gravity[3]:=0:

// Some relations for geometry.
lc := 1:
lr := 2:
alpha := arcsin(lc*sin(q[0])/lr):
x2 := lc*cos(q[0]) + lr*cos(alpha):

// Inertia characteristics.
mass[0]:=1:
mass[1]:=2:
mass[2]:=5:
Ixx[0]:=1:
Iyy[0]:=1:
Izz[0]:=mass[0]*lc^2/12:
Ixx[1]:=1:
Iyy[1]:=1:
Izz[1]:=mass[1]*lr^2/12:
Ixx[2]:=1:
Iyy[2]:=1:
Izz[2]:=1:

// The kinematic part only comes down to.
TOG[0]:= Trotz(q[0]) * Tdisp(lc/2,0,0):
TOG[1] := Tdisp(x2,0,0) * Trotz(-alpha) * Tdisp(-lr/2,0,0):
TOG[2] := Tdisp(x2,0,0):

// Initial conditions.
qi[0]:=1:

// Simulation condition.
FinalTime:=6:
```

```
StepSave:=0.01:
StepMax:=0.005:

// Optional flags.
SIMPLIFY:=1:
ANIM:=1:
PLOT:=1:
LATEX_EN:=1:
```

For this example, the following relation gives a view of the results complexity

$$\{\underline{\mathbf{v}}_{G,S_1}\} = \left\{ \begin{array}{c} \dfrac{-\frac{\dot{q}_0\,\sin(2\,q_0)}{2} - 4\,\dot{q}_0\,\sin(q_0)\sqrt{\frac{\cos(2\,q_0)}{8}+7/8}}{4\sqrt{\frac{\cos(2\,q_0)}{8}+7/8}} \\ \dfrac{\dot{q}_0\,\cos(q_0)}{2} \\ 0 \end{array} \right\} \tag{6.11}$$

The response of the system subjected to gravity has been simulated from initial conditions $q_0 = 1\,rad$ and $\dot{q}_0 = 0\,rad/s$. The time evolution of the dodies displacements, represented in figure 6.2, shows that the mechanism comes back to the same position after one oscillation, as there is no energy dissipation.

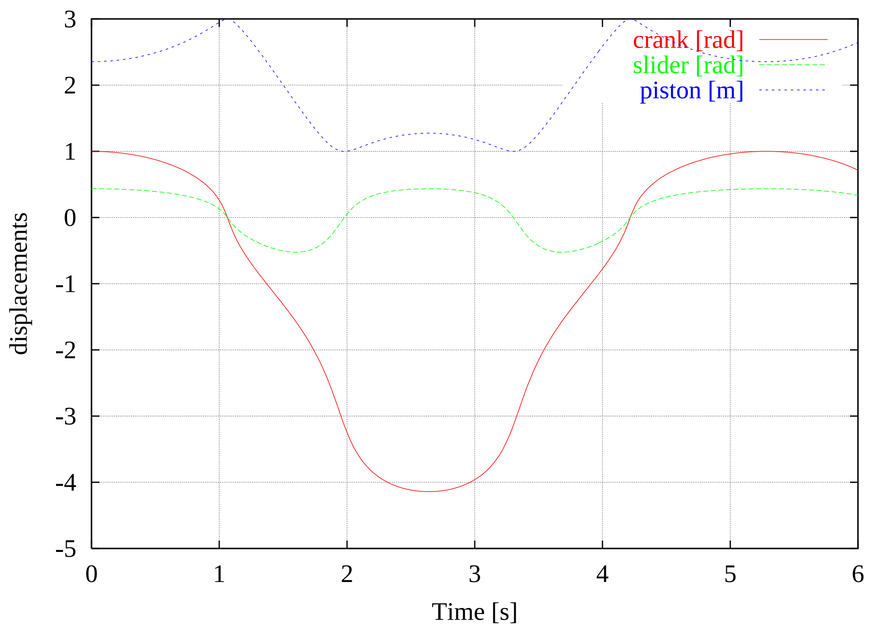


Figure 6.2: Simulation of the slider–crank mechanism

Another way to define the slider–crank mechanism consist in using dependent paramaters. So the two variables `alpha` and `x2` can be remplaced by the two parameters `p[0]` and `p[1]` defined in the same manner, after the script relative to the homogeneous matrices :

```
// The kinematic part only comes down to.
TOG[0]:= Trotz(q[0]) * Tdisp(lc/2,0,0):
TOG[1] := Tdisp(p[1],0,0) * Trotz(-p[0]) * Tdisp(-lr/2,0,0):
TOG[2] := Tdisp(p[1],0,0):

// The dependent parameters.
p[0] := arcsin(lc*sin(q[0])/lr):
p[1] := lc*cos(q[0]) + lr*cos(p[0]):
```

It is of course necessary to specify the number of dependent parameters on top of the script (`nbrdep:=2:`).