BASILE-MAPLE interface: a link between CACSD package and computer algebra system

Claude Gomez, Georges Le Vey, C. Rougerie

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports Techniques

N° 130

Programme 5
Automatique, Productique,
Traitement du Signal et des Données

BASILE-MAPLE INTERFACE : A LINK BETWEEN CACSD PACKAGE AND COMPUTER ALGEBRA SYSTEM

Claude GOMEZ
Georges LE VEY
Christine ROUGERIE

Juillet 1991

BASILE-MAPLE Interface: a Link between CACSD Package and Computer Algebra System

Interface *BASILE-MAPLE* : Un lien entre système de CAO pour l'automatique et système de calcul formel

Claude Gomez* Georges Le Vey† Christine Rougerie‡

*Projet META2, INRIA, Domaine de Voluceau, Rocquencourt, B.P. 105, 78153 Le Chesnay Cédex, France.
†SIMULOG S.A., 1 rue James Joule, 78182 Saint Quentin en Yvelines Cédex, France.
‡IBSI, 122 Avenue de Hambourg, 13008 Marseille, France.

Abstract

BASILE is a CACSD package of great efficiency for numerical computations. On another hand, automatic control is a field in which a lot of mathematical objects have to be manipulated in a symbolic way. For that, powerful computer algebra systems such as *MAPLE* exist. We describe in this paper an interface beween *BASILE* and *MAPLE*, allowing various data communication and combining in this way the respective powerfulness of numerical and symbolic computations.

Résumé

BASILE est un système de CAO pour l'automatique qui réalise principalement des calculs numériques. Pour résoudre les problèmes d'automatique, il est très souvent nécessaire d'effectuer des manipulations symboliques. Le but de l'interface présentée est de relier numérique et symbolique en permettant l'échange de données entre *BASILE* et le système de calcul formel *MAPLE*. Ainsi, l'utilisateur a la possibilité de mélanger la puissance du calcul symbolique et l'efficacité du calcul numérique.

1 Specifications

1.1 Introduction

BASILE (see [1]) is a CACSD package, essentially doing numerical computations. However, in many cases, one needs to do some symbolic computations, particularly in automatic control but also while dealing with general mathematical quantities, exact differentiation or integration. We will show, through a few examples, how such needs appear when trying to solve problems with *BASILE*. Among the numerous implications of computer algebra towards numerical purposes, let us mention first the symbolic differentiation as one of major interest for *BASILE*. Another interest, as we shall see in the sequel, is the possibility to automatically generate numerical codes (FORTRAN, C) from a formal description of dynamical systems, for simulation purposes (for example see [2]). We use *MAPLE* as the computer algebra system.

1.2 Examples

We present now some practical examples, and we will identify which data must be transferred between *BASILE* and *MAPLE*.

1.2.1 Example 1: derivative computation

BASILE knows how to numerically integrate explicit or semi-explicit differential systems. For that, the user must supply, through a FORTRAN program or a *BASILE* macro, the right member of the system under study. Moreover, it may be of interest to simultaneously give the derivative of the function with respect to the state vector (the jacobian matrix). Of course, such computations may be derived by hand but it is quite obvious that with a dozen of equations and variables, the preceding computations become tedious and always need precise verifications, for the symbolic manipulations and while writing the numerical code. So, we would like to use a computer algebra system for such derivative computations. Such computations appear in optimal control problems too, to calculate the gradient of a functional with respect to the control, by the state adjoint method. In such situations, we would like to transfer *BASILE* macros to *MAPLE* and to load a *BASILE* macro after processing by *MAPLE*. Let us give a simple example.

Suppose we have loaded into *BASILE* the macro describing the following function:

$$f(x) = \sin(x_1 + x_2) - 3(x_1 x_2 + e^{x_3})/(x_2 + x_3)$$

The corresponding *BASILE* macro is the following:

```
//<f>=f(x)
f=sin(x(1)+x(2))-3*(x(1)*x(2)+exp(x(3)))/(x(2)+x(3));
//end
```

As we mentioned previously, all the computations could be done by hand, or derived separately with *MAPLE* ; we could have written then the desired *BASILE* macro. Nevertheless, an automatic procedure would be of great value for bigger applications. In the example, we want to transfer the body of the *BASILE* macro to *MAPLE*, do symbolic differentiation of the function and load into *BASILE* the macro giving the derivative:

```
//<out>=grad(x)
out=<cos(x(1)+x(2))-3*x(2)/(x(2)+x(3)),cos(x(1)+x(2))-3*x(1)/(x(2)+x(3..
))+3*(x(1)*x(2)+exp(x(3)))/(x(2)+x(3))**2,-3*exp(x(3))/(x(2)+x(3))+3*(..
x(1)*x(2)+exp(x(3)))/(x(2)+x(3))**2>
//end
```

1.2.2 Example 2: change of variables

Change of variables is a frequent operation while dealing with mathematical problems. For example it may occur in order to reduce the size of a system or to give a better condition number. *BASILE* does not know how to do such an operation, except for a limited class of data. On the other hand, it is very easy to perform it with *MAPLE*. In that case, the data we want to transfer to *MAPLE* is again the body of a *BASILE* macro (description of a system of equations). Then the desired substitutions are done with *MAPLE* and we want to get a new *BASILE* macro describing the system after the relevant change of variable.

The following system describes the evolution of a simple planar pendulum in a normal coordinates system. λ is the tension of the pendulum, L is the length of the pendulum and g is the gravitational constant.

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= -\lambda x_1 \\ \dot{x}_3 &= x_4 \\ \dot{x}_4 &= -\lambda x_3 - g \\ 0 &= x_1^2 + x_3^2 - L^2 \end{aligned}$$

It is well known that eliminating the constraints leads to the following system of equations:

$$\begin{aligned} \dot{\phi}_1 &= \phi_2 \\ \dot{\phi}_2 &= g/L \sin \phi_1 \end{aligned}$$

where :

$$\begin{aligned} x_1 &= L \sin \phi_1 \\ x_3 &= -L \cos \phi_1 \end{aligned}$$

The *BASILE* macros describing the two systems are as follows:

For the first system:

```
//<res>=chvar(t,x,xd)
r=-xd(1)+x(2);
r=<r;-xd(2)-lambda*x(1)>;
r=<r;-xd(3)+x(4)>;
r=<r;-xd(4)-lambda*x(3)-g>;
//end
```

For the second system:

```
//<phid>=chvar2(t,phi)
phid(1)=phi(2);
phid(2)=g/l*sin(phi(1));
//end
```

Starting from the first as input, we want the second as an output. This example is similar to that in 1.2.1, as far as the data to be transferred between *BASILE* and *MAPLE* are concerned. Only the symbolic manipulations change.

1.2.3 Example 3: analytical expression of integrals or solution of particular O.D.E's

In a number of situations, it may be useful to get the exact expression of an indefinite integral. If a program evaluates a great number of times such an integral, much computing time may be saved as opposed to the numerical evaluation of the integral each time. Another interesting application is the exact solution (when it exists) of some differential equations or systems of equations, for example through the Laplace Transform. In such situations, the data to be transferred are again macros, for the description of the function to integrate or the equation to solve.

1.2.4 Example 4: unknown data types in *BASILE*

Suppose you want to study a transfer matrix $\frac{p_{ij}(s)}{q_{ij}(s)}$, depending on a formal parameter α. This data type does not exist in *BASILE*. We could plot a parameterized Bode plot the following way. First, let us compute $\frac{p_{ij}(s)}{q_{ij}(s)}$ as a function of α in *MAPLE* ; in a second step, we can compute and plot the frequency response in *BASILE*. In that case, the data to be transferred is a list that gives the rational matrix (remember that rational matrices are given as lists in *BASILE*).

1.3 Conclusion

The preceding examples allow the distinction of:

- Several data types that we need to transfer from *BASILE* to *MAPLE*:

  macros

  polynomials or lists representing rationals

  numerical data (scalars, vectors, matrices).

- Several ways to get back the relevant information into *BASILE*:

  macros to load into *BASILE* (mainly for non linear functions description)

  polynomials or rationals (*BASILE* lists)

  numerical data (scalars, vectors, matrices).

As we can see from this list, all the essential data types of *BASILE* are to be transferred to *MAPLE*.

2 User's Guide

2.1 Main features

In the sequel, the data transfer between the two packages *BASILE* and *MAPLE* will be held through files, transparently for the user.

For a comfortable use, the user should activate *BASILE* and *MAPLE* simultaneously on his workstation, with his favorite window manager. The two processes run each in a window. But a sequential use of the two processes is possible too when no multiple window terminal is available.

The following *BASILE* functionalities are given to the user:

tomaple to send data to *MAPLE*

frmaple to get data from *MAPLE*.

The following *MAPLE* functionalities are given to the user:

tobasile to send data to *BASILE*

frbasile to get data from *BASILE*.

The precise definitions of those functions are given hereunder.

2.2 Main functions of the interface

2.2.1 *BASILE* Functions

tomaple(var,string) takes a *BASILE* variable `var` as input and writes its *MAPLE* syntax translation into a temporary file. `string` is a character string.

If the *BASILE* variable `var` is the name of a *BASILE* macro, a *MAPLE* procedure with `string` as its name and the *MAPLE* syntax translation of the macro as its body is generated. Else, a *MAPLE* variable with `string` as its name and the value of the *BASILE* expression as its value is generated.

MAPLE expressions generated this way may be loaded into *MAPLE* by the *MAPLE* command `frbasile()`. The temporary file is destroyed after the execution of this command.

Successive calls to the function `tomaple` are possible, allowing several expressions to be stored in the temporary file. Each activation of `frbasile()` destroys the temporary file.

frmaple() loads *BASILE* macros generated by `tobasile()` from *MAPLE* and destroys the corresponding temporary file.

clmaple() destroys the temporary file where *BASILE* to *MAPLE* translations have been stored.

A complete description appears in table 1.

2.2.2 *MAPLE* Procedures

tobasile(exp,name{,arglist}) takes as input a *MAPLE* expression `exp` (it may be a procedure name) and writes the *BASILE* macro translation into a temporary file. The argument `name` is the name of the generated *BASILE* macro. The argument `arglist` is the argument list of the *BASILE* macro and is optional. If `arglist` is not given and `exp` is a procedure name, the arguments of the *BASILE* macro and those of the procedure are the same, else the argument list of the *BASILE* macro is empty.

The *BASILE* macros generated may be loaded into *BASILE* with the *BASILE* command `frmaple()`. The temporary file is destroyed with the activation of this command.

Successive calls to the function `tobasile` are possible, allowing several *BASILE* macros to be stored in the temporary file. Each activation of `frmaple()` destroys the temporary file.

frbasile() reads in *MAPLE* expressions generated with the commands `tomaple()` from *BASILE* and then destroys the corresponding temporary file.

clbasile() destroys the temporary file where *MAPLE* to *BASILE* translations have been stored.

basile(exp,file,{,arglist}) takes a *MAPLE* expression `exp` as input (it may be a procedure) and translates it into *BASILE* syntax.

This supplementary procedure gives the possibility of adding new functionalities for translating *MAPLE* expressions into *BASILE* ; but, strictly speaking, it is not part of the interface.

Expression to be translated: `a=1+x**2`

- **tomaple(a,'a');**

  Contents of the temporary file:

```
a := 1+x*(x) ;
```

  Loading into *MAPLE* with: `frbasile()`

Macro to be translated: `deff('<out>=f(x)','out=x+log(x)')`

- **tomaple(f,'f');**

  Contents of the temporary file:

```
f := proc(x)
    out :=(x+log(x));
   out;
end;
```

  Loading into *MAPLE* with: `frbasile()`

Table 1: Translating *BASILE* expressions into *MAPLE*

This procedure is able to treat several cases.

If **exp** is a *MAPLE* procedure name, the translation results in a *BASILE* macro. If the argument **file** is **terminal**, the macro definition (using the *BASILE* function **deff**) is printed on the terminal screen. If the argument **file** is different from **terminal**, a macro with name **file**, is written into the file **file**. This *BASILE* macro may then be loaded into *BASILE* with the *BASILE* command **getf**. The argument **arglist** is the argument list of the *BASILE* macro and is optional. When omitted, the arguments of the *BASILE* macro are those of the procedure.

If **exp** is a *MAPLE* expression and if the argument **file** is **terminal**, the *BASILE* syntax translation of the expression is printed on the terminal screen. If the argument **file** is not **terminal**, a *BASILE* macro with **file** as its name and the *BASILE* syntax translation of the *MAPLE* expression as its body is written into a file with **file** as its name. This *BASILE* macro may then be loaded into *BASILE* with the *BASILE* command **getf**. The argument **arglist** is the argument list of the *BASILE* macro. When omitted, the *BASILE* macro has an empty argument list.

While entering the command **basile(exp,terminal)**, we can decide to store the result in a file and then to load the translated expression into *BASILE* with the *BASILE* command **exec**. This is obtained with the following sequence of *MAPLE* commands:

```
writeto(file);
basile(exp,terminal);
writeto(terminal);
```

All the preceding functionalities are summarized in tables 2 and 3.
One simple way of doing things is to use **tobasile** solely.

2.3 Types of translated expressions

Table 4 shows which expressions the interface can translate.

In this paragraph, we describe some of the limitations of the *BASILE-MAPLE* interface and we introduce the new *MAPLE* function **&ev**.

2.3.1 Translating from *BASILE* to *MAPLE*

Presently, the interface does not know how to translate *BASILE* control structures (loops, conditional expressions). Moreover, it translates the *BASILE* expression `a(...)` into `a(...)`, because it has no means to distinguish between a functional expression and an element of a matrix. These limitations will be relaxed in the next version of the interface.

2.3.2 Translating from *MAPLE* to *BASILE*

MAPLE procedures are translated as such into *BASILE* macros. The interface considers that the name of the output variable is **out**. A user must define explicitly **out** as output variable for a correct execution of the generated macro (see the examples in section 3). Only one value may be returned from a *MAPLE* procedure, so if multiple values are needed, a list of the relevant values has to be returned from the procedure and this list is translated into a *BASILE* list.

Expression to be translated: `a:=1+x^2;`

- **tobasile(a,'a');**

  Contents of the temporary file:

```
//<out>=a ()
out=1+x**2
//end
```

  Loading into *BASILE* with: `frmaple()`

- **tobasile(a,'a',[x]);**

  Contents of the temporary file:

```
//<out>=a (x)
out=1+x**2
//end
```

  Loading into *BASILE* with: `frmaple()`

- **basile(a,toto);**

  Contents of the file `toto`:

```
//<out>=toto()
out=1+x**2
//end
```

  Loading into *BASILE* with: `getf('toto')`

- **writeto(toto);basile(a,terminal);writeto(terminal);**

  Contents of the file `toto`:

```
1+x**2
```

  Loading into *BASILE* with: `exec('toto')`

Table 2: Translating *MAPLE* expressions into *BASILE*

Procedure to be translated: `f:=proc(x,y) local out; out:=y+log(x) end;`

- **tobasile(f,'f');**

  Contents of the temporary file:

```
//<out>=f (x,y)
out=y+log(x),
//end
```

  Loading into *BASILE* with: `frmaple()`

- **tobasile(f,g,[x]);**

  Contents of the temporary file:

```
//<out>=g (x)
out=y+log(x),
//end
```

  Loading into *BASILE* with: `frmaple()`

- **basile(f,toto);**

  Contents of the file `toto`:

```
//<out>=f(x,y)
out=y+log(x),
//end
```

  Loading into *BASILE* with: `getf('toto')`

- **writeto(toto);basile(f,terminal);writeto(terminal);**

  Contents of the file `toto`:

```
deff('<out>=f(x)',<'out=x+log(x),'>)
```

  Loading into *BASILE* with: `exec('toto')`

Table 3: Translating *MAPLE* procedures into *BASILE*

MAPLE $\Longleftrightarrow$ *BASILE*
integer, float, complex numbers
constants π, e, i
algebraic expressions
matrices, arrays, indexed names
lists
trigonometric functions

MAPLE $\Longrightarrow$ *BASILE*
functions of `linalg` package:
`add`, `inverse`, `multiply`, `transpose`
in the body of procedures:
`:=`
`if` with boolean expressions (comparison operators, `true`, `false`, `and`, `or`, `not`)
`for` except the `for in` form
special function `&ev`
`evalm` is not translated

BASILE $\Longrightarrow$ *MAPLE*
macros defining no matrix expressions
in the body of these macros:
operators `+`, `*`, `-`, `/`, `\`, `**`
trigonometric functions
`if`, `for` and `select` are not translated
strings are not translated

Table 4: Expressions translated by the interface

2.3.3 The *MAPLE* function &ev

The function **&ev** is a new *MAPLE* function created for the needs of the interface. When used in a *MAPLE* procedure, it replaces its argument (which must be a variable name) by the evaluation of its name. This functionality is very useful when an expression is defined in *MAPLE* and we want to use it inside a generated *BASILE* macro.

We show hereunder an example of its use.

The following *MAPLE* program will be translated into a *BASILE* macro in which the array **a** is multiplied by $n!$. The array **a** is introduced into the *BASILE* macro with the help of **&ev(a)**.

```
with(linalg):

a:=array([[expand((1+x)^10),x],[1-x,2-x^2]]);

f:=proc(n,na)
  local fac;
  fac:=1;
  for i to n do fac := fac*i od;
  out := &ev(a);
  out := fac * out;
end;
```

After loading into *MAPLE*, we get the following in *BASILE*:

```
 <>frmaple()

 <>disp(f);

<out>=f(n,na)
fac=1,
for i=1:1:n,
fac=fac*i,
end,
out=<1+10*x+45*x**2+120*x**3+210*x**4+252*x**5+210*x**6+120*x**7+45*x**8+10*x**9
+x**10,x;1-x,2-x**2>,
out=fac*out,

 <>x=poly(0,'x')
 x         =

    x
```

```
 <>f(4,2)
 ans       =


         colonne    1
!                    2       3       4       5       6       7       8 !
!   24 + 240x + 1080x + 2880x + 5040x + 6048x + 5040x + 2880x + 1080x  !
!                   9    10                                            !
!           + 240x + 24x                                               !
!                                                                      !
!   24 - 24x                                                           !

         colonne    2
!   24x       !
!             !
!          2  !
!   48 - 24x  !
```

3 Examples

Let us first do the preliminary remark that it is often easier to define in *MAPLE* expressions to be used and then to translate them into *BASILE* with the help of the *BASILE -MAPLE* interface.

3.1 Derivative computation

We treat the example given in section 1.2.1.

In a first step, we load into *MAPLE* the following program:

```
with(linalg);

f:=sin(x[1]+x[2])-3*(x[1]*x[2]+exp(x[3]))/(x[2]+x[3]);

gf:=grad(f,[x[1],x[2],x[3]]);

tobasile(f,'f',[x]);
tobasile(gf,'grad',[x]);
```

Then we just have to execute the following sequence of instructions:

```
 <>frmaple()

 <>disp(f)

<out>=f(x)
out=sin(x(1)+x(2))-3*(x(1)*x(2)+exp(x(3)))/(x(2)+x(3)),
```

```
 <>disp(grad)

<out>=grad(x)
out=<cos(x(1)+x(2))-3*x(2)/(x(2)+x(3)),cos(x(1)+x(2))-3*x(1)/(x(2)+x(3))+3*(x(1)
*x(2)+exp(x(3)))/(x(2)+x(3))**2,-3*exp(x(3))/(x(2)+x(3))+3*(x(1)*x(2)+exp(x(3)))
/(x(2)+x(3))**2>,
```

3.2 Change of variables

Here we treat the example exposed in section 1.2.2.

We first load into *MAPLE* the following program:

```
with(linalg):

e1:=-diff(x1(t),t)+x2(t);
e2:=-diff(x2(t),t)-lambda*x1(t);
e3:=-diff(x3(t),t)+x4(t);
e4:=-diff(x4(t),t)-lambda*x3(t)-g;

e:=[e1,e2,e3,e4];
eb:=subs([diff(x1(t),t)=xd[1],diff(x2(t),t)=xd[2],diff(x3(t),t)=xd[3],
  diff(x4(t),t)=xd[4]],e);
eb:=subs([x1(t)=x[1],x2(t)=x[2],x3(t)=x[3],x4(t)=x[4]],eb);

tobasile(convert(eb,array),chvar,[t,x,xd]);

ee:=eval(subs([x1(t)=l*sin(phi1(t)),x3(t)=-l*cos(phi1(t))],e));

ee:=eval(subs([x2(t)=solve(ee[1],x2(t)),x4(t)=solve(ee[3],x4(t))],
  {ee[2],ee[4]}));

ee:=eval(subs([diff(phi1(t),t)=dphi1,diff(phi1(t),t,t)=dphi2],ee));

ss:=simplify(solve(ee,{dphi1,dphi2})[1]);

assign(ss);
dphi2:=subs(phi1(t)=phi[1],dphi2);
dphi1:=phi[2];

res:=[dphi1,dphi2];

tobasile(convert(res,array),chvar2,[t,phi]);
```

Then the following instructions are entered in *BASILE*:

```
 <>frmaple()

 <>disp(chvar);
```

```
<out>=chvar(t,x,xd)
out=<-xd(1)+x(2),-xd(2)-lambda*x(1),-xd(3)+x(4),-xd(4)-lambda*x(3)-g>,

 <>disp(chvar2);

<out>=chvar2(t,phi)
out=<phi(2),-sin(phi(1))*g/l>,
```

3.3 Help to curve plotting

We may be faced with the problem of plotting the graph of a function defined by an implicit equation such as: $f(t, y) = 0$. This cannot be done in this form in *BASILE*. To be able to plot the graph of y, we make use of the implicit functions theorem, in order to compute the derivative of y with respect to t. The derivative computation is done in *MAPLE*. The corresponding differential equation is then numerically solved in *BASILE*. The resulting solution is plotted using *BASILE*.

Let us take the simple example $f(t, y) = y - t^2$.

First we load into *MAPLE* the following program:

```
f:=y-t^2;
d:=-diff(f,t)/diff(f,y);
tobasile(d,kdot,[t,y]);
```

Then the following sequence of *BASILE* commands gives the desired result:

```
 <>frmaple()

 <>disp(kdot)

<out>=kdot(t,y)
out=2*t,

 <>tt=<0,1,2,3,4,5,6,7,8,9,10>;

 <>yt=ode(0,0,tt,kdot);

 <>plot(tt,yt)
```

The macro **kdot** describes the differential equation: $\frac{dy}{dt} = -\frac{\partial f}{\partial t} / \frac{\partial f}{\partial y}$.

tt is the sequence of time samples where the numerical solution is computed.

The call to the *BASILE* solver **ode** gives the desired result. Then the graph is plotted.

3.4 The inverted pendulum

The system equations are more complex here but the symbolic manipulations are of derivative type and of no particular difficulty.

A first *MAPLE* procedure, of general use, linearizes a system:

$$\begin{cases} \dot{X} = f(X, U) \\ Y = g(X, U) \end{cases}$$

into:

$$\begin{cases} \dot{X} = AX + BU \\ Y = CX + DU \end{cases}$$

The corresponding *MAPLE* procedure is as follows:

```
# x[1..nx], u[1..nu], y[1..ny], xdot[1..nx]
# x0[1..nx], u0[1..nu]

with(linalg):

lin:=proc(xdot,y,x,u,x0,u0)
  local a,b,c,d,jac,fmat0;
  fmat0:=proc(mat) map(eval,subs(x=x0,u=u0,op(mat))) end;
  jac:=jacobian(xdot,x); a:=fmat0(jac);
  jac:=jacobian(xdot,u); b:=fmat0(jac);
  jac:=jacobian(y,x); c:=fmat0(jac);
  jac:=jacobian(y,u); d:=fmat0(jac);
  [map(simplify,op(a)),
   map(simplify,op(b)),
   map(simplify,op(c)),
   map(simplify,op(d))]
end;
```

where `xdot` (resp. `y`) is an array representing $\dot{X}$ (resp. Y), `x` and `u` are the corresponding arrays for the variables X and U; `x0` and `u0` define the point where the linearization is done.

Let us recall briefly the goal of this problem (see figure 1): a control has to be designed so that the pendulum, in an inverted position on a moving chariot, keeps its equilibrium.

The differential system governing the pendulum evolution is:

$$\begin{cases} \dot{x}_1 = x_2 \\ \dot{x}_2 = \dfrac{u_1 + \sqrt{m_b}(\sin(x_3)x_4^2 - \cos(x_3)D_4)}{m_b + m_c} \\ \dot{x}_3 = x_4 \\ \dot{x}_4 = D_4 \end{cases}$$

$$\begin{cases} y_1 = x_1 \\ y_2 = x_3 \end{cases}$$

with

$$\left| \begin{array}{rcl} q_m & = & \dfrac{m_b}{m_b + m_c} \\ d & = & \dfrac{4}{3} - q_m \cos(x_3)^2 \\ D_4 & = & \dfrac{-\sin(x_3)\cos(x_3)q_m x_4^2 + \dfrac{2\sin(x_3)m_b g - q_m \cos(x_3)u_1}{m_b l}}{d} \end{array} \right.$$

The *MAPLE* program computing the tangent linear system is:

```
read lin:

x:=array(1..4); u:=array(1..1);
```

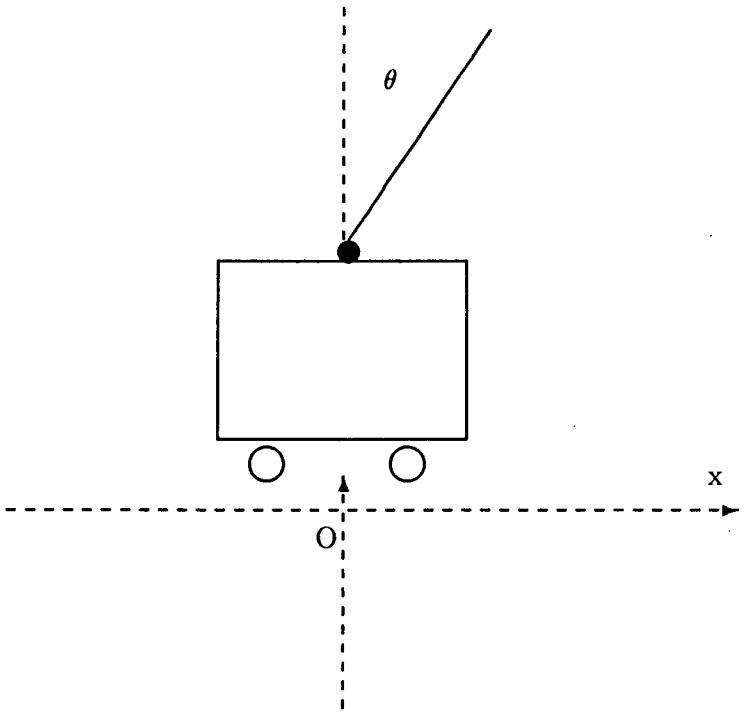

Figure 1: The inverted pendulum on a moving chariot

```
qm:=mb/(mb+mc);
cx3:=cos(x[3]);
sx3:=sin(x[3]);
d:=4/3-qm*cx3*cx3;
xd4:=(-sx3*cx3*qm*x[4]**2+2/(mb*l)*(sx3*mb*g-qm*cx3*u[1]))/d;
axdot:=array([x[2],
      (u[1]+mb*(1/2)*(sx3*x[4]**2-cx3*xd4))/(mb+mc),
      x[4],
      xd4]);
ay:=array([x[1],x[3]]);

x0:=array([0,0,0,0,0]); u0:=array([0]);

res:=lin(axdot,ay,x,u,x0,u0):
va:=res[1];
vb:=res[2];
vc:=res[3];
vd:=res[4];

u[1]:=u;
g:=9.81;

lres:=[''lss'',op(va),op(vb),op(vc),op(vd),op(x0),''c''];

tobasile(lres,'sys');
```

ay gives the original system equations, **x0** is the point where to linearize, **res** is the tangent linear system in (x_0, u_0) and **lres** is the construction of the *BASILE* list for correct description of the linear system.

We can now get the wanted result in *BASILE*:

```
 <>frmaple()

 <>disp(sys)

<out>=sys()
out=list('lss',<0,1,0,0;0,0,-0.2943D2*mb/(mb+4*mc),0;0,0,0,1;0,0,0.5886D2*(mb+mc
)/l/(mb+4*mc),0>,<0;4/(mb+4*mc);0;-6/l/(mb+4*mc)>,<1,0,0,0;0,0,1,0>,<0;0>,<0,0,0
,0,0>,'c'),

 <>mb=0.1;mc=1;l=0.3;

 <>ssprint(sys());

    | 0  1  0          0 |    | 0         |
.   | 0  0 -0.7178049  0 |    | 0.9756098 |
x = | 0  0  0          1 |x + | 0         |u
    | 0  0  52.639024  0 |    |-4.8780488 |

    | 1  0  0  0 |
y = | 0  0  1  0 |x
```

4 Implementation

4.1 Data communication through files

Using files for data transfer between *BASILE* and *MAPLE* implies the designation of two temporary files that *BASILE* and *MAPLE* must know.

In the case of *BASILE*, these two files have two be defined in the system file **basile.star** with the *BASILE* names **b_2_m** (for "basile to maple") and **m_2_b** (for "maple to basile").

This definition is machine dependent. For instance, in UNIX, it could be:

```
b_2_m='/tmp/b_2_m'
m_2_b='/tmp/m_2_b'
```

In the case of *MAPLE*, the two files must be defined in the system file **init** (usually in the **src** directory of the *MAPLE* library), with the *MAPLE* variable names **_tmpin** et **_tmpout**.

This definition is machine dependent. For instance, in UNIX, it could be:

```
_tmpin := `/tmp/b_2_m`:
_tmpout := `/tmp/m_2_b`:
```

The files defined for *BASILE* and *MAPLE* must have the same names, of course. In the presented implementation, it is important to point out that only one user can use the *BASILE -MAPLE* interface (to avoid overwriting of the temporary files). However, a slight modification can relax this restriction, for example by appending a user identification to the names of the temporary files.

4.2 Initializing *BASILE* and *MAPLE*

Practically, one *BASILE* function (`frmaple`) and two *BASILE* macros (`tomaple`, `clmaple`) are given to the user of the interface.

The function `frmaple` is resident (via the initial link edition of *BASILE*). For a correct access to the two macros `tomaple` and `clmaple`, the user has to load the file `tomaple.bas` with the *BASILE* command `getf`. It can be done automatically by inserting the command in the *BASILE* initialization file `startup.bas`. The directory containing the file `tomaple.bas` is machine dependent.

To get correct access to the *MAPLE* procedures `tobasile`, `frbasile`, `clbasile` and `basile`, the user must first execute the following command in *MAPLE*:

```
read ``.libname.`/map2bas.m`:
```

and again this can be done in the user *MAPLE* initialization file `.mapleinit`.

4.3 *BASILE* Functions

The new *BASILE* functionalities are mainly given by macros.

4.3.1 Function `frmaple`

This function is a FORTRAN program linked to *BASILE*. It is defined in the file `frmaple.f`. Its purpose is simply to execute `getf` from the temporary file.

4.3.2 The macro `clmaple`

This macro destroys the temporary file. It is defined in the file `tomaple.bas`.

4.3.3 The macro `tomaple`

Considering the translation from *BASILE* to *MAPLE* two situations have to be considered:

1. The translation is applied to simple data types such as matrices (real or complex numbers, polynomials, rationals), lists or character strings. In that case, the data transfer is done inside the macro `tomaple`.

2. We want to translate *BASILE* macros into *MAPLE* procedures. This situation would imply the use of a lexical analyzer and of a translator, leading to the writing of a *BASILE* expression parser. Instead, we have used the *BASILE* parser, taking the compiler output and translating it into *MAPLE* syntax. This is done by the FORTRAN subroutine `tramac` linked to *BASILE* during the initial link edition.

The *BASILE* macro `tomaple` is defined in the file `tomaple.bas` and the function `tramac` in the file `tramac.f`.

4.4 *MAPLE* procedures

MAPLE procedures are stored in the file `map2bas`. For the translation of a *MAPLE* expression into *BASILE* we took the ideas from the *MAPLE* function `fortran`. For the translation of *MAPLE* procedures into *BASILE* macros, we used their decompilation with the help of the *MAPLE* procedure `procbody`.

References

[1] Delebecque, F., Klimann, C., Steer, S., *BASILE: Guide de l'utilisateur*, INRIA, March 1989.

[2] Gomez, C., *MACROFORT: a FORTRAN code generator in MAPLE*, INRIA Report 119, May 1990.

Contents

List of Tables

List of Figures

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique.