Macro C : C code generation within maple

Patrick Capolsini

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

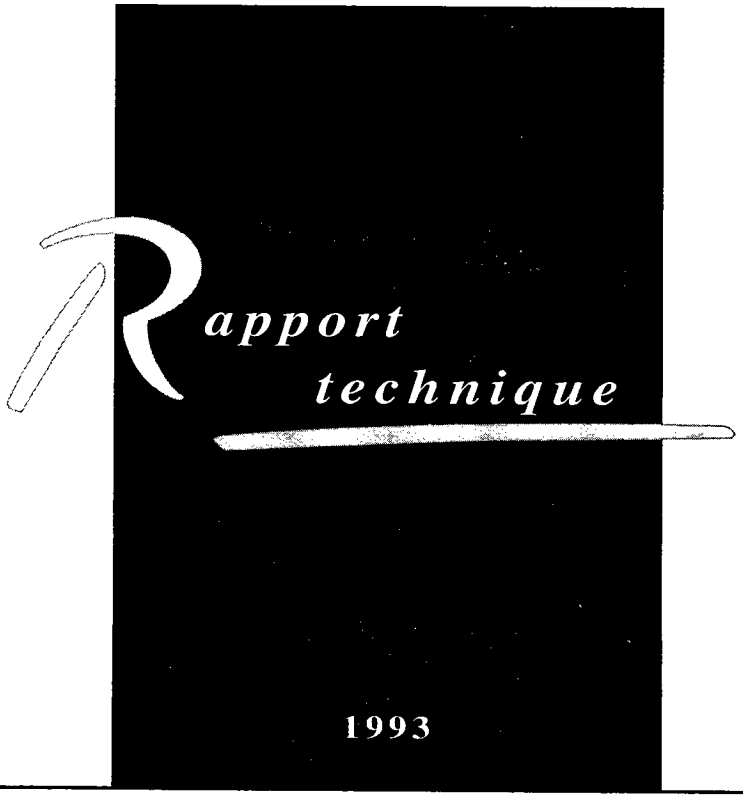
Macro C
C code generation within maple

Patrick CAPOLSINI

N° 151
Février 1993

PROGRAMME 2

Calcul Symbolique,
Programmation
et Génie logiciel



MacroC

C code generation within Maple

génération de code C depuis Maple

Patrick CAPOLSINI

Abstract

MacroC is a Maple package which provides a simple way to generate C language code without having to leave Maple. All C structures can be generated and the resulting code can be optimized. We have to notice that Claude Gomez (INRIA Rocquencourt) has already written such a package, called Macrofort, for the Fortran language. In this report, we describe macroC's syntax and features. Finally, we give two complete examples of its possible use.

Résumé

MacroC est un package de Maple qui offre la possibilité de générer du langage C sans avoir à quitter Maple. Toutes les structures du langage C peuvent être engendrées et le code obtenu peut être optimisé. Il faut noter que Claude Gomez (INRIA Rocquencourt) a déjà écrit un tel package, baptisé Macrofort, pour le langage Fortran. Dans ce rapport, nous décrivons la syntaxe et les possibilités de macroC. Enfin, nous donnons deux exemples d'utilisations possibles.

Chapter 1

What is macroC for ?

Computer Algebra Systems (such as Maple [1], [2], [3] or mathematica [4]) provide the capabilities to handle a very large set of symbolic computations : derivations, integrations, linear algebra, polynomial equations handling and solving, exact calculations on unbounded integers and floating-point numbers ...

Up to a few years ago, these Algebra systems were only used by specialists working in universities and specialized research laboratories but this is changing and Algebra Systems (especially Maple) are more and more used in different fields of industrial research and applications. Industrial engineers have found out that such systems could help them to establish symbolical models of the problem they are studying, but they still need numerical computations.

Fortran is the most common numerical language in industrial field, Ada begins to be used for spacecraft applications but the C language gets more and more followers as the Unix operating system becomes widely distributed. C stands out among other general-purpose languages since it combines portability, power, flexibility and elegance. This language is well suited for programs of various sizes and every Unix machine has a C compiler. Although there is nearly no C library routine (compared with Fortran ones), C is able to handle all numerical problems with a minimum amount of work. This paper is not intended to be a C course and we assume a small knowledge of C (the reader interested in learning about C can refer to [5] or [6]).

Maple ([1], [2], [3]) provides a few functions to deal with numerical computing (*evalf*, *evalhf*, *fsolve* or *dsolve*) but they are quite slow and can fail on some difficult problems. This is the reason why Maple's authors have written a Fortran and a C code generator.

Maple's C function takes a Maple expression as argument and returns the C corresponding code. An optional argument (*optimize*) can be given such that common subexpressions research is performed and an optimized code is provided. The example below illustrates the use of the C function.

```
> readlib(C);

proc() ... end


> e:= (ln(x^2*cos(x)) + sin(x^2-4/b) - Pi*x^23)/21.4*expand(cos(3*x-Pi/8)^2);
                        2              2             23          2       4
  e := .04672897196 (ln(x  cos(x)) + sin(x  - 4/b) - Pi x  ) (8 sin(x)  cos(x)

                2       4  1/2          2       2          2       2  1/2
      - 4 sin(x)  cos(x)  2    - 4 sin(x)  cos(x)  + 2 sin(x)  cos(x)  2
```

```
                1/2 1/2              5        1/2 1/2
      + 8 (2 - 2   )     sin(x) cos(x)   (2 + 2   )

                1/2 1/2              3        1/2 1/2             2
      - 8 (2 - 2   )     sin(x) cos(x)   (2 + 2   )     + 1/2 sin(x)

                  2  1/2             1/2 1/2              1/2 1/2
      - 1/4 sin(x)  2     + 3/2 (2 - 2   )     sin(x) (2 + 2   )     cos(x)

                6           6  1/2            4           4  1/2             2
      + 8 cos(x)  + 4 cos(x)  2     - 12 cos(x)  - 6 cos(x)  2     + 9/2 cos(x)

                  2  1/2
      + 9/4 cos(x)  2   )
```

```
> C(e);
      t0 = 0.4672897196E-1*(log(x*x*cos(x))+sin(x*x-4.0/b)-
0.3141592653589793E1*pow(x,23.0))*(8.0*pow(sin(x),2.0)*pow(cos(x),4.0)-
4.0*pow(sin(x),2.0)*pow(cos(x),4.0)*sqrt(2.0)-4.0*pow(sin(x),2.0)*
pow(cos(x),2.0)+2.0*pow(sin(x),2.0)*pow(cos(x),2.0)*sqrt(2.0)+8.0*
sqrt(2.0-sqrt(2.0))*sin(x)*pow(cos(x),5.0)*sqrt(2.0+sqrt(2.0))-8.0*
sqrt(2.0-sqrt(2.0))*sin(x)*pow(cos(x),3.0)*sqrt(2.0+sqrt(2.0))+
pow(sin(x),2.0)/2-pow(sin(x),2.0)*sqrt(2.0)/4+3.0/2.0*sqrt(2.0-sqrt(2.0))*
sin(x)*sqrt(2.0+sqrt(2.0))*cos(x)+8.0*pow(cos(x),6.0)+4.0*pow(cos(x),6.0)*
sqrt(2.0)-12.0*pow(cos(x),4.0)-6.0*pow(cos(x),4.0)*sqrt(2.0)+9.0/2.0*
pow(cos(x),2.0)+9.0/4.0*pow(cos(x),2.0)*sqrt(2.0));

> C(e, optimized);
      t1 = x*x;
      t2 = cos(x);
      t9 = t1*t1;
      t11 = t9*t9;
      t12 = t11*t11;
      t16 = sin(x);
      t17 = t16*t16;
      t18 = t2*t2;
      t19 = t18*t18;
      t20 = t17*t19;
      t21 = sqrt(2.0);
      t23 = t17*t18;
      t26 = sqrt(2.0-t21);
      t27 = t26*t16;
      t30 = sqrt(2.0+t21);
      t39 = t19*t18;
      t43 =
8.0*t20-4.0*t20*t21-4.0*t23+2.0*t23*t21+8.0*t27*t19*t2*t30-
8.0*t27*t18*t2*t30+t17/2-t17*t21/4+3.0/2.0*t27*t30*t2+8.0*t39+
4.0*t39*t21-12.0*t19-6.0*t19*t21+9.0/2.0*t18+9.0/4.0*t18*t21;
      t45 =
0.4672897196E-1*(log(t1*t2)+sin(t1-4.0/b)-0.3141592653589793E1*
t12*t9*t1*x)*t43;
```

If the user wants to get a complete C program to compute an expression, he has to translate the expression into the C syntax using the C function (such as in the example), write it into a file, edit the file and write C code "around" his expression. It is a bit tedious to have to exit from Maple, edit a file and so on. MacroC provides functions to make everything from Maple.

In fact, this idea is not new. Claude Gomez (INRIA - Rocquencourt) has already written a Fortran code generation package called *Macrofort* (see [7] for details) and we have tried to keep our syntax as near as Macrofort's as possible so that users can use both Macrofort and macroC with a minimum amount of work and time.

Chapter 2

Description of the macroC package

The macroC "language" is made of two kinds of instructions : single and macro instructions. Both of them have the same syntax : a Maple list which first operator is a keyword describing the C statement to generate and the other elements of the list are the parameters of the statement (if needed). Single instructions ordinarily describe a unique C statement and macros describe a set (usually a C block) of statements. If the first operator of a list is not a macroC keyword, the expression is intended to be a C simple expression and the generator will try to translate it into C syntax (this is particularly useful to invoke C functions without using the *callC* single instruction).

We give, in the next sections, the syntax of all the single and macro instructions with the corresponding generated C statements.

All optional parameters are given in *emphasized* style.

2.1 Single instructions

MacroC single instructions keywords are made of the C instruction name (or something close) followed by a "C". We give here the complete list of the existing single instructions.

PREPROCESSOR

[includeC,file_name]	→ # include file_name
[defineC,ident,var]	→ # define ident expr
[undefC,ident]	→ # undef ident
[ifdefC,ident]	→ # ifdef ident
[ifndefC,ident]	→ # ifndef ident
[ifC,expr]	→ # if expr
[elifC,expr]	→ # elif expr
[lineC,num,*file_name***]**	→ # line num *"file_name"*
[errorC,string]	→ # error "string"
[pragmaC,string]	→ # pragma "string"

DECLARATIONS

[declareC,*type***,type,[vars]]**	→ *type* type var1,var2,...;
[structC,*name***,** **[[type1,[vars]],** **[type2,[vars]],...],** *[ident1,...]*]	→ struct *name*

```
                                        {
                                            type1 var1,var2,...;
                                            type2 var3,var4,...;
                                        } ident1,ident2,...;
[unionC,name,
  [[type1,[vars]],
   [type2,[vars]],...],
  [ident1,...]]                     → union name
                                        {
                                            type1 var1,var2,...;
                                            type2 var3,var4,...;
                                        } ident1,ident2,...;
[enumC,name,[vars],[ident1,...]]    → enum name {var1,...} ident1,...;
[typedefC,type,ident]               → typedef type ident;
[typedefC,struct-union-enum,ident]  → typedef struct-union-enum
                                        {
                                          ...
                                        } ident;
```

MISC

```
[commentC,string]                   → /*string*/
[equalC,ident,expr]                 → ident = expr;
[callC,name,[vars]]                 → name(var1,var2,...);
[functionC,type,name,
  [[type1,[vars]],
   [type2,[vars]],...]]             → type name(var1,var2,var3,var4,...)
                                        type1 var1,var2,...;
                                        type2 var3,var4,...;
                                        ...
```

CONDITIONS

```
[if_thenC,expr]                     → if(expr)
[caseC,expr,[actions]]              → case expr :
                                        action1;
                                        action2;
                                        ...
[defaultC,[actions]]                → default :
                                        action1;
                                        action2;
                                        ...
```

ITERATIONS

```
[forC,expr1,expr2,expr3]            → for(expr1;expr2;expr3)
[whileC,expr]                       → while(expr)
```

EXITS AND LABELS

```
[returnC,expr]                      → return(expr);
[breakC]                            → break;
[continueC]                         → continue;
```

[gotoC,ident] → goto ident;
[labelC,ident] → ident:

INPUT - OUTPUT
[fopenC,pt_name,file_name,status] → pt_name = fopen("file_name" , "status");
[fcloseC,pt_name] → fclose(pt_name);

2.2 Macro instructions

MacroC macro instructions keywords are made of the C instruction name (or something close) followed by a "m". We give here the complete list of the existing macro instructions.

PREPROCESSOR

[ifdefm,ident,[actions],*[actions]*] →
```
# ifdef ident
    action1;
    action2;
    ...
# else
    action3;
    action4;
    ...
# endif
```

[ifndefm,ident,[actions],*[actions]*] →
```
# ifndef ident
    action1;
    action2;
    ...
# else
    action3;
    action4;
    ...
# endif
```

[ifm,expr,[actions],*[actions]*] →
```
# if expr
    action1;
    action2;
    ...
# else
    action3;
    action4;
    ...
# endif
```

[elifm,expr,[actions]] →
```
# elif expr
    action1;
    action2;
    ...
```

DECLARATIONS

[declarem,*type*,type,[vars]] → *type* type var1,var2,...;

MISC

[matrixm, ident, array] → *ident*[*i*][*j*] = *array*[*i*, *j*];
...

[functionm, *type*, name,
[[type1, [vars]],
[type2, [vars]], ...]]
[actions]] → *type* name(var1,var2,var3,var4,...)
type1 var1,var2,...;
type2 var3,var4,...;
...
{
action1;
action2;
...
}

CONDITIONS

[if_thenm, expr, [actions]] → if(expr)
{
action1;
action2;
...
}

[if_then_elsem, expr,
[actions], [actions]] → if(expr)
{
action1;
action2;
...
}
else
{
action3;
action4;
...
}

[switchm, expr,
[[caseC, case, [actions]],
...
[defaultC, [actions]]]] → switch (expr)
{
case case1 :
action1;
action2;
...
...
default :
action3;
action4;
...

```
                                    }
```

ITERATIONS

```
[form,expr1,expr2,expr3,
  [actions]]                    → for(expr1;expr2;expr3)
                                    {
                                        action1;
                                        action2;
                                        ...
                                    }
[whilem,expr,[actions]]         → while(expr)
                                    {
                                        action1;
                                        action2;
                                        ...
                                    }
[dowhilem,expr,[actions]]       → do
                                    {
                                        action1;
                                        action2;
                                        ...
                                    }
                                  while (expr);
```

EXITS AND LABELS

```
[labelm,ident,[actions]]        → ident:
                                    {
                                        action1;
                                        action2;
                                        ...
                                    }
```

INPUT - OUTPUT

```
[fopenm,pt_name,file_name,
  status, [actions]]            → pt_name = fopen( "file_name" , "status" );
                                    action1;
                                    action2;
                                    fclose(pt_name);
```

The **declareC** single instruction and **declarem** macro instruction have the same syntax and behavior but the latter can be found anywhere within a block (the corresponding declarations will be shifted to the beginning of the current block).

2.3 MacroC's use

MacroC's use is quite easy and can be described in four points :

Load the macroC package

The first thing to do is to load the macroC package. This can be done by issuing the command *read 'macroC';* (or *read 'macroC.m';*) intended that the file macroC (or macroC.m) can be found in the directory from which Maple has been invoked. This command initializes the global variables to their default values.

Eventually modify global variables

The macroC package provides the use of four global variables : *precision* which can be set to *single* (default) or *double* whether the user wants to have 8 or 15 significant digits for the floating point variables, the *shift* variable (default : -1) which represents the value the user wants to add to array indices, the boolean variable *optimized* (default : *false*) for the optimization of the generated code and the *autodeclare* variable (default : unassigned) which can be assigned a name representing the type of the intermediate variables generated with the *optimized* option. The use of these two variables is detailed in the next section. The *init_genC()* function call will initialize these variables to their default values.

Build the macroC list describing the C program

Once the package is loaded, the user has to build the list of macroC's single and macro instructions describing the C program he wants to generate.

Invoke the *genC* function

The *genC* function is the heart of the generator. Invoking *genC* on the list describing the program will generate the C program. A first optional argument specifies the amount of blank spaces at the beginning of each line of code. If this argument is omitted, the value 0 is set by default. Another last optional argument of the form *filename = 'foo.c'* will direct the output to the specified file *foo.c*.

Note that all macroC's instructions arguments are Maple names and have the same evaluation rules at toplevel, so, it can be required to quote them.

2.4 Other features

We describe here a few useful additional features to the single and macro instructions.

The "string" function

C syntax for strings uses the symbol " which is quite difficult to handle in Maple. We have written a small procedure which takes a string as arguments an return the string between ". Using this function, the generation of C functions using strings such as *printf, fprintf, scanf, fscanf* ... becomes easy.

```
> genC([fscanf(pt_name, string('%lf\n'), var)]);
fscanf(pt_name, "%lfn" ,var);
```

Boolean expressions

Maple's boolean relations (=, <>, <, <=) and values (*true*, *false*) can be used within macroC instructions and will be translated into C syntax. Logical expressions (*and*, *or*, *not*) will be translated too but after Maple's boolean evaluation. That means that the expression $a <> b$ *and* $c = d$ will be evaluated to *false* before translation by macroC. To cancel this problem we provide three new prefix operators : *AND*, *OR* and *NOT* such that expressions involving logical operators can be translated into the correct C statements.

```
> genC([a<=b or c<>d]);
1;
> genC([OR(a<=b, c<>d)]);
((a <= b) || (c != d));

> genC([NOT(AND(a<=b, c<>d, true))]);
!((a <= b) && (c != d) && 1);
```

The assignment new operator

The *equalC* instruction provides a simple way to generate an assignment but we have thought that it was interesting to provide another one which could handle the C composed assignment operators : ++, −−, + = and ∗ = (note that / = and − = are replaced by their corresponding expressions using ∗ = and + =). This new assignment operator is denoted & = and transforms an expression such as & = $(a, a*b)$ into a &∗ = b or & = $(a, a/b)$ into a &∗ = $1/b$. It is recommended to use this operator into the macroC iterative instructions to get an efficient C code.

```
>  &=(i, i+1);

                                    i++
> &=(i, -1+i);

                                    i--
> &=(a, a*b*(c-1));

                                a &*= (b c - b)
> genC(["]);
 *= b*c-b;a


> &=(a, a/(b*c));

                                      1
                                a &*= ---
                                      b c
```

Constants and functions transcription

Maple's constants such as *E*, *Pi*, *gamma* or *Catalan* are replaced with their respective floating point evaluations.

All C functions can be handled by macroC. The user can invoke them using three equivalent ways : the single instruction *callC*, as the right hand side of an assignment or even putting the function call between brackets. Mathematical particular functions which do not have the same name within Maple and C syntax will be translated in the right way. For example, the *csc* Maple function will be translated into *1/sin(x)*.

```
> genC(6, [E]);
      0.2718282E1;

> genC(6, [Pi]);
      0.3141593E1;

> genC(8, [strcmp(`string1`,`string2`)]);
        strcmp(string1,string2);

> genC([csc(psi)]);
1/sin(psi);

> genC(2, [cot(psi)]);
  1/tan(psi);

> genC(4, [coth(psi)]);
    1/tanh(psi);
```

Arrays transcription

All Maple arrays can be handled by macroC using the *matrixm* macro. By default, **array indices are shifted** to match C syntax. That is quite pleasant when translating an existing array but one has to be careful when referring to formal indices (for example in a loop process). In the last example below, each element of a mono-dimensional array is assigned the value of its indices divided by two (note the `tab[i+1]`). This mechanism can be cancelled or customized using the global variable *shift* which represents the value added to indices and is set by default to -1. Particularly, *shift* can be set to 0 if the user does not want to shift array indices.

```
> genC([matrixm, titi, array(1..3,[cot(a), csch(x),Pi])]);
titi[0] = 1/tan(a);
titi[1] = 1/sinh(x);
titi[2] = 0.3141593E1;

# The same with global variable shift equal to 0
> shift := 0:

> genC([matrixm, titi, array(1..3,[cot(a), csch(x),Pi])]);
titi[1] = 1/tan(a);
titi[2] = 1/sinh(x);
titi[3] = 0.3141593E1;

> shift := -1:
```

```
> genC([matrixm, toto, array(1..3,1..3,identity)]);
toto[0][0] = 1;
toto[0][1] = 0;
toto[0][2] = 0;
toto[1][0] = 0;
toto[1][1] = 1;
toto[1][2] = 0;
toto[2][0] = 0;
toto[2][1] = 0;
toto[2][2] = 1;

> genC([form, &=(i,0), i<=n, &=(i, i+1),
>           [equalC, tab[i+1],i/2]]);
for(i = 0;(i <= n);i++)
  tab[i] = i/2;
```

Generated code optimizations

As already mentioned, it is possible to perform code optimization while using macroC. If the global variable *optimized* is assigned the value *true* then common subexpressions optimization is performed using the Maple's procedure *optimize*. When invoked on a matrix, common expressions are shared for the whole matrix (see examples below).

```
> genC([equalC, var, cos(x)+cos(x)^3-tan(x)]);
var = cos(x)+pow(cos(x),3)-tan(x);

> optimized:=true:

> genC([equalC, var, cos(x)+cos(x)^3-tan(x)]);
t1 = cos(x);
t2 = t1*t1;
var = t1+t2*t1-tan(x);

> genC([matrixm, titi, array(1..3, [2*x^3-y*x^4, x^2, y*x^2])]);
t1 = x*x;
t4 = t1*t1;
titi[0] = 2*t1*x-y*t4;
titi[1] = t1;
titi[2] = y*t1;
```

Use of the *autodeclare* variable

This variable is used in conjunction with the *optimized* option. Assigning this variable to a name means that macroC has to generate the declaration of the intermediates variables (t1, t2, ...) used with the *optimized* option. The value of *autodeclare* specifies the type of the variables t1, t2, ...

```
> optimized:=true:
> autodeclare;

                                autodeclare
```

```
> genC([form, &=(i,0),i<=n,&=(i, i+1),
>           [equalC, expr, x^5]]);
for(i = 0;(i <= n);i++)
  {
  t1 = x*x;
  t2 = t1*t1;
  expr = t2*x;
  }

> autodeclare:=`double`:
> genC([form, &=(i,0),i<=n,&=(i, i+1),
>           [equalC, expr, x^5]]);
for(i = 0;(i <= n);i++)
  {
  double t1,t2;
  t1 = x*x;
  t2 = t1*t1;
  expr = t2*x;
  }
```

Chapter 3

Examples

It is always of great interest to see a full and practical example of the use of a new software. That is the reason why we expose here two examples. The first one is the problem which led us to writing this package and the second one is the "Generalized Newton method" (to solve a non linear system of equations) which has been exposed by C. Gomez in [7].

3.1 Mechanical systems simulations

We were interested in the computation of the equations governing the behavior of multibody mechanical systems involving forces, torques, inertia and so on. We have written, using Maple, a software which provides these equations using a matrix representation. These very large equations can be handled using a specific toolbox written in Maple but Mechanical engineers want to have a numerical simulation of the system's behavior. So we had to translate our equations into numerical code (Fortran and C) and to provide a full ready to compile and efficient code.

The behavior of such mechanical systems is given by a set of n second order differential equations (n being the number of degrees of freedom of the system). This set of equations can be represented by an equation of the form :

$$Masse\,\ddot{q} + Force = 0$$

Where $Masse$ is an n by n matrix, $Force$ an n by one vector and $\ddot{q}$ an n by one vector representing the accelerations of the n degrees of freedom. To simulate the system one has to perform different tasks :

1. Translate the Maple's matrix $Masse$ and the Maple's vector $Force$ into efficient numerical code,
2. Enter the numerical data (masses, inertia, initial conditions, ...),
3. At each time step :
   - Compute $Masse$ and $Force$,
   - Inverse the matrix $Masse$ to establish the system of differential equations

     $$\ddot{q} = -Masse^{-1}\,Force$$

   - Solve the system for the $\ddot{q}$ using an integration algorithm such as Runge/Kutta or Adams,
4. Write the results to a file.

There are at least four reasons why this type of problem can't be solved using only an algebra system :

- Mechanical engineers want to perform a lot of simulations using different sets of numerical data (masses, inertia, lengths, forces, torques, ...) and different sets of initial conditions (positions and velocities),
- All data are floating point numbers and systems such as Maple are much slower than Fortran or C to compute with floats,
- As soon as the number of degrees of freedom gets greater than two or three, it becomes nearly impossible to compute a formal inverse of the *Masse* matrix,
- The system has to be established and solved at each time step and the number of steps depends of the simulation time interval and of the requested accuracy. It is quite classical that a ten seconds simulation requires more than ten thousand steps !

We discuss, in the next sections, how each of the four tasks described below can be treated using macroC. For each of them we give macroC code and the corresponding generated C code for a two degrees of freedom system representing a double pendulum.

3.1.1 Translate matrices *Masse* and *Force* into C

Our matrices are established using formal linear algebra operators and we have written a special algorithm to translate these expressions into macroC code. We do the same work as with the *optimized* option but we use new intermediate variables called C_i which are declared in the routine's header.

MacroC code

```
[commentC, ], [commentC, 'Equations establishment '],

[functionm, void, calcul , [[double, [tps , *etat ]], [int, [dim ]]],

[

[declareC, double ,

[C1, C2, C3, C4[4], C5[2, 4], C6, C7, C8, C9[4, 4], C10[4, 4], C11[4, 4],

    C12[4, 4], C13[4, 4], C14[4], C15[4], C16[4], C17, C18, C19[4, 4], C20,

    C21[4], C22[4], C23[4], C24[4], C25[4], C26[4], C27[4], C28[4], C29[4],

    C30[4], C31[4], C32[2, 4], C33[4, 4], C34[4], C35, C36, C37, C38[4, 4],

    C39[4, 4], C40[4, 4], C41[4], C42, C43[2, 4], C44, C45, C46[4], C47, C48,

    C49, C50, C51, C52, C53[2, 4], C54, C55[2, 4], C56, C57, C58, C59, C60[4],

    C61, C62, C63, C64[4], C65, C66, C67[4, 4], C68[4], C69[4], C70[4],

    C71[2, 4], C72, C73, C74, C75[4], C76[4], C77[4], C78[4], C79[4], C80[4],

    C81[4, 4], C82[4], C83[4], C84[4], C85, C86[4], C87[4], C88[4, 4], C89[4],
```

```
    C90[4], C91[4, 4], C92[4], C93[4], C94[4], C95[4], C96[4], C97[4], C98,

    C99, C100, C101, C102, C103, C104[2, 4], C105, C106, C107]

],

[commentC,  'State vector '], [equalC, q1 , etat [1]],

[equalC, u1 , etat [3]], [equalC, q2 , etat [2]], [equalC, u2 , etat [4]],

[commentC,  'masse matrix and force vector '], [equalC, C1, sin(q1)],

                                                          [ C3  C2  0 ]
                                                          [           ]
[equalC, C2, - C1], [equalC, C3, cos(q1)], [matrixm, pas10, [ C1  C3  0 ]],
                                                          [           ]
                                                          [ 0   0   1 ]

[matrixm, l1, [ 0, 0, 1 ]], [equalC, C4[1], 0], [equalC, C4[2], 0],

[equalC, C4[3], 1], [equalC, C5[1, 1], C4[1]], [equalC, C5[1, 2], C4[2]],

[equalC, C5[1, 3], C4[3]], [equalC, C6, sin(q2)], [equalC, C7, - C6],

                                       [ C8  C7  0 ]
                                       [           ]
[equalC, C8, cos(q2)], [matrixm, pas20, [ C6  C8  0 ]],
                                       [           ]

                              . . . . . .

                [ C52  C37 ]
[matrixm, masse, [         ]],
                [ C37  C58 ]

                              . . . . . .


[matrixm, force, [ C102, C107 ]]]

]
```

Generated code

We call *genC* on the previous list of macroC instructions.

```
/* Equations establishment */
void calcul (tps ,etat ,dim )
double tps ,*etat ;
int dim ;
  {
  double  C1,C2,C3,C4[3],C5[1][3],C6,C7,C8,C9[3][3],C10[3][3],C11[3][3],C12[3][
3],C13[3][3],C14[3],C15[3],C16[3],C17,C18,C19[3][3],C20,C21[3],C22[3],C23[3],
C24[3],C25[3],C26[3],C27[3],C28[3],C29[3],C30[3],C31[3],C32[1][3],C33[3][3],C34
[3],C35,C36,C37,C38[3][3],C39[3][3],C40[3][3],C41[3],C42,C43[1][3],C44,C45,C46[
3],C47,C48,C49,C50,C51,C52,C53[1][3],C54,C55[1][3],C56,C57,C58,C59,C60[3],C61,
C62,C63,C64[3],C65,C66,C67[3][3],C68[3],C69[3],C70[3],C71[1][3],C72,C73,C74,C75
[3],C76[3],C77[3],C78[3],C79[3],C80[3],C81[3][3],C82[3],C83[3],C84[3],C85,C86[3
],C87[3],C88[3][3],C89[3],C90[3],C91[3][3],C92[3],C93[3],C94[3],C95[3],C96[3],
C97[3],C98,C99,C100,C101,C102,C103,C104[1][3],C105,C106,C107;
  /* State vector */
  q1  = etat [0];
  u1  = etat [2];
  q2  = etat [1];
  u2  = etat [3];
  /* masse matrix and force vector */
  C1 = sin(q1);
  C2 = -C1;
  C3 = cos(q1);
  pas10[0][1] = C2;
  pas10[2][2] = 1;
  pas10[0][0] = C3;
  pas10[1][0] = C1;
  pas10[0][2] = 0;
  pas10[2][1] = 0;
  pas10[2][0] = 0;
  pas10[1][2] = 0;
  pas10[1][1] = C3;
  l1[0] = 0;
  l1[1] = 0;
  l1[2] = 1;
  C4[0] = 0;
  C4[1] = 0;
  C4[2] = 1;
  C5[0][0] = C4[0];
  C5[0][1] = C4[1];
  C5[0][2] = C4[2];
  C6 = sin(q2);
  C7 = -C6;
  C8 = cos(q2);
  pas20[0][1] = C7;
  pas20[2][2] = 1;
  pas20[0][0] = C8;
  pas20[1][0] = C6;
  pas20[0][2] = 0;
  pas20[2][1] = 0;
  pas20[2][0] = 0;
  pas20[1][2] = 0;
  pas20[1][1] = C8;
```

```
                                . . . . . .

  masse[0][0] = C52;
  masse[0][1] = C37;
  masse[1][0] = C37;
  masse[1][1] = C58;

                                . . . . . .

  force[0] = C102;
  force[1] = C107;
  }
```

3.1.2 Enter numerical data

The algorithm computes which formal variables have not been assigned into Maple and builds a routine to read them from a file (*file.dat*) which has been written by the algorithm and must be filled by the user.

MacroC code

```
[commentC, ], [commentC,  'Data enter '],

[functionm, void, enter , [],

[[declareC, FILE, [*filepointr , *fopen()]], [declareC, int, [fclose(), i]],

[commentC, ], [commentC, 'Scalar data enter '],

[commentC, 'Simulation data '],

[commentC, 'Initial conditions '],

[fopenC, filepointr , file.dat, r],

[fscanf(filepointr ,

    string('%lf\n%lf\n%lf\n%lf\n%lf\n%lf\n%lf\n%lf\n%lf\n%lf\n%lf\n%lf\n'),

    &L, &grav, &m10, &m20, &tps_begin, &tps_end, &tps_pas, &q1, &q2, &u1, &u2)

    ],

[fcloseC, filepointr ]]

]
```

Generated code

```
/* Data enter */
void enter ()
  {
  FILE *filepointr ,*fopen();
  int fclose(),i;

  /* Scalar data enter */
  /* Simulation data */
  /* Initial conditions */
  filepointr  = fopen( "file.dat" , "r" );

  fscanf(filepointr , "%lf\n%lf\n%lf\n%lf\n%lf\n%lf\n%lf\n%lf\n%lf\n%lf\n%lf\n%lf\n\
" ,
&L,&grav,&m10,&m20,&tps_begin,&tps_end,&tps_pas,&q1,&q2,&u1,&u2);
  fclose(filepointr );
  }
```

3.1.3 Solve the set of differential equations

The system's state vector at time *tps* is assigned, the matrices *Masse* and *Force* are computed (call to the *calcul* routine), the *Masse* matrix inverse is computed (call to the *inversion* routine) and the final first order equivalent differential system is established.

MacroC code

```
[commentC, ], [commentC,  'Differential equations system '],

[functionm, void, diff_sys , [[double, [tps , *etat , *res ]]],

[[declareC, void, [calcul(), inversion()]], [commentC, ],

 [equalC, u1 , etat [3]], [equalC, res [1], u1 ], [commentC, ],

 [equalC, u2 , etat [4]], [equalC, res [2], u2 ], [commentC, ],

 [commentC,  'Equations establishment '], [callC, calcul , [tps , etat , 4]],

 [commentC, ], [commentC,  'Solve equations '],

 [callC, inversion , [masse , inv_masse , 2]],

 [equalC, res [3], - inv_masse [1, 1] force [1] - inv_masse [1, 2] force [2]],

 [equalC, res [4], - inv_masse [2, 1] force [1] - inv_masse [2, 2] force [2]]]

]
```

Generated code

```
/* Differential equations system */
void diff_sys (tps ,etat ,res )
double tps ,*etat ,*res ;
  {
  void calcul(),inversion();

  u1  = etat [2];
  res [0] = u1 ;

  u2  = etat [3];
  res [1] = u2 ;

  /* Equations establishment */
  calcul (tps ,etat ,4);

  /* Solve equations */
  inversion (masse ,inv_masse ,2);
  res [2] = -inv_masse [0][0]*force [0]-inv_masse [0][1]*force [1];
  res [3] = -inv_masse [1][0]*force [0]-inv_masse [1][1]*force [1];
  }
```

3.1.4 Write results into a file

The state vector (saved in the array *yp*) has been computed at different times stored in the array *xp*. The results are written into the file *file.res* at time intervals specified by the user (*tps_pas*) and a linear interpolation is made if the state vector has not been computed exactly at the desired time.

MacroC code

```
[commentC, ], [commentC,  'Write results to file '],

[functionm, void, sortie , [],

[[declareC, FILE, [*filepointw , *fopen()]],

[declareC, double, [tps , faux , lin_interpol()]],

[declareC, int, [binary_search(), fclose(), i, j]],

[declareC, extern, int, [kount ]],

[declareC, extern, double, [xp[] , * yp[] ]],

[fopenC, filepointw , file.res, w],

[form, tps  &= tps_begin , tps  <= tps_end  + tps_pas , tps &+= tps_pas,

[[equalC, j, binary_search(tps, xp, kount) ],
```

```
[fprintf(filepointw , string('\n%13e'), tps)],
[form, i &= 0, i <= 1, i++,
[
[equalC, faux ,
   lin_interpol(xp[j + 1], yp[j + 1, i + 1], xp[j + 2], yp[j + 2, i + 1], tps )
   ],
[fprintf(filepointw, string('%13e', faux))]
],
[fprintf(filepointw, string('\n%13e'), tps)],
[form, i &= 2, i <= 3, i++,
[
[equalC, faux ,
   lin_interpol(xp[j + 1], yp[j + 1, i + 1], xp[j + 2], yp[j + 2, i + 1], tps )
   ],
[fprintf(filepointw, string('%13e', faux))]
]
]
],
[fcloseC, filepointw ]]
]
```

Generated code

```
/* Write results to file */
void sortie ()
  {
  FILE *filepointw ,*fopen();
  double tps ,faux ,lin_interpol();
  int binary_search(),fclose(),i,j;
  extern int kount ;
  extern double xp[] ,* yp[] ;
  filepointw  = fopen( "file.res" , "w" );
  for(tps  = tps_begin ;(tps  <= tps_end +tps_pas );tps += tps_pas)
    {
    j = binary_search(tps, xp, kount) ;
    fprintf(filepointw , "\n%13e" ,tps );
    for(i = 0;(i <= 1);i++)
      {
      faux  = lin_interpol(xp[j],yp[j][i],xp[j+1],yp[j+1][i],tps );
      fprintf(filepointw , "%13e" ,faux);
      }
    fprintf(filepointw , "\n%13e" ,tps );
    for(i = 2;(i <= 3);i++)
      {
      faux  = lin_interpol(xp[j],yp[j][i],xp[j+1],yp[j+1][i],tps );
      fprintf(filepointw , "%13e" ,faux);
      }
    }
  fclose(filepointw );
  }
```

3.2 Generalized Newton method

We have chosen to expose the same application as C. Gomez has treated in [7] with the Fortran language and *Macrofort* that is : the generalized Newton method applied to a steel rolling problem.

3.2.1 A steel rolling system

We shall not explain here any mechanical consideration about how the equations below can be derived. The goal is to solve the following system :

$$
\begin{aligned}
F_1(f,h_2,\phi) &= h_2 - s - \frac{f + a_2\,(1 - e^{a_3 f})}{a_1} \\
F_2(f,h_2,\phi) &= f - lkgr\left(\frac{\pi\sqrt{h_2}\arctan(\sqrt{r})}{2\sqrt{gr}} - \frac{\pi\xi}{4} - \ln(\frac{h_n}{h_2}) + \ln(\frac{h1}{h_2})1/2\right) + \frac{gr\xi t_1}{h_2} \\
F_3(f,h_2,\phi) &= \arctan(\frac{\phi\sqrt{gr}}{\sqrt{h_2}}) - \sqrt{h_2}\left(\pi\ln(\frac{h_2}{h1})1/4 + \frac{\sqrt{gr}\arctan(\sqrt{r})}{\sqrt{h_2}} - \frac{t_1}{lkh_1} + \frac{t_2}{lkh_2}\right)1/2\frac{1}{\sqrt{gr}}
\end{aligned}
$$

with

$$
r = \frac{h_1 - h_2}{h_2},\quad \xi = \frac{\sqrt{h1 - h_2}}{\sqrt{gr}},\quad h_n = h_2 + gr\,\phi^2
$$

```
[fprintf(filepointw , string('\n%13e'), tps)],
[form, i &= 0, i <= 1, i++,
[
[equalC, faux ,
   lin_interpol(xp[j + 1], yp[j + 1, i + 1], xp[j + 2], yp[j + 2, i + 1], tps )
   ],
[fprintf(filepointw, string('%13e', faux))]
],
[fprintf(filepointw, string('\n%13e'), tps)],
[form, i &= 2, i <= 3, i++,
[
[equalC, faux ,
   lin_interpol(xp[j + 1], yp[j + 1, i + 1], xp[j + 2], yp[j + 2, i + 1], tps )
   ],
[fprintf(filepointw, string('%13e', faux))]
]
]
],
[fcloseC, filepointw ]]
]
```

Generated code

```
/* Write results to file */
void sortie ()
  {
  FILE *filepointw ,*fopen();
  double tps ,faux ,lin_interpol();
  int binary_search(),fclose(),i,j;
  extern int kount ;
  extern double xp[] ,* yp[] ;
  filepointw  = fopen( "file.res" , "w" );
  for(tps  = tps_begin ;(tps  <= tps_end +tps_pas );tps += tps_pas)
    {
    j = binary_search(tps, xp, kount) ;
    fprintf(filepointw , "\n%13e" ,tps );
    for(i = 0;(i <= 1);i++)
      {
      faux  = lin_interpol(xp[j],yp[j][i],xp[j+1],yp[j+1][i],tps );
      fprintf(filepointw , "%13e" ,faux);
      }
    fprintf(filepointw , "\n%13e" ,tps );
    for(i = 2;(i <= 3);i++)
      {
      faux  = lin_interpol(xp[j],yp[j][i],xp[j+1],yp[j+1][i],tps );
      fprintf(filepointw , "%13e" ,faux);
      }
    }
  fclose(filepointw );
  }
```

3.2 Generalized Newton method

We have chosen to expose the same application as C. Gomez has treated in [7] with the Fortran language and *Macrofort* that is : the generalized Newton method applied to a steel rolling problem.

3.2.1 A steel rolling system

We shall not explain here any mechanical consideration about how the equations below can be derived. The goal is to solve the following system :

$$
\begin{aligned}
F_1(f,h_2,\phi) &= h_2 - s - \frac{f + a_2\,(1 - e^{a_3 f})}{a_1} \\
F_2(f,h_2,\phi) &= f - lkgr\left(\frac{\pi\sqrt{h_2}\arctan(\sqrt{r})}{2\sqrt{gr}} - \frac{\pi\xi}{4} - \ln(\frac{h_n}{h_2}) + \ln(\frac{h1}{h_2})1/2\right) + \frac{gr\,\xi\,t_1}{h_2} \\
F_3(f,h_2,\phi) &= \arctan(\frac{\phi\sqrt{gr}}{\sqrt{h_2}}) - \sqrt{h_2}\left(\pi\ln(\frac{h_2}{h1})1/4 + \frac{\sqrt{gr}\arctan(\sqrt{r})}{\sqrt{h_2}} - \frac{t_1}{lkh_1} + \frac{t_2}{lkh_2}\right)1/2\frac{1}{\sqrt{gr}}
\end{aligned}
$$

with

$$
r = \frac{h_1 - h_2}{h_2},\quad \xi = \frac{\sqrt{h1 - h_2}}{\sqrt{gr}},\quad h_n = h_2 + gr\,\phi^2
$$

using the following set of initial values :

$$a_2 = 648, a_1 = 610, l = 1250, a_3 = -.00247, k = .014, gr = 360, t_1 = 12, t_2 = 35, h_1 = 24, s = 12$$

This system cannot be solved using the Maple's standard function **solve** or **fsolve** so the only remaining solution is to use numerical methods.

3.2.2 Newton's method

The method's aim is to compute a numerical solution of a non linear system of n equations and n unknown :

$$\begin{cases} f_1(x_1, \cdots, x_n) &= 0 \\ & \cdots \\ f_n(x_1, \cdots, x_n) &= 0 \end{cases}$$

Denoting X the n by 1 vector of the unknown, $F(X)$ the n by n matrix containing the equations and $\dot{F}(X)$ the Jacobian matrix of the system, Newton's algorithm can be described as follows :

X := X0 (* X0 arbitrary initial vector *)
while NORM(F(X)) > *Epsilon* **do** (* *Epsilon* is the desired accuracy for the result *)

Y := solution of the linear system $\dot{F}(X)Y = -F(X)$
Y := Y + X ;

end

As one can see, the algorithm requires the Jacobian matrix of the system which can easily be computed by an algebra system but would require a great amount of coding efforts with any other language.

3.2.3 Solving the problem using MacroC

MacroC code

The Maple's function describing the C program to handle the described problem is given below. **Resol** is a C routine for linear systems resolution (see for example [8]).

```
# STEEL ROLLING PROBLEM

f1 := h2-s-(f+a2*(1-exp(a3*f)))/a1;
f2 := f-l*k*gr*(Pi*sqrt(h2/gr)*arctan(sqrt(r))/2-Pi*csi/4-log(hn/h2)+
                log(h1/h2)/2) + gr*csi*t1/h2;
f3 := arctan(phi*sqrt(gr/h2))-sqrt(h2/gr)*(Pi*log(h2/h1)/4+sqrt(gr/h2)*
            arctan(sqrt(r))-t1/k/l/h1+t2/k/l/h2)/2;
r := (h1-h2)/h2;
csi := sqrt((h1-h2)/gr);
hn := h2+gr*phi^2;

# Numerical data
data := {a1=610, a2=648, a3=-.00247, l=1250, k=1.4*10^(-2), gr=360,
         t1=12, t2=35, h1=24, s=12};

F[1] := subs({op(data), f=x[1], h2=x[2], phi=x[3]}, f1);
F[2] := subs({op(data), f=x[1], h2=x[2], phi=x[3]}, f2);
F[3] := subs({op(data), f=x[1], h2=x[2], phi=x[3]}, f3);
```

```
# f : vector of the equations
# x : vector of the unknown values
# n : number of equations
Newton := proc(ff, x, n)
local ii, jj, l, l1, l2, l3, l4, jac;

# Jacobian matrix
jac := array(1..n,1..n);
for ii to n do
   for jj to n do
      jac[ii, jj] := diff(ff[ii], x[jj]);
   od;
od;

# Preprocessor commands
l1 := [includeC, '<stdio.h>'],
      [includeC, '<math.h>'],
      [defineC, m, 3],
      [commentC, ''],
      [declareC, double, [f[m+1], zj[m+1,m+1],x[m+1], eps]],
      [commentC, ''];

# The NORM function
l2 := [functionm, double, NORM, [[double, ['*f']]],
         [[declareC, double, [&=(res, 0)]],
          [declareC, int, [i]],
          [form, &=(i,0), i<n, &=(i, i+1),
                [equalC, res, res+f[i+1]^2]],
          [returnC, sqrt(res)]
         ]];

# Initialization of the vector x
l3 :=  [commentC, 'Initialization of the X vector'],
       [form, &=(i, 0), i<n, &=(i, i+1),
            [[callC, 'printf ', [string('x[%d] ='), i]],
             [scanf(string('%d\\n'), x[i+1])]
     ]];

# Calculation of the equations
for ii to n do
   l3 := l3, [equalC, f[ii], ff[ii]];
od;
```

```
  l4 := [commentC, 'The Jacobian matrix'],
        [matrixm, zj, jac],
        [commentC, ''],
        [commentC, 'Resolution by a linked C routine'],
        [callC, resol, [zj, f, n]],
        [commentC, ''],
        [form, &=(i, 0), i<n, &=(i, i+1),
              [equalC, x[i+1], -f[i+1]+x[i+1]]
        ];
  for ii to n do
     l4 := l4, [equalC, f[ii], ff[ii]];
  od;

  # The MAIN function
  l  := [commentC, ''],
        [commentC, 'The Main procedure'],
        [functionm, void, main, [],
           [[declareC, void, [resol()]],
            [declareC, int, [i]],
            [callC, 'printf ', [string('eps =')]],
            [scanf(string('%d'), eps)], l3,
            [whilem, NORM(f) >= eps,
                    [l4]
            ],
            [callC, 'printf ', [string(cat('%d\\n'$n)), '&x'[ii]$'ii'=1..n]]
           ]];


  genC([l1, l2, l], filename='Newton.c');
  end:
```

Generated code

```
# include <stdio.h>
# include <math.h>
# define m 3

double f[m],zj[m][m],x[m],eps;

double NORM(f)
double *f;
  {
  double res = 0;
  int i;
  for(i = 0;(i < 3);i++)
    res += pow(f[i],2.0);
  return (sqrt(res));
  }

/*The Main procedure*/
void main()
  {
```

```
  void resol();
  int i;
  printf( "eps =" );
  scanf( "%d" ,eps);
  /*Initialization of the X vector*/
  for(i = 0;(i < 3);i++)
    {
    printf( "x[%d] =" ,i);
    scanf( "%d\n" ,x[i]);
    }
  f[0] = x[1]-3984/305-x[0]/610+324/305*exp(-0.247E-2*x[0]);
  f[1] = x[0]-0.875E1*0.3141593E1*sqrt(x[1])*sqrt(360)*atan(sqrt(24-x[1])/sqrt(
x[1]))+0.4375E1*0.3141593E1*sqrt(24-x[1])*sqrt(360)+0.63E4*log((x[1]+360*pow(x[
2],2.0))/x[1])-0.315E4*log(24/x[1])+12*sqrt(360)*sqrt(24-x[1])/x[1];
  f[2] = atan(x[2]*sqrt(360)/sqrt(x[1]))-sqrt(x[1])*sqrt(360)*(0.3141593E1*log(
x[1]/24)/4+sqrt(360)/sqrt(x[1])*atan(sqrt(24-x[1])/sqrt(x[1]))-0.2857143E-1+
0.2E1/x[1])/720;
  while((eps <= NORM(f)))
    {
    /*The Jacobian matrix*/
    zj[2][1] = -x[2]*sqrt(360)/sqrt(pow(x[1],3.0))/(1+360*pow(x[2],2.0)/x[1])/2
-1/sqrt(x[1])*sqrt(360)*(0.3141593E1*log(x[1]/24)/4+sqrt(360)/sqrt(x[1])*atan(
sqrt(24-x[1])/sqrt(x[1]))-0.2857143E-1+0.2E1/x[1])/1440-sqrt(x[1])*sqrt(360)*(
0.3141593E1/x[1]/4-sqrt(360)/sqrt(pow(x[1],3.0))*atan(sqrt(24-x[1])/sqrt(x[1]))
/2+sqrt(360)/sqrt(x[1])*(-1/(sqrt(24-x[1]))/sqrt(x[1])/2-sqrt(24-x[1])/sqrt(
pow(x[1],3.0))/2)/(1+(24-x[1])/x[1])-0.2E1/pow(x[1],2.0))/720;
    zj[2][2] = sqrt(360)/sqrt(x[1])/(1+360*pow(x[2],2.0)/x[1]);
    zj[1][1] = -0.4375E1*0.3141593E1/sqrt(x[1])*sqrt(360)*atan(sqrt(24-x[1])/
sqrt(x[1]))-0.875E1*0.3141593E1*sqrt(x[1])*sqrt(360)*(-1/(sqrt(24-x[1]))/sqrt(x
[1])/2-sqrt(24-x[1])/sqrt(pow(x[1],3.0))/2)/(1+(24-x[1])/x[1])-0.21875E1*
0.3141593E1/sqrt(24-x[1])*sqrt(360)+0.63E4*(1/x[1]-(x[1]+360*pow(x[2],2.0))/
pow(x[1],2.0))/(x[1]+360*pow(x[2],2.0))*x[1]+0.315E4/x[1]-6*sqrt(360)/sqrt(24-x
[1])/x[1]-12*sqrt(360)*sqrt(24-x[1])/pow(x[1],2.0);
    zj[0][0] = -1/610-0.2623869E-2*exp(-0.247E-2*x[0]);
    zj[0][2] = 0;
    zj[1][2] = 0.4536E7*x[2]/(x[1]+360*pow(x[2],2.0));
    zj[0][1] = 1;
    zj[1][0] = 1;
    zj[2][0] = 0;

    /*Resolution by a linked C routine */
    resol(zj,f,3);

    for(i = 0;(i < 3);i++)
      x[i] += -f[i];
    f[0] = x[1]-3984/305-x[0]/610+324/305*exp(-0.247E-2*x[0]);
    f[1] = x[0]-0.875E1*0.3141593E1*sqrt(x[1])*sqrt(360)*atan(sqrt(24-x[1])/
sqrt(x[1]))+0.4375E1*0.3141593E1*sqrt(24-x[1])*sqrt(360)+0.63E4*log((x[1]+360*
pow(x[2],2.0))/x[1])-0.315E4*log(24/x[1])+12*sqrt(360)*sqrt(24-x[1])/x[1];
```

```
    f[2] = atan(x[2]*sqrt(360)/sqrt(x[1]))-sqrt(x[1])*sqrt(360)*(0.3141593E1*
log(x[1]/24)/4+sqrt(360)/sqrt(x[1])*atan(sqrt(24-x[1])/sqrt(x[1]))-0.2857143E-1
+0.2E1/x[1])/720;
    }
  printf( "%d\n%d\n%d\n" ,&x[3],&x[3],&x[3]);
  }
```

Chapter 4

Conclusion

The macroC package provides a new and easy way to connect Maple to the C numerical language. We have shown that it is possible to use Maple's large features to make a formal analysis of a numerical problem and to invoke C language to effectively compute the results with a rational amount of time and storage space. The actual package can surely be extended and improved and we expect users to report any comment on the software.

Bibliography

[1] Bruce W. Char, Keith O. Geddes, Gaston H. Gonnet, Benton L. Leong, Michael B. Monagan, and Stephen M. Watt. *MapleV Language Reference Manual.* Springer-verlag, 1992.

[2] Bruce W. Char, Keith O. Geddes, Gaston H. Gonnet, Benton L. Leong, Michael B. Monagan, and Stephen M. Watt. *MapleV Librairy Reference Manual.* Springer-verlag, 1992.

[3] Bruce W. Char, Keith O. Geddes, Gaston H. Gonnet, Benton L. Leong, Michael B. Monagan, and Stephen M. Watt. *MapleV First Leaves.* Springer-verlag, 1992.

[4] S. Wolfram. *Mathematica — A System for Doing Mathematics by Computer.* Addison-Wesley Publishing Company, Redwood City, California, 1988.

[5] R. Johnsonbaugh and M. Kalin. *Applications programming in C.* Macmillan Publishing Company - New York.

[6] B.W. Kernighan and D.M. Ritchie. *The C programming language.* Prentice-Hall.

[7] Claude Gomez. Macrofort : a fortran code generator in maple. Rapport technique 119, INRIA, 1990.

[8] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling. *Numerical recipes in C : the art of scientific computing.* Cambridge University Press, 1988.

Unité de Recherche INRIA Sophia Antipolis
2004, route des Lucioles - B.P. 93 - 06902 SOPHIA ANTIPOLIS Cedex (France)

Unité de Recherche INRIA Lorraine Technopôle de Nancy-Brabois - Campus Scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 VILLERS LES NANCY Cedex (France)
Unité de Recherche INRIA Rennes IRISA, Campus Universitaire de Beaulieu 35042 RENNES Cedex (France)
Unité de Recherche INRIA Rhône-Alpes 46, avenue Félix Viallet - 38031 GRENOBLE Cedex (France)
Unité de Recherche INRIA Rocquencourt Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)

EDITEUR
INRIA - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)

ISSN 0249 - 6399