Transfor Translating Maple Procedures into Fortran with BLAS Code Generation

Claude Gomez

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

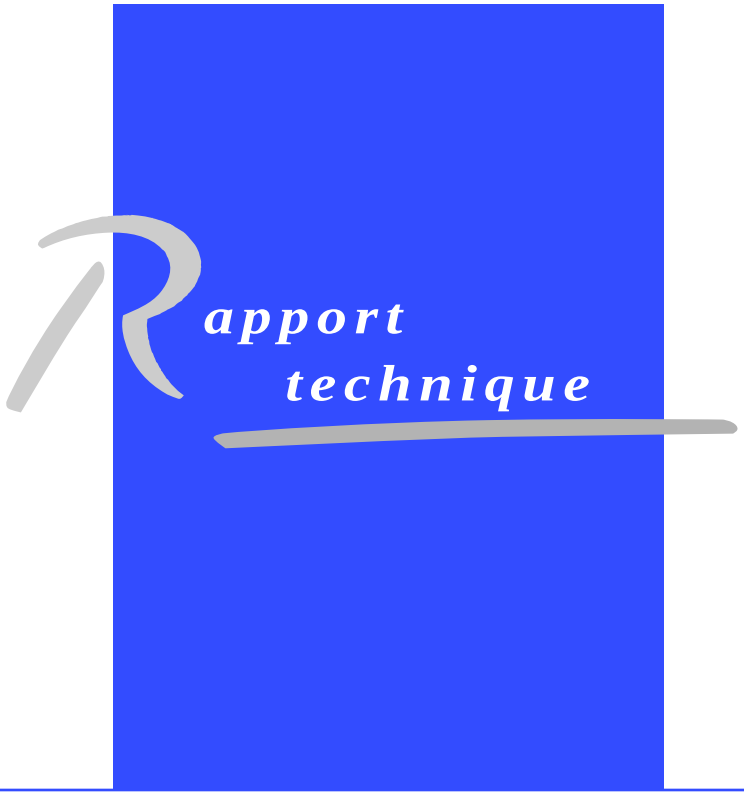
Transfor
Translating Maple Procedures into Fortran with BLAS Code Generation

Claude Gomez

No 0223

Août 1998

THÈME 4



ISSN 0249-0803

Transfor
Translating Maple Procedures into Fortran with BLAS Code Generation

Claude Gomez

Thème 4 — Simulation et optimisation
de systèmes complexes
Projet Meta2

Rapport technique n ° 0223 — Août 1998 — 20 pages

Abstract: Transfor is a Maple package which translates Maple procedures into Fortran 77 subroutines. Algebraic expressions, assignements, loops and conditionals are translated. Moreover, Matrix expressions are translated into BLAS calls.

Key-words: Maple, Fortran, BLAS, code generation, Macrofort.

(Résumé : tsvp)

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : 01 39 63 55 11 - International : +33 1 39 63 55 11
Télécopie : (33) 01 39 63 53 30 - International : +33 1 39 63 53 30

Transfor
Traduction de procédures Maple en fortran avec génération de code BLAS

Résumé : Transfor est un package Maple qui permet de traduire des procédures Maple en sous-programmes fortran 77. Il traduit les expressions algébriques, les assignations, les boucles et les expressions conditionnelles. De plus, les expressions matricielles sont traduites par des appels à des fonctions ou des sous-programmes BLAS.

Mots-clé : Maple, fortran, BLAS, génération de code, Macrofort.

1 Introduction

Transfor is a Maple package which translates Maple procedures into Fortran 77 subroutines. Matrix expressions are translated into BLAS [2] calls. Maple procedures are first translated into Macrofort [5], then Fortran code is generated.

The possibility of translating Maple procedures into Fortran already exists in Maple by using the Codegen package. The purpose of Codegen is to translate Maple procedures into C or Fortran code for the user who works within Maple and wants to achieve efficiency. The purpose of Transfor is not exactly the same. With Tranfor, Maple is not only used for its Computer Algebra capabilities but also as a friendly language to generate easily efficient Fortran code. Transfor is for the engineer or the scientist who only uses Computer Algebra as a front end for numerical computations: this user generally uses a system like Matlab [6] or Scilab [9] or even makes his own Fortran programs. He only wants to make symbolic computations before numerical computations (see for instance [1]) or he wants simply to generate easily BLAS Fortran code (see the example of section 5.1).

First Transfor translates procedure declarations into subroutine declarations. Then, most Maple statements are translated into Fortran statements. In particular, Transfor translates `if` conditionals, most `for` loops and the assign instruction `:=`. Transfor translates algebraic expressions. It is not able to translate Maple expressions which do not exist in Fortran such as lists, sets or ranges. Maple functions which do not exist in Fortran are not translated, for instance `map`, `op`... Transfor is able to translate matrix expressions in an efficient way. The efficiency is a very important point. Indeed, Fortran 77 is very efficient for matrix computations but it only knows scalar syntax (Fortran 90 [8] is able to do directly matrix operations but is not yet widely used by engineers). The question is: how to translate these matrix operations? There exists a set of very efficient Fortran subroutines doing the job. These are the Basic Linear Algebra Subprograms: BLAS [2]. These subroutines are now the basis of NAG Fortran library [7] and LAPACK library [4] and are machine-coded on a lot of computers. So, the best way to translate matrix expressions into Fortran seems to translate them into BLAS Fortran calls: Transfor does that.

2 Maple Expressions Translated by Transfor

It is clear that any Maple expression cannot be translated into Fortran. We cannot use expressions or functions which do not exist in Fortran. So, we must have in mind that we want to translate the Maple procedure into Fortran when we make it. In particular, Fortran is not case sensitive. So, you must not use names that differ only by the case of letters. Also, in standard Fortran 77, names must not exceed 6 letters.

2.1 Algebraic Expressions

Pure algebraic expressions are translated into Fortran in a straightforward way. When a function appears in such an expression, it can be a mathematical Maple function correspon-

ding to an intrinsic Fortran function or not. In the second case, there is a direct translation:
`foo(a,b)` $\longrightarrow$ `foo(a,b)`
In the first case the function is translated into the corresponding Fortran intrinsic function with good function name and good argument type:
`arcsin(123)` $\longrightarrow$ `asin(123.D0)`

Currently, the recognized mathematical Maple functions corresponding to intrinsic Fortran functions are: `abs`, `arccos`, `arcsin`, `arctan`, `cos`, `cosh`, `exp`, `ln`, `sin`, `sinh`, `sqrt`, `tan` and `tanh`.

Arguments of functions or subroutines must have the good expected type in Fortran. If you use numbers as arguments, you have to give the good type: integer or float. Integers are translated into Fortran integers and floats are translated into Fortran double precision numbers. There is a problem with zero. In Maple, zero is always an integer. If you want to specify double precision zero, you have to give the string `'0.0D0'` or the function `dble(0)`.

Conditionals `if` are translated into Fortran.

Most loops are translated into Fortran. But the following forms are not translated:

- `for in` loops
- infinite loops: `for var do` <> `od` and `do` <> `od`
- pure `for` with no variable: `to` <*end*> `do` <> `od`

In loops `next` and `break` are not translated.

The assign operator `:=` is translated into the Fortran assignment.

2.2 Matrix Expressions

Currently Transfor expects only matrix expressions involving double precision arrays. Moreover, these expressions must be only vectors or matrices, not general arrays (with 3 or more dimensions).

We call matrix expression in Maple every assignment of the form:
<*name*>`:=`<*matrix function*>`(`<>`)`
where <*matrix function*> can be `evalm` or `copy`, or one of the following functions of the `linalg` package of Maple: `augment` (or `concat`), `copy`, `dotprod`, `evalm`, `linsolve`, `matadd`, `multiply`, `norm`, `scalarmul`, `stackmatrix` or `transpose`. Other functions will be added in the future.

Only function of scalar expressions can appear in matrix expressions. For instance, Transfor is able to translate

```
a:=evalm(f(dotprod(a,b)) &* b)
```

where `a` and `b` are arrays and `f` is a scalar function, but not

```
a:=evalm(f(a) &* b)
```

where `f` is a matrix function.

The syntax `linalg[`<*matrix function*>`]` is not recognized by Transfor. So, you have to load the `linalg` package by issuing the command `with(linalg):` if you want to use the functions of this package.

The translation of a matrix expressions is announced in the generated Fortran code by a comment, for instance:

```
c       computing matrix value of d
```

The Fortran code corresponding to the matrix expression follows this comment.

Before translating a matrix expression, Transfor checks the dimensions of all the variables in the matrix operations. This is a complete checking of the coherency of the matrix expression. So, you have to declare and define the arrays occurring in the procedure. The way to do that is explained below.

Appendix A shows the class of matrix expressions which are recognized by Transfor and the way they are parsed.

2.2.1 Array Arguments

It is prohibited in Maple to declare array arguments of procedures with the `vector` and `array` commands. Transfor must know the size of these array arguments, so you need to declare them with the new Maple command `transdcl` within the procedure. Its syntax is:

`transdcl(a,d_1,...,d_n)`

`transdcl([a_1,...,a_n],d_1,...,d_n)`

where `a`, `a_1`, ..., `a_n` are names of matrices or vectors and `d_1`, ..., `d_n` are the values of the sizes which can be a range or a value `n` (which means `1..n`).

For instance, for declaring that `v` is a n-vector:

`transdcl(v,n)`

and for declaring that `a` and `b` are $m \times n$ matrices:

`transdcl([a,b],m,n)` or `transdcl([a,b],1..m,1..n)`.

2.2.2 Local Arrays

In Maple procedures, you must not use global arrays, but you can use local vectors or matrices. You must declare and define these local arrays in Maple procedures with the `vector` and `array` Maple commands. The only recognized syntax are `vector(n)` and `array(d_1,...,d_n)` where `d_1`, ..., `d_n` are ranges. Other syntaxes will be recognized in the future.

But, if the first time you use a local array is an assignment of this array, you need not declare and define it because its type and sizes can be inferred by the assignment. But declaring them does not hurt.

In Maple procedures, local arrays are dynamically allocated. This is not possible in Fortran. So, all Maple local arrays are put in a single Fortran array called `zloc` and it appears at the end of the calling list of the generated Fortran subroutine. The pointers to the beginning of the local arrays in `zloc` array are the integers `iz1`, `iz2`, ..., `izn`. The size needed for this array is given as a comment at the beginning of the subroutine.

For instance:

```
c       memory space of zloc : (6*n^2)
```

means that you have to give the local Fortran array `zloc` the size $6n^2$.

The equivalence between the local variable and the array `zloc` is given in the subroutine as a comment. For instance:

```
c       x0 : zloc(iz1)
          iz1 = 1
```

means that the Maple local array `x0` is replaced by Fortran `zloc(iz1)`.

2.2.3 Fortran Working Array

When performing matrix computations, Fortran working arrays are needed for intermediate computations. In the same way as local arrays, only one working array is declared in the Fortran subroutine and it is automatically managed by Transfor in an optimal way with respect to its size. Its name is `work` and it appears at the end of the calling list of the generated Fortran subroutine. The pointers to the beginning of intermediate variables in `work` array are the integers `iw1`, `iw2`, ..., `iwn`. The size needed for this array is given as a comment at the beginning of the subroutine.

For instance:

```
c       memory space of work : (max(n,n^2)+n^2+n)
```

means that you have to give the working array the size $2n^2 + n$.

2.2.4 Reserved Names

The following names `work`, `iw1`, `iw2`, ..., `iwn`, `zloc`, `iz1`, `iz2`, ..., `izn`, `istack`, `izero`, `jzero` are reserved. You must not use them in Maple procedures.

2.2.5 Fortran Subroutines Used for the Translation

The following BLAS subroutines are used for the translation into Fortran: `daxpy`, `dcopy`, `dgemm`, `dgemv` and `dger`.

The following BLAS functions are used for the translation into Fortran: `ddot` and `dnrm2`.

There is no BLAS subroutine performing the transposition. Indeed most of the time there are flags which allow to perform the transpositions when doing other operations with BLAS subroutines. So, a single transposition could be a bad formulation of the matrix operation. But, if you want to use a single transposition, Tranfor translates it into a call to `dtrans` subroutine:

```
call dtrans(m,n,a,b)
```

where `a` is the input $m \times n$ matrix and `b` is the output matrix (the transpose of `a`). The user must provide `dtrans` Fortran subroutine. In fact, Transfor uses a procedure called `gendtrans` with `m`, `n`, `a` and `b` as arguments which evaluates to the list `[dtrans,m,n,a,b]` by default which will then generate `call dtrans(m,n,a,b)`. The user can change it to customize the translation of the transposition. For instance, if you define

```
gendtrans:=proc(m,n,a,b) ['dtransc',m,n,a] end:
```

you obtain a call to `dtransc` subroutine as `call dtransc(n,n,b)`.

There is no BLAS subroutine solving a linear system. Many library subroutines exist for that. Tranfor translates this operation into a call to `dsolve` subroutine:

```
dsolve(n,a,b,x)
```

where `a` if the input $n \times n$ matrix, `b` is the input right hand side vector and `x` is the output vector (the solution). The user must provide `dsolve` Fortran subroutine. Transfor uses a procedure called `gendsolve` with `n`, `a`, `b` and `x` as arguments which evaluates to the list `[dsolve,n,a,b,x]` by default which will then generate `dsolve(n,a,b,x)`. The user can change it to customize the translation of the solution of a linear system, as for the transposition above.

There is no BLAS subroutine giving the power of a matrix. Tranfor translates this operation into a call to `dpow` subroutine:

```
dpow(n,m,a,b,w)
```

where `a` is the input $n \times n$ matrix, `m` is the power, `b` is the output matrix (`a` to `m` power) and `w` is a working array (needed for this operation). The user must provide `dpow` Fortran subroutine. Transfor uses a procedure called `gendpow` with `n`, `m`, `a` and `b` as arguments which evaluates to the list `[dpow,n,m,a,b,w]` by default which will then generate `dpow(n,m,a,b,w)`. The user can change it to customize the translation of the power of a matrix, as for the transposition above.

"concatenation" and "stacking" of arrays are translated by repeated calls to BLAS `dcopy` subroutine. Most of the time, a do loop is generated for the "stacking" operation and the Fortran integer variable `istack` is used.

2.3 Output of the Subroutine

In Maple the value returned by a procedure is the last evaluated expression. In Fortran subroutines values return from arguments. To have a good generated Fortran subroutine, you have to do the same thing in Maple. But remember it is prohibited to modify the value of an argument in Maple procedures, so you need to quote it when using the procedure (see the example of section 5.2).

3 How to Use Transfor

Using Transfor is very easy. When you want to translate a Maple procedure into Fortran, you use the `transfor` Maple procedure. It takes two arguments: the name of the Maple procedure you want to translate and the name of the Fortran subroutine you want to generate (they can be the same). If you want to have the result into a file, you use the `writeto` command.

For instance, if you want to translate the `foo` Maple procedure into the `foo` Fortran subroutine in file `foo.f`:

```
writeto(‘foo.f‘); transfor(foo,'foo'); writeto(terminal);
```

By default, the generated Fortran subroutine will have the same arguments as the Maple procedure. Sometimes, it is useful to give the Fortran subroutine others arguments. For

that, Transfor can have a third optional argument which is a list with *all* the arguments of the generated Fortran subroutines. For example, the Fortran subroutine often needs the sizes of the input matrices as arguments. If you want to translate:

```
f:=proc(a,b,c) transdcl(a,m,n); transdcl(b,n,m);
   c:=evalm(a &* b);
end;
```

you need to call Transfor the following way:

```
transfor(foo,'foo',[a,b,m,n,c]);
```

in order to generate:

```
c
c     SUBROUTINE foo
c
      subroutine foo(a,b,m,n,zloc(iz1),zloc)
        implicit doubleprecision (a-h,o-z)
        dimension zloc(*)
        dimension a(m,n)
        dimension b(n,m)
c       memory space of zloc : m**2
c       computing matrix value of c : zloc(iz1)
          iz1 = 1
          call dgemm('n','n',m,m,n,0.1D1,a,m,b,n,0.0D0,zloc(iz1),m)
      end
```

In fact, the Maple procedure is first translated into a Macrofort list describing the subroutine and then the Fortran code is generated. If you only want to generate the Macrofort list, for instance to use it as a basis for another Fortran program, you can use the `transmac` Maple procedure with the same arguments as `transfor`.

Appendix B shows briefly the various steps of Transfor when translating a Maple procedure into Fortran.

To compile the generated Fortran subroutine, you have to link double precision BLAS library to it. This library is public domain and you can get it for example at the following Web site:

`http://www.netlib.org/blas/`

4 Where to get Transfor

You can get Transfor package with source code, manual and examples at the following Web site:

`http://www-rocq.inria.fr/scilab/gomez/transfor/`

The latest version, corresponding to this manual, is release 2.0.

5 Examples

5.1 Simple Matrix Computation

We give below a small example. `comp` is a Maple procedure computing the value of the matrix expression $A * B^T * A + B * A^T * B$ where A and B are given square matrices. The result is returned into matrix C. This small program could be a repeated computation in a big program. The `comp` Maple procedure is:

```
comp := proc(a,b,c,n)
  local x,y;
  transdcl([a,b,c],1..n,1..n);
  x:=array(1..n,1..n);
  y:=array(1..n,1..n);
  x:=evalm(a &* transpose(b) &* a);
  y:=evalm(b &* transpose(a) &* b);
  c:=evalm(x + y);
end:
transfor(comp,'comp');
```

The generated Fortran subroutine is `comp`:

```
c
c     SUBROUTINE comp
c
      subroutine comp(a ,b ,c ,n ,work,zloc)
        implicit doubleprecision (a-h,o-z)
        dimension zloc(*)
        dimension work(*)
        dimension a (n ,n )
        dimension b (n ,n )
        dimension c (n ,n )
c       memory space of zloc : (2*n ^2)
c       memory space of work : n ^2
c       x  : zloc(iz1)
          iz1 = 1
c       y  : zloc(iz2)
          iz2 = 1+n **2
c       computing matrix value of x
          iw1 = 1
          call dgemm('n','t',n ,n ,n ,0.1D1,a ,n ,b ,n ,0.0D0,work(iw1),
     +n )
          call dgemm('n','n',n ,n ,n ,0.1D1,work(iw1),n ,a ,n ,0.0D0,zlo
     +c(iz1),n )
c       computing matrix value of y
          iw1 = 1
          call dgemm('n','t',n ,n ,n ,0.1D1,b ,n ,a ,n ,0.0D0,work(iw1),
```

```
     +n )
          call dgemm('n','n',n ,n ,n ,0.1D1,work(iw1),n ,b ,n ,0.0D0,zlo
     +c(iz2),n )
c       computing matrix value of c
          iw1 = 1
          call dcopy(n **2,zloc(iz1),1,work(iw1),1)
          call daxpy(n **2,0.1D1,zloc(iz2),1,work(iw1),1)
          call dcopy(n **2,work(iw1),1,c ,1)
       end
```

The result is also returned into array `c`. We can see in the argument list of Fortran subroutine `comp` the working array `work` and the local variable array `zloc`.

We stress the fact that both `comp` functions work in their respective languages. In this simple example, the only difference between a pure Maple procedure `comp` and the same to be translated into Fortran is the use of the `transdcl` command.

A main program using and testing the subroutine `comp` could have been generated by using Macrofort.

With this example, we can see that Transfor can be helpful when writing Fortran code involving matrix computations, even if we do not make symbolic computations.

5.2 Optimization Program

We give below an example of a complete program. `gmin` is a Maple procedure computing the following minimum of a matrix expression:

$$\min_x(x^T(AB^TA + BA^TB)x + c^Tx)$$

by an iterative gradient method. The result is returned into `x`. The initialization point `x0` of the algorithm is given in the procedure. The `gmin` procedure is:

```
gmin := proc(a,b,c,x,n)
  local d,eps,f,f0,i,ro,x0;
  transdcl([a,b],1..n,1..n);
  transdcl([c,x],1..n);
  x0:=vector(n);
  d:=array(1..n,1..n);
  for i from 1 to n do x0[i]:=0 od;
  d:=evalm(a &* transpose(b) &* a + b &* transpose(a) &* b);
  eps:=1.0*10^(-4); ro:=0.001;
  f0:=0; f:=1000.0;
  while (abs(f-f0) > eps) do
    f0:=f;
    x0:=evalm(x0 - ro * (2 * d &* x0 - c));
    f:=evalm(transpose(x0) &* d &* x0 - transpose(c) &* x0);
  od;
  x:=copy(x0);
end:
```

The generated Fortran subroutine is `gmin`. We want to test it and we use Macrofort to generate the main program. `a`, `b` and `c` are arrays to test the program. Maple procedure `gen_gmin` generates the main program and the subroutine `gmin`:

```
a:=array([[4.0,-2.0],[-2.0,4.0]]);
b:=array([[7.0,-1.0],[-1.0,8.0]]);
c:=vector([100.0,100.0]);

gen_gmin := proc(a,b,c,n)
  local pg;
  precision:='double';
  writeto(`mgmin.f`);
# main program generated with MACROFORT
  pg:=[['declaref','doubleprecision',[aa[n,n],bb[n,n],cc[n],x[n]]],
       ['declaref','doubleprecision',[w[2*n^2+n],z[n+n^2]]],
       ['matrixm','aa',a],[matrixm,'bb',b],
       ['matrixm','cc',c],
       ['callf','gmin',[aa,bb,cc,x,n,w,z]],
       ['writem',output,[`(2x,e14.7)`],[x]]];
  pg:=['programm','mgmin',pg];
  genfor(pg);
# subroutine translated from Maple procedure
  transfor(gmin,'gmin');
  writeto(terminal);
end:

gen_gmin(a,b,c,2);
```

The complete Fortran program is given below:

```
c
c     SUBROUTINE gmin
c
      subroutine gmin(a ,b ,c ,x ,n ,work,zloc)
        implicit doubleprecision (a-h,o-z)
        dimension zloc(*)
        dimension work(*)
        dimension a (n ,n )
        dimension b (n ,n )
        dimension c (n )
        dimension x (n )
c       memory space of zloc : (n +n ^2)
c       memory space of work : (max(n ,n ^2)+n ^2+n )
c       x0  : zloc(iz1)
          iz1 = 1
c       d  : zloc(iz2)
          iz2 = 1+n
```

```
c
        do 1000, i =1,n ,1
          zloc(iz1+i -1) = 0
 1000   continue
c
c       computing matrix value of d
          iw1 = 1
          iw2 = 1+n **2
          call dgemm('n','t',n ,n ,n ,0.1D1,a ,n ,b ,n ,0.0D0,work(iw1),
     +n )
          call dgemm('n','n',n ,n ,n ,0.1D1,work(iw1),n ,a ,n ,0.0D0,wor
     +k(iw2),n )
          call dgemm('n','t',n ,n ,n ,0.1D1,b ,n ,a ,n ,0.0D0,work(iw1),
     +n )
          call dgemm('n','n',n ,n ,n ,0.1D1,work(iw1),n ,b ,n ,0.1D1,wor
     +k(iw2),n )
          call dcopy(n **2,work(iw2),1,zloc(iz2),1)
        eps  = 0.1D-3
        ro  = 0.1D-2
        f0  = 0
        f  = 0.1D4
c
c       WHILE  (eps <abs ((f +(-1*f0 )))) DO <WHILE_LIST> (1)
c
c
c       WHILE LOOP BEGINNING
 1001   continue
c
c       WHILE LOOP TERMINATION TESTS
        if (eps .lt.abs (f -f0 )) then
c
c         NEW LOOP ITERATION
c
c         <WHILE_LIST>
            f0  = f
c           computing matrix value of x0
              iw1 = 1
              iw2 = 1+n
              iw3 = 1+n +n **2
              call dcopy(n ,zloc(iz1),1,work(iw1),1)
c
              do 1002, izero=1,n **2
                work(iw2+izero-1) = 0
 1002         continue
c
              call daxpy(n **2,2.D0,zloc(iz2),1,work(iw2),1)
```

```
               call dgemv('n',n ,n ,0.1D1,work(iw2),n ,zloc(iz1),1,0.0D0,
     +work(iw3),1)
               call daxpy(n ,-1.D0,c ,1,work(iw3),1)
               call daxpy(n ,-ro ,work(iw3),1,work(iw1),1)
               call dcopy(n ,work(iw1),1,zloc(iz1),1)
c            computing matrix value of f
               iw1 = 1
               iw2 = 1+n
               iw3 = 2+n
               call dgemv('n',n ,n ,0.1D1,zloc(iz2),n ,zloc(iz1),1,0.0D0,
     +work(iw1),1)
               work(iw2) = ddot(n ,zloc(iz1),1,work(iw1),1)
               work(iw1) = ddot(n ,c ,1,zloc(iz1),1)
               work(iw3) = -work(iw1)
               f  = work(iw2)+work(iw3)
          goto 1001
c
c         NORMAL WHILE LOOP TERMINATION
          endif
c         WHILE LOOP END (1)
c         computing matrix value of x
            call dcopy(n ,zloc(iz1),1,x ,1)
       end
```

We can see that both Maple procedure and Fortran program work and give the same result. The Maple session is:

```
> a:=array([[4.0,-2.0],[-2.0,4.0]]):
> b:=array([[7.0,-1.0],[-1.0,8.0]]):
> c:=vector([100.0,100.0]):
> gmin(a,b,c,'xx',2);
                          [.4769075489, .4303324980]
```

and the execution of the Fortran program gives:

```
% gmin
   0.4769075E+00
   0.4303325E+00
```

A Compiling Matrix Expressions

We describe in this appendix the way matrix expressions are translated into Macrofort-like intermediate language, *i.e.* as a list with a keyword defining the matrix operation to be done as the first element of the list. After this first translation, the lists are then translated into Macrofort lists corresponding to calls to BLAS routines. Before the translation into

BLAS calls, the user could modify the source code of Transfor and give its own translation of matrix operations.

The matrix operations taken into account are:

- matrix or scalar addition: $+$
- matrix product: $\cdot$
- product of a matrix by a scalar or scalar product: $\times$
- power of a matrix to an integer or power of scalars: $\hat{}$
- matrix inversion: forbidden
- matrix functions: **dotprod**, **stack**, **transpose** and others

A.1 Grammar of Matrix Expression

We first show the class of matrix expressions we consider. We formalize this general class of matrix expressions with a grammar in a Backus-Naur form [3]. It is not a pure context-free grammar because of the distinction between matrix expressions and scalar expressions.

The sign $\equiv$ means "definition" and explains the redundancy of the grammar for expressions which look the same but can be scalar or matrix. We need to distinguish them. Terms between $<$ and $>$ stand for themselves.

The grammar is:

$$
\begin{array}{rcl}
matrix\_expression & \longrightarrow & matrix\_expression + product \mid product \\
product & \longrightarrow & scalar\_prod \times mat\_prod \mid mat\_prod \\
scalar\_prod & \longrightarrow & scalar \mid scalar\_matrix\_expression \\
scalar & \longrightarrow & \text{<pure scalar>} \\
 & & \\
mat\_prod & \equiv & \text{<matrix product with no scalar value>} \\
mat\_prod & \longrightarrow & mat\_prod \cdot mat\_pfun \mid mat\_pfun \\
mat\_pfun & \longrightarrow & mat\_fun \mid mat\_power \\
mat\_power & \longrightarrow & mat\_fun \hat{} scalar \mid mat\_fun \hat{} scalar\_matrix\_expression \\
mat\_fun & \longrightarrow & f_1(matrix\_expression) \mid \\
 & & f_2(matrix\_expression, matrix\_expression) \mid \ldots \mid \\
 & & f_n(matrix\_expression, matrix\_expression, \ldots) \mid mat \\
mat & \longrightarrow & \text{<matrix>} \mid \text{<vector>} \\
f_i & \longrightarrow & \text{<matrix function with } i \text{ arguments, } i \geq 1\text{>} \\
 & & \\
scalar\_matrix\_expression & \equiv & \text{<}matrix\_expression\text{ with scalar value>} \\
scalar\_matrix\_expression & \longrightarrow & scalar + scalar\_matrix\_addition \mid scalar\_matrix\_addition \\
scalar\_matrix\_addition & \longrightarrow & scalar\_matrix\_addition + scalar\_product \mid scalar\_product \\
scalar\_product & \longrightarrow & scalar \times scalar\_prod \mid scalar\_prod \\
scalar\_prod & \longrightarrow & scalar\_prod \times scalar\_mat\_product \mid scalar\_mat\_product \\
scalar\_mat\_product & \longrightarrow & scalar\_mat\_prod \mid scalar\_mat\_fun \mid scalar\_mat\_power
\end{array}
$$

$$
\begin{array}{rcl}
scalar\_mat\_power & \longrightarrow & scalar\,\hat{}\,scalar\_matrix\_expression \mid \\
 & & scalar\_matrix\_expression\,\hat{}\,scalar \mid \\
 & & scalar\_matrix\_expression\,\hat{}\,scalar\_matrix\_expression \\
scalar\_mat\_fun & \longrightarrow & f_{s1}(matrix\_expression) \mid \\
 & & f_{s2}(matrix\_expression, matrix\_expression) \mid \ldots \mid \\
 & & f_{sn}(matrix\_expression, matrix\_expression, \ldots) \\
f_{si} & \longrightarrow & \text{<matrix function with } i \text{ arguments with scalar value, } i \geq 1\text{>} \\
 & & \\
scalar\_mat\_prod & \equiv & \text{<matrix product with scalar value>} \\
scalar\_mat\_prod & \longrightarrow & scalar\_mat\_prod \cdot mat\_fun
\end{array}
$$

A.2 Parsing of Matrix Expressions

We show below how the matrix expressions defined above are translated into a Macrofort-like syntax as a list. For each item, we first show the grammar of the matrix expression, then the matrix expression itself and at last the generated list.

- $matrix\_expression \longrightarrow matrix\_expression + product \mid product$
  $product_1 + \cdots + product_n$
  $[\texttt{addm}, product_1, \ldots, product_n]$
- $product \longrightarrow scalar\_prod \times mat\_prod \mid mat\_prod$
  $scalar\_prod \times mat\_prod$
  $[\texttt{scalarmul}, scalar\_prod, mat\_prod]$
- $mat\_prod \longrightarrow mat\_prod \cdot mat\_pfun \mid mat\_pfun$
  $mat\_pfun_1 \times \cdots \times mat\_pfun_n$
  $[\texttt{prodm}, mat\_pfun_1, \ldots, mat\_pfun_n]$
- $mat\_power \longrightarrow mat\_fun\ \hat{}\ scalar$
  $mat\_fun\ \hat{}\ scalar$
  $[\texttt{powerm}, mat\_fun, scalar]$
- $mat\_power \longrightarrow mat\_fun\ \hat{}\ scalar\_matrix\_expression$
  $mat\_fun\ \hat{}\ scalar\_matrix\_expression$
  $[\texttt{powerm}, mat\_fun, scalar\_matrix\_expression]$
- $mat\_fun \longrightarrow f_1(matrix\_expression)$
  $f_1(matrix\_expression)$
  $[f_1, matrix\_expression]$
  $\texttt{with}\ f_1 \equiv \texttt{transpose}$

- $mat\_fun \longrightarrow f_2(matrix\_expression,\ matrix\_expression)$
  $f_2(matrix\_expression_1, matrix\_expression_2)$
  $[f_2, matrix\_expression_1, matrix\_expression_2]$
  `with` $f_2 \equiv$ `linsolve`, $f_2 \equiv$ `stack`

- $\left\{ \begin{array}{rcl} scalar\_matrix\_expression & \longrightarrow & scalar + scalar\_matrix\_addition \\ scalar\_matrix\_addition & \longrightarrow & scalar\_matrix\_addition + scalar\_product \mid \\ & & scalar\_product \end{array} \right.$
  $scalar + scalar\_product_1 + \cdots + scalar\_product_n$
  $[$`add`$, scalar, scalar\_product_1, \ldots,\ scalar\_product_n]$

- $scalar\_matrix\_addition \longrightarrow scalar\_matrix\_addition$ `+` $scalar\_product \mid scalar\_product$
  $scalar\_product_1 + \cdots + scalar\_product_n$
  $[$`add`$, scalar\_product_1, \ldots,\ scalar\_product_n]$

- $\left\{ \begin{array}{rcl} scalar\_product & \longrightarrow & scalar \times scalar\_prod \\ scalar\_prod & \longrightarrow & scalar\_prod \times scalar\_mat\_product \mid scalar\_mat\_product \end{array} \right.$
  $scalar \times scalar\_mat\_product_1 \times \cdots \times scalar\_mat\_product_n$
  $[$`prod`$, scalar, scalar\_mat\_product_1, \ldots,\ scalar\_mat\_product_n]$

- $scalar\_prod \longrightarrow scalar\_prod \times scalar\_mat\_product \mid scalar\_mat\_product$
  $scalar\_mat\_product_1 \times \cdots \times scalar\_mat\_product_n$
  $[$`prod`$, scalar\_mat\_product_1, \ldots,\ scalar\_mat\_product_n]$

- $scalar\_mat\_power \longrightarrow scalar\ \hat{}\ scalar\_matrix\_expression$
  $scalar\ \hat{}\ scalar\_matrix\_expression$
  $[$`power`$, scalar, scalar\_matrix\_expression]$

- $scalar\_mat\_power \longrightarrow scalar\_matrix\_expression\ \hat{}\ scalar$
  $scalar\_matrix\_expression\ \hat{}\ scalar$
  $[$`prod`$, scalar\_matrix\_expression, scalar]$

- $scalar\_mat\_power \longrightarrow scalar\_matrix\_expression\ \hat{}\ scalar\_matrix\_expression$
  $scalar\_matrix\_expression_1\ \hat{}\ scalar\_matrix\_expression_2$
  $[$`power`$, scalar\_matrix\_expression_1,\ scalar\_matrix\_expression_2]$

- $scalar\_mat\_fun \longrightarrow f_{s2}(matrix\_expression,\ matrix\_expression)$
  $f_{s2}(matrix\_expression_1, matrix\_expression_2)$
  $[f_{s2}, matrix\_expression_1, matrix\_expression_2]$
  with $f_{s2} \equiv$ dotprod

A.3 Generation of BLAS Code

We are not going to describe the process of translation of the preceding lists into BLAS calls via Macrofort. This can be seen in the source code of Transfor.

This process is not obvious because the BLAS routines do not exactly correspond to the elementary operations of a naive translation of matrix expressions and if we want to keep efficiency, we need to use the BLAS routines in an optimal way.

B Internal Working of Transfor

We show briefly in this section the way Tranfor works and which are the main Maple procedures it calls internally. This section is not intended to completely describe Transfor but to help the user in customizing it.

We will follow the main steps of Transfor when translating the Maple procedure:

```
f:=proc(a,b,c,n)
  transdcl(a,n,n); transdcl(b,n,n);
  if n > 5 then
    c:=evalm(a &* b);
  else
    c:=evalm(2 * a &* b);
  fi;
end;
```

into Fortran by using the command `transfor(f,'foo');` to give the Fortran subroutine:

```
c
c     SUBROUTINE foo
c
      subroutine foo(a,b,zloc(iz1),n,zloc)
        implicit doubleprecision (a-h,o-z)
        dimension zloc(*)
        dimension a(n,n)
        dimension b(n,n)
c       memory space of zloc: n**2
        if (5.lt.n) then
c         computing matrix value of c: zloc(iz1)
            iz1 = 1
            call dgemm('n','n',n,n,n,0.1D1,a,n,b,n,0.0D0,zloc(iz1),n)
```

```
        else
c         computing matrix value of c
            call dgemm('n','n',n,n,n,2.D0,a,n,b,n,0.0D0,zloc(iz1),n)
        endif
      end
```

The different steps are described below.

1. First `transfor/transmac1` which translates the procedure into Macrofort is called. It calls `transfor/proc` which uses `procbody` to translate Maple procedure `f` into its inert form. The output of `transfor/proc` is the following:

```
[subroutinem, foo, [a, b, c, n],
 [[declarem, dimension, [a[n,n]]],
  [declarem, dimension, [b[n,n]]],
  [if_then_else_m, 5 < n,            ← conditional already translated into Macrofort
   [[linalg, c, `&function`(evalm,`&expseq`(            ← first matrix expression
     `&function`(`&*`,`&expseq`(`&args`[1],`&args`[2]))
                                        ))]],
   [[linalg, c, `&function`(evalm,`&expseq`(          ← second matrix expression
     `&function`(`&*`,`&expseq`(2*`&args`[1],`&args`[2]))
                                        ))]]
]]]
```

Note that the conditional `if then else` has already been translated into Macrofort syntax. Now we need only to deal with matrix expressions.

2. Then `transfor/lin` is called which scans the preceding list to translate matrix parts `[linalg, ...]` into corresponding Fortran statements as a Macrofort list. `transfor/linalg` is called for each matrix expression. For example, the two arguments of `transfor/linalg` for the second matrix expression `c:=evalm(2 * a &* b)` are:

```
c
`&function`(evalm,`&expseq`(`&function`(`&*`,
           `&expseq`(2*`&args`[1],`&args`[2]))))
```

and the corresponding translation is done by function `transfor/linalgp` which gives:

```
[[commentf, `computing matrix value of c`],
 [prodm, [scalarmul, 2, a], b]]
```

In the meantime function `transfor/dim` has been called which makes a complete checking of the sizes of all the matrix variables.

3. Now we need to translate the expression `[prodm, [scalarmul, 2, a], b]` into BLAS code. This done by calling the function `transfor/genblas`. Its two arguments are:

```
c
[prodm, [scalarmul, 2, a], b]
```

and the output of `transfor/genblas` is:

```
[callf, dgemm, [''n'', ''n'', n, n, n, 2., a, n, b, n, '0.0D0', c, n]]
```

the corresponding output of `transfor/linalg` (see step 2) is then:

```
[commentf, 'computing matrix value of c']
[[callf, dgemm, [''n'', ''n'', n, n, n, 2., a, n, b, n, '0.0D0', c, n]]]
```

Note that `transfor/genblas` could be replaced to give other translation of matrix expressions.

4. At last, the output of `transfor/lin` is:

```
[subroutinem, foo, [a, b, c, n],
 [[declarem, dimension, [a[n,n]]],
  [declarem, dimension, [b[n,n]]],
  [if_then_else_m, 5 < n,
   [[commentf, 'computing matrix value of c: zloc(iz1)'],
    [[equalf, iz1, 1]],
    [[callf, dgemm,
     [''n'', ''n'', n, n, n, 1.0, a, n, b, n, '0.0D0', c, n]]]],
   [[commentf, 'computing matrix value of c'],
    [[callf, dgemm,
     [''n'', ''n'', n, n, n, 2., a, n, b, n, '0.0D0', c, n]]]]]]]
```

But, during all the process, Transfor has accumulated the information about local and work variables. Here there is only one local variable, `c` which becomes `zloc(iz1)`. No work variable is needed here. So, by using the function `transfor/loc` a last work has to be done to replace `c` by its Fortran value, and the output of `transfor/transmac1` gives eventually the Macrofort translation of the procedure:

```
[subroutinem, foo, [a, b, zloc(iz1), n, zloc],
 [[declarem, 'implicit doubleprecision', ['(a-h,o-z)']],
  [commentf, 'memory space of zloc: n**2'],
  [declarem, dimension, ['zloc(*)']],
  [declarem, dimension, [a[n,n]]],
  [declarem, dimension, [b[n,n]]],
  [if_then_else_m, 5 < n,
   [[commentf, 'computing matrix value of c: zloc(iz1)'],
    [[equalf, iz1, 1]],
    [[callf, dgemm,
```

```
  [''n'', ''n'', n, n, n, 1.0, a, n, b, n, '0.0D0', zloc(iz1), n]]]],
[[commentf, 'computing matrix value of c'],
 [[callf,dgemm,
   ['n'', ''n'', n, n, n, 2., a, n, b, n, '0.0D0', zloc(iz1), n]]]]]]]
```

which is then translated into Fortran by using Macrofort.

References

[1] Delebecque, F. and Nikoukhah, R., *A mixed symbolic-numeric software environment and its application to control system engineering*, Recent advances in computer aided control, eds. Hergert, H. and Jamshidi, M., pp. 221–245, 1992.

[2] Basic Linear Algebra Subprograms for Fortran Usage, ACM Transactions on Mathematical Software 5, 308-325, 1979.

[3] Aho Alfred V., Ravi Sethi, Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[4] Anderson E. et al., *LAPACK User's Guide*, SIAM, 1992.

[5] Gomez Claude, *Macrofort: a Fortran Code Generator in Maple*, INRIA Technical Report 119, May 1990.

[6] Moler Cleve B., *MATLAB User's Guide*, Technical report CS81–1, Department of Computer Science, University of New Mexico, 1982.

[7] NAG LTD, *The NAG Fortran Library Manual – Mark 13*, 1988.

[8] Reid John, *Fortran 90, the New Fortran Standard*, .EXE Magazine, Vol 6, Issue 3, August 1991.

[9] Gomez Claude, editor, *Engineering and Scientific Computing with Scilab*, Birkhäuser, 1998, To appear.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399