Mascotte : A few Maple Routines for Real-Time Code Generation

Thierry Viéville

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Mascotte : A few Maple Routines for Real-Time Code Generation

Thierry Viéville

N 2826

Mars 1996

THÈME 3

ISSN 0249-6399

INRIA
SOPHIA ANTIPOLIS

Mascotte : A few Maple Routines for Real-Time Code Generation

Thierry Viéville

Thème 3 — Interaction homme-machine,
images, données, connaissances
Projet RobotVis

Rapport de recherche n2826 — Mars 1996 — 38 pages

Abstract:

The Mascotte Maple Package has been built to simplify the implementation of reactive software modules in vision and robotics and to facilitate the exchanges of data models and algorithms between builders of such reactive systems, using the Maple Symbolic calculator.

The main contributions of this package are :

- Designing routines for text manipulation
- A small add-on for linear algebra manipulations
- Some functions for code generation
  - A formater for equations and logic expressions: `format()`
  - A tool to manipulate program ressources: `msobj()`
  - A C-code generator for straight-line programs: `convert(,C)`
  - A mechanism to design maple routines from formal specifications: `procdefine()`

Key-words: Elementary symbolic computations, Vision and Robotics, Code generation, Linear algebra

(Résumé : tsvp)

Unité de recherche INRIA Sophia-Antipolis
2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex (France)
Téléphone : (33) 93 65 77 77 – Télécopie : (33) 93 65 77 65

Mascotte : Quelques Outils Maple pour la Génération de Code Temps-Réel

Résumé :

Le package Mascotte a été construit pour simplifier l'implémentation de modules de vision ou de robotiques de type réactifs et pour faciliter l'échange de models de données et d'algorithmes entre les concepteurs de tels systèmes, ceci à partir du système Maple.

Les principales contributions de cet ensemble de routines sont la manipulation de textes en Maple, quelques routines d'algèbre linéaire et des mécanismes de génération de code.

En bref, une manière de montrer qu'un calculateur symbolique ne sert pas qu'à faire des "maths", mais à manipuler d'autres entités informatiques.

Les principales contributions de cette petite extension sont :

- Des routines de manipulation de texte
- Un petit complément du package d'algèbre linéaire
- Des pour la generation de code
  - Un formateur d'équations et d'expression logiques: `format()`
  - Un outil pour manipuler les resssources d'un programme: `msobj()`
  - Un générateur de code C pour des calculs sans boucle: `convert(,C)`
  - Un méchanisme pour générer des routines maple à partir de spécifications formelles: `procdefine()`

Mots-clé : Calcul Symbolique élémentaire, Vision et Robotique, Génération de code, Algèbre linéaire

Contents

Index

1 Introduction

The Mascotte Maple Package has been built to simplify the implementation of reactive software modules in vision and robotics and to facilitate the exchanges of data models and algorithms between builders of such reactive systems, using the Maple Symbolic Calculator.

It is divided into three parts :

- A small enhancement of some `Maple` routines.
- A tiny complement of the `linalg` package.
- A few set of routines to manipulate equations and pieces of code.

As far as the first part is concerned, we have tried to formalize the different patchs and contribution made to some usual `Maple` routines in order to fit with our applications. We have only defined a new procedure for what can not be actually easily done using usual `Maple` routines, or what is rather tricky to program using `Maple` procedures and gain of being encapsulated.

As far as the second part is concerned, we have collected rather easy to program routines which are commonly used in linear algebra but where missing in the actual package. Some are very small, some are a bit more heavy. We have started to define some formal manipulation of vector sand matrix, without using their components, which is often required.

As far as the third part is concerned, we are still in a development phase, the main available and stable routine being the `format()` procedure. Other routines will make this package grown in a near future.

From a user point of view, all these routines form a `Maple` extension called Mascotte and can be easily used and accessed via the www page :

```
 <A HREF=
 "http://www.inria.fr/robotvis/personnel/vthierry/mascotte/main.html"
> Mascotte Package </A>
```

as shown in the User-Guide Appendix of the document, the installation is easily done by adding an element to the `libname` environment variable and calling `with(mso)`, plus `with(linalg1)` if required.

The documentation can be consulted on line, via the `?mso` command.

2 A short description of the Mascotte contribution

Improving `type(,function)`: `type(,func)`

(see section A.5)

The routine `type(,function)` does not accept Maple built-in operators, such as '+', '*' which behave as function such as '&+' or '&*'. Furthermore, we might want to restrict the selection to a given function, a subset of function, or a function with a certain number of argument. All this is implemented in `type(,func)`.

For instance `type(u,function) and op(0,u) = F and nops(u) = 3` is simply written `type(u,func(F,3`

Improving `readline()`: `textread()`

(see section A.3).

The main idea here is to provide an interface, not only to read a line, but a whole text-file and put it in a TEXT data-structure. This is a very simple routine but, extensively used in the sequel.

Improving `writeto()`: `textwrite()`

(see section A.4).

The dual routine allows to store a TEXT data structure in a text-file. Contrary, to the previous routine it is not that trivial, and in particular allows to avoid some interface problems which occur when `writeto()` is manipulated, since only one file can be open at a time.

Completing `save(lib|proc)()`: `saveinlib()`

(see section A.2).

Maple has hidden (!) procedures to help saving maple objects in a given library. We propose here a very simple and somehow powerfull mechanism to store a Maple object in the right place, including the help file.

Completing `makehelp()`: `convert(,TEXT)`

(see section A.6).

The `makehelp()` allows to store a TEXT data structure corresponding to the help information of a Maple object. As a complement, `convert(,TEXT)` allows to generate such a data structure, by concatenation of portions of TEXT or strings, by executing some maple commands (see next paragraph), the result being in the TEXT.

It must be noted, that this routine uses temporary files in your local directory, and make use of `writeto()`, but in full compatibility with `writetext()`.

While parse executes a Maple command and output the result, the `&parse()` operator of `convert(,TEXT)` ouptut the text of the command and its result in the TEXT data structure.

This routine is used to generate all help texts of this extension.

Improving `define()`: `procdefine()`

(see section A.11).

The `define()` procedure provides a mechanism to automatically generates a procedure from a set of properties. The `procdefine()` procedure is a release of this standard Maple procedure but with the following improvements :

- It has no side effects, i.e. does not assign any object, but simply return the procedure with the desired behavior.
- It has a much higher number of algebraic properties which can be defined (for instance all what exists in symbolic computations can be easily defined here), including the remember option, or a call back to a standard maple procedure to complete the definition of the procedure.

- We have the filling that `forall()` properties had a rather strange behavior in `define()` while `procdefine()` does not has this funny side effect.

This routine is extensively used in other parts of this extension.

Improving `cost()`: `cost()`

(see section A.13).

A new version of the `cost()` fucntion is proposed which simply accept non-algebraic expressions.

An alternative to `solve`: simplifying equations with `format()`

(see section A.12).

Sometimes, the goal of a user is not to solve equations, but to put them in a better form. This what the rather huge procedure `format` attempts to do.

Considering equations and inequalities defined from logical operators, or composite equations, format apply all what is easy to do to simplify the equations :

- Logic operators are simplified,
- When the sign of an expression can be simplified, format uses it,
- Redundant equations are eliminated,
- Some syntaxic compact forms are understood.

This routine has three applications:

- It put equations in a normal form and separate different components of the set of solution of these equations.
- It is a syntaxic mechanism which allows to define complex problems via equations, with first-order logic operators and to reduce them.
- It contains several simplification rules to eliminate all trivial parts of the equations.

This allows, for instance, to evaluate the true complexity of a given problem, since all what is simple has been eliminated.

Improving `C()`: `convert(,C)`

(see section A.10).

The philosophy of the `convert(,C)` function is completly different from what is found in `C()`. In our case:

- we do not generate C-lines code but a complete compilable C-routine, with its interface and a documentation of the source code (maple expression which is computed and computation cost);
- several non-algebraic expression are compiled, including logic expressions and conditional statements;

- the code is output as a TEXT data structure.

May be the drawback of our routine is that the generated code has a lot of parenthesies, whereas in the C-lines generated by `C()` care is taken of having a minimal number of parenthesies. So our code is less pretty. But how cares ?

This procedure allows to convert expressions to a "straight-line program" (slp) i.e. a piece of code which can be computed their input data being given with a constant execution time. The execution time is function of the data components sizes (and is a polynomial function of them). In such pieces of code, there is no random access to data structures and the index bounds can be evaluated, at compilation time. Such pieces of code never create a memory fault or an infinite loop and their semantic is easily defined.

Improving `scanf()`: `textparse()`

(see section A.8).

While `scanf()` allows to parse lexical elements corresponding to numeric data structure or strings, `textparse()` is a Maple transcription of the lex/yacc mechanisms. It has a standard fixed lexical analyser and a parser which is entirely dynamic and can parse any context-free grammar. The parsing mechanism is -of course- not optimal, since it contains a form of backtracking, but is the best solution if we want to be able to change the grammar at will, which is the case here.

Completing the `linalg` package: `linalg1`

(see section A.14).

Considering the rather extensive use of Maple in the field of computer vision and robotics we have been able to identify new linalg routines, often used. Some are very small, close to a macro, but save time and avoid repetitive mistakes. Some are more complex at the implementation level, but very standard in the field of algebra.

Manipulating rotations: `tilde()`, `tilded()`, `rot3d()`

These routines allows to manipulate infinitesimal or non-infinitesimal 3D rotations. The parameterization of 3D rotations is a recurrent problem, and every-body has its own "best" solution. Here several solutions are proposed, including our "best solution", based on the non-real part of a unary quaternion (although the formalism of quaternions is totaly avoided here).

Specifing that a matrix is singular: `subrank()`

This simple routine allows to generate a minimal number of equations to insure that a given matrix is not of full rank. A valse with subdeterminants, nice to have in one box.

Putting a symetric matrix in its Gauss form: `gaussform()`

This very usefull routine allows to find a "diagonal" form for symmetric matrices: it is known that although every symmetric matrix has a diagonal form, the transformation being an orthogonal matrix, the algorithm to find this decomposition corresponds to the solution of an algebraic equation which is in general not possible, except for dimensions less than 5.

It is less known, that if we accept to have not an orthogonal matrix, but a less restrictive matrix of transformation, a triangular matrix in our case, we can find such a pseudo-diagonal form, from rational expressions only.

This problem is equivalent to put a second degree polynomial in a form where it appears as a sum of squares and both forms are available here.

An optimal form is found here, in the sense that among the different possible decompositions, the decomposition with the minimal cost is given.

Computing the Bezoutian: `bezoutian()`

The Bezoutian of two polynomials is computed here.

Computing the signature of an object: `signature()`

The signature of an expression (list of value, array, etc...) is somehow always easy to compute when the expression is numeric. When the expression involves variables, only a conditional expression can be issued. This is the case here, where the routine tries to output a simple conditional expression which evaluation corresponds to the given signature.

Formal manipulation of matrices and vectors: `evalsm()`

There is often a need to manipulate vectors and matrices "without the components", expecially in the case of Hilbert spaces, with infinite dimensions.

In fact several simplifications can be made, from symbolic algebraic properties of matrices as implemented in this routine.

However, we must remind that contrary to the polynomials for instance, the matrix algebra is very unfair: non-commutative, non-integral, etc... so that simplifications is more a heuristic than the generation of any pseudo-canonical form.

This might be the reason why "good" mathematicians have never program such a routine, while we have done it because we need it!

Furthermore, we have, in this routine, a formal derivator which can very easily simplify the computations of Jacobians.

A Programmer Guide

A.1 mso

HELP FOR:

```
Introduction to the Mascotte extension of the maple system
```

CALLING SEQUENCE:

```
with(mso)
```

SYNOPSIS:

- The Mascotte extension is a small set of maple routines which extend the capabilities of the actual maple system. They are made of functions to manipulate text structures, generate procedures, plus linalg functions.

- The objective of these functions is to generate reactive modules in robotics and vision systems, but may have also some other applications

- The presentation is a standard package, with some ‘‘readlib-defined’’ routines defined in it. It is thus simply loaded via with(mso):

- The linalg package must be loaded using with(linalg1). The trace function is renamed proctrace to avoid confilc with linalg[trace]

- Available routines in this extension are:

SEE ALSO:

```
proctrace, textread, textwrite, type[func], convert[TEXT],
         procdefine, format, cost, linalg1
```

A.2 saveinlib

FUNCTION:

```
saveinlib - store a maple-object with its help in a libary
```

CALLING SEQUENCE:

```
saveinlib(lib_name, obj_name)
```

PARAMETERS:

```
lib_name  - the name of the library in which the object is to be stored
obj_name - the name of the object to be stored
```

SYNOPSIS:

- saveinlib store a maple procedure and all its sub-objects and help file in a maple library.

- The obj_name can be of the form 'name', or 'name1/name2'. Then all objects of name 'obj_name' or 'obj_name/*' will be stored in the file of name 'lib_name/obj_name'. Furthermore the resulting help name will have the name 'help/text/name' or 'help/name1/text/name2' and be stored in the corresponding file, after conversion via convert[TEXT].

- Since you are a serious maple programmer, it always complains if no help file can be created.

SEE ALSO:

makehelp, convert[TEXT]

A.3 textread

FUNCTION:

textread - read a text data structure from a file

CALLING SEQUENCE:

```
textread(filename)
```

PARAMETERS:

```
filename        - the name of a file
```

SYNOPSIS:

```
- The function textread() returns the text read in the specified file.
  It is returned as a text data structure of the form :
                    TEXT(
                     'line-1-of-text',
                       .../...
                    )
```

SEE ALSO:

```
 TEXT, cat, textwrite
```

A.4 textwrite

FUNCTION:

```
 textwrite - write a text data structure in a file
```

CALLING SEQUENCE:

```
 textwrite(filename, text)
```

PARAMETERS:

```
  filename        - the name of a file
  text            - a text data structure of the form :
                    TEXT(
                     'line-1-of-text',
                       .../...
                    )
```

SYNOPSIS:

```
- The function textwrite() writes the text into the specified file
```

SEE ALSO:

```
 TEXT, cat, writeto
```

A.5 type[func]

FUNCTION:

```
type/func - generalized check for a function
```

CALLING SEQUENCE:

```
type(expr,func(name,nbargs))
```

PARAMETERS:

```
expr   - any expression
name   - the name of the generalized function (optional)
         or a set of names of the generalized function
nbargs - the number of arguments of the function (optional)
```

SYNOPSIS:

```
- Check if the expression is a generalized function
  of eventually of a given name
  and eventually of a given number of argument(s).
- The following types: list, set,
  `+`, `*`, `^`, `=`, `<>`, `<`, `<=`, `**`, `and`, `or`, `not`
  are considered as generalized function.
```

SEE ALSO:

```
type[function]
```

A.6 convert[TEXT]

FUNCTION:

```
convert/TEXT - convert string or expr to a text data structure
```

CALLING SEQUENCE:

```
convert(string,TEXT)
```

PARAMETERS:

```
  string - string or list of string or TEXT data structures
           with some possible unevaluated functions
```

SYNOPSIS:

```
- This function attemps to construct a TEXT data structure from its
  argument. Each string is taken as a line of text. List of TEXT/string
  are concatenated.

- Strings may contain the following unevaluated functions:
 - `&cat`() which is the unevaluated form of cat()
 - `&parse`(string1,string2,...) which is a sequence of statements
   evaluated and put in the text as if run though a maple session.
   The `&parse` function is typically used in help text structure.
 - `&print`(expr1,expr2,...) to print maple expressions within the text.
```

EXAMPLE:

```
Please run:
`convert/TEXT/example`();
```

SEE ALSO:

```
TEXT, cat, textwrite, textread
```

A.7 convert[maple2inert]

FUNCTION:

```
convert/maple2msotext - convert a maple object to an msotext
         convert/msotext2maple - convert an msotext to an maple object
         convert/maple2inert   - convert a maple object to an inert form
         convert/inert2maple   - convert back to a maple object
```

CALLING SEQUENCE:

```
convert(expr, maple2msotext)
convert(text, msotext2maple)
convert(expr, maple2inert)
convert(expr, inert2maple)
```

PARAMETERS:

```
  expr   - a maple exprssion.
  text   - a text data structure.
```

SYNOPSIS:

```
- These complementary pairs of functions allows to convert a maple object
  into :
   * an inert form, involving only string and function
   * an easily parsed text, as used by the mso C/C++ library.

- In fact, these routines are not really useful for a maple user,
  but are part of the interface with the C/C++ library.
```

A.8 textparse

FUNCTION:

```
 textparse - defines a text parser via a formal grammar
```

CALLING SEQUENCE:

```
 textparse(grammarule1, grammarrule2, ...)
```

PARAMETERS:

```
  grammarrule.i - an equation defining a grammar rule.
```

SYNOPSIS:

```
- The function textparse defines a maple procedure of the form:
      proc(t:TEXT) ... end:
  which parses a TEXT data structure, according to a formal grammar.
- A lexical element of the grammar is defined by the regular expression:
[

]*                                # which is skipped
 (
  [@#$&]? [A-Za-z_]+ |
  ( [-]? ( [0-9]* [.] )? [0-9]+ ( [Ee] [+-] [0-9] )? ) |
  [@#$&]? [+-*%/~^=<>?!:|.]+ |
  ( [‘] ( [‘][‘] | [^‘] )* [‘] ) |   # not yet implemented
  ["’’#$0{[()]}]
```

```
 )
- The grammar is defined by a sequence of grammar rules of the form:
   nonterminal(expr) = pattern
  where expr is an inert expression which represent the syntax structure
  and might contain some variables names as defined in the sequel
  where a pattern is either a:
   - a sequence of (sub)pattern
   - any lexical element &.(v) where v is an optional variable name
     which will match the lexical element.
   - a string, corresponding to a given lexical unit
   - a nonterminal represented by an inert function with as optional
     argument, a variable name which will match the syntax structure
   - the &*(pattern) to denote 0, 1 or more occurence of the pattern
   - the &+(pattern) to denote 1 or more occurence of the pattern
   - the &?(pattern) to denote 0 or 1 occurence of the pattern
   - the &!([pattern1], [pattern2], ...) operator to denote alternative,
     [] can be used to separate between different sequence of pattern.
- The root of the grammar is the non-terminal of the first grammar rule.
```

EXAMPLE:

```
>   g := textparse(add(add(x,y))=(&.(x),‘+‘,&.(y)));
proc()
options ‘I’ve been created with textparse() !!!‘,remember;
    if type(args[1],TEXT) then
        ‘textparse/yacc‘(
          add,[add(add(x,y)) = (‘&.‘(x),‘+‘,‘&.‘(y))],‘textparse/lex‘(args[1])
          )
    else ERROR(‘Bad argument type‘)
    fi
end

>   g(TEXT(‘a + b‘));
                                    add(a, b)
```

A.9 msobj

FUNCTION:

```
 msobj - elementary mechanism of object definitions
```

CALLING SEQUENCE

```
S:
 msobj(obj, att [,satt...]) - to retrieve the value of an attribute
 msobj(obj, att=val, ...)   - to define the value of an attribute
```

PARAMETERS:

```
 obj  - the name of an object
 att  - the name of an attribute
 satt - the name of subattributes, either for dymanic attributes or
        to reference an attribute of a subobject.
 val  - any value
```

SYNOPSIS:

```
 - Provide an elementary mechanism of object definition.
   An object is defined by a list of attributes of a given value.
   The value of an object can be a name, an expression or another
   subobject.
   Objects are simply defined via Maple tables.
 - Object reference : an object is referenced by its Name attribute,
   automatically inserted in the table. Therefore:
    `type/msobj` := u -> type(u,table) and u[Name]=u:
 - Object heritage : when an attribute is undefined it default value is
   its own name, unless the attribute Type is defined. In this case, the
   value attribute is given by the first object in u[Type] wich has this
   attribute defined.
 - Object encapsulation : the attribute can itself be an object and
   referenced as such.
 - Object dynamic attribute or methods : if an object value is a function
   then the value of the object is given by the evaluation of the function.
   Arguments of the functions are the sub-attributes
```

EXAMPLE:

```
>   msobj(myobj,a=1,b=hello,c=log(x-1)):
>   op(myobj);
                              table([
                                  a = 1
                                  b = hello
                                  c = ln(x - 1)
                                  Name = myobj
                              ])

>   msobj(yourobj,Type=[myobj,anotherobj],a=2,d=[1,2,3]):
>   msobj(yourobj,a);
```

```
>   msobj(yourobj,b);
                                    hello

>   msobj(yourobj,d);
                                  [1, 2, 3]

>   msobj(yourobj,e);
                                      e
```

SEE ALSO:

assume, table

A.10 convert[C]

FUNCTION:

convert/C - convert an expression to a piece of C-code

CALLING SEQUENCE:

convert(expr, C, name)

PARAMETERS:

expr - an expression, an array of expression or a list of equations
name - the name of the C-routine to be generated

SYNOPSIS:

- Convert the input to C code for evaluating it. It outputs a TEXT,
  which contains a full ANSI-C procedure.
- The input must be either:
  - a single algebraic, numeric or logic expression,
  - a named array of such expressions,
  - a list of equations of the form name = expression
    understood as a sequence of assignment statements.
- The ouput is always optimized, using:
  - the Maple optimize function
  - 23 digits constants for double precision
  - monodimensional arrays indexed from 0:
    (A[i,j,...] is replaced by _A[(i-1)+(j-1)*rowdim(i)+...])

```
 - The following "documentation" is provided in the code header:
  - The ANSI-C interface
  - The maple expression to be evaluated
  - The cost, i.e. the number of operations
 - The following naming conventions are followed:
  - t0,t1,... variables are reserved for internal temporary variables
  - variables names starting with _, e.g. _a, _M, are reserved for
    indesed variables

- Comparing to the original C() routine or one of its variant, this
  function manipulates less restrictive piece of codes, has a simpler
  interface and a better optimization. The conditional expression &if()
  is understood and used to translate logic expressions. However, the
  generated code is less "pretty" than with C(), e.g. parenthesies are
  systematically inserted as in convert(,string).
```

EXAMPLES:

```
>   s := ln(x)+2*ln(x)^2-ln(x)^3:
>   textwrite(‘tmp1.c‘,convert(s,C,tmp1)):
>   s := array([[x^2,ln(sin(x^2))],[sin(x^2),x^(-2)]]):
>   textwrite(‘tmp2.c‘,convert(s,C,tmp2)):
>   s := [v=&if(u>=0, u^2, csc(u^2)), w=v/cos(u^2), a[1]=arctan(w/f(u,w))]:
>   textwrite(‘tmp3.c‘,convert(s,C,tmp3)):
```

SEE ALSO:

```
  optimize, cost, C, fortran, textwrite, TEXT
```

A.11 procdefine

FUNCTION:

```
 procdefine - define an operator via properties and rules
```

CALLING SEQUENCE:

```
 procdefine(property1, property2, ...)
```

PARAMETERS:

```
  property.i - either name, equation or forall construct
               defining a property or a rule.
```

SYNOPSIS:

```
- procdefine defines a maple procedure of the form
      proc(args) ... end:
  via properties and rules.
- Contrary to define(), it has no side-effects, and does not affect
  functions such as simplify, eval, expand, diff, series, and testeq.
- Properties for the operator can be defined in a "declarative" way
  using this enhanced forall construct:
  - forall({vars}, (f(args),cond) = res) define rewritting rules, where
     {vars}  - is a list of names, eventually empty
     f(args) - is the pattern on which this rule will be used. f must be
               the function defined by procdefine. It must oftern be in
               an inert form.
     cond    - is an (optional) inert boolean expression
     res     - is the desired result
    - The operator produces the expression res each time:
       (1) the [args] pattern matchs the operator arguments and
       (2) the cond pre-condition is evaluated to true after substitution.
    - The names defined in vars are used as matching arguments which can
      take any possible value. The substitutions are then done for all
      possible such values.
    - To constrain an expression x to be of a certain type t and/or
      have a given property p as defined in the assume facility,
      the cond expression must contain a relation of the form:
       `&is`(x,t)
      where the is function is given in a "freezed" form
      Any other conditions can be introduced, using procmake syntax.
- Properties can also be defined in a "procedural" way via a standard
  maple procedure:
  - procedure=p; In that case the operator is also defined by a procedure
    of name p which will be called to evaluate the arguments.
- Otherwise the following properties are understood:
  - remember; The operator has the remember option
  - unary;  The operator is unary and must have 1 argument.
  - binary;  The operator is binary.
  - type=t; The argument(s) must be of the given type
  - type=[t1,t2,...,tn]; Each arguments must be of the given type
    according to this list. The operator must have n arguments
  - associative; The operator is associative (n-ary), so f(x,f(y,z)) =
    f(f(x,y),z) = f(x,y,z).
  - commutative; The operator f is commutative, so f(x,y) = f(y,x).
    The arguments will be ordered to guarantee uniqueness.
  - antisymmetric; The operator f is antisymmetric, so f(x,y) = -f(y,x)
    and f(x,x)=0.
  - idempotent; The operator is idem-potent, so f(f(x)) = x,
    it is consider as a unary operator
```

```
- nopower; The operator reduces powers so, f(x,x) = f(x)
  such as in and/or boolean operators
- linear; Define the operator as being linear, i.e.:
   L(K1*a + K2*b, ...) = K1*L(a, ...) + K2*L(b, ...) and
   L(K3, ...) = K3*L(1, ...)
  for any constants K1, K2 and K3.
- zero=x; Define the zero to be x, so if any argument is x then
  the result is x.
- identity=x; The expression x is to be considered the left and right
  identity of the operator.
- inverse=g; Define the inverse left-side and right-side unary
  operator to be the operator g.
- power=g; Define the power binary operator to be the operator g,
  so f(x,x) = f(g(x,2)), etc...
- powerof=g; Define the this binary operator is the power of an
  operator g
- derivative=[&g,..]; The operator is a derivative operator for
  the multiplicative operator(s) &g,..., so if K1, K2 are constant,
  i.e. do not contain one of the derivative variable:
   L(K1*a + K2*b, ...) = K1*L(a, ...) + K2*L(b, ...) and L(K1, ...) = 0
  while for the multiplicative operator(s)
   L( a &g b, ... ) = a &g L(b, ...) + L(a, ..) &g b
- scalarprod; The operator is mixed with scalar products, so:
   f(K * a, b) = K * f(a,b) for any constant or scalar K
- expand=[g,...]; The operator left and right expanded over the
  operator(s) g,..., so f(...,g(y,z),...) = g(f(...,z,...),f(...,y,...))
  note that arguments of g are inverted.
- expandon=[g,...]; The operator left and right expanded over the
  operator(s) g,..., so f(...,g(y,z),...) = g(f(...,z,...),f(...,y,...))
  note that arguments of g are inverted AND expand() is map on
  arguments.
- verbosecall; The operator executes a print('procname(args)') on
  each call, for modular debugging purpose.
```

SEE ALSO:

```
define, procmake, mso
```

A.12 format

FUNCTION:

```
format - format a set/logic expression of [in]equalities
```

CALLING SEQUENCE:

```
format(eqns)
format(eqns,real)
```

PARAMETERS:

```
eqns       - a system of real equations, either a set of equation
             or a boolean expression of equations.
real       - an optional directive to assume all variables are real.
```

SYNOPSIS:

```
- This function format a set of [in]equalities or a logic combination
  of these and attempt to reduce them to obtain a normal form.

- The final form is a &or(&and( ... )) expression, i.e. the union
  of conjonction of equations, unless quantificators occur.

- The real directive assumes that all variables are real. [In]equalities
  with complex values are false

- The following tranformations are performed:
  - [In]qualities are expanded over vectors and matricies
  - [In]equalities are put in a normal form, i.e.:
     e = 0, e <> 0, e < 0, 0 > e, e <= 0, e >= 0
    and the sign of e is reduced (via csgn) when possible.
  - For complex expressions the sign is understood as for csgn.
  - For expressions with a denominator the constraint that the
    the denominator is not zero is introduced.

- During the computation, we must set :
   _Envsignum0 := 0;
  and restore its old value at the end, in order to get coherent sign values

- Logic expressions are simplified. They are builded from the following
   primary operators:
  &and(x,y)     means : x and y
  &or(x,y)      means : x or y
  &exists(v,x)  means : there exists variable(s) v, for which x is true
  &forall(v,x)  means : for all variable(s) v, x is true
  and the following secondary operators, which are reduced:
   &not &implies &iff &nor &nand &xor
  - In these operators x and y are [in]equalities
    or the boolean constant false or true, or subexpressions of these,
    while v is the name of a variable or a set of variable(s).
  - In this context &and() and &or() are commutative operators
```

```
    evaluated in any order, contrary to the maple and/or operators.
  - A set is taken as a &and operator and a sequence as a &or operator
    in order to be compatible with solve() outputs

- Conditional expression are simplified, using the operator:
   &if(cond, expr-if-true, expr-if-false)
 - The multi-arguments inter &if operator:
    &if(cond1, expr1[, cond2, expr2[,cond3, expr3]]... expr0)
   is reduced to the 3-arguments form. If the expr-if-false argument
   is omitted it takes the false value.
 - The If() and `if`() names of the operator are also accepted.
 - The `and` and `or` operators are rewritten using &if:
    x and y           is rewritten as &if(x, y, false)
    x or  y           is rewritten as &if(x, true, y)

- List of equations are taken as equation sequences.

- Several piece-wise continous fonctions are rewritten using &if:
  the maple inert functions Heaviside, Dirac, Signum, Csgn, Abs and :

                                 /
 Win(t) : Window Function     : | = 0     if t < -1  or t > 1
                                 \ = 1     if t > -1 and t < 1

                                 / = t-1   if t > 1
 Thr(t) : Threshold Function  : | = 0     if t > -1 and t < 1
                                 \ = t+1   if t < -1

                                 / = 1     if t > 1
 Sat(t) : Saturation Function : | = t     if t > -1 and t < 1
                                 \ = -1    if t < -1

- Some nested constructions with relational operators are also allowed:
   a < (b < c)      means (a < b)  and (b < c)
   a <> (b <> c)    means (a <> b) and (b <> c) and (c <> a)
   a <> (b and c)   means (a <> b) and (a <> c)
   d < &if(c,a,b) means &if(c,d<a,d<b)
   a =  (b or c)    means (a = b)  and (a = c)
  etc ...

- The form &delete(expr,o) where o is a subset of {`<>`,`<`,`<=`}
  allows to delete inequalities and transform them to equations,
  adding a new variable, as stated by the following rules:
   - f <> 0 <=> &exists( t, t   * f - 1 = 0)
   - f <= 0 <=> &exists( t, t^2 + f     = 0)
   - f <  0 <=> &exists( t, t^2 * f - 1 = 0)
```

EXAMPLES:

```
>   format({a*x+b*y=0,x=0,b<>0});
                          &and(b <> 0, x = 0, y = 0)

>   format({a<>(x &and y), If(u,v,w)=0});
              &and(a - x <> 0, &if(u, v = 0, w = 0), a - y <> 0)

>   format(x=u*RootOf(_Z^2+4));
                     (I x + 2 u = 0) &or (I x - 2 u = 0)

>   format({(b<>c)<>(a=0),a<=0,a>=0,d+log(1+a)>=0});
                   &and(b <> 0, a = 0, b - c <> 0, 0 <= d)

>   format(array([a,b,c])=array([a,c,b]));
                                  b - c = 0

>   format(&exists({x,y},&exists(z,x^3+z=0)));
                                        3
                          {x, z} &exists (x  + z = 0)

>   format(&or(a^10<=0,a=0,-a=0,10*a=0,a+c^2*a=0),real);
                                    a = 0

>   format({b*a>=0,b>=0,c^2+1>0,b^2-d^2=0},real);
     &or(&and(b - d = 0, 0 <= a, 0 <= b), &and(0 <= a, 0 <= b, b + d = 0),

         &and(a <= 0, b = 0, d = 0))

>   format({d^8*c^2+d^10*b^2>=0,-c^4-a^8<=0},real);
                                    true

>   format(&or(b=0,c<=0,&and(e=0,b<>0,c>0)));
                          &or(b = 0, e = 0, c <= 0)

>   format(&if(a and (b or not c), d, e));
                   &if(a, &if(b, d, &if(&not c, d, e)), e)

>   format(Thr(u-2)/Sat(v));
                                                                 u - 3
 &if(0 < - u + 1, &if(v + 1 < 0, 3 - u, &if(1 - v < 0, u - 3, ——-)),
                                                                  v

                                                                    u - 1
     &if(3 - u < 0, &if(v + 1 < 0, - u + 1, &if(1 - v < 0, u - 1, ——-)), 0))
                                                                     v
```

```
>   format(&delete({x<0,y<>0}),real);
                                        2
                  ({t0} &exists ((- x t0  + 1 = 0) = 0)) &and

                      ({t1} &exists ((- y t1 + 1 = 0) = 0))
```

SEE ALSO:

csgn, signum, solve, logic, evalc, procdefine

A.13 cost

FUNCTION:

cost - (enhanced) operation evaluation count

CALLING SEQUENCE:

```
cost(e1, e2,...,en)
```

PARAMETERS:

e1, e2 ... any expression(s)

SYNOPSIS:

- Cost is used to compute an operation count for the numerical evaluation of the given arguments. The operation count is expressed as a polynomial in the names additions, multiplications, divisions, functions, subscripts and assignments with non-negative integer coefficients. Comparisons or operations in logical expressions are considered as ‘‘assignments’’.

- Assignment of positive real values to these global names yields a weighted cost.

- Note that the cost used for computing powers is as follows. For an

```
    integral power, repeated multiplication is assumed. For a general
    power it is assumed to be computed using exp and ln.

  - This function is similar to the maple cost function except that
    it is valid also on logical expressions
```

EXAMPLES:

```
>   cost(x+x^2+x^3+x^4);
                        3 additions + 6 multiplications

>   cost(w=&if(a<>0,b+1,c-2));
                   2 additions + functions + 2 assignments
```

SEE ALSO:

```
 optimize
```

A.14 linalg1

HELP FOR:

```
 The mso extension to the linalg package
```

CALLING SEQUENCE:

```
   with(linalg1)
```

SYNOPSIS:

```
 - The linalg1 pseudo-package is a small extension of the linalg package.
   It is loaded using with(linalg1) but all functions belong to linalg

 - Using with(linalg1) the linalg package is loaded with additional routines

 - The trace function is renamed proctrace, avoiding conflict with
   linalg[trace]

 - Avalaible additional routines are listed below :
```

SEE ALSO:

```
linalg, mso, linalg[tilde], linalg[tilded], linalg[rot3d],
         linalg[gaussform], linalg[subrank], linalg[bezoutian],
         linalg[signature], linalg[evalsm],
```

A.15 linalg[tilde]

FUNCTION:

```
tilde tilded - representation of a skew-symmetric matrix
```

CALLING SEQUENCE

S:

```
         SkewSymmetricMatrix:=tilde(Vector)
         Vector:=tilded(SkewSymmetricMatrix)
```

PARAMETERS:

```
      SkewSymmetricMatrix : a skew-symmetric 3x3 matrix
  Vector              : a 3 dimensional vector
```

SYNOPSIS:

```
-  tilde and tilded convert a vector V to a skew-symmetric
matrix H such that :  V ^ x = H * x. We have :

V = (X, Y, Z)

   (0  -Z   Y)
H= (Z   0  -X)
   (-Y  X   0)
```

SEE ALSO:

```
linalg - maple package for Linear Algebra.
```

A.16 linalg[rot3d]

FUNCTION:

```
 rot3d - compute the matrix of a 3D-rotation
```

CALLING SEQUENCE

```
S:
         R:=rot3d(u,a)
         R:=rot3d(v)
         R:=rot3d(alpha,beta,gamma)
         r:=rot3d(R)
         r:=rot3d(R,'alpha','beta','gama')
```

PARAMETERS:

```
        R  a rotation matrix
        u  a unitary vector parallel to the axis of rotation
        a  a scalar equal to the rotations angle
        v  a vector parallel to the axis of rotation which norm is
           equal to 2*tan(a/2)
        r  a vector parallel to the axis of rotation which norm is
           equal to sin(a). We have r = 4/(4+v^2) v.
        alpha, beta, gama scalars equal to the Euler angles of the rotation
        'alpha','beta','gama' names equal to the Euler angles variables
```

SYNOPSIS:

```
- rot3d computes the rotation matrix related to the unitary vector u
  and the angle a, if called with two parameters (trigonometric
  representation of the rotation).
or
- rot3d computes the rotation matrix related to the vector v, as defined
  above, if called with one parameter (algebraic representation of the
  rotation).
or
- rot3d computes the rotation matrix related to Euler angles defined as
  as follows :
   rot3d(array([1,0,0]), alpha) *
   rot3d(array([0,1,0]), beta)  *
   rot3d(array([0,0,1]), gamma)
or
- rot3d computes the rotation vector r related to matrix R
```

```
or
- rot3d computes the equations of the Euler angles related to matrix R
```

EXAMPLE:

```
>   rot3d(array([0,0,1]),a);
                              [ cos(a)  - sin(a)  0 ]
                              [                     ]
                              [ sin(a)   cos(a)   0 ]
                              [                     ]
                              [    0        0     1 ]

>   rot3d(array([0,0,1]));
                                  [ 3/5  -4/5  0 ]
                                  [              ]
                                  [ 4/5   3/5  0 ]
                                  [              ]
                                  [  0     0   1 ]

>   assume(beta>0);
                                  [ 3/5  -4/5  0 ]
                                  [              ]
                                  [ 4/5   3/5  0 ]
                                  [              ]
                                  [  0     0   1 ]

>   rot3d(rot3d(alpha,beta,gama),Alpha,Beta,Gama);
                 sin(gama)                sin(alpha)
    {tan(Gama) = ---------, tan(Alpha) = ----------, sin(Beta) = sin(beta~)}
                 cos(gama)                cos(alpha)
```

SEE ALSO:

```
 linalg - maple package for Linear Algebra.
```

A.17 linalg[subrank]

FUNCTION:

```
 subrank - generates equations for a matrix to be singular
```

CALLING SEQUENCE:

```
subrank(M)
```

PARAMETER:

```
M - a matrix
```

SYNOPSIS:

```
- subrank computes the set of equations which must vanish for
  the matrix to be singular i.e. not of full rank.

- If M is a square matrix the equation
   { det(M) = 0 }
  is returned, else a minimal set of minor to vanish is returned.
```

EXAMPLE:

```
>   subrank(array([[t,0],[0,b],[a,0]]));
                              {t b = 0, - b a = 0}
```

SEE ALSO:

```
linalg - maple package for Linear Algebra.
```

A.18 linalg[gaussform]

FUNCTION:

```
linalg/gaussform - convert a symmetric matrix to its Gauss form
```

CALLING SEQUENCE:

```
gaussform(<matrix>)
```

PARAMETERS:

```
<matrix> - a symmetric matrix
```

SYNOPSIS:

```
 - The Gauss decomposition of the nxn matrix is returned as a exprseq
   of two matrices in the following form : (L, P)
   where:
    - L is a diagonal matrix, i.e. n coefficients, which corresponds,
      to the diagonal of M in the general case (unless one diagonal
      element is zero)
    - P is upper-triangular matrix (unless all diagonal elements are 0)
      with 1 as diagonal terms, i.e. n(n-1)/2 coefficients for the upper
      triangle, such that the n(n+1)/2 coefficients of M are defined
      though: M = P^T L P

 - Such decompositions always exists and are in finite number,
   among them, the one which is the fastest to compute is returned.
```

EXAMPLES:

```
>   P:=array([[1,2,4],[0,1,2],[0,0,1]]):
>   L:=array([[a,0,0],[0,b,0],[0,0,c]]):
>   M:=linalg[multiply](linalg[transpose](P),L,P);
                     [  a      2 a          4 a      ]
                     [                               ]
                     [ 2 a   4 a + b      8 a + 2 b   ]
                     [                               ]
                     [ 4 a  8 a + 2 b  16 a + 4 b + c ]

>   G:=map(u->map(normal,u),gaussform(M));
                          [ a  0  0 ]  [ 1  2  4 ]
                          [         ]  [         ]
                         [[ 0  b  0 ], [ 0  1  2 ]]
                          [         ]  [         ]
                          [ 0  0  c ]  [ 0  0  1 ]
```

SEE ALSO:

```
linalg
```

A.19 linalg[bezoutian]

FUNCTION:

```
linalg/bezoutian - Compute the bezoutian of one or two polynomials
```

CALLING SEQUENCE:

```
bezoutian(P, t)
bezoutian(P, Q, t)
```

PARAMETERS:

```
  P, Q - polynomials
  t    - a variable
```

SYNOPSIS:

```
 - Computes the Bezoutian of two polynomials P(t) and Q(t) in one variable.
   It is a symmetric matrix B defined as :
      -d,   d                              P(x) Q(y) - P(y) Q(x)
  b = >         B[i,j] x^(d-i) y^(d-j) = ———————————-
      -i=0, j=0                                  x - y
   where d = min(deg(P),deg(Q))

 - When called for one polynomial, it is understood as the Bezoutian of P
   with its derivative P'(t). The matrix is thus of size deg(P) x deg(P)
   and its signature is equal to the number of real roots of P
```

EXAMPLE:

```
>   P    := t->a*t^2+b*t+c:
>   B    := bezoutian(P(t),t);
                              [    2            ]
                              [ 2 a       b a   ]
                              [                 ]
                              [        2        ]
                              [  b a  b  - 2 c a ]

>   d    := degree(P(t),t):
>   b1   := sum(sum(B[i,j]*x^(d-i)*y^(d-j),i=1..d),j=1..d):
>   b2   := (P(x)*diff(P(y),y)-P(y)*diff(P(x),x))/(x-y):
>   zero := normal(b1-b2);
                                     0
```

SEE ALSO:

```
linalg, bezout, signature
```

A.20 linalg[signature]

FUNCTION:

```
linalg/signature - compute the signature of list of value
```

CALLING SEQUENCE:

```
signature(v [,t])
```

PARAMETERS:

```
  v - a list, vector, symmetric-matrix or polynomial
  t - a variable t, for the case v is a polynomial
```

SYNOPSIS:

```
 - Compute the signature of a list of values. A signature is the number
   of positive values minus the number of negative values.
 - If the argument is a symmetric matrix, the signature of the diagonal
   of the Gauss decomposition of the matrix is returned.
 - If the argument is a polynomial in one variable t, the signature of
   the Bezoutian of P, i.e. the number of real roots, is returned.
 - If the values are not constant a conditional expression is returned.
```

EXAMPLE:

```
>   signature([a,a*b]);
      &if(a = 0, 0, &if(a < 0, &if(b a = 0, -1, &if(b a < 0, -2, 0)),

                          &if(b a = 0, 1, &if(b a < 0, 0, 2)))          )

>   signature(a*t^2+b*t+c,t);
                              unable to execute seq
```

SEE ALSO:

```
linalg, bezoutian, gaussform
```

A.21 linalg[evalsm]

FUNCTION:

```
linalg/evalsm - perform a symbolic evaluation of linalg operators
```

CALLING SEQUENCE:

```
'linalg/evalsm'(<expr> [,expand])
```

PARAMETERS:

```
<expr> - a symbolic expression involving matrix and vector
expand - a directive to require the expression to expanded
```

SYNOPSIS:

```
- The function evaluate the following symbolic matrix operators :
 A + B    : addition of scalar/vector/matrix with 0 as unary element
 A &* B   : multiplication of scalar/vector/matrix with 1 as unary element
 s *  B   : scalar multiplication
 A &^ n   : integer power of scalar/vector/matrix.
 &t(A)    : transpose of scalar/vector/matrix
 &x(u,v)  : the cross-product of two 3D-vectors
 &.(u,v)  : the dot-product of two vectors
 &D(A,X)  : differential of a scalar/vector/matrix with respect to
            a scalar/vector/matrix
 Det(A)   : determinant of a matrix
 Trace(A) : trace of a matrix

- Note that a constant will be considered as a constant multiple of the
  identity matrix.

- The following properties of a matrix are used for simplification:
   - symmetric
   - antisymmetric
   - singular
   - idempotent
   - nilpotent
   - ScalarMatrix
   - orthogonal
```

```
    - scalar
   as available via the assume() facility

 - Unassigned names will thus be considered either as symbolic matrices
   or as scalars depending on their properties.
```

EXAMPLES:

```
>   unassign('c','A','K','p','C','G','B','x','y','f','g'):
>   assume(c,scalar):
>   assume(A,symmetric):
>   evalsm(&*(&t(&t(&t(A))),1));
                                        A~

>   evalsm(&*(K &^ p, (Pi * K)));
                                 Pi (K &^ (p + 1))

>   evalsm(&*(C &^(-1),&t(&t(C)))+&*(&*(1,K),G,K));
                                  1 + &*(K, G, K)

>   evalsm(C+&*(&t(&t(G))+0)+(&*(&*(1,K),K &^(-1)))-1-G);
                                        C

>   evalsm(&D(&*(&t(A),x)+x+B^y+c &* f(x) &* g(x),x));
         A~ + 1 + c~ (((f(x) &D x) &* g(x)) + (f(x) &* (g(x) &D x)))

>   evalsm(&t(A+c*(&t(B) + G &* F)),expand);
                       c~ ((&t F) &* (&t G)) + c~ B + A~

>   evalsm(&t(A &* (&t(B) + c * G + F)),expand);
                 (B &* A~) + c~ ((&t G) &* A~) + ((&t F) &* A~)

>   evalsm(&t(A &x (&t(B) &x &t(C))),expand);
                            (A~ &* C) B - (A~ &* B) C
```

SEE ALSO:

procdefine, evalm, assume

B User Guide

The **Mascotte Maple Package** has been built to simplify the implementation of reactive software modules in vision and robotics and to facilitate the exchanges of data models and algorithms between builders of such reactive systems, using the Maple Symbolic calculator.
Technical support: vthierry@sophia.inria.fr .

- To know more about Mascotte : HERE also available as ps-file .
- To access to the on-line Maple/Mso doc, just choose your subject:

  and your request.

- To install Mascotte with your maple : HERE
- To feedback to the Mascotte authors : HERE


In order to get this package, you must generate an additional Maple library, from the make.map source file, under the following copyright :

- If you are at at INRIA-Sophia you just have to put the line :
  libname := '/u/krakatoa/0/ftp/pub/html/vthierry/mascotte/lib', libname: with(mso):
  in your *$HOME\.mapleinit* configuration file.
- If you are in a **UniX environment** :
  - Get the source file make.map and the script mso-install file, and store them in a suitable directory, that we will now call **MSO_DIR** .
  - Run *mso-install* in this directory and add the line :
    libname := MSO_DIR, libname: with(mso):
    in the configuration file *$HOME\.mapleinit* of the user(s).

  You can easily maintain this package putting this mso.makefile in *MSO_DIR* .
  More details on the installation can be found in mso-install .
- If you have a **MsDos PC-Compatible installation,** in the *c:/maplev3* directory :
  - Get the source file make.map and the script install.bat file, and store them in the same directory.
  - Run *install.bat* and add the line :
    libname := 'c:/maplev3/mso/', libname: with(mso):
    in the configuration file *c:\maplev3\lib\maple.ini* of your maple installation.
  - Run your standard maple and enjoy it !

  More details on the installation can be found in install.bat .
- If you have **Macintosh machine,** sail it, by a PC or a UniX machine and choose one of the items above.

Other source files, not in the library are also available:

- The ugeometry small set of routines for 2d and 3d geometry.
- More to come !

Please send report any bug, or suggestion :

- Your Email Address **(mandatory)**:

- The subject of your message:
- The contents of your message:

.

Acknowledgments :

- We are thankfull to Michel Buffa for the small icons used in this page.
- This work has been done in the RobotVis project, belonging to Robotic, Image and Vision program of the INRIA institute, located at Sophia Antipolis.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité 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)
ISSN 0249-6399