ALIAS-Maple

The Maple interface for ALIAS

Version 2.8

January 2018

The HEPHAISTOS project

Chapter 1

Introduction

1.1 Preliminaries

The ALIAS-C++ manual has introduced the `ALIAS` C++ library. When using this library to solve a particular system it is necessary to write the C++ code that allows to deal with the system at hand.

The primary purpose of the ALIAS-Maple library was to allow the automatic generation of the C++ code being given the Maple description of an equations system and then to compile and run the generated code in order to get the result within the Maple session. But when developing this library it appears that the use of symbolic computation may allow to increase the efficiency of the C++ solving procedure. Hence the procedures available in the ALIAS-Maple library may be divided into two categories:

- procedures that allow to solve a given problem. In that case the procedure will:
  1. generate the necessary C++ code
  2. compile this code
  3. run the created program
  4. return the result to Maple
- procedures that allow to improve the efficiency of a solving procedure (which may eventually use C++ program).

The ALIAS-Maple library `ALIAS.m` has initially been developed by Didier Bondyfalat. ALIAS-Maple is compiled for Maple V.5 and 9.5 and this version has 49356 lines of code.

In this package there are numerous Maple procedures that may be used to solve or to help to solve a specific problem. The behavior of these procedures may be modified by changing the values of some Maple variables (that will correspond to the parameters of the ALIAS-C++ procedures). An ALIAS-Maple variable is always defined using the following syntax:

`‘ALIAS/XX‘`

The use of some of these variables may need an understanding of the algorithms of ALIAS-C++.

The C++ procedures created by ALIAS-Maple procedures have usually a fixed name (e.g. `_GS_` for the general purpose solving procedures) but this behavior may be changed for some of them if the string variable `‘ALIAS/ID‘` is defined (and on the long term it is planned that all ALIAS-Maple procedures will propose this possibility): in that case all files created by a procedure have the `‘ALIAS/ID‘` string appended to their name.

Before explaining how to use this package very **important** remarks have to be done:

- an equation $f(x) = 0$ is defined in ALIAS-Maple by $f(x)$ while an inequality is defined by $f(x) \leq 0$
- not all expression may interval evaluated. For example interval evaluating 1/x with x being an interval including 0 will cause the C++ program created by ALIAS-Maple to crash. ALIAS-Maple provides procedures to deal with this problem (see sections 2.1.4,2.1.5)

- it has been noticed that for large expressions the compilation time of their C++ equivalent may be quite large (or even may be impossible). It is sometime possible to avoid this problem by turning off the optimization flag. Indeed by default the C++ program will be compiled with the optimization flag `-O` and it is sometime better to turn off this option by setting the flag `'ALIAS/optimized'` to 0. Another approach is to use some procedures provided in ALIAS-Maple that simplify the compilation by using dummy intervals or decompose expression into elementary components as will be seen later on. This problem may also be solved by using the `ALIAS` parser in order to avoid having to compile very large expressions.

The reader interested only in the solving procedures available in ALIAS-Maple may skip to chapter 3. But the concept of simplification procedures explained in chapter 4 are worth considering afterward as they may drastically improve the efficiency of the algorithms.

This manual describes the use of the `ALIAS` Maple library both for version 5.5 and for version 9.5, with minor difference when loading the library. This interface is intended to be used through a command line and not a worksheet.

1.2 Installing the Maple library

It is assumed here that you have installed the `ALIAS`-C++ library together with the `Profil` for example, in the directory `/u/ALIAS`. To install the MAPLE library you need first to copy the library `ALIAS.m` and the sub-directory `ALIAS` in some specific directory, say `/u/ALIAS-MAPLE`. You end up with:

- the directory `/u/ALIAS-MAPLE` containing the file `ALIAS.m` and a sub-directory `/u/ALIAS-MAPLEALIAS`
- the directory `/u/ALIAS` with the directories `Lib` and `Profil` of the `ALIAS`-C++ library

To use the `ALIAS` MAPLE library you must first update your `libname` variable by:

```
libname:=libname,"/u/ALIAS-MAPLE":
```

Then you will have to indicate where the `ALIAS`-C++ libraries are located. This is done by:

```
'ALIAS/profil':="/u/ALIAS/Profil":
'ALIAS/lib':="/u/ALIAS/Lib":
```

You must also indicate the full path of your C++ compiler using the variable `'ALIAS/gpp_sun'` (if you are using a SUN computer) or `'ALIAS/make_linux'` (for a Linux computer).

Note that if you are planning to use the parallel implementation of the algorithms (see the corresponding chapter) on a network of heterogeneous machines (PC's and SUN), then you must indicate the locations of the Profil and `ALIAS` libraries for the machines whose architecture is different from the machine that you are using with the variable `'ALIAS/libN'` and `'ALIAS/profilN'`.

Thus a typical `Maple` file for using the `ALIAS`-Maple library with Maple 5.5 on a SUN computer will begin with:

```
libname:=libname,"/u/ALIAS-MAPLE":
with(ALIAS):
'ALIAS/profil':="/u/ALIAS/Profil":
'ALIAS/lib':="/u/ALIAS/Lib":
'ALIAS/gpp_sun':="/usr/local/bin/g++":
```

Alternatively you may choose to initialize these variables in your `.mapleinit` file. The only difference when using Maple9.5 is that you cannot use the Maple command `with` for compatibility problem with Maple 5.5. For Maple9.5 and a linux workstation loading and starting the library is done as follows:

```
libname:=libname,"/u/ALIAS-MAPLE":
read "Start_ALIAS.maple"
'ALIAS/profil':="/u/ALIAS/Profil":
'ALIAS/lib':="/u/ALIAS/Lib":
'ALIAS/gpp_linux':="/usr/local/bin/g++":
```

The Maple version is given in the global variable `'ALIAS/maple_version'`. Beside this loading difference there is no other difference between the use of `ALIAS` version 5.5 and version 9.5 (apart of the fact that linear algebra in the procedure uses the `linalg` package instead of the newer `LinearAlgebra` which is much less efficient for the operations involved in the procedure).

1.3 On-line help and ALIAS-On-Line

Most of the procedures of ALIAS-Maple have a built-in help. This help is obtained using the classical Maple syntax, with a ? followed by the name of the procedure. For example:

```
?ALIAS
```

will give the names of all the available procedures of ALIAS-Maple. Help for a particular procedure is obtained by:

```
?ALIAS,name
```

where `name` is the name of the procedure or equivalently by

```
?ALIAS,name
```

except for the `Parameters` procedure.

1.4 Interval valuation of expression

There are few exceptions for which an interval evaluation of an expression cannot be computed for some ranges of the unknowns that appear in the expression. Namely the following problems may occur when intervals are involved:

- denominator that may include 0
- argument of square should be positive
- argument of arcsin and arccos should be included in [-1,1]
- argument of log,ln,log10 should be positive
- argument of arccosh should be greater than 1
- argument of arctanh cannot have an intersection with the interval [-1,1]
- argument x of x^y where y is not an integer should be positive
- argument x of $exp(x)$ should not be too large to avoid overflow problem.

In that case the `BIAS/Profil` interval arithmetic package will issue a fatal error. Hence attention should be paid to these evaluation problems. `ALIAS`-Maple offers strategies to deal with these problems, see sections 2.1.4,2.1.5.

Chapter 2

Interval evaluation in ALIAS-Maple

Clearly for any interval analysis method that deal with a problem involving expressions it is necessary to have a program that is able to interval evaluate these expressions. We will see also that having a program to interval evaluate the first and second derivatives may help to solve the problem. ALIAS-Maple provides various procedures that allow to create automatically such C++ program, being given only the expressions in Maple. These procedures will be presented in the next section. For testing purposes it may be interesting to have in Maple a procedure that interval evaluate an expression: such procedure will be presented in section 2.2. Special C++ code generating procedures will be presented in the section 2.3.

2.1 Equations, Gradient and Hessian

2.1.1 MakeF, MakeJ, MakeH

This set of procedures enable one to generate automatically C++ code for later use with the ALIAS-C++ library:

- MakeF: generates the procedure for evaluating a set of equations. These equations may be composed of most classical mathematical functions (see section 3.1.1 for a list of allowed operators) but may also include unexpanded determinants (see section 2.1.3).
- MakeJ: generates the procedure for evaluating the gradient of a set of equations
- MakeH: generates the procedure for evaluating the Hessian of a set of equations

The syntax for these procedures is:

```
Make[FJH]("[C++ file name]","[procedure name]",[list of equations],[list of unknowns]);
```

Thus:

```
with(ALIAS):
expr := [x^2+y^2-1,x+y]:
MakeF("Test.C","Test",expr,[x,y]);
```

enable to create the following C++ program Test.C

```
/* Code automatically written by Maple */
INTERVAL_VECTOR Test(int l1, int l2, INTERVAL_VECTOR & v_IS) {
        INTERVAL_VECTOR V(2);
        if (l1<=1 && 1<=l2)
                V(1)=Sqr(v_IS(1))+Sqr(v_IS(2))-1;
        if (l1<=2 && 2<=l2)
                V(2)=v_IS(1)+v_IS(2);
        return V;
}
```

Note that the program generated by `MakeF` has a specific form that allows to interval evaluate all expressions or a specific set of expressions: such format will be called the *MakeF format*. There are also a `MakeJ` and `MakeH` format. The procedures `Make[FJH]` makes an extensive use of the procedure `MinimalCout` that try to find the optimal formulation of an expression with respect to interval arithmetic. Indeed not all mathematically equivalent formulation of an expression are interval equivalent (i.e. they produce the same interval evaluation). For example $x^2 + 2x + 1$ and $(x + 1)^2$ are equivalent but the second formulation always produces the optimal interval evaluation as there is only one occurrence of x. For further detail on `MinimalCout` see section 9.7. Note that the variable `'ALIAS/mincout'` plays an important role in the calculation time of these procedures.

Note that not all expression can be interval evaluated (e.g. 1/x cannot be evaluated if the interval for x includes 0). `ALIAS` provide an automatic way to avoid evaluation problems (see section 2.1.5). If you use your own procedure and are aware of evaluation problems and modify the returned values it will be a good policy to set C++ flags `ALIAS_ChangeF`, `ALIAS_ChangeJ` to 1 (default value 0) if a change occurs: this will allow the `ALIAS` C++ library to avoid using improperly the returned value (e.g in the interval Newton scheme).

There are two "optimized" versions of `MakeF` and `MakeJ` called `MakeFO` and `MakeJO` that may produce a better code from the view point of interval evaluation but with a larger computation time. Note that the `Code` procedure may be used to produce simple C++ equivalent of a formula. Note also that inequalities are also recognized by these procedures.

In most of the ALIAS-Maple procedures it may be possible for the user to provide its own procedure for the evaluation of the expressions and of their derivatives. This is obtained by setting the flag `'ALIAS/user_func'` to a string that is the name of the C++ file that defines a procedure which will be used for evaluating the expressions (the name of the procedure must be `F`. Similarly the flag `'ALIAS/user_derivative'` may be used for the derivative of the expressions in which the procedure name must be `J`.

For the `MakeF` the end-user may also indicate that he has already written the expressions in a form that is optimal and that `MakeF` should only translate as it the expression in C++ without any modification by setting the flag `'ALIAS/as_itF'` to 1.

For the `MakeJ` procedure the end-user may also indicate that he has already computed the derivatives by setting the flag `'ALIAS/as_itJ'` to 1 and by defining the variable `'ALIAS/as_itJ_array'` as an array that contains the derivatives of all the expressions (the i-th row of this array contains the derivatives of the i-th expression with respect to the variables). If you have determinant in the expression you have to set the flag `'ALIAS/as_itJ_mat'` to 1 to avoid `MakeJ` trying to find a better evaluation function.

Instead of generating C++ code these procedures may produce a C++ code that will use a parser. A motivation of using the parser is that the compilation time of the program necessary to interval evaluate a complex expression may be very large (note that the procedure `Auto_Diff`, section 2.3.2, may allow to reduce this compilation time).

Each expression to evaluate will be written in a file and the interval evaluation will be done at run-time by parsing the file. This allow to deal with large expressions at a small cost. This is obtained by setting the variable `'ALIAS/use_parser'` along the following rules:

- 5: only the Hessian will be calculated using the parser
- 6: only the Jacobian and Hessian will be calculated using the parser
- 7: the expressions, their Jacobian and Hessian will be calculated using the parser
- 10: the expressions, their Jacobian and Hessian will be calculated using the parser

To determine if a procedure offers this possibility check the on-line help and look if the variable `'ALIAS/use_parser'` is in the list of the global variables for the procedure.

2.1.2 Improving the efficiency of the code

It may happen that the evaluation of an expression involves many time the evaluation of a sub-expression. Clearly evaluating only once these sub-expressions will speed up the code. This may be done through a user-provided Maple procedure that must be called `ALIAS_FSIMPLIFY`. This procedure takes as input a file descriptor and an expression expr. It will be first called right after the creation of the evaluation file with a string: detecting that expr is a string allow to write some initialization. Then it will be called before writing any equation to allow

for simplification. For example assume that an expression that will be treated by the `Make[FJH]` procedure involves numerous time the evaluation of the sine and cosine of the first variable `x` (whose name in ALIAS-C++ is `v_IS(1)`).The `ALIAS_FSIMPLIFY` procedure may be written as:

```
ALIAS_FSIMPLIFY:=proc(fid,expr)
local aux:
if type(expr,string) then
    fprintf(fid,"INTERVAL SS,CC;\n"):
    fprintf(fid,"SS=Sin(v_IS(1));\n"):
    fprintf(fid,"CC=Cos(v_IS(1));\n"):
    RETURN(0):
fi:
aux:=expr:
aux:=subs(sin(x)=SS,cos(x)=CC,aux):
RETURN(aux):
end:
```

This procedure will be first called with a string for `expr` and a consequence is that at the beginning of the evaluation file `fid` the interval variable `SS,CC` will be defined and then assigned to the value of $\sin(x), \cos(x)$. Then the procedure will be called for each expression that will be assigned to `expr`: each occurrence of the sine and cosine of x in the expression will be substituted by `SS, CC`.

The procedure `Math_Func`, see section 9.4, may be used to identify mathematical functions occurring in an expression and the list provided by this function may be used to write a generic `ALIAS_FSIMPLIFY` procedure that will automatically compute only once the more complex components of an expression. The procedure `Auto_Diff`, see section 2.3.2, may also be used to speed up the interval evaluation of an expression. Note also that a similar mechanism exists for expression involving determinants of matrices.

2.1.3 Function involving determinants

A determinant of a matrix `A` may appear in an equation using one of the syntax:

```
Fast_Determinant(A)  Medium_Determinant(A)  Slow_Determinant(A)
```

the differences between the syntax being only in the computation time (from the fastest to the slowest) and in the width of the interval evaluation (from the largest to the narrowest). If you are interested in determining if a determinant may cancel you may also use `Slow_NonZero_Determinant` which is faster than `Slow_Determinant`.

Note that if determinants are present in the equations there are different ways to compute the determinant:

1. by using Gaussian elimination. This method will be used only if the flag `‘ALIAS/use_gaussian_elim_det‘` is set to 1
2. either by computing only the intervals for each component of the matrix and using a row or column expansion to determine the interval evaluation of the determinant
3. same as above but the gradient of the coefficients are used to improve their interval evaluation
4. by computing symbolically every minors of dimension n of the matrix and using the interval evaluation of these expressions to compute the interval evaluation of the determinant. The value of n is given by `‘ALIAS/minor22` (default value: 2)
5. a mix of method 2 and 3

or pre-computing an interval expression for all the dimension 2 minors (thereby using, for example, either the `Fast_Determinant` or the `Fast_Determinant22` procedures. The second procedure will, in general, lead to better interval evaluation but Maple may take some time to produce the corresponding source code. You may choose one of these ways by setting the flag `‘ALIAS/det22‘` to 0, 1, 2 or 3 (default value: 0). Clearly the values 2 and 3 should be avoided for large matrix as the Maple computation time and the size of the generated code

may be large. If the determinant of minors is used, then the expansion will be done according either to the row or to the column according to the value of ‘ALIAS/row22‘. A value of 0 (the default value means that the expansion will done along the rows and for any other value according to the column).

Note that you may speed up the interval evaluation of functions including determinant of matrices (which are often computer intensive) by defining intermediate interval variables and substituting these variables in the expression. For example assume that the coefficients of a matrix A involve a large number of sine and cosine depending upon the unknown x, the first unknown in our list of unknowns: you first define a Maple procedure intro_A that return an array of strings that contain all the definition of the intermediate variables. In our case the procedure is:

```
intro_A=proc()
local h;
h:=["INTERVAL SX;SX=Sin(v_IS(1));","INTERVAL SY;SY=Cos(v_IS(1));"]
RETURN(h);
end:
```

Note that the name of the unknowns in the code generated by ALIAS is v_IS. The two strings of the array h will be automatically inserted in the C++ procedure generated by ALIAS. Hence the intermediate variable SX, SY will contain the value of the interval evaluation of $sin(x), cos(x)$. You may then write the simplification procedure that will substitute every occurrence of Sin(v_IS(1)) and Cos(v_IS(1)) by SX and CX. The name of this procedure must be simplify_A and is written as:

```
simplify_A:=proc(eq)
local eq1;
eq1:=subs(Sin(v_IS(1))=SX,Cos(v_IS(1)=CX,eq):
RETURN(eq1);
end:
```

2.1.4 Dealing with undefined expressions

Not all expression may be evaluated using interval arithmetic. For example expression involving a denominator whose interval evaluation contain 0 cannot be evaluated. Similarly expressions involving square root of terms whose lower bound is negative are not allowed. This does not mean that such expressions cannot be dealt with ALIAS but that the code must take care of such cases. The procedures MakeF and MakeJ allows the user to deal with such cases. Note also that a package described in section 2.1.5 allows one to to produce automatically the procedures that are described in this section.

The purpose of the control mechanism is to allow the user to calculate the interval evaluation of terms that may cause a problem for the evaluation and if this a case to attribute a default value to the expression that use these terms. This is done directly at the level of the C++ code.

First of all it will be necessary to define some C++ interval that will be used during the auxiliary computation. MakeF will look at the string ‘ALIAS/user_FINIT‘ and if it is not of 0 length will write it directly after the beginning of the procedure. Hence writing:

```
‘ALIAS/user_FINIT‘:="INTERVAL U;":
```

will allow to use the C++ interval U for the auxiliary computation. The procedure MakeJ uses for a similar purpose the variable ‘ALIAS/user_JINIT‘. For the MakeF procedure the user will have to define a Maple procedure ALIAS_F that will be called before the generation of the C++ code of each equations or inequalities involved in the calculation. The syntax of this procedure is:

```
ALIAS_F:=proc(fid,i)
```

where fid is the Maple file descriptor in which the C++ code is written and i is the number of the expression that is considered. This procedure allows to include some C++ code right before the evaluation of the expression i.

For example assume that you have a set of inequalities function of the variable x, y that are defined in the list INEQ and that some of these inequalities may have interval denominator. Hence before the evaluation it is

necessary to check if the denominator evaluation may include 0, in which case the whole expression has to be evaluated to a large interval including 0 (indeed the algorithm of ALIAS will then consider that this inequality is not satisfied). An ALIAS_F procedure for this case may be written as:

```
ALIAS_F:=proc(fid,i)
global INEQ:
local j,aux:

# denom of inequality i is numeric: do nothing
if type(denom(op(1,INEQ[i])),numeric) then RETURN(0): fi:
# denom is not numeric, evaluate the denominator
        aux:=denom(op(1,INEQ[i])):
#
#in the C++ evaluation procedure the unknown are in the table v_IS
    aux:=subs(x=v_IS(1),y=v_IS(2),aux):
#substitute the mathematical operator by their interval equivalent
#using ALIAS procedure
    aux:=`ALIAS/ReplaceText`("..",",",convert(aux,string)):
#write the denominator evaluation in the C++ file
    fprintf(fid,"U=(%s);\n",aux):
#if the denominator evaluation include 0 return a large interval
#for expression i
    fprintf(fid,"if((0<=U))V(%d)=INTERVAL(-1.e6,1.e6);\n",i):
#otherwise proceed with the real evaluation
    fprintf(fid,"else\n"):
RETURN(0):
end:
```

Note that for MakeF the C++ evaluation of the i-th expression is preceded by the label nexti (hence expression has label next2, expression 3 next3 and so on). Hence you may use a goto next3 to skip the evaluation of the second expression.

A similar mechanism is available for the MakeJ procedure. Before writing the code for the evaluation of the derivative of the expression i with respect to the unknown j the procedure ALIAS_J will be called. The syntax of this procedure is

```
ALIAS_J:=proc(fid,i,j)
```

Note that in this case is compulsory to return a large interval if the evaluation cannot be done as the derivative may be used to improve the evaluation using a first order Taylor expansion. Note that the C++ procedure created by MakeJ evaluates the components of the jacobian element by element although it returns an interval matrix V.

2.1.5 Interval valuation and the Problem_Expression package

This package allows to deal automatically with expressions that cannot be evaluated using interval arithmetic for some range for the unknowns. Their purpose is to create ALIAS_F and ALIAS_J procedures that will be used by MakeF and MakeJ.

The first procedure in this package is Problem_Expression that will create a list A_CONSTS of constraints that must be verified to be able to evaluate an expression. The end-user may provide information on constraints that will always be satisfied by writing them in the global list variable A_CONSTS_ALWAYS_OK.

The syntax of Problem_Expression is:

```
Problem_Expression(expr)
```

For example

```
Problem_Expression(x/(y-2)+sqrt(x+1));
```

will produce the list

```
[y - 2 <> 0, -x - 1 <= 0]
```

that indicates that the denominator of `x/(y-2)` must not include 0 and that the argument of `sqrt` cannot be an interval that includes negative numbers. Note that some element of this list of constraints may be used to speed up the solving of a problem by using the `HullIConsistency` procedure (see section 4.2.1) that will modify the ranges of the unknowns so that the expressions may be evaluated or even reject ranges for which one (or more) expression cannot be evaluated.

This procedure deals with the following evaluation problems:

- denominator that may include 0
- argument of square should be positive
- argument of arcsin and arccos should be included in [-1,1]
- argument of log,ln,log10 should be positive
- argument of arccosh should be greater than 1
- argument of arctanh cannot have an intersection with the interval [-1,1]
- argument x of x^y where y is not an integer should be positive
- argument x of $exp(x)$ should not be too large to avoid overflow problem.

Using the constraints produced by `Problem_Expression` the `Verify_Problem_Expression` procedure will create an `ALIAS_F` procedure to be used by `MakeF`. It uses the following rules for the evaluation of an expression `expr`:

- if the constraint is $x \leq 0$ where $x = [\underline{x}, \overline{x}]$
  - if $\underline{x}$ is positive `expr` will be set to [`` `ALIAS/low_value_expr_violated` ``,`` `ALIAS/high_value_expr_violated` ``]
  - if $\underline{x} \geq$ -`` `ALIAS/close_to_zero` `` and $\overline{x} > 0, \leq$ `` `ALIAS/close_to_zero` `` `expr` will be set to [`` `ALIAS/low_value_expr_violated` ``, `` `ALIAS/high_value_expr_violated` ``]
  - if $\underline{x} < 0$ and $\overline{x}$ is positive `expr` will be set to the interval [`` `ALIAS/lower_bound_uneval_expr` ``, `` `ALIAS/upper_bound_uneval_expr` ``]
- if the constraint is $x \neq 0$
  - if $|\underline{x}|, |\overline{x}| \leq$ `` `ALIAS/close_to_zero` `` `expr` will be set to [`` `ALIAS/low_value_expr_violated` ``, `` `ALIAS/high_value_expr_violated` ``]
  - if x includes 0 or ($\underline{x} < 0$ and $|\underline{x}| <$ 'ALIAS/close_to_zero') or ($\overline{x} > 0$ and $|\overline{x}| <$ 'ALIAS/close_to_zero') `expr` will be set to the interval [`` `ALIAS/lower_bound_uneval_expr` ``, `` `ALIAS/upper_bound_uneval_expr` ``]
- if the constraint is $x \cap [-1, 1] = \emptyset$ (for the arccoth function) and x has an intersection with this interval then `expr` will be set to the interval [`` `ALIAS/lower_bound_uneval_expr` ``, `` `ALIAS/upper_bound_uneval_expr` ``]
- if we deal with e^x
  - if $\underline{x} <$200 and $\overline{x} >$ 200 then `expr` is set to the interval $[e^{\underline{x}}, e^{\underline{x}} + 10^{50}]$
  - if $\underline{x} >$200 then `xpr` is set to $[10^{50}, 10^{60}]$
- if the constraint is $x \subset [-1, 1]$ (for the arcsin, arccos functions) then
  - if $\underline{x} < -1$ or $\overline{x} > 1$ then `expr` will be set to [`` `ALIAS/low_value_expr_violated` ``, `` `ALIAS/high_value_expr_violated` ``]
  - if $\underline{x} < -1$ and $\overline{x} \leq 1$ then x will be set to $[-1, \overline{x}]$
  - if $\overline{x} > 1$ and $\underline{x} \geq -1$ then x will be set to $[\underline{x}, 1]$

The syntax of the procedure is

```
Verify_Problem_Expression(EQ,VAR)
```

where `EQ` is a list of expressions and `VAR` a list of unknown names. The global list variable `‘ALIAS/Problems` will contain the number of evaluation problems detected for each element of `EQ`.

This procedure generates also a C++ file `ALIAS_F_AUTOMATIC.c` with two procedures

- `Is_Evaluable(INTERVAL_VECTOR X)` that returns 1 if all the expressions in `EQ` can be evaluated when the unknown `VAR` have as range `X`
- `Is_Evaluable(INTERVAL_VECTOR X), int i` that returns 1 if all the i-th expression in `EQ` can be evaluated when the unknown `VAR` have as range `X`

Note that of the string `‘ALIAS/ID‘` is not empty the created procedure will be `ALIAS_F_AUTOMATIC‘ALIAS/ID‘.c` and will contain the procedures named `Is_Evaluable‘ALIAS/ID‘`. For example if ‘ALIAS/ID‘ has been set to `"100"` the file will be `ALIAS_F_AUTOMATIC100.c` which will include the routines `Is_Evaluable100`.

If these procedures have been created, then the flag `‘ALIAS/has_test_evaluate‘` will be set to 1 (the procedure uses also the internal flag `‘ALIAS/test_evaluate‘`) to determine what should be done when a non-evaluability condition is satisfied for a given variable `U`. The code will be

```
if (non evaluable condition) {
          U=default value;
          goto next[i];} #if ‘ALIAS/test_evaluate‘=0, i being computed
          return 0;}     #if ‘ALIAS/test_evaluate‘=1
          }              #‘ALIAS/test_evaluate‘=2
         goto next[i];}   #‘ALIAS/test_evaluate‘=3, i =ALIAS_Next
U=real U value;
next[i]: ;
if (non evaluable condition) {
           ....
```

If `‘ALIAS/test_evaluate‘` is set to 3 the label number is given by the global ALIAS variable `ALIAS_Next`.

These procedures may be used to test if an expression may be evaluated before using them for a filtering algorithm. For example the `HullConsistency`, `HullIConsistency` and `Simp2B` that implement the 2B filtering use automatically these procedures as soon as `Verify_Problem_Expression` has been called before their use (but `Verify_Problem_Expression` has to be called with the same list of unknowns and variables than the consistency procedures).

Note that `Verify_Problem_Expression` may have a third argument which is a string (e.g. `"ALIAS_Coeff"`). In that case instead of creating the `ALIAS_F_AUTOMATIC` file with the `ALIAS_F` Maple procedure the file `ALIAS_Coeff` will be created describing the Maple procedure `ALIAS_Coeff(fid,i)`. If this procedure is called for expression `i` of `func` the procedure will create in the file described by `fid` the C++ code necessary to determine if this expression can be interval-valuated. If not then it is necessary to affect the i-th element of the interval vector that will be returned by the procedure. As we do not know the name of this vector it shall be indicated as the fourth argument of `Verify_Problem_Expression`. Hence

```
Verify_Problem_Expression(func ,Vars,"ALIAS_F_AUTOMATIC","V")
```

is equivalent to `Verify_Problem_Expression(func ,Vars)`. See an application of this use in section 8.3.

The `Verify_Problem_ExpressionJ` procedure creates in the same way a Maple procedure `ALIAS_J`. The rules are identical but the syntax of the procedure is

```
Verify_Problem_ExpressionJ(J,VAR)
```

where `J` is an array that define the jacobian of `EQ` with respect to `VAR`. A typical call to this procedure is

```
with(linalg):
J:=jacobian(EQ,VAR):
Verify_Problem_ExpressionJ(eval(J),VAR)
```

The name of the variables that allows one to specify what should be the return value of the expression are: `'ALIAS/low_value_exprJ_violated'`, `'ALIAS/high_value_exprJ_violated'`.

Note that in the C++ procedures generated by `MakeF`, `MakeJ` if an evaluation problem occurs the C++ flags `ALIAS_ChangeF`, `ALIAS_ChangeJ` will be set to 1 (default value 0) to indicate that the resulting computed value should be used with some care.

2.2 Interval evaluation of an expression in `Maple`

Maple provides the package `evalr` that calculate the interval evaluation of an expression. An example follows:

```
readlib(evalr):
X:=INTERVAL(0..1):
evalr(sin(X)*cos(X)+X);
```

But this package does not evaluate correctly some expressions (for example X-X will be evaluated to 0 whatever is the interval for X). ALIAS-Maple provides the procedure `Interval` that allows for the interval evaluation of an expression as shown in the following example:

```
Interval([x*cos(y)+sin(x*y),x^2-cosh(y)],[x,y],[[-2,2],[-2,2]]);
               [[[-3, 3]], [[-7.3890560989307,3.8646647167634]]]
```

The first argument is a list of expression, the second argument is the name of the variables and the third a list of intervals for the variables. The procedure generates a C++ program and hence may take some seconds. But as soon as the executable has been created it is possible to re-use it with other intervals using the `Restart` procedure

```
Restart("Interval",[[-0.5,0.5],[-0.5,0.5]]);
```

Note however that as soon as you have defined the `'ALIAS/ID'` string before generating an executable it is necessary to reset this string to the same name before using the `Restart` procedure to re-run the same executable

An optional 4th argument allows to control the evaluation. Indeed by default the expression will be transformed into a compact form (as provided by the procedure `MinimalCout`) that may lead to a better evaluation. The fourth argument may be:

- `"AsIt"`: the expressions are not transformed and are evaluated as they are provided
- `"Gradient"`: the derivatives of the expressions are used to try to improve the evaluation

Here is an example:

```
Interval([x*sin(x)+cos(x)],[x],[[0,1]]);
          [[[.54030230586814, 1.8414709848079]]]

Interval([x*sin(x)+cos(x)],[x],[[0,1]],"Gradient");
                  [[[1, 1.381773290676]]]
```

in which the use of the "Gradient" option allows to get the exact interval evaluation. It must be reminded that the interval evaluation of an expression is very sensitive to the manner with which the expression is written. For example the expression $x^2 + 2x + 1$ for x in the interval [-1,1] is evaluated as:

```
Interval([x^2+2*x+1],[x],[[-1,1]],"AsIt");
                         [[[-1, 4]]]
```

If the option `AsIt` is not used `ALIAS` will convert the expression into the Horner form $x(x + 2) + 1$:

```
Interval([x^2+2*x+1],[x],[[-1,1]]);
                         [[[-2, 4]]]
```

Here it may be noticed that the Horner form leads to a worst interval evaluation. But we may factor this expression as $(x + 1)^2$:

```
Interval([(x+1)^2],[x],[[-1,1]],"AsIt");
                         [[[0, 4]]]
```

2.3 Generating code

2.3.1 Transforming expressions into C++ code

The BIAS/Profil C++ version of an arbitrary Maple expression may be obtained by using the procedure `Code` that will display on the standard output the C++ equivalent of the Maple expression. The syntax of this command is:

```
Code(eq);
Code([eq1,eq2,eq3]);
```

In the later case the components of the interval vector `v_IS` will be affected to each component in the list.

If this procedure has a second argument which is a list of variable the procedure will provide a C++ evaluation formula where the unknown has been substituted by the element of the interval vector `v_IS` (which the name of the interval vector used internally by ALIAS):

```
Code(x^2+2*x+1,[x]);
               Sqr(v_IS(1)) + 2 v_IS(1) + 1
```

An optional last argument, the string `"minimal"` may be added: in that case the procedure will try to produce a compact form for the expression that may be more appropriate for interval evaluation and is the result of the procedure `MinimalCout`:

```
Code(x^2+2*x+1);
v_IS=Sqr(x)+2*x+1;
Code(x^2+2*x+1,"minimal");
v_IS=1+(2+x)*x;
```

2.3.2 Interval evaluation and Taylor remainder

The procedure `Auto_Diff` addresses two main problems:

1. when dealing with large or complex expression the compilation time of the C++ program that is needed to interval evaluate the expression may be quite large
2. for some problem it may be interesting to have a procedure that computes the Taylor remainder of a given expression. But this remainder may have a very large expression and the first item apply

The syntax of this procedure is

```
Auto_Diff(Func,Vars,N,REM)
```

where `N` is the order of the Taylor remainder (if `N` is set to 0 this remainder is the expression itself) and `REM` is a string that will be the name of the C++ program that will compute the remainder (the program is written in the file `REM.c`). Note that `REM` is a procedure using `MakeF` format.

As the remainder may be a very large expression but which includes multiple occurrences of the same elementary components `Auto_Diff` uses the `Decompose_Diff` procedure to reduce the number of interval evaluation of the same elementary components.

Consider the expression

$$(sin(x)^2 + cos(x)^3)/x + x * (sin(x)^2 + cos(x)^3)$$

and apply `Auto_Diff` for the evaluation of this expression

```
Auto_Diff((sin(x)^2+cos(x)^3)/x+x*(sin(x)^2+cos(x)^3),[x],0,"REM");
           (ALIAS_B4 + ALIAS_B5) ALIAS_B1 + x (ALIAS_B4 + ALIAS_B5)
```

This indicates that the terms $\sin(x)^2, \cos(x)^3, 1/x$ will be calculated only once and affected to the dummy interval `ALIAS_B4`, `ALIAS_B5`, `ALIAS_B1`.

The calculation of these dummy variables is done in the procedure `ALIAS_DIFF.C`:

```
#include <fstream.h>
#include "Functions.h"
#include "Vector.h"
#include "IntervalVector.h"
#include "IntervalMatrix.h"
#include "IntervalMatrix.h"
#include "IntegerVector.h"
#include "IntegerMatrix.h"
INTERVAL_VECTOR ALIAS_DIFF(INTERVAL_VECTOR &v_IS)
{
INTERVAL_VECTOR B(3);
INTERVAL_VECTOR AD(5);
Clear(AD);
B(1)=Sqr(Sin(v_IS(1)));
B(2)=Power(Cos(v_IS(1)),3);
B(3)=1/v_IS(1);
AD(1)=B(3);
AD(2)=B(1);
AD(3)=B(2);
AD(4)=B(1);
AD(5)=B(2);
return AD;
}
```

Note that `ALIAS_DIFF.C` is a self-contained program that can be compiled independently. The code for `REM.c` is

```
INTERVAL_VECTOR ALIAS_DIFF(INTERVAL_VECTOR &v_IS);
#include "ALIAS_REM.c"
INTERVAL_VECTOR REM(int l1,int l2,INTERVAL_VECTOR & v_IS)
{
INTERVAL_VECTOR V(1),X(5),XX(6);
int i;
//computation of the elementary components
X=ALIAS_DIFF(v_IS);
for(i=1;i<=1;i++)XX(i)=v_IS(i);
for(i=1;i<=5;i++)XX(i+1)=X(i);
//XX contains the unknowns and then the elementary components
V=ALIAS_REM(1,1,XX);
return V;
}
```

Here the interval vector `XX` contains first the variable and then the dummy variables. The calculation of the expression is then performed by the `ALIAS_REM` procedure:

```
INTERVAL_VECTOR ALIAS_REM(int l1, int l2, INTERVAL_VECTOR & v_IS) {
        INTERVAL_VECTOR V(1);
next1:
        if (l1<=1 && 1<=l2)
        {
        V(1)=(v_IS(5)+v_IS(6))*v_IS(2)+v_IS(1)*(v_IS(5)+v_IS(6));
        }
next2: ;
        return V;
}
```

For a remainder let's use

```
Auto_Diff((sin(x)^2+cos(x)^3)/x+x*(sin(x)^2+cos(x)^3),[x],1,"REM");
```

which returns

```
(ALIAS_B4 ALIAS_B3 + ALIAS_B7 + ALIAS_B8 + ALIAS_B4 ALIAS_B2
     + (ALIAS_B7 + ALIAS_B8) ALIAS_B1 + x (ALIAS_B3 + ALIAS_B2)) W_x_1
```

Here `W_x_1` represents $x-h$ where h is the expansion point (`W_x_2` will represent $(x-h)^2$ and so on). This terms is calculated in the program `ALIAS_DIAM.C`.

Note that `Auto_Diff` try to improve the interval evaluation by using a Horner form. But it can be seen that it does not always provide the best one. In the previous example

```
((ALIAS_B7 + ALIAS_B8)(1+ ALIAS_B1) + (x+ALIAS_B4) (ALIAS_B3 + ALIAS_B2)) W_x_1
```

will have been optimal.

Chapter 3

The solving procedures

3.1 Introduction

3.1.1 Allowed mathematical operators

The following `Maple` mathematical operators are recognized in an expression:

- arithmetic: `+,-,*,/,^,**`
- trigonometry: `sin,cos,tan,cot,sec,csc,arcsin,arccos,arctan,arccot`
- hyperbolic trigonometry : `sinh,cosh,tanh,coth,sech,csch,arcsinh,arccosh,arctanh,arccoth`
- log and exponential: `log,ln,log10,exp`
- miscellaneous: `abs, signum, signum(1,..),abs(1,...),INTERVAL,ceil,floor,round`

Note that the value of `signum(x)=|x|/x` where `x` is an interval whose absolute value is lower than `‘ALIAS/value_sign_signum‘` is set to the Maple variable `_Envsignum0`. This allows one to specify the value of `signum(0)`.

3.1.2 Basic principles

Many of the algorithm existing in the ALIAS-C++ library can be used directly from `Maple`. Their efficiency may be improved by using the simplification procedures described in the previous chapter.

All these procedures use the same mechanism:

- they generate the C++code specific for the problem at hand (using the ALIAS-Maple procedures `Make[FJH]`)
- they generate a main program that call specific algorithms of the ALIAS-C++ library
- the C++ code is compiled into an executable program
- the executable program is run and the result is written in a file
- the procedure read the result file and returns the result in Maple format

All these procedures allow for an optional last argument which is the name of a simplification procedure (see chapter 4).

There are numerous parameters that allows to change the behavior of the solving procedures: they are described in the section 3.5.

3.1.3 Bisection mode

All the solving procedures proceed by considering a box for the variables. Some calculation is performed with the current box (for example trying to reduce the width of the box). Then this box will be bisected. Hence the bisection process is a key element of the algorithms and `ALIAS` offers various bisection modes that are controlled through the variable `ALIAS/single_bisection`:

- 0: all the ranges of the n variables are bisected at their middle point. Hence 2^n new boxes are created after the bisection process. This is a mode that has to be avoided
- > 0: a single variable will be bisected. The choice of the bisected variable will depend on the the value of `‘ALIAS/single_bisection` variable: see the C++ manual for the possible values of this variable
- 20: you have defined your own bissection procedure. By default this procedure should be written in the file `BISSECTION_USER.C` which must include the procedure. Alternatively you may give the file name in the variable `‘ALIAS/bisection_user‘` which should be a string. .

  ```
  int Select_Best_Direction_Interval_User(int DimVar,int DimEq,
  INTERVAL_VECTOR (* TheIntervalFunction)(int,int,INTERVAL_VECTOR &),
  INTERVAL_VECTOR &Input)
  ```

  where `DimVar` is the size of the unknown vector, `DimEq` the number of equations, `TheIntervalFunction` the evaluation function of your problem (by default `F`)and `Input` the current box. This procedure shall return the variable number that will be bisected.
- a mix of these 2 modes in which m ranges among the n available will be bisected. `‘ALIAS/mixed_bisection`. gives the number of bisected ranges

3.1.4 Storage mode

The boxes created after a bisection are stored in a list for further processing. There are different ways to store the new boxes:

- immediately at the end of the list: this is a called the direct mode
- immediately in place of the current box and after it: this is called the reverse storage mode
- as a mix between these two modes: this is the mixed bisection mode

The reverse storage mode is the one that ensure that we will use the minimal amount of memory storage.

The storage mode is controlled through the variable `‘ALIAS/storage_mode`: 0 is used for the direct storage mode while reverse storage will be used when `‘ALIAS/storage_mode` is set to a number equal or larger than the number of unknowns+1. Mixed storage may also be used using the variable `‘ALIAS/mixed_storage` that indicates how many boxes are stored in place of the current box. For more details on the storage mode see the ALIAS-C++ manual.

3.1.5 Simplification procedures

For a specific problem the end-user may have some knowledge of the problem that implies that he may determine that for some intervals for the variables the problem has no solution, that the intervals for some variables may be reduced or that a solution may be found. Most of the ALIAS C++ solving programs have an optional argument that allows then end-user to provide this knowledge as a procedure, that will be called a *simplification procedure*.

Usually this procedure will take as input a set of intervals for the variables and will return an integer which is typically:

- -1 if there is no solution to the problem at hand for the given set of intervals
- 1 if the routine has allowed to reduce the width of the intervals of the set

- 0 if the routine has not changed the set of intervals
- > 1 if the procedure has provided a solution to the problem

As these simplification procedures are very important for solving efficiently most problems we will devote the specific chapter 4 for those used in system solving. More specific simplification procedures will be presented together with the solving algorithms.

3.2 General purpose system solving procedures

You may solve a system of equations and inequalities by calling the `ALIAS-Maple` procedures:

```
GeneralSolve  GradientSolve  HessianSolve
```

Most of the solving procedures may deal with inequalities constraints but in the list of functions you must first define the equations and then the inequalities. For using these solvers it is needed to create C++ programs that interval evaluate the equations and their derivatives in the appropriate format. By default these procedures will use `Make[FJH]` to create these programs but they offer the possibility of using user's pre-defined C++ programs.

The name of the executable created by ALIAS-Maple is `_GS_` but this may be changed by assigning another string to the variable `'ALIAS/name_executable'` or by setting the string `'ALIAS/ID'`.

The calculation done by these procedures may be stopped although the calculation has not been completed by setting the variable `'ALIAS/time_out'` to a floating point that indicates the maximum computation time in minutes. In that case the procedure will return a time out signal together with the solutions that have been obtained up to now. For example for a problem with 2 variables the message will be:

```
[[TIME_OUT],[[1.,1.],[2.,2.]],[[2.5,2.5],[3.,3.]]]
```

which indicates that 2 solutions have been found before the time-out occurs.

3.2.1 GeneralSolve

This is the most basic procedure for solving systems. It uses only the the interval evaluation of the expressions and can deal with equations involving intervals as coefficients, while the other solving procedure cannot be used in that case.

This procedure will return as solution a set of *boxes*: a box is defined as a set of intervals, one for each unknowns. Such box will be obtained as soon as either the width of all the intervals is lower than `'ALIAS/epsilon'` or the width of the interval evaluation of the expressions for the box is lower than `'ALIAS/fepsilon'`. To avoid returning a large number of solutions it is assumed that all the solutions of the system are at a distance at least larger than `'ALIAS/dist'`.

Hence the results provided by this procedure exhibit the following features:

- if `'ALIAS/dist'` is set to 0: all the solutions of the system will be included in one of the result boxes. But there may be some result boxes that do no include a solution of the system
- if `'ALIAS/dist'` is not set to 0: the number of result boxes will be in general lower than in the previous case but some solutions of the system may not be included in the result boxes and be only close to a solution box. There still may be some result boxes that do no include a solution of the system

For example:

```
with(ALIAS):
u:=GeneralSolve([x^2+y^2-1,x+y],[x,y],[[-2,2],[-2,1]]);
```

will provide in `u` an approximation of the 2 solutions of the system. $x^2 + y^2 - 1 = 0, x + y = 0$ for x, y included in the range [-2,2], [-2,1].

The efficiency of the calculation may be improved by using a simplification procedure, see chapter 4. Hence the following code will be faster:

```
with(ALIAS):
HullConsistency([x^2+y^2-1,x+y],[x,y],"Simp"):
u:=GeneralSolve([x^2+y^2-1,x+y],[x,y],[[-2,2],[-2,1]],"Simp");
```

Note that by default when an evaluation of all the equations is needed in the C++ program, the program will proceed by successive calls to the procedure created by `MakeF`. This behavior may be changed by setting `'ALIAS/equation_alone'` to 0: in that case all equations will be evaluated by a single call to the C++ evaluation procedure. This may be important in term of computation time if the evaluation of simplification terms is performed before doing the evaluation (e.g. if the procedure `ALIAS_FSIMPLIFY` has been defined, see the `MakeF` section 2.1.1).

Note that there is a specific bisection mode that can be used for this solving algorithm. The table `'ALIAS/table_ordered_bisection'`, with as many lines as equations, will contain variable number in a row such as for the first row [1,3,4]. This will indicate that only variables 1, 3, 4 will first be bisected until the first equation is satisfied. Then we will move to the second equation and the second row of the table. This bisection method is validated as soon as the flag `'ALIAS/ordered_bisection'` is set to 1.

3.2.2 `GradientSolve` and `HessianSolve`

The syntax of these procedures is the same as `GeneralSolve` but there are two main differences between these procedures and `GeneralSolve`:

- the derivatives of the expressions are used to improve their interval evaluation: hence the expressions should be at least C^1 for `GradientSolve` and C^2 for `HessianSolve`
- provided that the jacobian of the system is regular these algorithms are able to compute *exactly* the solutions of a system in the following sense: each of the solution boxes are guaranteed to contain one and only one solution and furthermore there is a numerical scheme that allows to determine this solution. If this is the case these procedures will return as result boxes for which each interval is reduced to a point (otherwise interval solutions are returned: this is a good way to determine if `ALIAS` has been able to calculate an exact solution). Furthermore the result provided by `HessianSolve` may be used as input for the `Newton` procedure that will allow to compute the solutions with an arbitrary accuracy, see section 9.8.

When `ALIAS` has returned a point M as solution it is always possible to retrieve the boxes in which it was found that there was a unique solution whose approximation is M: all interval solutions are available via the list `'ALIAS/Solutions'`.

In spite of the additional computation time involved by the use of the derivatives, these procedures are in general faster than `GeneralSolve`. Indeed the algorithms that are used in these procedures may allow to determine relatively large box in which an unique solution will be found, thereby avoiding a large number of bisection. As this knowledge is used to manage the list of boxes it may have a drastic effect on the computation time of the algorithm.

Note that it is possible to avoid using the Hessian (which may be computer intensive) for the evaluation of the expressions by setting the variable `'ALIAS/no_hessian'` to 1. The flag `'ALIAS/rand'` may be used to switch of bisection mode from time to time (it fixes the value of the ALIAS-C++ `ALIAS_RANDG` variable). The variables `'ALIAS/size_tranche_bisection'` and `'ALIAS/tranche_bisection'` may be used for the bisection mode 8 of `GradientSolve` (see the ALIAS-C++ manual).

For systems with m equations and n unknowns with $m > n$ the algorithm computes the exact solutions of the first n equations. Then if the interval evaluation of all the $m-n$ remaining equations has an absolute value lower than `'ALIAS/fepsilon'` the solution is supposed to be valid. For systems with $m < n$ see section 3.4.

3.3 Specific solving procedures

ALIAS-Maple provides solving procedures for systems having specific structures.

3.3.1 The `SolveSimplex` and `SolveSimplexGradient` procedures

This procedure allows to solve systems that have algebraic terms in at least at two equations. Each equation is transformed as the sum of a linear part and a non linear part. The linear part is substituted into a new variable that is submitted to a linear constraints. Then the simplex method is used to improve the range for the unknowns. The difference between the `SolveSimplex` and `SolveSimplexGradient` procedure is that in the second case the derivatives of the non linear parts of the equations are used to improve their interval evaluation and hence the linear constraints used by the simplex.

The following variables plays a role in these procedures:

- `'ALIAS/diam_simplex'`: the simplex will not be applied if the width of the current box is lower than this value (default value:0.1)
- `'ALIAS/full_simplex'`: if the value of this variable is n the simplex will be applied only for searching the minimum and maximum of the first $n-1$ variables when they are ordered by decreasing order of width. **This is probably the most important parameter to set**: a compromise has to be found between the gain of considering all variables and the computation time of the simplex
- `'ALIAS/init_simplex_linear'`: if some of the coefficients of the linear part of the expressions have an identical value you may assign this value to this variable so that a faster assignation of the coefficients will be performed (available only for the parallel version of these procedures). For example if a majority of the linear coefficients are 2 setting `'ALIAS/init_simplex_linear'` to 2 implies that the C++ procedure that calculate the linear coefficients will just compute the coefficients that are not equal to 2
- `'ALIAS/min_diam_simplex'`: the simplex will not be applied if the width of the current box is larger than this value (default value: 1.e7)
- `'ALIAS/min_improve_simplex'`: if the width of one range of the box has been improved by a value larger than this variable, then the simplex method will be repeated (default value: 0.1)
- `'ALIAS/no_simplex'`: if this parameter is set to 1 the simplex will not be used. This may be interesting for the parallel implementation where the computation involved by the simplex may be to computer intensive for the master
- `'ALIAS/simplex_expanded'`: for the simplex method using the gradient setting this flag to 1 indicates that the expression will be expanded with respect to the lower bound of each variable (i.e. the unknown x with lower bound $\underline{x}$ will be written as $x{=}X{+}\underline{x}$ and the simplex method will use the variable X)

3.3.2 Systems of distance equations

3.3.2.1 The `SolveDistance` procedure

The `SolveDistance` procedure allows to solve systems of distance equations. A distance equation describes that the distance between m points in a n-dimensional space is given. The unknowns are the n coordinates $x_1^k, \ldots, x_n^k$ of the points. Hence a distance equation may be written as:

$$\sum_{j=1}^{j=n}(x_j^k - a_j)^2 = l_k^2$$

$$\sum_{j=1}^{j=n}(x_j^k - x_j^l)^2 = l_{kl}^2$$

where a_j are numerical values. Furthermore the algorithm allows for the use of *virtual points*. The coordinates of a virtual point M are linear combination of the coordinates of k real points:

$$x_j^M = \sum_{l=1}^{l=k} b_l x_j^l$$

Hence a distance equation may also be written as:

$$\sum_{j=1}^{j=n}(x_j^M - x_j^l)^2 = l_{kl}^2$$

The syntax of `SolveDistance` is:

```
SolveDistance(Func,Vars,Init)
```

The initial search domain for `SolveDistance` may be determined using the `Bound_Distance` procedure.

3.3.2.2 Specificity of the procedure

Note that there is no need to use the `HullConsistency` simplification procedure when using `SolveDistance` as it is already embedded in the C++ method. The consistency method in the algorithm updates the value of the variables and starts again the update if the change in at least one interval exceed the threshold `'ALIAS/seuil2B'` (which has the default value 0.1).

As the `SolveDistance` algorithm uses systematically a Newton scheme with as estimate of the solution the center of the box, it may be interesting to switch the current box with the largest one in the list of boxes to process in order to find new solutions to the system very quickly. Indeed if the Newton scheme appears to converge toward an approximate solution S we will use a special version of the Kantorovitch test to determine a box B centered at S that contains only one solution. This box will be further enlarged by using a specific version of the Neumaier test (this is called an *inflation* of the box). Then we will determine if B has an intersection with each box in the list of boxes to process and if this is the case we will modify the list of boxes so that it has only boxes that are the complementary of B: this process will speed up the algorithm.

Switching the current box with the largest one is done by setting the flag `'ALIAS/permute'` to the number of bisection after which the boxes will be permuted. The default value for this flag is 1000 and if it is set to 0 no permutation will be done.

Note that if you have a system of distance equations with one parameter, you may fix the value of this parameter to a given value a and then follow the solutions when the parameter changes in the range $[a, b]$ by using a specific continuation procedure (see section 7.3)

3.3.3 Linear algebra

3.3.3.1 Enclosure of an interval linear system

Let consider the family of linear systems defined by the matrix equality

$$A(Y).X = B(Y)$$

where Y is a set of unknowns, A a square $n \times n$ matrix whose elements are functions of Y and B a n dimensional vector whose elements are also functions of Y. The *enclosure* of the set of solutions of this family of linear systems is a box that includes the solution of all linear systems in the family.

The enclosure can be computed using the `LinearBound` procedure whose syntax is

```
LinearBound(A,B,Derivative,Vars,Init)
```

where:

- `A` is a square array whose elements are function of the unknowns in `Vars`
- `B` is an array whose elements are function of the unknowns in `Vars`
- `Derivative` is an integer. If set to 0 the procedure uses the classical interval Gaussian elimination scheme to calculate the enclosure. If set to 1 it will use an `ALIAS` specific version of the Gaussian elimination scheme that uses the derivatives of the elements of `A, B` to improve the enclosure calculation
- `Vars`: a list of unknowns names

- `Init`: a list of ranges for the unknowns

A typical example is:

```
with(ALIAS):
with(linalg):

A:=array([[x,y],[x,x]]):
B:=array([x,y]):
VAR:=[x,y]:
LinearBound(A,B,0,[x,y],[[3,4],[1,2]]);
LinearBound(A,B,1,[x,y],[[3,4],[1,2]]);
```

which returns the enclosure [[.76923076923077, 10], [-13, -.076923076923077]] if `Derivative` is set to 0 and [[.91666666666667, 2.6666666666667], [-2, -.66666666666667]] if `Derivative` is set to 1 (note that these values are computer dependent), the exact answer being [[1.25,1.66666],[-1]].

3.3.3.2 Regularity of parametric matrices

We consider a matrix whose coefficients are functions of a set of variables. The procedure `RegularMatrix` allows one to determine if the set of matrices includes only regular matrices. Its syntax is

```
RegularMatrix(mat,vars,init,cond,typedet)
```

where

- `mat`: the matrix definition (here called A)
- `vars`: the set of variables
- `init`: a list of ranges, one for each variable
- `cond`: an integer that indicates if matrix conditioning is used. It is 0 if no conditioning is used, 1 if the conditioning KA is used, 2 if AK is used and 3 if both conditioning are used. The matrix K used here is Inverse(A(Mid(vars))).
- `typedet`: a list of 3 integers. This integer indicates which determinant calculation procedure is used. A value lower than 2 indicates the use of `Fast_Determinant`, a value of 2 the use of `Medium_Determinant` and a value of 3 `Slow_Determinant`. A value of -1 indicates that the user has provided its own procedure to compute the determinant: the procedure name is `‘ALIAS/user_determinant_matrix‘` and is written in the file `‘ALIAS/user_determinant_matrix‘`.C. It is also possible to use specific procedures for computing the determinant of the left and right conditioned matrices by using the procedure `‘ALIAS/user_determinant_cond_left‘` and `‘ALIAS/user_determinant_cond_right‘`.

The procedure returns 1 if all matrices are regular, -1 if the set includes a singular matrix and 0 if the algorithm has failed.

Note that the procedure generates the C++ code for calculating the elements of the matrix by using the procedure `MakeF`. If these elements includes several times the same complex expressions it is advised to use the mechanism described in `MakeF` (section 2.1.1) to interval evaluate these expressions only once.

Note also that the used bisection method is to bisect the unknown range having the largest width. Scaling the unknown is therefore important

Note a special case that may occur if the matrix may include interval coefficients apart of the parameters or if the matrix is badly numerically conditioned. In that case it may happen that for a specific value of the parameters the sign of the determinant of the matrix cannot be ascertained (such point will be called unsafe). Hence at this point we cannot state the regularity of the matrix. But it may happen that at 2 other points the determinants will have opposite sign indicating the presence of a singularity. So the following cases may occur:

1. at point P1 the determinant has not a constant sign while at points P2, P3 the signs are opposite

2. at a set of points the determinant has not a constant sign while at all other points the determinant have a constant sign

Hence the output of the algorithm may be -1 in the first case but cannot be neither -1 nor 1 in the second one.

The procedure may behave in 2 different ways:

1. stop as soon as an unsafe point has been found

2. continue if an unsafe point is detected until either a case 1 is detected or until only unsafe points remains

The procedure behavior is controlled through the flag `'ALIAS/unsafe_det'`. If this flag is set to 1 behavior 1 will be used while behavior 2 is obtained by setting the flag to 0. The default value of this flag is 0. For behavior 1 the return code of the procedure is -3 if an unsafe point is found.

The procedure reorders the boxes every 'ALIAS/rand' iteration (default value=0 i.e. at each iteration). The ordering may be controlled through the 'ALIAS/order' variable. Assume that the interval evaluation for a box is [a,b]. At some point the procedure will have find a box with a constant sign s for the determinant:

- if the order is 0 the box are sorted by decreasing order of b+a is s> 0 or -(b+a) if s< 0
- otherwise the box are sorted by decreasing order of b/(b-a) if s> 0 or -a/(b-a) if s< 0

The default value for 'ALIAS/order' is 0.

The conditioning may play an important role especially as the conditioned matrix is calculated symbolically. Assume for example that the first row of A is `[x x]` and that AK is used. If the first column of K is `[a1 a2]`, then the first element of the conditioned matrix is `a1x+a2x`, that the procedure will arrange as `x(a1+a2)`, thereby leading to an optimal form in term of interval evaluation, which is better than the numerical conditioning.

The procedure accepts an optional sixth argument which is a string such as `SIMP` that defines a *matrix simplification procedure*. The syntax of such procedure is

```
SIMP(int dimA,INTERVAL_MATRIX & A)
```

where `dimA` is the dimension of matrix A. This procedure shall return -1 if all the matrices in the set A are regular, 0 otherwise. The C++ program corresponding to this procedure is written in the file `SIMP.C`. Such program may be obtained for example by using the `RohnConsistency` or `SpectralRadiusConsistency` procedures.

A parallel version of this procedure is `ParallelRegularMatrix`.

3.3.3.3 The `RohnConsistency` procedure

This procedure creates a matrix simplification program to help testing the regularity of a set of matrices. Its syntax is:

```
RohnConsistency(name)
```

where `name` indicates the name of the procedure, that will be written in the file `name.C`. This simplification procedure uses Rohn regularity test that is based on the calculation of 2^{2n-1} determinants of scalar matrices (where n is the dimension of the matrices). It shall therefore not be used with very large matrices.

Note that if the flag `Use_Simp_Cond` is set to 0 this test will not be used if the C++ flag `Simp_In_Cond` is not equal to 0. For example in `RegularMatrix` this is the case if the matrix received by Rohn is a conditioned matrix.

A remembering mechanism may allow to reduce the computation time by storing information on already processed matrix in the Rohn test. For using this mechanism you must set `'ALIAS/rohn_remember'` to the number of matrix that will be stored and `'ALIAS/rohn_size_matrix'` to the dimension of the matrix.

3.3.3.4 The `SpectralRadiusConsistency` procedure

This procedure creates a matrix simplification program to help testing the regularity of a set of matrices. Its syntax is:

```
SpectralRadiusConsistency(name,eps,iter)
```

where `name` indicates the name of the procedure, that will be written in the file `name.C`. The floating point `eps` is used to compute a safe upper bound of the spectral radius: usually a value of 10^{-6} is a good value. The integer `iter` indicates the maximal number of iteration that the algorithm is allowed to perform.

Note that if the flag `Use_Simp_Cond` is set to 0 this test will not be used if the C++ flag `Simp_In_Cond` is not equal to 0. For example in `RegularMatrix` this is the case if the matrix received by the procedure is a conditioned matrix.

3.3.3.5 The `LinearMatrixConsistency` procedure

This procedure should be used in conjunction with `RegularMatrix`. It should be used only if the matrix includes rows or columns in which a variable appears with degree 1 in at least 2 elements of a row or a column. Its syntax is:

```
LinearMatrixConsistency(A,VAR,vars,row,context,rohn,
                        Func,FuncKA,FuncAK,name)
```

The parameters are:

- `A`: the considered matrix
- `VAR`: a list of the variable name that appears in the matrix
- `vars`: a list of variable names that appears linearly in the row or column of `A`
- `row`: 1 if the procedure look at the row of `A`, 2 for the column
- `context`: see description
- `rohn`: 1 if the Rohn consistency procedure is used in the program
- `Func, FuncKA, FuncAK`: the name of C++ procedures to compute respectively the matrix, the left conditioned matrix and the right conditioned matrix. If this procedure is used in conjunction with `RegularMatrix` these names are we have `Func="F"`, `FuncKA="AKA"`, `FuncAK="AAK"`.
- `name`: the name of the C++ procedure that will be created in the file `name`.C

Linearity has to be understood in a very loose sense. For example if a row of the matrix is

```
x    xy    x+y
```

this row may be considered as linear in `x` in which case we have 3 linear elements in the row. If the row is considered as linear in `y` we have 2 linear elements in the row. Note that the linearity is checked according to the order provided in `Vars`.

This procedure may be used for the matrix `A`, the conditioned matrix `KA` or the conditioned matrix `AK`. Context is used to indicate the choice according to the following code:

- 0: for `A` only
- 1: for `KA` only
- 2: for `AK` only
- 3: for `A, KA, KA`
- 4: for `AK, KA`
- 5: for `A, KA`
- 6: for `A, AK`

The complexity of this procedure is approximately $\prod(2_i^m)$ where m_i is is the number of linear terms in the elements of row or column i of `A`. Hence it should not be be used with a large `Vars`. The use of Rohn matrix may also be expensive as it requires to calculate $2^(2n-1)$ scalar determinant, with n the dimension of the matrix `A`.

For example we may use

```
LinearMatrixConsistency(A,[x,y],[x],1,5,"F","AKA","","Simp"):
```

In this example we will consider the row of matrix `A` and its elements that are linear in `x`. The procedure will be used for both the matrix `A` and the conditioned matrix `KA` (hence this procedure should be used with the parameter `cond` of `RegularMAtrix` set to 1 or 3. Note that we assume here that the conditioning matrix is $\mathtt{A}^{-1}(Mid(X))$.

3.3.3.6 The `GerschgorinConsistency` procedure

Any `ALIAS` maple procedure involved in the calculation on eigenvalues of a matrix A may use the Gerschgorin circles methods that states that all the eigenvalues of a matrix are enclosed in the union of a set of circles (in the complex plane) whose center and radii are calculated as functions of the coefficients of the matrix.

The purpose of the `GerschgorinConsistency` procedure is to generate the C++ code for a simplification procedure that may be used by the ALIAS-Maple procedures doing calculation on the eigenvalues of a square matrix. The syntax is:

```
GerschgorinConsistency(Func,Vars,Gradient,n,procname)
```

where:

- `Func`: list of constraints (equation or inequality), the last element must be the matrix.
- `Vars`: list of parameters name including as first element an auxiliary name that will be the name of the unknown in the characteristic polynomial name
- `Gradient`: a flag that indicates if the derivatives of the matrix coefficients with respect to the unknowns may be used (1) or not (0)
- `n`: an integer, the order of the method, see below
- `procname`: the name of the simplification procedure. The name of the created file will be procname.C

Here we try to improve the bounds given by the Gerschgorin method using the fact that for any diagonal matrix D with positive components the eigenvalues of DAD^{-1} are the same than the eigenvalues of A. The method is not able to determine D such that the Gerschgorin circles are minimal (i.e. give the best bounds) and only try a set of at most n^2 different D where n is the order of the method.

To check only if the eigenvalues lie in the interval provided by the C++ procedure without modifying this interval use a negative n

3.4 Non 0-dimensional system

Although the solving procedures of `ALIAS` are mostly devoted to be used for 0 dimensional system (i.e. systems having a finite number of solutions) most of them can still be used for non 0-dimensional system. Specific procedures for systems of dimension 1 are described in chapter 7. For larger dimension systems the solving procedures of ALIAS-Maple will provide an approximation of the result as a set of input intervals that are written in a file. To deal with non-0 dimensional system it is necessary to set the flag `‘ALIAS/ND‘` to 1 and to set the name of the file in the variable `‘ALIAS/ND_file‘` (its default value is `.resultND`.

The total volume of the boxes written in the file may be obtained through the variable `‘ALIAS/VolumeIn‘`. During the process we call *neglected boxes* the boxes that cannot be eliminated as not containing solutions of the problem but can neither be discarded as part of the boxes may contain a solution. A box will be neglected if its width is lower than `‘ALIAS/epsilon‘`. The total volume of the neglected boxes may be obtained through the variable `‘ALIAS/VolumeNeglected‘`.

The result (or cross-sections of it if the problem has more than 3 variables) may be visualized using the `DrawND` procedure described in section 9.10.

3.5 Parameters for the solving procedures

The behavior of the solving algorithms may be adjusted using various parameters that are presented in the on-line help (see `ALIAS[Parameter]`): basically all the parameters that may be adjusted in the C++ library may be also adjusted via `Maple`.

3.5.1 General parameters for the solving procedures

We indicate here parameters that appear in most procedures, their meaning, their default value and the corresponding name in the C++ library.

Parameter name	Meaning	C++ equivalent
`'ALIAS/debug'`	flag for debug purpose	`Debug_Level_Solve_General_Interval`
`'ALIAS/dist'`	minimal distance between 2 distinct solutions (for `GeneralSolve`)	
`'ALIAS/epsilon'`	maximal width of a solution box	
`'ALIAS/fepsilon'`	maximal width of the expressions evaluation for a solution box	
`'ALIAS/lib'`	string that indicates the location of the ALIAS C++ library	
`'ALIAS/maxbox'`	maximal number of stored boxes	
`'ALIAS/maxsol'`	maximal number of solutions	
`'ALIAS/rand'`	allow to switch between bisection modes	`ALIAS_RANDG`
`'ALIAS/rand'`	exchange largest box with current every xx iter.	`ALIAS_RANDG`
`'ALIAS/mixed_bisection'`	bisection mode, see section 3.1.3	`ALIAS_Mixed_Bisection`
`'ALIAS/mixed_storage'`	allows to switch between direct and reverse storage	`Switch_Reverse_Storage`
`'ALIAS/optimized'`	flag to indicate if the C++ files are compiled with -O	
`'ALIAS/order'`	ordering for the storage of the boxes, see the ALIAS-C++ manual	
`'ALIAS/profil'`	string that indicates the location of the BIAS/Profil library	
`'ALIAS/single_bisection'`	bisection mode, see section 3.1.3	`Single_Bisection`
`'ALIAS/storage_mode'`	bisection mode, see section 3.1.4	`Reverse_Storage`
`'ALIAS/name_executable'`	name of the executable	
`'ALIAS/allows_n_new_boxes'`	maximum number of boxes that can be created when intersecting a box and a unicity box	`ALIAS_Allows_N_New_Boxes`
`'ALIAS/type_n_new_boxes'`	if the maximum number of boxes that is created when intersecting a box and a unicity box is lower than the maximum determine the rule to create the new boxes (see ALIAS-C++)	
`'ALIAS/tranche_bisection'`	for bisection mode 8	`ALIAS_Tranche_Bisection`
`'ALIAS/size_tranche_bisection'`	for bisection mode 8	`ALIAS_Size_Tranche_Bisection`

The variable `'ALIAS/stop_first_sol'` allows to exit from a solving procedure without completing the full calculation. The following behavior will be obtained according to the value of this flag:

- 0: (default value) the procedure complete the whole solving process
- 1: the procedure exit as soon as one solution has been found
- 2: the procedure exit as soon as the maximal number of solution has been found

For the expression involving the determinant of a matrix we have some specific variables:

Parameter name	Meaning	C++ equivalent
`'ALIAS/det22'`	see section 2.1.3	
`'ALIAS/minor22'`	see section 2.1.3	
`'ALIAS/row22'`	see section 2.1.3	

3.5.2 Parameters for the procedures using the derivatives

- ‘ALIAS/maxgradient‘: the maximal width of a box for using the gradient for the evaluation of the expression (default value: 1.e10). (C++:ALIAS_Diam_Max_Gradient)
- ‘ALIAS/maxkraw‘: the maximal width of a box before using the Krawczyk test (default value: 1.e10) (C++: ALIAS_Diam_Max_Kraw)
- ‘ALIAS/maxnewton‘: maximal width of a box before using the interval Newton method (default value: 1.e10) (C++: ALIAS_Diam_Max_Newton)
- ‘ALIAS/newton_max_dim‘: for the numerical C++ Newton scheme the iteration is stopped if the absolute value of the i-th parameter is greater than the i-th of this table
- ‘ALIAS/store_gradient‘: the signs of the derivatives for each box processed by the algorithm are usually stored. Setting this variable to 0 allows to decrease the size of the storage memory
- ‘ALIAS/transmit_gradient‘: the master program will usually transmit to the slaves the sign of the elements of the jacobian for the box that is send to the slave. This may be avoided by setting this flag to 0
- ‘ALIAS/use_inflation‘,‘ALIAS/eps_inflation‘: as soon as a solution is found we will try to inflate the box in which the solution has been found by at least ‘ALIAS/eps_inflation‘. This process may be computer intensive and may be invalidated by setting ‘ALIAS/use_inflation‘ to 0
- ‘ALIAS/type_n_new_boxes‘, ‘ALIAS/allows_n_new_boxes‘: if a unicity box has been discovered solutions may be sought in the following boxes only in the complementary part of the box with respect to the unicity box: this is allowed by setting the flag type_n_new_boxes to 1. This may create a large number of boxes and the flag allows_n_new_boxes allows to specify how many new boxes may be created
- ‘ALIAS/Grad_Equation‘: an integer array used by the HessianSolve procedure. If the i-th element of this array is 0 the derivatives of the i-th equation are not used for the interval evaluation of the equation
- ‘ALIAS/apply_kanto‘: an integer used by the HessianSolve procedure. If set to 2 the algorithm will always perform ‘ALIAS/newton_iteration‘ (default value: 100) iterations of a Newton scheme with as initial guess of the solution the center of the current box. If the scheme converge it is first verified that a solution has indeed been found and if this solution lie inside the initial search domain. The box in which a solution is found is then inflated and the search domain is reduced by the solution box. Using this method allows to often determine quickly solutions of a system.

3.6 Generating program without running it

You may also directly obtain a full program for one of the three main solving procedures of ALIAS by setting the variable ‘ALIAS/runit‘ to 0.

You may then compile the created program using the makefile _makefile. These procedures may be used with inequality constraints. Alternatively you may run Maple with a solving program embedded in the Maple code and stop the session when the compilation starts. According to the algorithm you are using the name of the main program will start with _G and has the extension .C (alternatively you may define the name of the main program using the variable ‘ALIAS/name_executable‘) Alternatively you may define a string in ‘ALIAS/ID‘ and all files that will be created during the solving process will have this string appended to their name.

Chapter 4

Simplification procedures

4.1 Introduction

For a specific problem the end-user may have some knowledge of the problem that implies that he may determine that for some intervals for the variables the problem has no solution or that the intervals for some variables may be reduced.

ALIAS-Maple allows one to introduce such knowledge in the generated C++ code. The mechanism that is used is to allow one to specify a C++ routine, that will be called a *simplification procedure*. We will present in this chapter simplification procedures devoted to system solving. A simplification procedure takes as input a set of intervals for the variables and will return an integer:

- -1 if there is no solution to the problem at hand for the given set of intervals
- 1 if the routine has allowed to reduce the width of the intervals of the set
- 0 if the routine has not changed the set of intervals
- 11 if the simplification procedure provides a root that may be found with the Newton scheme (see for example `TryNewton`)

Procedures returning -1,0 or 1 will be called *filtering procedures* while procedures returning 11 are called *root procedures*.

This routine will be called by all the solving procedure of ALIAS-Maple whenever a new set of intervals is considered in the algorithm i.e. right after the bisection process (see the ALIAS-C++ manual). System solving procedures in ALIAS C++ are designed to discard box for which a simplification procedure returns -1 and to update it if the simplification procedure returns 1.

The syntax of such a routine `Simp` is:

```
int Simp(INTERVAL_VECTOR &v_IS)
{
....
return 0;
}
```

Alternatively ALIAS-Maple allows to automatically produce simplification procedures according to classical methods used in interval analysis.

Note that numerical round-off errors in Maple may produce solving errors as the filtering procedures uses frequently an expanded version of the expressions. To reduce the impact of this problem it is possible to adjust the number of digits that will be used when doing numerical computation in the filtering procedure by using the `ALIAS` variable `‘ALIAS/digits‘` which is set by default to 40.

4.2 Filtering simplification procedures

An important point: some of these procedures have as argument a list of variable names and this list must be the same than the one that is used for the solving procedure.

4.2.1 The `HullConsistency`, `Simp2B` and `HullIConsistency` procedures

The idea underlying this simplification procedure (known in the community of constraint programming as the hull-consistency or 2B-consistency) is to rewrite the equation and check if it is consistent at each time. For example imagine that one of the equation is $x^2 - 2x + 1 = 0$. The procedure will introduce a new variable X such that $X = 2x - 1$ and compute its interval evaluation. As X should be equal to x^2 if X has a negative upper bound the simplification procedure will return -1, which indicate to the solving algorithm that it may discard the current input interval. If the upper bound U of X is positive the procedure will check if x lie in $[-\sqrt{U}, \sqrt{U}]$ and update the interval for x if this is not the case.

In a second step the procedure will introduce another new variable Y such that $Y = (x^2+1)/2$ and compute its interval evaluation. As Y should be equal to x the procedure will check if the interval value for Y is consistent with the interval value for x and if not update the interval for x accordingly.

The `HullConsistency` and `HullIConsistency` procedures of ALIAS-Maple partly implements this principle respectively for equalities and inequalities. They may be used if any of the equation has linear or square terms in the unknown (e.g. $x * y, x^2 * y$). The procedure will isolate the linear and square terms of each unknown and produce the C++ code that will check is their interval evaluation is consistent.

The syntax of the `HullConsistency` procedure is:

```
HullConsistency([x^2+y^2-1,x*y-1],[x,y],"Simp");
```

The first argument is a list of equations, the second a list of variable names and the third argument is the name of a file. In the above example the procedure will create the C++ program `Simp.C` that may be later be given to a solving procedure.

Until version 2.4 was a `HullConsistencyStrong` procedure available, that has been contributed by G. Chabert. The procedure was intended to deal with terms that were different from linear and square terms that are managed by `HullConsistency`. Its drawback was that it was based on a semantic tree of the expression and was not able to decrease the dependency problem. For example if the expression is

$$x^2 + x * y + x$$

and the final node is x^2, then the 2B will be implemented using

$$x^2 = -x * y - x$$

while `HullConsistency` will use

$$x^2 = -x * (y - 1)$$

which is more efficient. For example in the equation

$$\sin(x_1 + x_2)\cos(x_1^2) * x_2 + x_1 = 0$$

we will use

$$\sin(x_1 + x_2) = \frac{-x_1}{\cos(x_1^2) * x_2} \quad x_2 = \frac{-x_1}{\sin(x_1 + x_2) * \cos(x_1^2)} \quad x_1 = -\sin(x_1 + x_2)\cos(x_1^2) * x_2$$

to update successively the range for $(x_1, x_2), (x_2), (x_1)$. Note that there is not an extensive check of the possibility of interval evaluating the right hand side term of these new equations and thus the simplification procedure may lead to an error during this evaluation. The syntax of `Simp2B` is similar to the one of `Hullconsistency`.

The new procedure `Simp2B` manages in a better way the dependency problem but is less complete than `HullConsistencyStrong`, which is no more available for Maple 9.5, as it manages only the following expressions

$$\sin, \cos, \tan, \log, exp, x^n (n \textit{ being an integer})$$

It is also usually more computer intensive than the `HullConsistencyStrong` procedure. Note that you have some control over the simplification rule that will be used. Consider for example the equation

$$x^2 + e^z y + 1 = 0$$

By default `Simp2B` will use the following equalities: $x^2 = -1 - e^z y$, $e^z = (-1 - x^2)/y$, $y = (-1 - x^2)/e^z$ to reduce the range respectively for x, z, y. But if you specify in the list `ALIAS_FORBIDDEN_TERMS`:

```
ALIAS_FORBIDDEN_TERM:=[e^z]:
```

then only the first and third rule will be used.

The syntax of the `HullIConsistency` procedure is:

```
HullIConsistency([x^2+y^2-1<=0,x*y-1<=0],[x,y],0,"Simp");
```

Here the third argument is an integer which should be either 0 or 1. It 1 it indicates that the derivatives of the left side of the inequalities are used for their interval evaluation.

There is an optional last argument to these procedures which is a floating point number f. If during the simplification there is change in the interval for a variable such that the width of the new range is lower by more than f from the width of the initial range, then the simplification process will be repeated until either the procedure detect that there is no solution to the system or that no change occur in the variable or that the amplitude of the change is lower than f.

If you have numerous equations that have such terms the C++ procedure may be quite large. To reduce this size you may choose to look only at the linear term or only at the square term by setting the flag `‘ALIAS/hullC‘` to 1 or 2 (its default value is 0). Note that this option is not valid for the `HullConsistencyStrong` and `HullIConsistency` procedures.

A typical use of this procedure is given below, assuming that you have a list of equations `EQ` and a list of variable `VAR`:

```
HullConsistency(EQ,VAR,"SS"):
HessianSolve(EQ,VAR,Init,"SS");
```

Here the procedure generate the C++ procedure `SS` (which will be written in the file `SS.C`) and the solving algorithm `HessianSolve` will use this procedure during the computation.

The efficiency of the hull consistency approach is usually quite good but decreases quite quickly after the first improvements. Hence it is in general not a good idea to repeat the procedure when only small improvements on the range are obtained.

A test to determine if an expression involved in the equations and inequalities can be interval-valuated may be performed if the `Verify_Problem_Expression` procedure has been called before the call to `HullConsistency` or `HullIConsistency` (see section 2.1.5). If different calls to the consistency procedures are made with different set of expressions it is necessary to specify a different string ‘ALIAS/ID‘ string before each call to `Verify_Problem_Expression`. For example

```
‘ALIAS/ID‘:="Simp1":
Verify_Problem_Expression(EQ1,VAR):
HullIConsistency(EQ1,VAR,"Simp1"):
‘ALIAS/ID‘:="Simp2":
Verify_Problem_Expression(EQ2,VAR):
HullConsistency(EQ2,VAR,"Simp2"):
‘ALIAS/ID‘:="Simp3":
Verify_Problem_Expression(EQ2,VAR):
HullConsistencyStrong(EQ3,VAR,"Simp3"):
‘ALIAS/ID‘:="other":
```

is valid.

A special key-word may be used for the procedures `HullConsistency` and `HullIConsistency` in the case of an optimization problem. In that case the ALIAS-C++ variable `ALIAS_Optimize` is set to -1 (minimum

problem) or 1 (maximum problem), the flag `ALIAS_Has_Optimum` is set to 1 as soon as an estimation of the optimum is found while the interval `ALIAS_Optimum` is set to the estimation of the optimum. If an equation or an inequality has the key-word "Optimum" in it and if `ALIAS_Has_Optimum` is 1, then "Optimum" will be substituted by `Sup(ALIAS_Optimum)` (minimum problem) or `Inf(ALIAS_Optimum)` (maximum problem).

For example assume that you deal with a problem involving the minimum of $x\sin(x) + y\cos(y)$: you may then use the following call

```
HullIConsistency([x*sin(x)+y*cos(y)-Optimum<=0],[x,y],0,"Simp");
```

to create a simplification procedure that may reduce the current box. See another example in section 5.2

4.2.2 The `HullConsistencyTaylor` procedure

This procedure uses a third order Taylor extension of an equation and an interval evaluation of the remainder. This leads to a second order equation in each of the variables, whose coefficients is computed. The analytic solution of this second order equation is used to improve the variable range. This filter is very efficient for non algebraic equations and if the width of the range for the variable is not too large.

The syntax is:

```
HullConsistencyTaylor(func,vars,procname)
```

An optional fourth argument may be used. It consists in a list of one or two number. If this argument is [0.1,0.01], then the filter will be used only if the maximal width for the ranges is less than 0.1 and the filter will be used again if the gain in a range if larger than 0.01. If only one number is given in the list, then it will indicate the maximal width for the ranges.

4.2.3 The `BiCenteredForm` procedure

This procedure is based on the interval evaluation of equations using centered form with as centers two possible centers which may be optimal (see the `ALIAS` C++ manual for the calculation of these centers). The centered form uses the fact the interval evaluation of a function f in the n unknown $X = \{x_1, \ldots, x_n\}$ is included in $f(c) + J(X)(X - c)$ where c is point called the center. As mentioned earlier two possible centers are calculated. Furthermore the calculation of the product $J(X)(X - c)$ may involve several occurrences of the same variables, that are not detected when computing the product numerically. This procedure allows, as an option, to compute symbolically the product and to re-arrange it in order to try to reduce multiple occurrences.

The syntax of this procedure is:

```
BiCenteredForm(func,funcproc,Jfuncproc,vars,custom,procname)
```

where

- `func`: a list of equations
- `funcproc`: the name of a procedure in `MakeF` format that computes the interval evaluation of `func`
- `Jfuncproc` :the name of a procedure in `MakeJ` format that computes the interval evaluation of the derivatives of `func`
- `vars`: a list of variable names
- `custom`: a string that indicate how `funcproc` and `Jfuncproc` are obtained.
  - `"none"`: the procedure will assume that funcproc and Jfuncproc are the procedures `F` and `J` that are generated by `GradientSolve` and `HessianSolve`
  - `"F"`: the procedure will assume that `funcproc` is a customized procedure that, for example, has been generated by `MakeF`
  - `"J"`: the procedure will assume that `Jfuncproc` is a customized procedure that, for example, has been generated by `MakeJ`

  - `"FJ"`: the procedure will assume that `funcproc` and `Jfuncproc` are customized procedures

- `procname`: the name of the simplification procedure that will be created in the file `procname.C`

In addition the procedure admits a 7th optional argument `"prod"`. In that case the procedure will compute symbolically the product $J(X)(X-c)$ and re-arrange its terms. Both the numerical form and the one resulting from the symbolic calculation will be used in the simplification procedure.

4.2.4 The `GlobalConsistencyTaylor` procedure

This procedure uses a second order Taylor extension of an equation and an interval evaluation of the remainder. The equation system is transformed into a linear part and an interval remainder and an interval linear system solver is used to determine the bounds for the solutions. Hence this filter is *global* (i.e. it takes into account all equations of a system) while most of the other filter are *local* (i.e. they deal only with one equation at a time). This filter is usually very efficient for non algebraic equations and if the width of the range for the variable is not too large.

The syntax is:

```
GlobalConsistencyTaylor(func,vars,procname)
```

An optional fourth argument may be used. It consists in a list of one or two number. If this argument is [0.1,0.01], then the filter will be used only if the maximal width for the ranges is less than 0.1 and the filter will be used again if the gain in a range if larger than 0.01. If only one number is given in the list, then it will indicate the maximal width for the ranges.

4.2.5 The `SimpAngle` procedure

This procedure may be used if one of the equation indicates that the scalar product of two vectors must have some given value. Hence such equation may be written as:

$$\sum (x_i - x_j)(x_k - x_m) = 0$$

where the x_i, x_k are unknowns and the x_j, x_m unknowns or constants.

A typical use of this procedure is given below, assuming that you have a list of equations `EQ` and a list of variable `VAR`:

```
eq1:=(x1-3)*(x2-4)+(y1-1)*(y2-4)-10:
SimpAngle([eq1],VAR,"SS"):
GeneralSolve(EQ,VAR,Init,"SS"):
```

Here the procedure generate the C++ procedure `SS` (which will be written in the file `SS.C`) and the solving algorithm `GeneralSolve` will use this procedure during the computation.

There is an optional last argument to this procedure which is a floating point number f. If during the simplification there is change in the interval for a variable such that the width of the new range is lower by more than f from the width of the initial range, then the simplification process will be repeated until either the procedure detect that there is no solution to the system or that no change occur in the variable or that the amplitude of the change is lower than f.

4.2.6 The `SimplexConsistency` procedure

This simplification procedure may be used to solve systems of equations as soon as the system has at least two equations with polynomial terms in at least one variable. The principle used for this simplification procedure is explained in the ALIAS-C++ manual section of `Solve_Simplex`.

The syntax of this procedure is:

```
\SimplexConsistency(EQ,VAR,"Simp"):
```

where `EQ` is a list of equation, `VAR` a list of variable name and `Simp` is the name of the simplification procedure that will be written in the file `Simp.C`.

The behavior of this simplification procedure may be modified by using the following variables:

- `'ALIAS/full_simplex'`: an integer n. The simplification procedure will try to improve the width of the n largest variable ranges. If n is -1 (the default value for this variable) the procedure will just return -1 if there is no solution to the system but no improvement on the range of the variable will be obtained. n may be larger than the number of variables.
- `'ALIAS/diam_simplex'`: a floating point number f (default value: 0.1). The procedure will not improve the range of variable whose width is lower than f
- `'ALIAS/min_diam_simplex'`: a floating point number f (default value: 1e7). The procedure will not try to improve the range of variables whose width is greater than f
- `'ALIAS/mid_simplex'`: by default the procedure uses a linearization at the lower bound of the variables. If you set this flag to 1, then the linearization point will be the mid point of the range

Two remarks about this procedure:

1. it is usually very effective
2. it may be computer intensive for large systems: hence its use should in general be avoided within the 3B method (see section 4.5)
3. it is not numerically safe with the current implementation of the simplex i.e. in some cases it may lead to miss some solutions

4.2.7 The `IntervalNewton` procedure

This procedure allows one to take into account the specific structure of the system to solve when using the interval Newton method. In this method the classical algorithm, for example used by default in `GradientSolve` or `HessianSolve`, computes numerically for a box X the interval evaluation of $M = J^{-1}(X_m)J(X)$ where X_m is the mid-point of X. This numerical calculation does not allow to to take into account the dependency between the elements of J when computing M. This procedure compute symbolically the matrix M and re-arrange its elements (with `MinimalCout`) in order to reduce the dependency problem.

The syntax of this procedure is

```
IntervalNewton(func,nfunc,funcproc,njfunc,Jfuncproc,typemid,grad,grad3B,incl,vars,procname)
```

where

- `func`: the list of equations,
- `nfunc`: the number of equations that will be evaluated by `funcproc`. It may not be the same than the number of equations in `func`
- `funcproc`: the name of the procedure that is used to evaluate the equations. It may be the name of a procedure designed by the end-user with `MakeF` or the name of the procedure that will be used by the solving algorithm (usually `"F"`)
- `njfunc`: the number of equations that will be used when computing the derivatives. It may not be the same than the number of equations in `func`
- `Jfuncproc`: the name of the procedure that is used to evaluate the derivative of the equations. It may be the name of a procedure designed by the end-user with `MakeJ` or the name of the procedure that will be used by the solving algorithm (usually `"J"`)
- `typemid`: the interval Newton method uses a conditioning matrix. This flag will indicate which type of conditioning matrix is used. If 0 we use $J^{-1}(X_m)$, if 1 $Mid(J^{-1}(X))$, if 2 both above conditioning matrices will be used

- `grad, grad3B`: it is possible to use the derivatives of M to improve the interval evaluation. If `grad` is set to 1 this will be used for all boxes in the algorithm. If `grad3B` is set to 1 it will also be used for the sub-boxes that are considered when using the `3B` method
- `incl`: a list. If the end-user uses a specific procedure to compute the interval evaluation of `func`, then `incl[1]` should be set to 1. Similarly if it uses its own procedure to compute the derivatives of `func`, then `incl[2]` should be set to 1.
- `vars`: the list of variables. All n unknowns in the n elements of `func` must appear first in this list
- `ProcName`: the name of the simplification procedure that will be created in the file `ProcName.C`

4.3 Roots simplification procedures

A special type of simplification procedure is the one that are used not for filtering boxes but for finding roots of a system of equations. The general solving algorithm of `ALIAS` accept such type of procedure that returns 11 if a root is provided by the simplification procedure.

4.3.1 `TryNewton`

This procedure will create a C++ program that is intended to be used within an equation solving procedure as a simplification procedure. For each call to this program the program will run at most `newton_iteration` iterations of the classical Newton scheme with as initial guess for the solution the mid-point of the interval vector that is the argument of a simplification procedure. If Newton seems to converge (i.e. the residues have a width lower than `fepsilon`) the program first checks that the approximation of the root lies within a given search domain and then uses the Kantorovitch theorem and the epsilon-inflation scheme to certify that there is a single solution within a given ball. In that case the program will return 11, otherwise it returns 0. The general use of this procedure is for finding quickly a root of the system although in some cases it may speed up the process of finding all solutions of the system.

The syntax of this procedure is

```
TryNewton(func,vars,Init,procname)
```

- `func`: a list of n equations
- `vars`: a list of n unknowns names
- `Init`: a search domain for the solutions
- `procname`: the name of the simplification program that will be written in the file `procname.C`

This procedure may be used with `GeneralSolve` and `GradientSolve`. It should not be used with `HessianSolve` as the corresponding C++ program is already embedded in the C++ code of this procedure.

4.3.2 `Rouche`

The `Rouche` procedure has to be used for a system of equations and will create a simplification procedure that may return 11, i.e. it provide a ball that include a single solution of the system, that may be found using a Newton iteration. A key-point for using this procedure is to consider the matrix constituted of the k-th derivatives of the equations with respect to the variable, that will be denoted $F^(k)$ and to be able to find k_0 such that there is a k_j in $[2, k_0]$

$$\frac{||(F^{(1)}(X_0))^{-1}F^{(k)}(X_0)||^{1/(k-1)}}{k!} \leq \frac{||(F^{(1)}(X_0))^{-1}F^{(k_j)}(X_0)||^{1/(k_0-1)}}{k_j!}$$

for all X_0 in the search space and all possible $k \geq 2$. The value k_0 will be called the *order* of the Rouche method. Two different cases will be considered here:

- algebraic equations
- non-algebraic equations

For algebraic equations there will be clearly a k_1 such that $F^{(k)} = 0$ for all $k \geq k_1$. Hence we may use $k1 - 1$ as order for the Rouche method.

As soon as there are non algebraic terms in at least one equation, then we cannot determine in advance the order of the Rouche method without an in-depth analysis of the system.

The syntax of this procedure is:

```
Rouche(Func,nfunc,Order,Vars,ProcName)
```

where

- `Func`: the list of equations
- `nfunc`: the number of equations in `Func`
- `Order`: the order of the Rouche method. For algebraic equation you may specify "p" for this order as the procedure will determine automatically the correct order. Otherwise it is your responsibility to provide a numerical value for the order
- `Vars`: a list of variable name
- `ProcName`: the name of the simplification procedure which will be written in the file `ProcName.C`

A good example is

```
EQ:=[x^2+y^2+z^2-9,(x-1)^2+(y-1)^2+z^2-9,(x-1)^2+(y+1)^2+z^2-16]]:
VAR:=[x,y,z]:
Rouche(EQ,"p",VAR,"rouche");
```

This procedure will be called only if the width of the box is lower than `ALIAS/maxnewton` and uses at most `ALIAS/newton_iteration` of the Newton scheme to compute an approximation of the root such that the residues of the system are lower than `ALIAS/fepsilon`. The generated C++ program will also use the epsilon inflation method to enlarge as much as possible the ball that includes the root.

The Rouche procedure may be quite powerful to find a ball that includes a single root of the system (even more powerful than the Kantorovitch scheme). For example by using `Rouche` in combination with `DeflationUP` (see section 8.1.2.1) we have been able to solve the Wilkinson polynomial of order 19 (see section 12.3), while the general procedure failed certifying roots starting at order 13.

4.4 Concatenation of simplification procedures

In some cases you may have the opportunity to write more than one simplification procedure (for example by using the `HullConsistency` procedure and writing you own simplification procedure). Currently the procedures in the C++ library and in the Maple library accept only one simplification procedure. But you may concatenate an unlimited number of simplification procedures by using the procedure `CatSimp`. The first argument of this procedure is the name of the procedure that will be used as unique simplification procedures by ALIAS-C++. The following arguments are names of simplification procedures. Note that the name of the file containing the simplification procedure must always be the name of the procedure with the extension `.C`. Hence

```
CatSimp("WholeSimp","Simp1","Simp2")
```

will produce the file `WholeSimp.C` defining the procedure `WholeSimp` which in turn calls the procedures `Simp1`,`Simp2` defined in the file `Simp1.C`, `Simp2.C`. Hence the arguments of the `CatSimp` procedure are usually strings.

This procedure accepts a numerical value as optional last argument: in that case the whole chain of simplification procedures will be repeated if the width of at least one range has decreased by more than the specified numerical value during the previous run of the simplification procedures.

Some of the simplification procedures may be computer intensive and therefore their use in the 3B method (see section 4.5) may not be recommended. For that purpose it is possible to specify within the `CatSimp` procedure that a given simplification procedure shall not be used within the 3B method: for that purpose just add the string `NO_3B` right after the procedure name. Hence:

```
CatSimp("Simp","Simp1","Simp2","NO_3B"):
```

will create the simplification procedure `Simp` that will usually calls `Simp1` and `Simp2` except for the 3B method where only `Simp1` will be used.

4.5 Using the 3B method

4.5.1 Principle

The 3B method is an usually efficient method to reduce the width of the variable intervals. This method can be used for any algorithm of `ALIAS` and is embedded in the code of the solving procedures of the C++ library.

The principle of the method is quite simple. Let assume that we deal with a problem with n variables $x_1, \ldots, x_n$ and that the range for the variable x_i is $[a_1, b_1]$ while ϵ is a small number. The procedure will examine if the problem may have a solution if the range for x_i is reduced to $[a_1, a_1 + \epsilon]$. If it is not the case the range for x_i may be reduced to $[a_2, b_2]$=$[a_1 + \epsilon, b_1]$. The procedure will then be repeated by using 2ϵ as a new value for ϵ until the algorithm either determine that there is no solution to the problem or fails to show that that is no solution for the range $[a_n, a_n + n\epsilon]$. The algorithm will then try to reduce the range for x_i on the "right" side i.e. by examining if the problem has no solution for the range $[b_1 - \epsilon, b_1]$. This process is repeated in turn for each variable.

Within Maple the value of ϵ is given by the variable `‘ALIAS/Delta3B‘` which set an identical value of ϵ for all variables. Alternatively you may specify a specific value of ϵ for each variable by assigning a list to the variable `‘ALIAS/Delta3B_ARRAY‘` with as many elements as unknowns: each element of this list is the value of ϵ for each variable.

The 3B method will not be used for variables whose range have a width greater than a given threshold. This threshold is defined by the variable `‘ALIAS/Max3B‘` (default value: 5). You may assign a new value for this threshold by:

```
‘ALIAS/Max3B‘:=10:
```

Alternatively you may specify a specific threshold for each variable by using the array `‘ALIAS/Max3B_ARRAY‘`. For example:

```
‘ALIAS/Max3B_ARRAY:=array([0.1,0.1,0.05]):
```

indicates that the 3B method will be used only if the width or the input intervals is lower than 0.1 for the 2 first unknowns and 0.005 for the third one.

As soon as the `Max3B, Delta3B` values have been set you activate the 3B method by setting the `‘ALIAS/3B‘` variable to 1 or 2. At iteration n of the 3B method the algorithm will try to reduce the range for the variable by $n\epsilon$. Assume that n is larger than 1 and the algorithm fails:

- if `‘ALIAS/3B‘` is set to 1 the algorithm will move to the next variable (or try to reduce the range on the "right" side if we dealing with the "left" side)
- if `‘ALIAS/3B‘` is set to 2: the algorithm was considering for the variable the range $[a_k, a_k + n\epsilon]$ when it has failed. Instead of moving to the next step the algorithm will consider the range $[a_k, a_k + \epsilon]$ and repeat the process until the method fails for a gain exactly equal to ϵ

4.5.2 Repeating the 3B method

An improvement may be obtained for the variable x_i with $i > 1$ while no improvement was obtained for the variable $x_1, \ldots, x_{i-1}$. But during the calculation for these variables the range for the variable x_i was the initial one and not the improved value. Hence it may be interesting to repeat the procedure as soon as one the variable is improved.

The 3B procedure will be repeated if:

- the variable ‘ALIAS/Full3B‘ is set to 1 and the change in at least one variable is larger than ‘ALIAS/Full3B_Change‘
- the variable ‘ALIAS/Full3B‘ is set to 1 or 2 (default value: 0) and if there are two changes on the variable (a change is counted when a variable is changed either on the left or right side) or the change in at least one variable is larger than ‘ALIAS/Full3B_Change‘

The 3B method usually exhibit an asymptotically slow convergence speed and it is hence bad policy of repeating the process for very small improvements in the range.

4.5.3 Other parameters for the 3B method

At one step of the algorithm the 3B method will determine that there is no solution either by considering the simplification procedures provided by the end-user or by a direct evaluation of the expressions involved in the problem.

As the simplification procedures may be computer intensive you may avoid using them within the 3B method by setting the flag ‘ALIAS/Use_Simp_3B‘ to 0. Alternatively you may specify that only some of the simplification procedures are to be used within the 3B method by using the mechanism described in the CatSimp procedure (section 4.4). You may also set ‘ALIAS/Use_Simp_3B‘ to 2: in that case if some variable ranges have been improved the algorithm will check if the whole improved set of ranges may not be a solution of the problem by using all available simplification procedures (including the one having a "NO_3B" in CatSimp).

As for the interval evaluation of the expressions involved in the problem some of the procedures proposed in ALIAS-Maple may use the derivatives of the expression to improve the interval evaluation. This may be computer intensive and you may avoid using these derivatives within the 3B method by setting the variable ‘ALIAS/Grad_3B‘ to a list with as many elements as number of variables. The values in this list are thresholds for the width of the variable ranges such that the derivatives are not used if at least one range width exceed the value.

4.6 Simplification procedures for matrix

For some procedure involving matrices there are specific simplification procedures with a different syntax:

```
SIMP(int dimA,INTERVAL_MATRIX & A)
```

This procedure shall return -1 if the set of matrices A satisfies a given property, 0 otherwise (dimA is the dimension of the matrices).

Chapter 5

Optimization

5.1 Introduction

You may solve an optimization problem with the `ALIAS` library by using one of the following procedures:

```
Minimize    Maximize        MinimizeGradient  MaximizeGradient
MinMax      MinMaxGradient
```

.

Note that these procedures are only special variants of the general solving procedures of `ALIAS`-Maple. Hence most of the parameters setting for these procedures are also valid for the optimization procedures. Specific parameters for these procedure are presented in section 5.3.

In the list of functions that are provided to these procedures the last one must be the function whose extremum are looked for, while the other one are supposed to be constraints. The interval solution that are returned will verify:

- for the constraints:
  - for inequality constraint: for any value of the unknowns within the solution ranges the inequality is strictly satisfied
  - for equality constraint: if the width of the interval solution is less than `‘ALIAS/epsilon‘` then the interval evaluation of the constraint include 0, while if this is not the case then the width of the interval evaluation of the function include 0 and its width is less than `‘ALIAS/fepsilon‘`
- for the function that is to be minimized or maximized: the real minimum or maximum of the function is at a distance at most `‘ALIAS/eepsilon‘` (default value: 1e-6) from the result provided by `ALIAS`

You may also us the gradient of the function to be minimize or maximized to improve the efficiency of the procedure. The `MinimizeGradient` and `MaximizeGradient` procedures may be used for this purpose and have the same argument than the `Minimize` and `Maximize` procedure.

There are also procedures to compute at the same time the minimum and maximum values of a function, eventually under constraint. The `Minmax` and `MinMaxGradient` procedures may be used for this purpose. The only difference with the previous procedures is that the arguments should be ended by `Min`, `Max` which provide the minimum and maximum of the function (being understood that the function that is considered is always the last in the list, the other one being the constraints).

However in that case a global variable of the `ALIAS` Maple library may play an important role: `‘ALIAS/stop_minmax`. Its default value is set to 0 but:

- if set to 1: the procedure will search the minimal and maximal value of the last function in the list but will exit if at some point it is shown that the minimum and maximum value have opposite sign. If this is not the case the procedure will still provide the minimal and maximal value of the function up to the given accuracy

- if set to 2: here the main purpose is to determine if the minimum and maximum of the function have the same sign. None of the `Min`, `Max` value will be significant from an optimum viewpoint: they are used only to indicate the sign of the extremum.

The flag `allow_storage` may also be of interest: if set to 1 (default value: 0) the current estimation of the optimum is written in the file `.opti` in a sequential way.

All these procedures accept optionally as least argument a string `Simp` that indicates the name of a simplification procedure (see chapter 4) whose C++ code is available in the file `Simp.C`. Among the simplification procedures provided by ALIAS-Maple the use of the `HullIConsistency` procedure with the key-word `Optimum` may be interest for an optimization problem, see section 4.2.1.

You may also specify initial value for the minimum or maximum value of the optimized function by setting `‘ALIAS/opt_min‘`, `‘ALIAS/opt_max‘`.

5.2 Examples

Let consider the following example:

```
Minimize([cos(x)+y*cos(y)^2-0.2,x^2+y^2],[x,y],[[-Pi,Pi],[-Pi,Pi]],Min);
```

will return the value of x, y in the range $[-\pi, \pi]$ which minimize the value of $x^2 + y^2$ under the constraint $\cos(x) + y * \cos(y)^2 - 0.2 = 0$ and `Min` will contain the minimal value. In the previous example we will get the following result:

```
>Minimize([cos(x)+y*cos(y)^2-0.2,x^2+y^2],[x,y],[[-Pi,Pi],[-Pi,Pi]],Min);
[[[-.957582, -.957582], [-.474626, -.474626]],

    [[-.957582, -.957582], [-.474626, -.474626]]]
>Min;
                                [[1.14223, 1.14223]]
```

You may notice that `Min` is a range and that `Minimize` returns two solutions for `x,y`. This is so because `Minimize` may deal with function having interval coefficients: hence the procedure will return a range for the minimal value of the function and the values of `x,y` at which the lower and upper bound have been obtained. Hence we may change the previous example with:

```
Minimize([cos(x)+y*cos(y)^2-0.2,(x+INTERVAL(0..0.01))^2+y^2],[x,y],[[-Pi,Pi],[-Pi,Pi]],Min);
[[[-.958271, -.958271], [-.473241, -.473241]],

    [[-.957582, -.957582], [-.474626, -.474626]]]

> Min;
                                [[1.12307, 1.14233]]
```

Note that for this example we may have generated a simplification procedure with:

```
HullIConsistency([x^2+y^2-(Optimum-‘ALIAS/eepsilon‘)<=0],[x,y],0,"Simp");
Minimize([cos(x)+y*cos(y)^2-0.2,x^2+y^2],[x,y],[[-Pi,Pi],[-Pi,Pi]],Min,"Simp");
```

which allows to decrease almost by half the computation time.

5.3 Specific parameters for the optimization procedures

We indicate here parameters that appear in the optimization procedures, their meaning, their default value and the corresponding name in the C++ library.

Parameter name	Meaning	C++ equivalent
‘ALIAS/eepsilon‘	Accuracy on the optimum	
‘ALIAS/optimize‘	type of optimization (automatically set by the procedure)	ALIAS_Optimize
‘ALIAS/optimize_lo_a‘	threshold on the distance between the current minimum and a new minimum to call the local optimizer	ALIAS_LO_A
‘ALIAS/optimize_lo_b‘	threshold on the distance between the current maximum and a new maximum to call the local optimizer	ALIAS_LO_B
‘ALIAS/stop_minmax‘	allows to exit from the algorithm if the optimum have opposite sign	ALIAS_Stop_Sign_Extremum
‘ALIAS/stop_opt_sol‘	allows to exit from the algorithm if the optimum is better than a threshold	ALIAS_Stop
‘ALIAS/opt_sol_min‘ ‘ALIAS/opt_sol_max‘	threshold for stop_opt_sol threshold for stop_opt_sol	ALIAS_Extremum ALIAS_Extremum
‘ALIAS/opt_max‘,‘ALIAS/opt_min‘	allows to specify initial values for the minimum and/or maximum	

Chapter 6

Integration

ALIAS provides various procedures for the certified calculation of definite integrals. These procedures will return a range that is guaranteed to include the real value of the integral. The width of the range will be lower than the global variable `'ALIAS/fepsilon'`. In the current implementation only finite bounds may be used.

Some of the following procedures involve the use of the derivatives and you should be careful when the interval evaluation of these derivative cannot be performed. For example when considering $e^x + 2\sqrt{x} + 1$ you should not 0 in the integration domain. In that case you should integrate around 0 by using the `Integrate` problem while using another procedure for the remaining part of the domain.

6.1 Integral in one variable

The basic integration procedure is `Integrate(Func , Init)` where `Func` is the integrand and `Init` the range for the integration. For example

```
Integrate(1/x,[1,2]);
```

returns the range [.69264842756679057, .69364643155883843] if `'ALIAS/fepsilon'` is set to 1e-3. This procedure should not be used to determine accurate range.

For accurate computation for function that are at least twice differentiable you may use `IntegrateTrapeze` or `IntegrateRectangle` which have the same syntax than `Integrate`. You may also use `IntegrateTaylor` that use a Taylor expansion of order `N` of the function. The syntax is

```
IntegrateTaylor(Func,Init,N)
```

All these procedures will return `UNABLE` if the integral cannot be computed with the desired accuracy or `FAILED` if the number of boxes exceed the global variable `'ALIAS/maxbox'`. As the default value of this variable is 20 000 (i.e. relatively small) it may be necessary to set the variable to a larger number before calculating the integral.

Note that the derivatives of the function and the function itself may involve expression that cannot be interval evaluated everywhere. Hence it is of good policy to check this point by using the `Problem_Expression` package and that may require to set some parameters of this package (see section 2.1.5).

For example assume that we have to integrate the function $\sqrt{1-\cos^2(x)}$ in the range 0, $\pi/2$. The derivatives of this expression involves the expression of $1/(\sqrt{1-\cos^2(x)})$ which is not defined in 0. If we use `IntegrateTaylor` of order 2 we need to ensure that the function and the derivatives up to the third one can be interval evaluated.

```
eq:=sqrt(1-cos(x)^2):
'ALIAS/close_to_zero':=1e-20:
'ALIAS/low_value_expr_violated':=1:
'ALIAS/high_value_expr_violated':=1:
Verify_Problem_Expression([eq,diff(eq,x\$1),diff(eq,x\$2),diff(eq,x\$3)],[x]):
IntegrateTaylor(eq,[0,evalf(Pi)/2.],2);
```

The setting of the parameters allows to calculate the integral.

6.2 Integral in several variables

The basic multiple integral procedure of `ALIAS` is

```
IntegrateMultiRectangle(Func,Vars,Init)
```

that may be used as in this example:

```
IntegrateMultiRectangle(x*sin(y),[x,y],[[0,1],[0,1]]);
```

This procedure requires that the function is at least twice differentiable.

A more sophisticated procedure is based on Taylor expansion. Its syntax is

```
IntegrateMultiTaylor(Func,Vars,Init,N,Type)
```

where `N` is the order used for the Taylor expansion. If `N` is large (i.e. say larger than 6) the remainder of the Taylor expansion may be a very large expression, that may be difficult to compile. `Type`, which should either "explicit" or "autodiff", may be used to deal with this problem. If it is set to "explicit" the remainder will be calculated exactly and be compiled as it is. If `Type` is set to "autodiff" the procedure will try to use forward automatic differentiation and elementary components identification (see the procedures `Decompose_Diff` and `Auto_Diff`) to reduce the compilation time. Still you cannot expect to be able to use very large value for `N` if the integrand is complicated.

Chapter 7

Continuation for one-dimensional system

7.1 Introduction

The purpose of continuation here is to consider a system with one parameter and n unknowns and to determine what are the possible values of the unknowns when the parameters lie in a given range [a,b]. The values of the unknowns will be calculated for values of the parameter separated by a constant value, starting at one extremity of the range for the parameter.

The principle is to solve the system for the initial value a or b of the parameter range and then to obtain the solution for other values of the parameter using a certified Newton scheme (hence it is necessary that the equations of the system are at least C^1).

As soon as the initial starting point of the branches have been obtained the continuation procedure is usually very fast. The continuation will stop if the jacobian become singular. The algorithm will then increase the parameters value until we can start again a certified procedure.

7.2 The `Continuation` procedure

The full continuation procedure described in the ALIAS-C++ manual is available in `Maple`. It enable one to obtain initial points on the different branches of a one dimensional system and then to follow these branches. These points are written in files. The syntax of this procedure is:

```
Continuation([equation set],[variable set],[parameter name],
  [list of ranges for the variable],[range for the parameter],
  dz,epsilon,epsiloni,Sens,file base name);
```

where `dz` is the parameter increment for the storage (a point will be stored each time the parameter change value by `dz`), `epsilon` has a small value, `epsiloni` defines the accuracy with which the starting points of the branches are determined (if we change the parameter value of the starting point by `epsiloni`, then the system has no solution) and `Sens` is 1 if we compute the branches by increasing values of the parameter (-1 for decreasing values). Note that determining the initial starting points of the branches with an accuracy on the parameter `epsiloni` smaller than `epsilon` may a costly process: this will be done only if the flag `'ALIAS/allow_backtrack'` is set to 1 (its default value is 0). Note also that according to the value of `Sens` you may not obtain the same number of branches: `Sens` should be chosen so that at the initial value of the parameter (the first one for which the system has solutions) the number of solutions is maximal. A major change with respect to version 1.2 is that a backtrack mechanism is used to avoid such problem: if at some point the procedure has followed n branches until a problem for Newton iteration has been detected and if the number of solutions for the next value of the parameter is m with $m > n$, then the procedure will store the value of the m solution and, after having completed the computation for the current `Sens`, will start again a continuation from this point and will follow the m branch using the opposite value of `Sens`. Note however that only a maximum

of 10 such points are stored and hence some branches may still be missing for complex expression. Note also that among the m branches we will have n branches that will be identical to the one that have been computed in the process before the backtrack.

The points on the branches are written in files whose names start with the file base name and is followed by the branch number. Hence if `Branch` is file base name and the algorithm has found two branches, then the points are written in the files `Branch1`, `Branch2`. In these files is written on each line the values of the variables in the order given by `variable set` followed by the value of the parameter.

By default we use only Kantorovitch theorem to follow the branches but you may also both this test and Moore test: this is done by setting the flag `'ALIAS/kraw'` to 1.

For this procedure the Newton scheme will be used and will run for a limited number of iteration defined by `'ALIAS/newton_iteration'` (default value: 100).

Note that the `Draw` procedure (section 7.5) allows to directly obtain a Maple plot of the branches as soon as there is only one or two unknowns.

7.2.1 Optional arguments

Note that optionally you may add two arguments at the end of the list of arguments. These arguments are the name of a C++ program and the name of the procedure that is defined within this program. This procedure must update the range of the unknowns according to the value of the parameter.

Assume for example that you are considering the equation:

$$(x-1)^2 + (y-1)^2 = l_{25}$$

where l_{25} is the parameter of the system and x, y the unknowns. Clearly x and y cannot exceed $l_{25}+1$ and cannot be lower than $-l_{25}+1$.

The syntax of this simplification procedure is:

```
int Simp(double param, INTERVAL_VECTOR v_IS)
```

where `param` is the current parameter value and `v_IS` the current ranges for the parameters. Unfortunately this simplification procedure cannot be produced directly by the ALIAS-Maple procedure described in chapter 4 as the argument of the simplification procedure are different.

7.2.2 Example

Let consider the curve described by the system:

$$x^2 - (y-1)^2 + z^2 - 1 = 0$$
$$x^2 - y^3 = 0$$

where z may be chosen as the parameter. To obtain in the files `B1, B2, B3, B4` points of the 4 branches of this curve you will write:

```
with(ALIAS):
Continuation([x^2+(y-1)^2+z^2-1,x^2-y^3],[x,y],[z],
             [[-100,100],[-100,100]],[[-2,2]],0.05,1e-6,1.e-4,1,"B");
```

7.3 Continuation with initially known starting points

If the starting point of branches is known it is possible to follow the branches without initially solving the system by using the procedure

```
StartContinuation(Func,Vars,Par,Init,RPar,DPar,MinD,MinI,Sens,Base,P)
```

where

- `Func`: the list of equations

- `Vars`: a list with the name of variables
- `Par`: name of the variable that will be used a parameter
- `Init`: a list of intervals ranges for the unknowns
- `RPar`: a list which define the minimal and maximal value of Par
- `DPar` a point is added to the plot each time the current value of Par is at DPar from the previous one
- `MinD` a parameter that is used to determine singular points. Its value should be small
- `MinI`: the accuracy on the parameter with which the initial starting point of the branch is determined: if the parameter is z and the curve start at z0, then if `Sens`=1 the system has no solution for z=z0-MinI
- `Sens` 1 is the curve is drawn from Min(Par) to Max(Par) or -1 if drawn from Max(Par) to Min(Par).
- `Base` a string that indicate the base-name for the files that will be created
- `P`: a list of starting points for the branches. For two unknowns this list has the form [[[a1,a1],[a2,a2]],...]. These points must be a solution of the system in the Kantorovitch sense and be obtained using of the solving procedure using the Hessian

A mechanism allows the user to stop the determination of a branch. For that purpose the user has to write a C++ program such as

```
int End_Cont(MATRIX &BRANCH,int nb,int Sens, double z, INTERVAL &LIMITS)
```

This procedure receives the current value of the parameter (`z`), possible limits that have been defined for this parameters (`LIMITS`) and the direction (`Sens`) in which the value of the parameter is changed. The values of the other variables are stored in `BRANCH`, while their value for the current `z` is `BRANCH(nb,...)`. If this procedure returns -1, then the branch will be no more followed, otherwise the procedure must return 0.

The user may indicate that such procedure is available by setting the variable `‘ALIAS/user_stop_continuation‘` to a string which indicates the name of the C++ procedure, being understood that the file that includes the code is `‘ALIAS/user_stop_continuation‘.C`

A specific procedure exists for distance equations:

```
StartContinuationDistance(Func,Vars,Par,Init,RPar,DPar,MinD,MinI,Sens,Base,P)
```

7.4 Specific parameters for the continuation procedures

Parameter name	Meaning	C++ equivalent
`‘ALIAS/allow_backtrack‘`	allows for backtracking the parameter value	`ALIAS_Allow_Backtrack`
`‘ALIAS/kraw‘`	allows to use the Krawczyk test	`ALIAS_Use_Kraw_Continuation`

7.5 Drawing procedure

For one dimensional system with 1 or 2 variables it is possible to obtain a plot of the curve. The syntax of this procedure is:

```
Draw([equation set],[variable set],[parameter name],
  [list of ranges for the variable],[range for the parameter],
  parameter increment for the storage,epsilon,epsiloni,Sens);
```

where the arguments are the same than the previous procedure. It returns a `PLOT` structure for system with one variable and a `PLOT3D` structure for systems with two variables. Hence to draw the previous curve you may write:

```
with(ALIAS):
with(plots):
Draw([x^2+(y-1)^2+z^2-1,x^2-y^3],[x,y],[z],[[-100,100],[-100,100]],[[-2,2]],
0.05,1e-6,1);
```

Note that optionally you may add two arguments at the end of the list of arguments. These arguments are the name of a C++ program and the name of the procedure that is defined within this program. This procedure must update the range of the unknowns according to the value of the parameter.

Note that the points used for drawing the plot are stored in the file `/tmp/Branch1` for the first branch, `/tmp/Branch2` for the second branch and so on.

For problems involving more than 2 variables you may obtain partial plots by using the `DrawND` procedure, see section 9.10.

Chapter 8

Univariate and parametric polynomial

8.1 Univariate polynomials

Although Maple can deal efficiently with univariate polynomials we still provide some utilities that may help to design a C++ program to solve and manage univariate polynomial.

8.1.1 Utilities

8.1.1.1 Bounds for the real roots

The procedure `BoundUP` allows one to create a C++ program that will determine bounds for the real roots of an univariate polynomial. Its syntax is

```
BoundUP(Func,Vars,name)
BoundUP(x^3-6*x^2+11*x-6,x,"SIMP");
‘ALIAS/ID‘:="nd":
BoundUP(y*x^3-6*x^2+11*x*y-6,[x,y],"SIMP");
```

where

- `func`: the polynomial
- `Vars`: the variable name of the polynomial or a list of variable names, whose first element should be the polynomial variable
- `name`: the name of the created C++ program, that will be written in the file `name.C`

The C++ program will take as input an interval vector which will represent the range for all variables in `Vars`. The program will then possibly update the ranges

```
[AA_low_pos, AA_high_pos]
[AA_low_neg, AA_high_neg]
```

that will contain respectively the ranges for the positive and negative real roots of the polynomial. If the program has been able to update the range for the positive roots it will set the global flag `AA_Flag[0]` to 1 and `ALIAS_Flag[1]` for the negative roots.

Note that several such procedures may be used as soon as a different ‘ALIAS/ID‘ are given between the procedure: if ‘ALIAS/ID‘ is not an empty string all global variables `AA, ALIAS_Flag` will have ‘ALIAS/ID‘ appended to their names. Hence in the second above example the lower bound for the positive roots will be `AA_low_posnd`.

An optional 4th argument may be a list of variable name. In that case `Vars` must be a single variable name. This will allow to create multiple simplification procedures for a parametric polynomial. For example if P is the polynomial

$$y^3 * x^3 - 6 * x^2 + 11 * x * y - 6, x$$

then

```
‘ALIAS/ID‘:="1":
BoundUP(y^3*x^3-6*x^2+11*x*y-6,x,"SIMP1",[x,y]);
‘ALIAS/ID‘:="2":
BoundUP(y^3*x^3-6*x^2+11*x*y-6,y,"SIMP2",[x,y]);
```

will create the simplification `SIMP1` that consider P as a polynomial in `x` while `SIMP2` consider P as a polynomial in `y`. This allows to have a single list of variable names, as required by the solving procedure, but still to generate multiple simplification procedures.

8.1.2 Simplification procedures

8.1.2.1 Deflation

The procedure `DeflationUP` will create a simplification procedure based on a re-writting of the univariate polynomial $P(x)$ of degree n. Let us assume that approximate roots $x_1, \ldots x_m$ of the polynomial P has been found. Then the polynomial may be written as

$$P(x) = \prod_{j=1}^{j=m} (x - x_j)(Q(x) + R)$$

where $Q(x)$ has degree $n-m$. In the generated simplification procedure a C++ program will use the `ALIAS_Nb_Solution` approximate roots stored in the interval matrix `ALIAS_Solution` to compute safely Q, R and will then use the above form of P to compute the interval evaluation of P for the current box. If this evaluation does not include 0, then the simplification procedure will return -1, allowing the current box to be discarded.

The syntax of this procedure is:

```
DeflationUP(Func,Vars,EvalProc,JevalProc,name)
```

where

- `Func`: the P polynomial
- `Vars`: the name of the polynomial variable
- `EvalProc`: the name of a C++ procedure in `MakeF` format that will be used to evaluate the polynomial. If `GradientSolve` or `HessianSolve` are used as solving procedure this name is by default "F". Otherwise the user may use its own procedure, for example by using `MakeF`.
- `JevalProc`: the name of a C++ procedure in `MakeJ` format that will be used to evaluate the derivative of the polynomial. If `GradientSolve` or `HessianSolve` are used as solving procedure this name is by default "J". Otherwise the user may use its own procedure, for example by using `MakeJ`.This procedure is used to compute accurately the approximate roots of P by using the Newton scheme with as initial guess the mid-point of the global C++ variable `ALIAS_Solution` until the residues are lower than `‘ALIAS/fepsilon‘`. Alternatively you may specify `"none"` for `JevalProc` in which case the mid-point of `ALIAS_Solution` will be used as approximate solution
- `name`: the name of the simplification procedure that will be written in the file `name.C`

To be used the coefficients of P must be either real numeric or intervals.

8.2 Parametric polynomial

A *parametric polynomial* is a polynomial whose coefficients are functions of a set of parameters (in other words it is a set of polynomials).

8.2.1 Utilities

8.2.1.1 Decomposition

Being given an expression it may be interesting to determine if it can be written as a polynomial in one variable. For example

```
eq:=x^2*sin(x)+x/sin(y)+ x*exp(x):
```

may be considered as a second order polynomial in x with coefficients $\sin(x), 1/\sin(y), exp(x)$ but not as a polynomial in y. The procedure `Decompose_Algebraic` will provide such decomposition. The call `Decompose_Algebraic(eq,x)` will return

```
AA1*x^2+AA2*x+AA3*x
```

Note that the coefficients are not collected. while the global variable `ALIAS_TERMS` will contain the value of the `AA`. If the returned polynomial has degree 0 in the variable, then the expression is not a polynomial in this variable.

8.2.2 Simplification procedures

8.2.2.1 Routh table: `Routh`

The Routh table allows one to determine if all the roots of a polynomial have a negative real part. The number of sign changes in the first column of the table is the number of roots with positive real parts. The Routh table has n+1 rows, where n is the degree of the polynomial.

The syntax of this procedure is:

```
Routh(poly,Vars,Type,Where,ProcName)
```

where:

- `poly` is the polynomial
- `Vars`: a list of variable name whose first element is the polynomial variable
- `Type`: see below
- `Where`: if P(x) is the polynomial we will compute the Routh table for P(x+`Where`). If `Where` is not 0 the number of sign changes will indicate the number of roots with real part greater than `Where`.
- `ProcName`: the name of the simplification procedure

`Type` allows to control the output of this procedure:

- 0 : the procedure will simply returns the Routh table of the polynomial
- $n > 0$ and lower than the degree of the polynomial +1 : the procedure will create a C++ simplification procedure (whose name is the last argument of the procedure) that will compute an interval evaluation of the n first element of the first column of the Routh table, using the derivatives of these elements. This simplification procedure will return -1 if the polynomial has a root with real part larger than `Where`. The interval input of this procedure will be an interval vector corresponding to `Vars`. Note that if n is large the procedure may take a long time to be generated.
- $[k, [n, var1, var2..], [m, var3]]$: if $k > 0$ the same simplification principle than described in the above section will be generated. Furthermore we will calculate the sign of the first element of the first row of the Routh table. If this element has a constant sign (say positive) we will consider the element at the n-th row of the first column. Assuming that this element S has a linear term in var1 (S= a var1+b) with a positive we will write that S is positive when var1 is greater than -b/a and we will update var1 accordingly. We will proceed similarly with the element at of row m and with variable var3

Consider the polynomial $x1 + (x1 + x2) * s + x2^2 * s^2$ in the variable s. Its Routh table at 0 is:

```
Routh(x1+(x1+x2)*s+x2^2*s^2,[s,x1,x2],0,0);
                    [    2             ]
                    [  x2       x1    0]
                    [                  ]
                    [x1 + x2    0     0]
                    [                  ]
                    [  x1       0     0]
```

The first element of the Routh table is positive and the second element of the first row is $x1 + x2$. Hence a necessary condition for the polynomial for not having root with real part positive is that $x1 + x2 > 0$. Hence writing:

```
Routh(x1+(x1+x2)*s+x2^2*s^2,[s,x1,x2],[2,[2,x1,x2]],0,"ROUTH");
```

will imply that the simplification procedure will use the simplification rules $x1 > -x2$ and $x2 > -x1$. If x1=[-5,5] and x2=[0,4] $x1 > -x2$ will imply $[-5, 5] > [-4, 0]$ which leads to the new range [-4,5] for $x1$

8.2.2.2 The `KharitonovConsistency` procedure

This procedure may be used when dealing with the problem of determining if the roots of a parametric polynomial all lie within a given range.

For a parametric polynomial there is usually four Kharitonov polynomials i.e. polynomials have that have constant coefficients. It can be shown that if all the Kharitonov polynomials have the real parts of their roots of the same sign, then all the polynomials in the set will have the real part of their of the same sign. The following procedure allows to use the Kharitonov polynomials to test if there they may be roots of a parametric polynomial within a given range. Its syntax is:

```
KharitonovConsistency(Func,Vars,Gradient,procname)
```

with the following parameters:

- `Func`: a polynomial. If `Func` is a matrix, then the polynomial is supposed to be the characteristic polynomial of the matrix
- `Vars`: list of unknowns. The first one must be the unknown of the polynomial. If `Func` is a matrix the first name must be `AXX`
- `Gradient`: a flag that indicates if the derivatives of the polynomial with respect to the unknowns may be used (1) or not (0)
- `procname`: the name of the simplification procedure. The name of the created file will be procname.C

This procedure will calculate for the polynomial $P(x)$ the coefficients of the polynomial $P(x - a)$. Then the simplification procedure will consider that its input box has as first element a range for x and update a so that the Kharitonov polynomial will indicate if the polynomial may have a root larger than the lower bound of the first element of the box or lower than the upper bound of this interval. This procedure is intended to be used in conjunction with the procedure dealing with parametric polynomials.

8.2.2.3 The `WeylFilter` procedure

Let P be a polynomial and be `maxroot` the maximal modulus of the root of P. From P we may derive a the *unitary polynomial* Q such that the roots of Q have a modulus lower or equal to 1 and if w is a root of Q then `maxroot`w is a root of P.

Let $Q = sum_{i=0}^{i=n} a_i x^i$ which may also be written as $sum_{i=0}^{i=n} b_i (x - a)^i$ where a is some fixed point.

Let a range $[a, b]$ for x and let z_0 be the mid point of the range. We consider the square in the complex plane centered at z_0 and whose edge length is $b - a$. Let δ be the length of the half-diagonal of this square. If

$$|b_0| > \sum_{j=1}^{j=n} |b_j| \delta^j$$

then the polynomial has no root in the square. For a given list of equations we consider in turn each equation and determine if it may be considered as a parametric polynomial. For example the equation $x^2\sin(x) + x\sin(y) + x\ exp(x)$ will be considered as a second order polynomial in x with coefficients $\sin(x), \sin(y) + exp(x)$ (but not a polynomial in y). The procedure

```
WeylFilter(Func,Vars,FullVars,MaxRoot,TypeB,name)
```

will consider each equation in the list `Func` and examine if it may considered as a parametric polynomial successively in each variable in the list `Vars`. If yes the Weyl filter will be used on the polynomial whose coefficients are functions of the variables in the list `FullVars` (all variables in `Vars` must be a member of `FullVars`). `MaxRoot` is a list which indicates for each variable in `Vars` what is the maximum modulus of the roots of all parametric polynomials in this variable. An element of `MaxRoot` may be a numerical value of the key-word "automatic" which indicates that the C++ program will try to determine the maximal modulus. The list `TypeB` indicates for each variable in `Vars` how are computed the b_i i.e. numerically with the keyword "numeric", or symbolically (which is usually more efficient) with the keyword "symbolic". The simplification procedure will be named `name` and be written in the file `name.C`. For the previous example the procedure will be

```
WeylFilter([x^2*sin(x)+x*sin(y)+ x*exp(x)],[x],[x,y],["automatic"],["symbolic"],"SIMP");
```

Another example of the use of this procedure is presented in section 12.3.

8.2.3 Minimal and maximal real roots of a parametric polynomial

The basic procedures for computing the minimal and maximal roots of a parametric polynomial are:

```
MinMax_Polynom(Func,Vars,Init,Type,Solve,Rand,Points,Min,Pm,Max,PM)
MinMax_Polynom_Gradient(Func,Vars,Init,Type,Solve,Rand,Points,Min,Pm,Max,PM)
```

where

- `Func`: the list of constraints between the parameters and polynomial unknown. The polynomial must be the last element of this list. If this last element is a matrix then the considered polynomial is the characteristic polynomial of the matrix (but this is not the more efficient procedure for this case, see next section). If this last argument is `Fast_CharPoly(S)`,`Medium_CharPoly(S)` or `Slow_CharPoly(S)` where `S` is a matrix the polynomial is the characteristic polynomial of the matrix but this polynomial will not be computed by Maple. Instead the procedure `Coeff_CharPoly` will be used to compute the coefficients of the polynomial: this may be interesting for relatively large matrix.
- `Vars`:a list giving the name of the unknown starting with the name of the polynomial unknown
- `Init`: a list of ranges for the unknown starting with the range for the polynomial unknown (this element should be set to a large interval and is updated automatically) followed by the ranges for the parameters
- `Type`: 0 to find only the minimal root, 1 to find only the maximal root and 2 to find both.
- `Solve`: a flag that should usually be set to 3, see the on-line help
- `Rand`, `Points`: the equivalent of the `rand` and `Nb_Points` arguments of the C++ procedure `ALIAS_Min_Max_EigenValues`, see the ALIAS-C++ manual
- `Min, Max`: minimum and maximum root
- `Pm,PM`: values of the parameters at which the minimum and maximum have been obtained

The difference between these two procedures is that the second one uses the derivatives of the polynomial with respect to the variables together with the derivative of the coefficients of the polynomial.

For these procedures the flag `'ALIAS/stop_opt_sol'`, `'ALIAS/opt_sol_max'` will play the same role than `Stop`, `Seuil[0]` and `Seuil[1]` of the C++ procedure `ALIAS_Min_Max_EigenValues`, see the ALIAS-C++ reference manual.

In the special case where the polynomial is the characteristic polynomial it may be interesting to use the specific procedures:

```
MinMax_Char_Poly(Func,Vars,Init,Type,Solve,Rand,Points,Min,Pm,Max,PM)
MinMax_Char_Poly_Gradient(Func,Vars,Init,Type,Solve,Rand,Points,Min,Pm,Max,PM)
```

which has the same argument that the previous procedures except that the last argument of `Func` must be a matrix.

8.2.4 Possible parameters values for a given range for the roots

It is possible to determine an approximation of the region of the parameters space (a n dimensional space where each of the dimension corresponds to one of the n parameters) that contain all the possible values of the parameters such that the corresponding polynomial has all its roots in a given range. This approximation will be constituted of a set of n dimensional boxes written in a file. The procedure to be used is:

```
MinMax_Polynom_Area(Func,Vars,Init,Strong,Solve,Rand,Points,Out,Lim,Type)
```

where

- `Out`: a string which is the name of the file in which the result will be written
- `Lim`: during the calculation all the boxes that have a width lower than this value will be neglected
- `Type`: a string that may be "Real" (if only the real root or the polynomial are considered), "RealPart" (for the real part of the root), "AllReal" (for polynomial having only real root), "OneReal" (for polynomial having at least one real root)

`Strong` is a list of two elements: the first element may be 1 or 2. If it is 2 the derivative of the coefficients of the polynomial will be used. This algorithm uses a secondary bisection process and the second element of `Strong` gives the maximum number of boxes that can be used for the secondary algorithm.

The range for the real roots of the polynomial is defined as:

```
[‘ALIAS/opt_sol_min‘,‘ALIAS/opt_sol_max‘]
```

It is possible to improve incrementally the quality of the approximation. During the first run the flag `‘ALIAS/ND‘` will be set to 1 and the neglected boxes will be written in the file `‘ALIAS/ND_file‘`. During the next run `Lim` has to decreased and the flag `‘ALIAS/ND‘` has to be set to 2. This has the effect that the initial box considered by the algorithm will not be `Init` but the set of boxes stored in `‘ALIAS/ND_file‘`. In this run the neglected boxes will still be stored in the file, thereby allowing another run of the algorithm with a lower `Lim`.

The largest square enclosed in the parameters space such that the polynomials defined by parameter values within the square have all their real roots within a given range can be computed using:

```
MinMax_Square_Polynom_Gradient(Func,Grad,Vars,Range,Init,Rand,Points,Center,Edge)
```

where `Center` will be the center of the largest square while `Edge` will be the lengths of the edge of the square. If you have imperative constraint you may use the variable `‘ALIAS/Imperatif‘` as a list with 0 for non imperative constraints and 1 for imperative constraints. Note that this algorithm uses a main algorithm for getting the largest square and a secondary algorithm for calculating the minimal and maximal root of the polynomial in a given square. You may specify two different bisection mode, one for the main algorithm (`‘ALIAS/single_bisectiong‘`) and one for for the secondary algorithm (`‘ALIAS/single_bisection`). Similarly the maximal number of boxes used by the main algorithm may be specified by `‘ALIAS/maxboxg‘` while the number of boxes used by the secondary algorithm is specified by `‘ALIAS/maxbox‘`

8.2.5 Condition number

The condition number of a polynomial may be defined either as the ratio lowest root over largest root or as the ratio Min($|x_i|$) over Max($|x_i|$) where x_i are the roots of the polynomial. In the later case the condition number has a value between 0 and 1. The minimal and maximal values of the condition number of a parametric polynomial in both form may be calculated using the procedure:

```
MinMax_Condition_Number_Gradient(Func,Vars,Init,Type,Absolute,Rand,Min,Pm,Max,PM)
```

where `Absolute` has to be set to 0 if looking for the ratio minimal root over maximal root or 1 if looking for the ratio in absolute value. The parametric polynomial should appear as last argument of `Func`. Note that is this last element is a matrix then the considered polynomial will be the characteristic polynomial of the matrix. This procedure may hence be used to compute the minimal and maximal value of the condition number of a matrix. Note here that the derivatives of the polynomial and of its coefficients must be available.

8.3 Specificity for the analysis of parametric polynomials

The following variables play a role in the procedures involving a parametric polynomial:

- `‘ALIAS/opt_min‘`, `‘ALIAS/opt_max‘`: initial value for the minimum and maximum real roots of a parametric polynomial
- `‘ALIAS/stop_opt_sol‘`: if set to 1 the algorithm will exit soon as a maximum greater than `‘ALIAS/opt_sol_max‘` **or** a minimum lower than `‘ALIAS/opt_sol_min‘` has been found. If set to 2 while looking for a minimum and a maximum the algorithm will exit if both the minimum is lower than `‘ALIAS/opt_sol_min‘` **and** the maximum is greater than `‘ALIAS/opt_sol_max‘`

For the evaluation of the coefficient of the polynomial there may be evaluation problem (for example one coefficient involves a division by an expression whose interval evaluation may include 0). To deal with this case a procedure similar to what is done for `MakeF` can be used:

- the variable `‘ALIAS/user_CoeffINIT‘` allows to define auxiliary variable in the C++ procedure that implement the evaluation of the coefficients of the polynomial
- a procedure `ALIAS_Coeff(fid,i)` may be created that determine if the coefficient `i` may be evaluated and if not affect to him an arbitrary large value.

To create the procedure `ALIAS_Coeff` we may also use the `Problem_Expression` package (see section 2.1.5). The procedure `ALIAS_Coeff` will be obtained from a list of coefficients `Coeff` in the variable `Vars` with the following maple code:

```
‘ALIAS/low_value_expr_violated‘:= -1e20;
‘ALIAS/high_value_expr_violated‘:= 1e20;
Verify_Problem_Expression(Coeff,Vars,"ALIAS_Coeff","ALIAS_Coeff");
```

Finally the above procedures may use the simplification procedure `BoundUP`, see section 8.1.1.1, that uses general roots bound algorithms for determining if a polynomial may have a root within a given interval.

Chapter 9

Utilities procedures of ALIAS-Maple

9.1 Reusing a compiled program: the `Restart` procedure

It may be possible to restart a procedure with new parameters (usually a new search space) and reusing an already existing executable (hence it is not necessary to create a new executable). This is done with the `Restart` procedure that takes as first argument a string that indicate which procedure should be restarted and as following arguments new parameters for the procedure. See the on-line help to know which procedures can be restarted.

Note however that as soon as you have defined the `'ALIAS/ID'` string before generating an executable it is necessary to reset this string to the same name before using the `Restart` procedure to re-run the same executable

9.2 Expression conversion: `Convert_Frac_Cons`

`ALIAS` uses a special mechanism to convert an expression into an equivalent C++ sentence that can be interval evaluated. This conversion is based on the conversion of the expression into a string in which all mathematical functions are substituted by their interval equivalent. This causes a problem if rational numbers are present in the expression as rational such as 1/6 will lead to 0 in C++. This procedure returns a string that describes the interval equivalent of the expression in which rational such as 1/6 as 1./6. and constant such as Pi have been converted into their C++ equivalent. For example

```
Convert_Frac_Cons(1/6*Sin(v_IS(1))+Pi+2/3*Cos(v_IS(2)));

1./6.Sin(v_IS(1))+BiasPi+2./3.*Cos(v_IS(2))
```

9.3 Multiple occurrences of variables: `MultipleOccurence`

This procedure takes as input an expression and a list of variables and returns 1 if there are multiple occurrences of at least one variable. For example

```
 MultipleOccurence(x^2+y^2-1,[x,y])
```

will return 0 while

```
MultipleOccurence(x^2+x+y^2-1,[x,y]);
```

will return 1.

This procedure is useful to test the transformation of an expression: is there is no multiple occurrences of variables in an expression, then the interval evaluation of this expression will be exact in the sense that it will return the exact lower and upper bounds of the expression for given intervals for the variables (which is not the case when multiple occurrence of variables appear in an expression).

9.4 Mathematical functions in an expression

The procedure `Math_Func` allows one to obtain a list of mathematical functions (in the `Maple` sense) and of their power in a given expression. For example

```
Math_Func([(sin(x)^2+cos(x)^3)/x+sin(x)^2*cos(x)^3+1/sin(x)^2])
```

returns in the global variable `ALIAS_MATH_FUNC` the list

```
      3        2     1
[cos(x) , sin(x) , -------]
                         2
                    sin(x)
```

If these terms appears more than once in the expression the `MakeF` procedure (see section 2.1.1) will generate a code where they are evaluated for each occurrence, which is costly. The `ALIAS_FSIMPLIFY` mechanism described in the `MakeF` section may be used to evaluate only once these expressions and substitute them by interval when calculating the interval evaluation. The end-user write its own `ALIAS_FSIMPLIFY` procedure but the result of `Math_Func` may be used to write such a procedure. A typical `ALIAS_FSIMPLIFY` using this functionality is as follows:

```
ALIAS_FSIMPLIFY:=proc(fid,expr)
local aux,ai:
if type(expr,string) then
   fprintf(fid,"INTERVAL_VECTOR ALIAS_S(%d);\n",nops(ALIAS_MATH_FUNC)):
   for ai from 1 to nops(ALIAS_MATH_FUNC) do
       aux:=Convert_Frac_Cons(Code(ALIAS_MATH_FUNC[ai],[x,y,z,w])):
       fprintf(fid,"ALIAS_S(%d)=%s;\n",ai,aux):
   od:
RETURN(0):
fi:
aux:=expr:
for ai from 1 to nops(ALIAS_MATH_FUNC) do
   aux:=subs(ALIAS_MATH_FUNC[ai]=ALIAS_S(ai),aux):
od:
RETURN(aux):
end:
```

Note the use of the `Code` procedure to convert the expression into an interval equivalent.

9.5 Ordering a list of variables

The procedure `Sort_Variable` sort a list of variable in decreasing order according to the number of occurrences of the variable in a given expression. This list may be used as the argument of the `Maple` procedure `convert,horner` to convert an expression into a form that is more convenient for interval evaluation. For example consider the expression $x^2 + x + y * x$: by using the procedure we get the sorted list [x,y] and

```
convert(x^2+x+y*x,horner,[x,y])
```

leads to $(1 + y + x)x$ while

```
convert(x^2+x+y*x,horner,[y,x])
```

leads to $(1+x)x+yx$ which is less efficient. The number of occurrences of a variable may be found in the global variable `ALIAS_OCCURENCES`.

9.6 Finding elementary components in a function

The procedure `Decompose_Diff` decomposes an expression in a list of elementary components i.e. powers of terms and functions. For example $sin(x)^2 + cos(x)^3)/x + 1/(sin(x)^2 + cos(x)^3)$ will be decomposed into $\sin^2(x), \cos^3(x), 1/x, 1/(\sin^2(x)+\cos^3(x))$. This procedure may be used to design a `ALIAS_FSIMPLIFY` procedure of `MakeF` so that the elementary components are evaluated only once even if they have multiple occurences in the expression. The list of elementary components may be found in the global variable `ALIAS_TERMS`. For a list of expression one has to use `Decompose_Diff_List` that will provide a list sorted by decreased order of length (as defined by the `length` procedure of `Maple`.

9.7 Transformation of an expression: `MinimalCout`

It is well known that the interval evaluation of an expression is sensitive to the manner with which the expression is written. For example in term of interval analysis $x^2 + 2x + 1$ is not equivalent to $(x + 1)^2$ as the interval evaluation of these two forms will usually be different. Unfortunately there is usually no known rule to determine which form will lead to the best interval evaluation of an expression (except if there is exactly one and only one occurrence of each variable in the expression). Hence ALIAS-Maple uses a set of heuristics to transform an expression into a form that leads usually to a good interval evaluation. The procedure `MinimalCout` takes as argument an expression and returns a mathematical equivalent of this expression but in a form that may be better for the interval evaluation.

Among the heuristics that are used in this procedure we use a conversion into Horner form. For a multivariate expression there are different Horner form according to the ordering of the variable. `MinimalCout` will consider all possible ordering of the variable to produce the best form but this may be computer intensive if there is a large number of variables. Hence all the ordering will be tested only if the number of variables does not exceed `'ALIAS/mincout'` with a default value of 8. Reducing the value of `'ALIAS/mincout'` will allow to reduce the computation time for calculating the best form for the expression as the possible expense of getting a less sharper interval evaluation. If the use of `MinimalCout` has to be avoided it is sufficient to set `'ALIAS/mincout'` to a negative value.

`MinimalCout` accepts an optional second argument which is a list of variables. The procedure will try find an expression that is the best with respect to this list.

9.8 Newton scheme

We have a specific `Maple` implementation of the Newton scheme which is mostly be intended to be used in conjunction with the result of the solving algorithms of ALIAS-C++. The main purpose of this procedure is to allow to refine the accuracy of the result provided by the solving procedures of ALIAS-C++ which have an accuracy of `double`. It will allow to calculate the roots of a system with an accuracy that is specified in term of number of digits. For example when specifying an accuracy of 200 digits the result provided by this procedure (when it succeeded) will be a number with 200 digits, the last digit being guaranteed to be the correct digit for the solution.

```
Newton(Func,Vars,Init,Maxiter,Acc)
```

where

- `Func`: a list with the system of equations to be solved (which must be square)
- `Vars`: a list with the name of the unknowns
- `Init`: a list for the initial guess. This list may be either an element of the list returned by the solving procedure of `ALIAS` (i.e. with 2 elements for each variable) or a list with one element for each variable
- `Maxiter`: maximum number of iteration allowed for the scheme
- `Acc`: accuracy for the solution: the Acc-th bit is guaranteed to be exact

This procedure returns:

- -4: number of `Init` element not compatible with the number of elements in `Vars`
- -2: not a square system
- -3: singular jacobian matrix
- -1: more than `MaxIter` iterations
- 1: success

and the estimation of the solution in `ALIAS_Newton`.

Clearly this procedure may fail to return a correct result in some cases: for example if the correct result of an univariate equation is 2 it may happen that the numerical Newton scheme always converge to 1.999999999....

9.9 The `Bound_Distance` procedure

This procedure is specific for systems of distance equations (see section 3.3.2.1) for determining an initial search domain. It may determine either determine an initial guess for the input intervals or to improve a given initial guess.

The syntax is:

```
Bound_Distance(Func,Var,Result)
```

where `Func` is a list of distance equations, `Var` a list of variable name and `Result` is the initial search domain.

An optional fourth argument of this procedure is an initial guess for the search domain. The procedure will return:

- -1: there is no solution to the system of equations
- 0: the procedure has not been able to find an initial search domain or to improve the initial guess
- 1: the procedure has been able to determine an initial guess or to improve the initial guess

9.10 Drawing for non-0 dimensional systems: `DrawND`

The ALIAS-Maple solving procedure allows to compute an approximation of the solutions of non-0 dimensional system as a a set of boxes (see section 3.4). The procedure `DrawND` allows on to visualize 2D or 3D cross-sections of the result:

```
DrawND(File,Nb,PLotL,Compact)
```

where

- `File`: is the name of the result file provided by the solving algorithm
- `Nb`: the number of variables in the result file
- `PlotL`: a list of number that indicates which variable will be plotted. This list must have 2 or 3 elements. For example if the list if [1,10], then a 2D plot with variables 1, 10 will be returned while if the list is [1,2,3] a 3D plot will be returned
- `Compact`: if this integer is set to 1 the procedure will try to regroup boxes in the file in order to reduce the plotting time

The procedure returns a plot structure that may be visualized with the `Maple` procedure `display`.

Chapter 10

Parallel version of ALIAS-Maple

10.1 Introduction

Clearly most of the algorithms of the ALIAS-C++ library have a structure that allows a parallel implementation. Hence ALIAS-Maple offers a parallel version of most its problem solving procedures.

The general principle of the parallel version of the procedure available in ALIAS-Maple is to create a master program that will process the initial search domain until a fixed number N of boxes remains to be processed. This master program will be run on the computer on which is run the Maple session. Then the master program, that maintains a list of unprocessed input intervals, will dispatch by an appropriate mechanism the input intervals to be processed to a slave program that will be run on different computers. This slave program will run a bisection process until some stop criteria is fulfilled: at this point the slave program will send back to the master program the list of unprocessed input intervals and, eventually, the solutions that have been found.

As soon as all the available computers have received an interval input to process the master process will check if any slave computer has terminated its job: if this is the case the master will retrieve the unprocessed boxes from the slave and add them to the list of boxes to be processed. Then new boxes will be sent to the free computers. If none of the slaves have terminated their jobs the master program will process the current box while monitoring if a slave as returned its result. As soon as this computation is performed the master program will again check if any slave computer has terminated its computation: if not it will process the next box in its list.

This process will be repeated until all the boxes in the master list have been processed and all the slaves are free.

The master and slave programs are created by writing an initial part which is problem dependent and by concatenating to this initial part a fixed code that depends only on the procedure that is used. The ALIAS distribution includes generic fixed codes (which are called `Master` and `Slave` followed by the Maple procedure name, e.g. `MasterGeneralSolve.C`, `SlaveGeneralSolve.C`).

For some procedures you may customize the fixed code by assigning the variables `‘ALIAS/master_program‘`, `‘ALIAS/slave_program‘` to a string that gives the name of your C++ code. To determine if you have such possibility look at the global variables at the on-line help for the procedure.

10.2 Stopping an ALIAS-C++ procedure

For using the previous mechanism it is necessary to be able to stop an ALIAS-C++ procedure and recover the current state of the process i.e. the solutions if any and the remaining unprocessed boxes.

There are different mechanisms to stop the bisection process of an algorithm:

1. the number of boxes still to be processed is greater than a given threshold N (this is obtained by setting the global ALIAS-C++ variable `ALIAS_Parallel_Max_Bisection` to $N-1$). This is the mechanism that is used when the master starts processing the initial search box. .

2. as it may be necessary to perform a large number of bisection before obtaining N boxes it is necessary to have a safety mechanism that will limit the number of bisection (otherwise a slave may be stuck in a

lengthy computation while the other slaves are free). The maximal number of bisection may be indicated by setting the global ALIAS-C++ variable `ALIAS_Parallel_Max_Split` and the algorithm will return if this number is reached **and** the number of unprocessed boxes is lower than `ALIAS_Parallel_Max_Bisection` (otherwise the process will continue until this number is reached).

3. the number of performed bisection is greater than a given threshold M whatever is the number of boxes still to be processed: this is obtained by setting the global variable `ALIAS_Parallel_Max_Bisection` to $-M$

4. the number of performed bisection is greater than a given threshold M **and** the number of boxes still to be processed is lower than a given threshold N: this is obtained by setting the global ALIAS-C++ variable `ALIAS_Parallel_Max_Bisection` to $-M$ and the global ALIAS-C++ variable `ALIAS_Parallel_Max_Box` to N . A safety mechanism is enforced in that case to avoid that only one of the slave computer will perform all the computation: if $M \times S$ bisections have been done and there is more than N boxes to be processed the algorithm will return the error code -1. The value of S is given by the global C++ variable `ALIAS_Safety_Factor‘` with a default value of 2

5. if we are using the reverse storage mode we may indicate that the number of performed bisection must be greater than a given threshold M **and** the number of boxes still to be processed is lower than a given threshold N: this is obtained by setting the global ALIAS-C++ variable `ALIAS_Parallel_Max_Bisection` to N and the global ALIAS-C++ variable `ALIAS_Parallel_Max_Reverse` to M

6. the slave computation time has exceeded a fixed amount of time, which probably indicate that it will be preferable to distribute the treatment of the processed box among different slaves. This could be done using the ALIAS-C++ variable `ALIAS_TimeOut` which indicates the maximum amount of time (in minutes) during which a slave may run (this amount will be respected only approximately).

Using this stop criteria we may implement a parallel version of the `ALIAS`-Maple procedures.

10.3 Message passing mechanism and `pvm`

For a parallel implementation it is hence necessary to have a message passing mechanism that enable to send a box to a slave program on another computer and to receive data from the slave computers. For the parallel implementation we use the message passing mechanism `pvm`[1].

10.3.1 Example of message passing mechanism

A very simple master program using `pvm` may be found in the `ALIAS` distribution under the name `MasterGeneralSolve.C` while the corresponding slave program is `SlaveGeneralSolve.C`: these programs are used by Maple for creating the parallel implementation of the `GeneralSolve` procedure. Here we use text message passing:

- the master program send a box with n unknowns by writing in a text buffer the keyword `B` followed by $2n$ reals which are the extremity of the box ranges for each unknown.

- then it append to this message the value `N` of `ALIAS_Parallel_Max_Bisection` with the keyword `SP` (hence `SP N` is appended to the buffer. If `N` is negative then the value `M` of `ALIAS_Parallel_Max_Box` is appended to the buffer by appending the text `P I M` (the value of `ALIAS_Parallel_Max_Bisection` is computed by the master program: if the box is large a positive value is chosen while for small box a negative value is given).

This buffer is then sent to a slave computer with the `pvm_pkstr` procedure. As soon as a buffer is received by the slave computer the value of the box to be processed is extracted from the message using the `ALIAS` procedure `ALIAS_Read_Buffer` (which also update the value of `ALIAS_Parallel_Max_Bisection` and `ALIAS_Parallel_Max_Box`. As soon as the slave computer has terminated its computation it send back to the master in a text buffer:

[1] `pvm` may be downloaded from `http://www.netlib.org/pvm3/index.html`

- the boxes still to be processed (using the keywords `B` for each box or `N` is no box remain to be processed)
- the solution that have been eventually found (using the keyword `S` followed by the $2n$ reals)
- a failure code if the slave has not been able to process the box (using the keyword `F` followed by the error code)

For the algorithms using the derivatives the jacobian matrix is also transferred to the slaves. For large system this may slow down considerably the communication between the slaves and the master. Hence it may be a good policy to inhibit the transfer of the jacobian by setting the variable `'ALIAS/transmit_gradient'` to 0.

10.3.2 pvm

The message passing mechanism that has been chosen is `pvm`[2]. Before using the parallel procedures it is necessary to run `pvm`. This is usually done by first setting the environment variable `PVM_ROOT` to the name of the directory where you have installed `pvm`. Typically this variable should contain something which looks like:

```
/u/PVM/pvm3
```

Then you create a file, called the `hostfile`, in which you indicate the name of the slave machine that you intend to use in the parallel computation. An interesting option is to indicate the full path for the `pvm` daemon program that will be run by the slaves. This option is indicated by the keyword `dx=path`. A typical hostfile will look like:

```
cygnusx1 dx=/u/PVM/pvm3/lib/pvmd
penfret dx=/u/PVM/pvm3/lib/pvmd
camarat dx=/u/PVM/pvm3/lib/pvmd
```

Note that you may have as slave computers a mixture of SUN and PC. As soon as this hostfile has been created you run `pvm` by:

```
pvm hostfile
```

After some time you will get a prompt and you may verify that all the slave computers have been registered with the command `conf`. You may then exit from the `pvm` console program with the command `quit` (you may also kill the `pvm` program with the command `halt`).

10.4 Parallel procedures in Maple

Most of the procedures available in the `ALIAS`-Maple library have a parallel implementation. They have the same name than the non parallel procedures except that they are prefixed by the word `Parallel`. Hence `ParallelGeneralSolve` is the name of the parallel implementation of the procedure `GeneralSolve`. There is an exception with `ParallelContinuationSimplex` which exists only in a parallel implementation.

The parallel procedures have the same initial arguments as the normal procedure except that they have at the end two additional arguments:

- a list of strings: this list gives the name of the slave computers. For example `["cygnusx1","molotova"]` indicates that the slave computers will be `cygnusx1` and `molotova`.
- a list of pair of integers like `[[N1,M1],[N2,M2]]`. There must be as many pairs of integers as the number of slave computers. The first number in the pair indicates that the corresponding slave will return approximately `N1` new boxes provided that the maximal width of the box given as input for the slave is greater than `'ALIAS/diam_switch'` and that the mean diameter is larger than `'ALIAS/mean_diam_switch'`: the exact number of returned boxes is such that right after a bisection process the number of unprocessed boxes is equal or larger than `N1`. Note that there is a safety mechanism to avoid that a slave has a too large

[2] `pvm` may be downloaded from `http://www.netlib.org/pvm3/index.html`

workload: if the number of bisection that has been performed is larger than ‘ALIAS/max_split‘ (default value: 10000) and the number of unprocessed boxes is lower or equal to N1, then the slave will return. A good policy is to increase the value of ‘ALIAS/diam_switch‘ if you observe that all the computation is done by the slave computers.

Otherwise if the maximal width of the box given as input for the slave is lower than ‘ALIAS/diam_switch‘ the slave program will perform at least M bisection (provided that there are still boxes to process after this number of bisection) and will return at most ‘ALIAS/bisection_slave‘ (default value: 30). If M S bisections have been done and still the number of unprocessed boxes is larger than ‘ALIAS/bisection_slave‘, then the slave will stop its processing and will return an error code. In that case the master program will bisect the range having the largest width in the initial box, two new boxes will be created and added to the list of unprocessed boxes. The value of S is defined by ‘ALIAS/safety_factor‘ with a default value of 2. Clearly the efficiency of the parallel implementation is heavily dependent on the values of N and M and diam_switch. Another way to stop the computation of a slave is to set the flag ‘ALIAS/time_out‘ to the maximum number of minutes that a slave may run before sending the processed boxes to the master. By default this value is set to 0 which means that a slave calculation will never be stopped by a time-out signal.

10.5 Minimal parameters setting

Before using the parallel implementation it is compulsory to set some variables:

- ‘ALIAS/bisection_master‘: the number of unprocessed boxes that the master will have to generate before dispatching the boxes to the slaves (default value: 30). It is also the bound on the number of unprocessed boxes that the master will compute when all the slaves are busy before checking again if any slave is free (within the master algorithm there is a mechanism to monitor if a slave has completed its calculation but stopping the master algorithm allows to ensure that all the computation is not done by the master). This number should be small so that the master is not doing any heavy computation while the slaves are waiting for boxes. This variable corresponds to the C++ variable ALIAS_Parallel_Max_Bisection
- ‘ALIAS/working_directory‘: the full path of the directory in which the C++ master and slave programs will be created
- ‘ALIAS/pvm‘: the path of the include files for the pvm library e.g. "/u/caphorn/PVM/pvm3/include"
- ‘ALIAS/libpvm‘, ‘ALIAS/libpvm_linux‘: the path of the pvm library respectively for SUN and LINUX PC's. You should have typically something looking like "/u/caphorn/PVM/pvm3/lib/SUN4SOL2" and "/u/caphorn/PVM/pvm3/lib/LINUX"
- ‘ALIAS/gpp_sun‘, ‘ALIAS/make_sun‘: the full path for the C++ compiler and the make program for SUN workstation. Typically "/usr/local/bin/g++" and "/usr/ccs/bin/make"
- ‘ALIAS/gpp_linux‘, ‘ALIAS/make_linux‘: same for PC's under linux.

Hence a typical Maple file for solving a system of equations EQ in the variable VAR within the ranges S on a mixture of SUN and PC will be:

```
‘ALIAS/profilN‘:="/net/coprin/intervalles/Profil-linux":
‘ALIAS/libN‘:="/net/coprin/intervalles/Lib-linux":
‘ALIAS/working_directory‘:="/u/Interval/Maple":
‘ALIAS/bisection_master‘:=2:
‘ALIAS/diam_switch‘:=0.3:
‘ALIAS/bisection_slave‘:=15:
ParallelHessianSolve(ef2,VAR,S,["otway","molotova"],[[30,4000],[30,4000]]);
```

As soon as the Maple procedure run two C++ program will be generated: one contained the keyword MASTER, the other one with the keyword SLAVE. The program MASTER will be compiled on the machine that run Maple and will control the process. The program that will run the slaves is the SLAVE program: one or two such

programs will be generated according to the architecture of the slave computers (slave program running on a Sun workstation will have their name ended by `SUN` while the program running under Linux will be ended by `LINUX`).

10.6 Specific parameters for the parallel implementation

The solving parameters described in section 3.5 are the one that will be used by the algorithm run by the slaves. Hence it is necessary to specify the parameters for the algorithm that is run by the master.

There is a simple rule: the parameters for the master program have the same name with `_master` appended. Hence `'ALIAS/maxkraw_master'` is the parameter `'ALIAS/maxkraw'` that will be used by the master while the slaves will use the value of `'ALIAS/maxkraw'`. There is an exception to this rule: the parameters for the 3B method have `_Master` appended.

We indicate in table 10.1 parameters that appear in the parallel version of the procedures, their meaning, their default value and the corresponding name in the C++ library.

The parameters that play a role in the calculation are the same for the slave program than for the non parallel version except for the maximal number of boxes for the slave program which is defined by `'ALIAS/maxbox_slave'`.

For the calculation of the master involving the simplex method you may define the parameters of the simplex using one of the following names `'ALIAS/min_improve_simplex_master'`, `'ALIAS/full_simplex_master'`.

Parameter name	Meaning	C++ equivalent
‘ALIAS/bisection_master‘	maximal number of boxes that a master program may create before checking for free slaves	ALIAS_Parallel_Max_Bisection
‘ALIAS/bisection_slave‘	maximal number of boxes returned by a slave	ALIAS_Parallel_Max_Box
‘ALIAS/diam_switch‘	maximal width of a box before switching to direct storage mode in a slave	Diam_Switch
‘ALIAS/3B_Master‘	1 if 3B is activated for the master	
‘ALIAS/Full3B_Master‘	full 3B for the master	
‘ALIAS/Full3B_Change_Master‘	minimal change for the full 3B for master	
‘ALIAS/Delta3B_Master‘	delta for master 3B	
‘ALIAS/Delta3B_ARRAY_Master‘	individual delta for master 3B	
‘ALIAS/Max3B_Master‘	maximal box width for applying 3B for master	
‘ALIAS/Max3B_ARRAY_Master‘	individual maximal box width for applying 3B for master	
‘ALIAS/Use_Simp_3B_Master‘	0: don’t use simplification procedure in master	
‘ALIAS/gpp_sun‘	location of the C++ compiler for Sun	
‘ALIAS/gpp_linux‘	location of the C++ compiler for Linux	
‘ALIAS/libpvm‘	location of the pvm library for Sun Os	
‘ALIAS/libpvm_linux‘	location of the pvm library for Linux	
‘ALIAS/make_sun‘	location of the make program for Sun Os	
‘ALIAS/make_linux‘	location of the make program for Linux	
‘ALIAS/maxgradient_master‘	maxgradient flag for the master program	
‘ALIAS/maxkraw_master‘	maxkraw flag for the master program	
‘ALIAS/maxnewton_master‘	maxnewton flag for the master program	
‘ALIAS/no_hessian_master‘	don’t use hessian for evaluation in master	
‘ALIAS/allows_n_new_boxes_master‘	allows_n_new_boxes for the master	
‘ALIAS/max_reverse‘	see section 10.2	ALIAS_Parallel_Max_Reverse
‘ALIAS/max_split‘	see section 10.2	ALIAS_Parallel_Max_Split
‘ALIAS/max_split_master‘	max_split for the master	
‘ALIAS/profilN‘	string that indicates the location of the BIAS/Profil library for slaves having a different architecture than the master	
‘ALIAS/pvm‘	location of the pvm include file	
‘ALIAS/safety_factor	see section 10.2	ALIAS_Safety_Factor
‘ALIAS/store_gradient_slave‘	gradient storage activated for the slave	ALIAS_Store_Gradient
‘ALIAS/time_out‘	maximal computation time of a slave	ALIAS_TimeOut
‘ALIAS/working_directory‘	directory where is located the slave program	

Table 10.1: Parameters for parallel procedures

Chapter 11

Customizing and improving the code produced by ALIAS-Maple

11.1 Source code

The source code produced by ALIAS-Maple may be customized to improve the efficiency. The name of the programs that are produced by the `ALIAS` Maple library follows some simple rules:

- `_G*.C`: name of the main program for the non-parallel implementation. The characters following `G` vary according to the procedure
- `_F*.c`: name of the program providing the equation evaluation
- `_J*.c`: name of the program providing the estimation of the gradient of the equation
- `_H*.c`: name of the program providing the estimation of the Hessian of the equation
- `_makefile*`: name of the make program
- `_Sol_, _Func_`: result of the computation
- `_MASTER*.C`: master main program for parallel implementation
- `_SLAVE*.C`: slave main program for the parallel implementation
- `_sys_,_temp_,_Func_` : auxiliary files, may be removed at will

11.2 Some rules to improve efficiency

11.2.1 Function evaluation

- when formulating your problem find the best compromise between the number of equations and their complexity. Having the lowest possible number of equations may not be the best if the equations are very complex
- remember that you are not stuck to algebraic equations
- first focus on the problem at hand, not on the equations (see the example in the next section and the example in section 13.2.3.1)
- give the equations from the simplest to the more complex
- if you are using methods involving the gradient and have equations that are similar in the number of terms give first the equations that may have a gradient with a constant sign

- try to factor our most of the term (e.g. $(x-1)^2$ will in general be better than x^2-2x+1). ALIAS-Maple will try to do this job for you but any help is welcome!
- try to normalize your equation and variable

11.2.2 A counter-example

ALIAS-Maple is designed to help generating C++ code and improving the efficiency of the algorithm. But it will not solve all the problems that may be treated by interval analysis as illustrated by the following example that originated from the Geometrica project.

Consider 2 ellipses E1, E2 in the plane that are perfectly defined and the so-called *bi-tangent* i.e. the lines that are tangent to both ellipsis. There is 4 such lines but we assume that some criteria allows to choose two bi-tangents L1, L2. L1 intersects E1 at point P, E2 at and L2 intersects E1 at point R and E2 at point S. We construct now the vectors **PQ**, **RS** and then the determinant D=|**PQ RS**|. The problem is to determine the sign of D in a guaranteed manner (this sign is then used for another algorithm and a mistake in the sign has a very negative influence).

A classical approach to solve this problem is to consider that the 8 coordinates of the points P, Q, R, S are the unknowns, write 8 equations to determine these unknowns, solve the system and then determine the sign of D. But this approach has drawbacks:

- the system of equations is complex
- it is difficult to solve *exactly*

We may evidently use ALIAS-Maple to solve the system and uses the `Newton` procedure to compute the solution with a sufficient accuracy to state the sign of D. But this is overkill: we do not really want to determine the values of the coordinates of P, Q, R, S but only the sign of D.

Now we may note that the coordinates of the points are bounded as the points are included in the bounding box of the ellipses. Being given ranges for the unknowns we may compute the interval evaluation of the determinant. If the lower bound is positive or the upper bound is negative we may directly state the sign of the determinant. If not we bisect one the unknown coordinates: we has thus a list of boxes with 8 ranges in each box, one for each coordinate. For each box we check if the coordinates ranges allows the points P, Q, R, S to belong to the ellipsis and to the bi-tangents, otherwise we reject the box. Then we compute the interval evaluation of D: if the lower bound is positive we store the box in a special array of boxes, called the positive array and discard it from the list. Similarly if the upper bound is negative we store the box in the negative array and discard it from the list. When the list of boxes is exhausted we look at the number of elements in the positive and negative array: if one array has 0 element, then we have determined the sign of D. Otherwise we consider the array that has the lowest number of elements and restart the bisection procedure on this new list, eliminating boxes that do not describe points on the ellipsis or that are not on the bi-tangent. At the end of this process we may have eliminated all the boxes of the list or, in the worst case, we will have computed the coordinates of P, Q, R, S i.e. we will end up with the classical approach. Our test show however that this is seldom the case and the above procedure is much more faster than solving using the classical approach.

Still in some cases the sign of D cannot be safely computed: this may happen for example when some coordinates have very large values while other have very small one. Numerical round-off errors then prohibit the exact determination of the sign of D, a fact that is detected by interval analysis. But we may still in some cases solve the problem: for that purpose instead of using Cartesian coordinates we may use polar coordinates with different ranges for the unknowns in the determinant or we may use an extended arithmetic. In any case the algorithm is safe as it provides either a sign, which is guaranteed, or will state that the sign cannot be determined with the current arithmetic.

In this example the `MakeF, MakeJ` procedures may be used to generate the C++ code for evaluating the value of D and to check if the points belong to the ellipsis. But then specific C++ code should be written, mostly based on the ALIAS-C++ library.

11.2.3 Simplification procedures

- introduce any a-priori simplification rules

- use the 3B and 2B consistency approach. You may start using the `HullConsistency` and `Simp2B` procedure but you may still have a lot of opportunity to produce efficient simplification procedure. For example if you have an expression like $x\sin(x) + (F(x,y)\sin(x) + G(x,y)) = 0$ you may write down (provided that $\sin(x)$ is a range that does not contain 0) $x = -F(x,y) - G(x,y)/\sin(x) = U$ and consider as new range for x the intersection of U with the current range for x

Chapter 12

Examples

We present here some examples of `Maple` file for solving systems of equations.

12.1 6-body

The problem is here to find the solutions of a set of 6 equations (`eq1` to `eq6`). Two simplification procedures are used together with the 3B method. The `HessianSolve` procedure will allow to get exact results.

```
eq1:=3*(b-d)*(B-D)+B+D-6*F+4:
eq2:=3*(b-d)*(B+D-2*F)+5*(B-D):
eq3:=3*(b-d)^2-6*(b+d)+4*f+3:
eq4:=B^2*b^3-1:
eq5:=D^2*d^3-1:
eq6:=F^2*f^3-1:
with(ALIAS):
‘ALIAS/lib‘:="../Test/Lib":
‘ALIAS/profil‘:="../Profil":
‘ALIAS/single_bisection‘:=2:
‘ALIAS/3B‘:=0:
‘ALIAS/Full3B‘:=1:
‘ALIAS/Full3B_Change‘:=0.1:
‘ALIAS/Delta3B‘:=0.01:
‘ALIAS/Max3B‘:=10^16:
‘ALIAS/Use_Simp_3B‘:=0:
‘ALIAS/debug‘:=1:
‘ALIAS/storage_mode‘:=10:
‘ALIAS/full_simplex‘:=9:
‘ALIAS/min_diam_simplex‘:=1000:
‘ALIAS/diam_simplex‘:=0.1:
u:=[0,10^5]:
SimplexConsistency([eq1,eq2,eq3],[B,D,F,b,d,f],"Simp1");
HullConsistency([eq1,eq2,eq3,eq4,eq5,eq6],[B,D,F,b,d,f],"Simp2");
CatSimp("Simp","Simp1","Simp2",0.01);
```

```
HessianSolve([eq1,eq2,eq3,eq4,eq5,eq6],[B,D,F,b,d,f],
[u,u,u,u,u,u],"Simp");
```

12.2 Ferraris

Here we deal with a non polynomial system of 2 equations with a very large initial search domain. An additional problem is that the second equation involves the interval evaluation of the exponential of a variable that may leads to an overflow. Hence we use the mechanism described in section 2.1.4 to avoid the evaluation of the exponential if the upper bound of the first variable is larger than 1e4.

```
eq1:= -0.25/Pi*x2 - 0.5*x1  + 0.5*sin(x1*x2);
eq2:= exp(1)/Pi*x2 - 2*exp(1)*x1  + (1 - 0.25/Pi)*(exp(2*x1) - exp(1));

EQ:=[evalf(eq1),evalf(eq2)]:
VAR:=[x1,x2]:
INIT:=[[-10^8,10^8],[-10^8,10^8]]:

with(ALIAS):
‘ALIAS/debug‘:=1:
‘ALIAS/single_bisection‘:=2:
‘ALIAS/storage_mode‘:=40:
‘ALIAS/lib‘:="../Test/Lib":
‘ALIAS/profil‘:="../Profil":
‘ALIAS/3B‘:=1:
‘ALIAS/Max3B‘:=5*10^9:
‘ALIAS/Delta3B‘:=0.0001:
‘ALIAS/Full3B‘:=0:
‘ALIAS/Full3B_Change‘:=0.0001:
‘ALIAS/maxgradient‘:=20:
‘ALIAS/maxkraw‘:=1:
‘ALIAS/maxnewton‘:=1:

‘ALIAS/user_FINIT‘:="INTERVAL U;":
ALIAS_F:=proc(fid,i)
if i=2 then
fprintf(fid,"if(Sup(v_IS(1))>1.e4)V(2)=INTERVAL(-1.e6,1.e12);else\n"):
fi:
RETURN(0):
end:

HullConsistency(EQ,VAR,"Simp",0.1);
HessianSolve(EQ,VAR,INIT,"Simp"):
```

To manage the possible overflow problem instead of designing the ALIAS_F procedure by hand we may have also used the procedures described in the section 2.1.5 that create automatically the ALIAS_F and ALIAS_J procedures.

```
with(ALIAS):
with(linalg):
‘ALIAS/gpp_linux‘:="/usr/local/gcc-2.95/bin/g++":
‘ALIAS/debug‘:=1:
‘ALIAS/single_bisection‘:=1:
‘ALIAS/storage_mode‘:=40:
‘ALIAS/lib‘:="../Test/Lib":
‘ALIAS/profil‘:="../Profil":
‘ALIAS/3B‘:=1:
‘ALIAS/Max3B‘:=5*10^9:
‘ALIAS/Delta3B‘:=0.0001:
‘ALIAS/Full3B‘:=0:
```

```
‘ALIAS/Full3B_Change‘:=0.0001:
‘ALIAS/maxgradient‘:=20:
‘ALIAS/maxkraw‘:=1:
‘ALIAS/maxnewton‘:=1:
‘ALIAS/optimized‘:=0:

J:=jacobian(EQ,VAR):

Verify_Problem_Expression(EQ,VAR):
Verify_Problem_ExpressionJ(eval(J),VAR):

HullConsistency(EQ,VAR,"Simp1",0.1);
HessianSolve(EQ,VAR,INIT,"Simp1"):
```

12.3 Wilkinson polynomial

The Wilkinson polynomial of order n is obtained as the product of the terms $(x - i)$ for i from 1 to n. Its expanded form is difficult to solve as soon as n is larger than 12. If `eq` is the polynomial a procedure for solving it is:

```
EQ:=[eq]:
VAR:=[x]:
INIT:=[[-100,100]]:

‘ALIAS/fepsilon‘:=1e-9:
‘ALIAS/epsilon‘:=1e-14:
‘ALIAS/storage_mode‘:=10:
‘ALIAS/3B‘:=1:
‘ALIAS/Max3B‘:=10000:
‘ALIAS/Delta3B‘:=1e-9:
‘ALIAS/optimized‘:=0:
‘ALIAS/rand‘:=500:
‘ALIAS/eps_inflation‘:=1e-10:
‘ALIAS/dist‘:=1e-16:

Rouche(EQ,"p",VAR,"rouche"):
WeylFilter(EQ,VAR,VAR,["automatic"],["symbolic"],"SimpWeyl");
DeflationUP(eq,x,"F","J","simp_quo_new",debug);

CatSimp("SIMP","rouche","NO_3B","simp_quo_new","SimpWeyl",0.1);

SOL:=GradientSolve(EQ,VAR,INIT,"SIMP");
```

To filter the boxes we use `WeylFilter, DeflationUP` and to find the solutions we use `Rouche`.

Chapter 13

Troubleshooting: ALIAS-Maple does not work!

13.1 Compilation problems

In some cases for complex expressions the code produced by ALIAS-Maple cannot be compiled for lack of memory. C++ seems to be indeed quite sensitive to the size of the expressions.

A first possibility is to turn off the optimization flag that is used for the compilation by setting the flag `'ALIAS/optimized'` to 0.

If the compilation cannot still be completed you are confronted to a difficult problem: basically the only solution is to try to simplify the generated expression using auxiliary variables for terms that appear at least twice in the code. The mechanisms described in sections 2.1.2,2.3.2 may allow to perform part of this task automatically.

But many time it will be necessary to modify the C++ program that has been produced especially the procedure involving the expressions, their jacobian and Hessian. The name of this procedure are usually `F`, `J` and `H` respectively and are located usually in the files `_F_.c`, `_J_.c`, `_H_.c`.

Another possibility is to compile separately these files which are included in the main program (usually named `_G..._.C`). For that you will need to modify the code and makefile (which is usually named `_makefile`). For example for the `_F_.c` you will first rename this program `_F_.C`, add the following header:

```
#include <fstream>
#include "Functions.h"
#include "Vector.h"
#include "IntervalVector.h"
#include "IntervalMatrix.h"
#include "IntervalMatrix.h"
#include "IntegerVector.h"
#include "IntegerMatrix.h"
```

Then the makefile will be modified so that `_F_.o` will appear in the dependency. Hence the line:

```
_GS_:_GS_.C $(LIB_SOLVE)
```

will be changed to:

```
_GS_:_GS_.C _F_.o $(LIB_SOLVE)
```

Then indicate that `_F_.o` should be linked by adding this file name before the library flag `LIB_SOLVE`. Hence:

```
$(CC) -I$(INCL) -I$(INCLI) _GS_.C -o _GS_ \
$(LIB_SOLVE) -L$(LIB) $(LD) -lm
```

will be changed to:

```
$(CC) -I$(INCL) -I$(INCLI) _GS_.C -o _GS_ \
_F_.o $(LIB_SOLVE) -L$(LIB) $(LD) -lm
```

An alternative is to avoid generating C++ file for the interval evaluation of expression but rather use the ALIAS-C++ parser to get the interval evaluation of the expressions by an interpretation mechanism. This is possible for most of the ALIAS-Maple procedures by setting the variable `'ALIAS/use_parser'` to 1. To determine if a procedure offers this possibility check the on-line help and look if the variable `'ALIAS/use_parser'` is in the list of the global variables for the procedure.

13.2 Execution problems

To run any program generated by ALIAS-Maple it is necessary that your `PATH` variable always include the current directory.

13.2.1 Wrong results

If ALIAS-Maple returns a wrong result the most probable reason is that the theoretical system of expression has not been correctly translated into a C++ interval equivalent. This may occur if the expression involves numerical values and the value of the `Maple Digits` variable has a low value (by default the value is 10). With a too low value there may be a round-off of any numerical values that may indeed cause a solving problem. Hence `Digits` and the `'ALIAS/digits'` of ALIAS-Maple should be set to a large value (typically 40).

You must also be careful with expression involving rational numbers that may be not translated correctly in C++ (for example 3/9 is usually quite different from 3./9.).

13.2.2 Running out of memory

Even for very large problem it is highly improbable that a program generated by `ALIAS`-Maple will run out of memory as soon as the single bisection mode is chosen (by setting `'ALIAS/single_bisection'` to a value larger than 0) and by choosing the reverse storage mode (by assigning `'ALIAS/storage_mode'` to a value larger than the number of variables.

13.2.3 Large computation time

13.2.3.1 Changing the formulation of the problem

Interval analysis is a very specific domain and require some expertise to be efficient. We give in this section some tricks that may help you.

As a naive user you may want to solve a specific problem but the problem that you have submitted to `ALIAS` may be already a transformed version of the problem you want really to solve and this transformation may be not favorable to the use of interval analysis. Focusing on the problem you want to solve is important for the use of interval analysis.

Rule: *focus on the problem you want to solve*

Let us give you an example: you have a third order polynomial F whose coefficients depend on a set of parameters P and you want to find if there are values of these parameters such that the root x of the polynomial verifies some properties. A natural way of doing that is to use the available generic analytical form of the root and check if you may find parameters such that x satisfy the properties. In terms of interval analysis you have already transformed your problem and this may not be a good idea (the analytical form of the roots of a 3rd order polynomial is badly conditioned).

Rule: *it is usually a bad idea to try to adapt to interval analysis the last step of a solving problem that has already been transformed to fit another method*

Thinking backward what you want really is to verify if the x satisfying $F(x, P_i) = 0$ satisfy the properties. Hence you should use as unknown not only the P but also the x: you add an unknown but the calculation will then involve only the evaluation of the polynomial which is much more simple than the evaluation of the

analytical form of the roots. Note that `ALIAS` C++ provides procedures to determine bounds for the unknown x.

Rule: *introducing new variables for simplifying expressions may be a good strategy*

See section 11.2.2 for another example on how focusing on the problem may help to solve it efficiently. A good strategy is also to learn what are the main problems that have been addressed in interval analysis so that you can later on transform your problem into another one for which solving algorithms are available.

Rule: *investigate which main problems have been addressed in interval analysis so that you may later on transform your problem into another one for which solving algorithms are available*

Typical of such approach is to determine if a parametric matrix may include singular matrices. You may be tempted to derive the determinant of the interval matrix and then to determine if there are values of the unknowns that cancel it. But if you have not been able to manually determine these values this probably means that the determinant is quite complicated, and therefore not appropriate for interval analysis. But determining if a parametric matrix may include a singular matrix is a well-known problem in interval analysis and for solving this problem there are efficient methods that do not require the calculation of the determinant (see section 3.3.3.2).

Another case of failure occurs when the system is badly conditioned. This may occur, for example, when you have large coefficients in an equation while the values of the variable are very small: as `ALIAS` takes into account the round-off errors you will get large interval evaluation even for a small width of the intervals in the input interval. There is no standard way to solve the problem. The first thing you may try is to normalize the unknowns in order to reduce the size of the coefficients. A second approach may be to switch the solving problem to an optimization problem.

Another trick for system solving is to combine the initial equations to get new one. For example consider the system:

```
cons1 := a1 + a2 + a3 + 1 ;
cons2 := a1 + 2*a2*x1 + 3*a3*x1^2 + 4*x1^3 ;
cons3 := a1 + 2*a2*x2 + 3*a3*x2^2 + 4*x2^3 ;
cons4 := a1 + 2*a2*x3 + 3*a3*x3^2 + 4*x3^3 ;
cons5 := a1*(x1+x2) + a2*(x1^2 + x2^2) + a3*(x1^3 + x2^3) + x1^4 + x2^4 ;
cons6 := a1*(x2+x3) + a2*(x2^2 + x3^2) + a3*(x2^3 + x3^3) + x2^4 + x3^4 ;
cons7 := x1 + delta -x2<= 0;
cons8 := x2 + delta -x3<= 0;
```

(which is the `fredtest` example of our Web tests page). As it this system is difficult to solve (the computation time is over 10 minutes with a 3B increment of 0.001). Now we may notice that monomials involving `a1` appear in many equations. Hence we consider adding as equations `cons2-cons1`, `cons2-cons3` and so on. With this additional equations the computation time reduces to about 1mn 20 seconds.

Rule: *Very often introducing additional constraints derived from the set of initial ones will reduce the solving time*

Interval arithmetics plays evidently a large role in the computation time. The interval evaluation of mathematical operator such as sine, cosine,... is relatively expensive. If similar complex expressions appear more than once in an expression it is usually very efficient to compute it only once and to substitute it in the other places (see section 2.1.2 for a procedure that allows this manipulation).

Rule: *try to avoid multiple interval evaluation of the same complex expressions*

13.2.3.2 Choosing the right heuristics

This is a very complex issue: in interval analysis we use a lot of heuristics but it is quite difficult to determine what is the right combination that will be the most efficient to solve a problem. And choosing the right parameters for these heuristics may have a very large influence on the computation time (and we mean a *really* large with a decrease factor of the computation time that may be 10^4).

Usually it is good policy to use filters that may be generated with simplification procedures (see section 4.2). For example the 2B-consistency, see the `Hull_Consistency` procedure, section 4.2.1) with a repeat factor f that is about the diameter of the initial range divided by 1000. Similarly it is good policy to use the 3B method

(see section 4.5) with an ‘ALIAS/Delta3B‘ value similar to the f factor. You may also consider looking at the ALIAS-C++ library that provide specific filters (not all of them are incorporated in ALIAS-Maple) that you may use to write your own specific simplification procedure.

13.2.4 Crash

You have found an ALIAS bug. First check the error message if any. If you get a message of type Abort: Division by Zero (which originates from Bias/Profil) this means that you have tried to evaluate with interval arithmetic an expression that is not allowed: see sections 2.1.4 and 2.1.5 to correct such behavior. A call to the procedures Verify_Problem_Expression, Verify_Problem_ExpressionJ before the filtering and solving procedures will usually be sufficient to solve the problem.

In all other cases we will be happy to have a look at your program and see what has gone wrong.

The bug report should contain as much information as possible so that we can repeat your problem on our computers. Clearly this report should indicate what type of computer you are using, the operating system and the C++/Maple files that have been used. It may also be interesting to give some background information on the problem you want to solve. By giving background information on the problem we may also be able to suggest another, maybe simpler, way to solve it.

13.3 Setting the debug option

If you cannot succeed in finding the right parameters you may use the debug option and to try to understand what is going on.

The algorithms of ALIAS are in general giving some information when running. There is a debug tool with three different levels: the first one (which is the default) is user-oriented and will help you to figure out what is going on, the second one is for expert and the third one is no debug at all. To set the debug option add to your program one of the following sentence:

```
‘ALIAS/debug‘:=1:
```

The value 0 indicates no debug while level 2 is the highest (1 being the default). Then, when you run your program ALIAS will print lines looking like the following one:

```
Current box (11/23,remain:13), Sol: 0 (W=0.1,73.275)
```

This indicates that the algorithm is dealing with box 11 in a list that has 23 elements. Thus 13 boxes are still to be considered as the algorithm will end when all the elements in the list have been considered.

The following part Sol:0 indicates that up to now ALIAS has found 0 solutions (for an optimization problem the value you get here is the current minima or maxima).

The last element, (W=0.1,73.275) indicates that for the current box the minimal width of the intervals in the set is 0.1 while the largest width is 73.275. With these information you may figure out what is going on in the algorithm.

Setting the debug variable to 2 will give you a large amount of information but it is reserved for experts.

Contents

In the index keywords in typeset font indicate variable that are used either in the C++ library (with the exception of the C++ procedure of `BIAS/Profil` that are displayed in normal font) or in the Maple library. In the later case if the keyword is, for example, `permute` the name of the Maple variable is `‘ALIAS/permute‘`.